


ASMesh: Anonymous and Secure Messaging in Mesh Networks Using Stronger, Anonymous Double Ratchet

Alexander Bienstock 
New York University
abienstock@cs.nyu.edu

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

Yi Tang
University of Michigan
yit@umich.edu

ABSTRACT

The majority of secure messengers have single, centralized service providers that relay ciphertexts between users to enable asynchronous communication. However, in some scenarios such as mass protests in censored networks, relying on a centralized provider is fatal. Mesh messengers attempt to solve this problem by building ad hoc networks in which user clients perform the ciphertext-relaying task. Yet, recent analyses of widely deployed mesh messengers discover severe security weaknesses (Albrecht et al. CT-RSA’21 & USENIX Security’22).

To support the design of secure mesh messengers, we provide a new, more complete security model for mesh messaging. Our model captures forward and post-compromise security, as well as forward and post-compromise anonymity, both of which are especially important in this setting. We also identify novel, stronger confidentiality goals that can be achieved due to the special characteristics of mesh networks (e.g., delayed communication, distributed network and adversary).

Finally, we develop a new protocol, called ASMesh, that provably satisfies these security goals. For this, we revisit Signal’s Double Ratchet and propose non-trivial enhancements. On top of that, we add a mechanism that provides forward and post-compromise anonymity. Thus, our protocol *efficiently* provides strong *confidentiality and anonymity* under *past and future* user *corruptions*. Most of our results are also applicable to traditional messaging.

We prove security of our protocols and evaluate their performance in simulated mesh networks. Finally, we develop a proof of concept implementation.

CCS CONCEPTS

• **Security and privacy** → **Public key encryption**;

1 INTRODUCTION

Civil protest movements are often targeted by surveillance and censorship [Yee22, Sta21, GKH16, EHM17, DSKB21, EM22]. In a worst case scenario, protesters cannot communicate via public Internet at all due to a censored or entirely blocked infrastructure. One strategy [Koe19] to circumvent this issue is to set up an ad hoc *mesh network* spanned only by the client devices of individual participants. Using simple smartphone apps, such a mesh network can be realized without the need for special technical equipment. Each participant’s phone is a network-node in such a mesh, only connected via a wireless protocol (WiFi or Bluetooth) to neighbors in its immediate range. In order to realize communication between nodes out of immediate range, the mesh network simply propagates

communication through the entire mesh until the actual receiver is reached.

SECURE MESH MESSAGING. Since protesters are prime targets for adversaries, having secure communication protocols in mesh networks is critical. We focus on this problem in this work.

Having the layered OSI model in mind, one could naively argue that the (messaging) application layer and its underlying network layout can be designed independently. Following this argument, one could simply deploy standard secure messaging protocols (e.g., Signal’s Double Ratchet) in mesh networks. However, as shown by Albrecht et al. [AEP22], deploying secure messaging protocols in mesh networks effectively is non-trivial.

In fact, we illustrate with our definitions and constructions that it is expedient to take account of the special character of ad hoc mesh networks when designing a messaging application (protocol) on top of them. Consider, for example, the following peculiarities of messaging in mesh networks: Depending on the density and the size of the network, information travels for a long time across many hops until it reaches its destination; Each hop has only limited computational and memory resources; The participants of mesh networks both have a larger demand of security and are typically faced with stronger adversaries; Adversaries may not be in *control* of the entire network but only of a sub-set of hops therein; Moreover, such adversaries may not even *eavesdrop* all traffic due to the distributed data transmission.

INSECURE MESH MESSENGERS. Only a few applications provide messaging in mesh networks—the comprehensive list of applications in [ABJM21] shows that almost none directly offer messaging and are actively maintained. The most prominent and widely used application, Bridgefy [Bri], was recently analyzed by Albrecht et al. [ABJM21]; Among other discovered weaknesses, Bridgefy’s protocol allowed tracking of users and extracting the users’ social network. One reason for this is that each packet in the mesh network contained sender and receiver identifiers as well as a list of all nodes who forwarded it. In response to the analysis, Bridgefy implemented Signal’s Double Ratchet [PM16a] and encrypted the list of forwarding nodes with a network-wide shared symmetric key. This was, however, insufficient as a second analysis [AEP22] showed attacks against confidentiality that bypass the implementation of the Double Ratchet; additionally, user tracking and social network extraction was still possible because network-wide encryption only protects against outsiders but not against other, potentially malicious, Bridgefy users.

ANONYMOUS MESSAGING. Preventing user tracking, hiding metadata, and providing anonymity is important for mesh messaging users. Yet, it is notable that implementing Signal’s Double Ratchet

(DR) protocol does not immediately support this goal as the DR does not hide metadata natively. In contrast, DR ciphertexts implicitly reveal to which session they belong and by which user they were sent. The reason is that these ciphertexts re-send the same cryptographic values within communication round-trips and contain counters. Nevertheless, the DR provides confidentiality for communicated payload, even if the local secrets of a session participant are ever exposed in the future—*Forward Security* (FS)—or were exposed in the past—*Post-Compromise Security* (PCS).

To add sender anonymity on top of this, the Signal messenger wraps all DR ciphertexts with the Sealed Sender protocol [jlu18]. This protocol uses the receiver’s static long-term key to hide metadata. Therefore, once the receiver’s local secret key is exposed, metadata of all past and future ciphertexts sent to them is revealed—i.e., anonymity is only preserved if user secrets are never corrupted.

	Confidentiality			Anonymity					CC
	Base	FS	PCS	Base	S	R	FS	PCS	
Signal (DR+Sealed Sender) [PM16a, jlu18]	✓	✓	✓	✓	✓	✗	✗	✗	✗
Bridgefy ¹ [Bri]	(✓)	(✓)	(✓)	✗	✗	✗	✗	✗	✗
Moby [PJW ⁺ 22]	✓	✓	✗	✓	✓	✓	✓	✗	✗
PSEB [PSEB22]	✓	✗	✗	✓	✓	✓	✗	✗	✗
DHRR (★) [DHRR22]	✓	✓✓	✓✓	✓	✓	✓	✓✓	✓✓	✗
ASMesh 1	✓	✓	✓✓	✓	✓	✗	✓	✓✓	✓
ASMesh 2	✓	✓	✓✓	✓	✓	✓	✓	✓✓	✗

Table 1: Property comparison of relevant (mesh) messaging protocols. Base: Confidentiality/Anonymity without corruptions; FS: With Forward Security; PCS: With Post-Compromise Security. S: Anonymity for senders; R: For receivers. CC: Confidentiality & Anonymity with Contact Cooperation (see Sec. 2.2). Note on DHRR [DHRR22]: Inefficient construction for only unidirectional, in-order communication. Two check marks indicate stronger FS resp. PCS guarantees than achieved by plain Double Ratchet [PM16a].

Instead of wrapping ciphertexts within another protocol, various recent works opt to explicitly *remove* certain metadata from ciphertexts but thereby either lose PCS guarantees [PJW⁺22] or do not even aim to provide security under corruption of user secrets [PSEB22]. Furthermore, purely removing metadata comes at the cost of trial decrypting all ciphertexts received in the mesh network, even those not intended for a user, incurring a large efficiency cost.

In a recent work, Dowling et al. [DHRR22] design a sophisticated protocol that hides metadata and provides anonymity with strong FS and PCS guarantees—considerably stronger than those achieved by our or any other related work. However, since their protocol only works for unidirectional communication, fails under re-ordered or dropped ciphertexts, and is rather inefficient, it can be considered primarily as a theoretical benchmark.

Finally, several works (e.g., [PJW⁺22, PSEB22, PHE⁺17]) provide a means to prevent traffic analysis from revealing sender and receiver identities, especially in the case that the adversary eavesdrops the traffic of *all nodes* in the entire network. One of the studied techniques [PHE⁺17, PSEB22] is to let the nodes regularly send dummy ciphertexts for hiding actual communication patterns. Furthermore,

they propose mechanisms to mitigate denial of service and flooding attacks [PJW⁺22]. We view these *engineering* techniques as complementary to any *cryptographic* techniques for anonymity, and thus out of scope for this paper. Yet, we emphasize that *both* techniques are important for a truly anonymous system. Moreover, if the adversary is completely global, i.e., sees every message sent and received by every user, provable anonymity is likely impossible (without blowing up complexity by letting all users flood the network with dummy ciphertexts). Our work aims for a solution that *does not reduce* other security guarantees if there is a *global* adversary, whereas it indeed *strengthens anonymity* compared to related work if the adversary is *not global*, which we consider a realistic assumption in various settings.

Table 1 summarizes the security properties of related work.

CONTRIBUTIONS. Motivated by the lack of secure applications and practical building blocks, as well as the security properties covered by recent analyses [ABJM21, AEP22], we begin in Section 2 with a **new security model for Mesh Messaging**, capturing important guarantees **absent in previous models**. Indeed, our model accounts for **FS** not present in that of [PSEB22], **PCS** not present in that of [PJW⁺22, PSEB22], and (cryptographic) **sender anonymity with FS and PCS** for bidirectional communication not present in that of [PJW⁺22, PSEB22, DHRR22]. Beyond this, we capture *strengthening of all security* against corruptions if the adversary is non-global, called **Contact Cooperation**, which is not present in any of the aforementioned works. In this base model, we opt to forgo receiver anonymity in favor of Contact Cooperation for scenarios in which confidentiality and sender anonymity with stronger resilience against corruptions are more important. With a tweaked version of this model, we require **sender and receiver anonymity with FS and PCS**, but *not* Contact Cooperation.

We construct our first Anonymous and Secure ASMesh protocol to efficiently achieve the security of the base model. One building block of our new ASMesh protocol is based on Signal’s **Double Ratchet** (DR). We revisit the original DR and enhance it with efficient public-key based extensions that **increase its security** against corruptions in Section 3. To overcome the lack of practical solutions for hiding metadata, we then develop a new, **highly efficient Message Anonymizer** (MA) protocol in Section 4 that uses only symmetric cryptography. When composed with our (enhanced) DR variant, this MA protocol offers **strong sender anonymity** guarantees with FS and PCS. Due to its simplicity, it is applicable beyond the mesh messaging setting—e.g., for replacing Sealed Sender in Signal.

The composition of these two building blocks is enriched with routing and re-encryption procedures, which, in total, yields our **new ASMesh** that we detail in Section 5. A (slightly) adjusted version of ASMesh indeed achieves the security of the aforementioned tweaked security model, taking full advantage of the MA protocol to achieve **strong sender and receiver anonymity** guarantees with FS and PCS (with no Contact Cooperation).

We build **proof of concept implementations** with open source code [BRT23c], and evaluate their performance in Section 6. By **simulating** mesh networks for **several deployment scenarios** (also published open source [BRT23c]), we also demonstrate the practicality of our constructions in (down-scaled) reality.

All formal definitions and security proofs are in the appendix.

2 SECURE MESH MESSAGING

We outline our assumptions on the execution environment, considered threats, limitations of our work, and future research directions.

Assumed Environment We consider Secure Messaging in a Mesh Network and focus on (already non-trivial) two-party messaging. In such a mesh network, each *user* is considered as a *node*. There is no other external network which users can use to communicate; users can only send something from their device if they are in close proximity to another node. Therefore, messages may have to travel along a large number of *hops* through the network before reaching the destination. Also, messages may arrive at the recipient out-of-order or even be completely dropped. Furthermore, because of the structure of the network, message receipt acknowledgments are not very useful and thus several copies of the same message may travel through the network, even after successful delivery. Since nodes in the network are often users’ mobile devices, their computational and memory resources can be limited. Thus, Secure Mesh Messaging protocols must be reasonably efficient.

Threat Model Since we consider the usage of mesh networks to come during times of heightened security threats (e.g., government shutdown of the internet during protests), users expect a high level of security against powerful adversaries. Indeed, as a *worst-case*, we consider a global adversary that can eavesdrop on all traffic as it leaves users’ devices. Yet, because of the distributed nature of the mesh network and communication through it, adversaries may not be “all-powerful” in the following sense: they may only have control of a subset of nodes within the mesh network and, hence, may not actually be able to eavesdrop on all traffic. In this situation, our threat model requires stronger security properties.

In this work, we focus on strong *confidentiality* and *anonymity* of ciphertexts under *temporary state exposures*. Due to the already complex nature of our work, we leave the study of *authentication* as future work. Intuitively, we want to retain all confidentiality guarantees that Secure Messengers in normal network settings provide (including Forward and Post-Compromise Security due to users’ state updates). However, we enhance these guarantees along several dimensions: (i) we believe sacrificing marginal efficiency for better confidentiality is important in mesh networks, thus we correspondingly strengthen confidentiality; (ii) we believe anonymity is particularly important in mesh networks, thus we correspondingly strengthen anonymity (if the adversary is non-global); and (iii) we take advantage of the (potentially) limited adversarial view of communication through the mesh network to enable strong security guarantees, which we call *Contact Cooperation*.

Illustration of Contact Cooperation We provide a motivating example for (iii): Suppose that Alice sends a ciphertext c to Bob across the mesh network. While c travels to Bob in the network, Bob is *temporarily* exposed, and then afterwards issues a state update u which he sends through the network, too. Then, before some adversarial node in the network sees ciphertext c , honest party Charlie (who has a session with Bob) sees the update u , then c , and *re-encrypts* c to obtain c' . Thus, if the adversary thereafter only sees c' , all security is still guaranteed *despite* the earlier exposure rendering the *original* ciphertext c insecure.

We again emphasize that Contact Cooperation only *strengthens* security when faced with such a situation; if the adversary is global, we can no longer hope for this cooperation effect, yet, we still require all other mentioned guarantees. In a similar way, our *cryptographic* notion of anonymity downgrades for global adversaries: We only require that the ciphertexts themselves do not reveal information about their origin (and destination, for receiver anonymity), but we do not require obfuscation techniques for the case that all sent and received traffic is observed.

Limitations and Future Work Our work is focused on the formal specification, design, and analysis of efficiently achievable strong confidentiality and anonymity guarantees in two-party communication via mesh networks. For deployment in practice, several challenges remain to be solved by future work. First and foremost, analyzing our construction in the presence of *active attackers* would be a natural, yet complex next step (see [PR18, BRV20, RSS23, CCD⁺20, ACD19, CJSV22, BFG⁺22] for centralized two-party communication definitions with varying levels of active security). Similarly, extending our ideas to *group communication* is important, especially for the motivating mass protest scenario. Notably, none of the mentioned mesh messaging applications in Section 1 propose a solution for groups and several challenges that are already complicated for groups in centralized networks (e.g., concurrency [BDR20] or dynamic membership [BDG⁺22]). We refrain from capturing both aspects—active adversaries and groups—in this work to keep our formal definitions and proofs (in the appendix) comprehensible. Finally, some of our contributions are particularly meaningful against non-global attackers. Focusing on *global attackers* and investigating *non-cryptographic techniques* composed with our ideas is left for future work, too.

2.1 Mesh Messaging API

A Secure Mesh Messaging protocol MM may consist of the following algorithms, the first four of which provide basic functionality:

- $\text{MM.gen} \rightarrow_{\S} (st, pk)$: Generates new public key pk (which also serves as user identifier) with corresponding secret state st for a new user.
- $\text{MM.enc}(st, m, pk) \rightarrow_{\S} (st, (t, i))$: Issues the encryption of message m to receiver public key pk and outputs resulting new state st with encryption of m cached inside; output counters t and i uniquely specify the current point in the session with partner pk
- $\text{MM.bc}(st) \rightarrow_{\S} (st, C)$: Broadcasts the cached set of ciphertexts C to network neighbors.
- $\text{MM.proc}(st, C) \rightarrow_{\S} (st, PKM)$: Processes received ciphertexts C and decrypts those messages that were directed to the executing user; each decrypted message m is output with the sender’s public key pk and corresponding session position (t, i) s.t. $(pk, m, (t, i)) \in PKM$; remaining ciphertexts are cached until next broadcast.

To account for practical and modular constructions, we add:

- $\text{MM.init}(st_0, pk_0, st_1, pk_1) \rightarrow_{\S} (st_0, st_1, C)$: Abstractly captures session initialization between two users pk_0 and pk_1 (with secret states st_0 and st_0 , resp.) with public transcript C .

- $\text{MM.sup}(st, pk) \rightarrow_{\S} st$: Issues a state update for a specific session with public key pk ; the update is cached inside new output state st .

Example Execution When Alice and Bob become protocol users, they generate fresh key material: $\text{MM.gen} \rightarrow_{\S} (st_A, pk_A)$; $\text{MM.gen} \rightarrow_{\S} (st_B, pk_B)$. To exchange messages, they initialize a joint session: $\text{MM.init}(st_A, pk_A, st_B, pk_B) \rightarrow_{\S} (st_A, st_B, c)$. Alice can now encrypt a message to Bob: $\text{MM.enc}(st_A, m, pk_B) \rightarrow_{\S} (st_A, (t, i))$, which her device caches until she is in range of another user’s device; when this happens, these devices exchange cached ciphertexts: $\text{MM.bc}(st_A) \rightarrow_{\S} (st_A, c_A)$; $\text{MM.bc}(st_C) \rightarrow_{\S} (st_C, c_C)$; $\text{MM.proc}(st_A, c_C) \rightarrow_{\S} (st_A, M_A)$; $\text{MM.proc}(st_C, c_A) \rightarrow_{\S} (st_C, M_C)$. Such ciphertext exchanges happen whenever two members of the mesh network meet (e.g., C meets D , then D meets E , etc). When Bob meets a user who caches a ciphertext that decrypts to Alice’s message m , Bob will receive m : $\text{MM.bc}(st_E) \rightarrow_{\S} (st_E, c_E)$; $\text{MM.bc}(st_B) \rightarrow_{\S} (st_B, c_B)$; $\text{MM.proc}(st_B, c_E) \rightarrow_{\S} (st_B, M_B)$; $(pk_A, m, (t, i)) \in M_B$.

Whenever Bob is concerned that his local state was exposed (e.g., his device was temporarily confiscated or stolen), he can update it on demand for each of his sessions: E.g., $\text{MM.sup}(st_B, pk_C) \rightarrow_{\S} st_B$ updates his session with Charlie. Like payload ciphertexts, this update propagates through the network via user meetings. When updates of Bob and payload ciphertexts to Bob are processed by the same user Charlie, Charlie can internally use her refreshed session with Bob to re-encrypt the payload ciphertext—strengthening the security for the remaining transmission through the mesh network.

2.2 Security Properties

In our base security model, formally defined in Appendix B, we capture strong confidentiality and sender anonymity of ciphertexts under temporary state exposures against passive adversaries. (However, we stress that our protocol still uses authenticated encryption in the same manner as the DR, and thus can be conjectured to inherit the DR’s authenticity properties (at least).)

Parties P can execute the protocol with the above defined algorithms. We consider adversaries \mathcal{A} that can temporarily **expose the state** of any party P . This reveals all secrets locally stored by P at the moment of exposure.² Furthermore, \mathcal{A} can **watch and unwatch** any party P at any time, potentially every party in the mesh network. As long as P is watched, \mathcal{A} observes all their meetings and sees all their exchanged ciphertexts. Yet, if neither of two meeting parties is watched, their ciphertext exchange remains hidden, which models that an adversary may only control a subset of the nodes in the mesh network. Against such adversaries, we require:

- **Full Confidentiality and Sender Anonymity**: Ciphertexts only reveal *payload length*, *receiver identity*, and *remaining transmission lifetime* to anyone besides the actual receiver. Yet, exposures of receiver and sender states can inevitably break anonymity or even confidentiality (e.g., of so far unreceived ciphertexts).
- **Confidentiality without Sender Anonymity**: Due to state exposures, some ciphertexts may be confidential,

while not anonymous. Such ciphertexts additionally reveal *sender identity* and *technical session details* (e.g., position within the session, number of passed round-trips, etc.) beyond the above information.

- **Forward Security (FS)**: Confidential (and anonymous) ciphertexts maintain their guarantees even if the states of sender and receiver are exposed in the future.
- **Post-Compromise Security (PCS)**: Even if the states of sender and receiver are exposed, future ciphertexts between them will be confidential (and anonymous) after a short recovery time.
- **Contact Cooperation**: If a node along a ciphertext’s transmission path corresponds to a contact of the receiver, then the security of the corresponding session between this contact and the receiver is added to the ciphertext (via re-encryption; i.e., security may only be strengthened).

Definition 2.1 (Informal). Mesh Messaging protocol MM is secure (according to our base model) if it provides Confidentiality and Sender Anonymity with Forward Security, Post-Compromise Security, and Contact Cooperation.

We provide a formal, *simulation-based* definition in Appendix B. This definition directly models the above ideal protocol behavior, where the main technicality is to capture FS and PCS guarantees via permitted exposures and their effect on the confidentiality and anonymity of challenge ciphertexts. Looking ahead, we require (and achieve) stronger FS and PCS than what is typically achieved by Secure Messenger protocols, such as the Double Ratchet [CCD⁺20, ACD19, CJSV22, BFG⁺22]. More details on this are provided in Section 3.

Our tweaked security model *in addition* to the above items requires **Receiver Anonymity** for fully secure ciphertexts (i.e., the first item above). However, it no longer requires Contact Cooperation. We provide this alternative definition in Appendix B.1.

Definition 2.2 (Informal). Mesh Messaging protocol MM is secure (according to our alternative model) if it provides Confidentiality as well as Sender and Receiver Anonymity with Forward Security and Post-Compromise Security.

2.3 Protocol Overview

We now give a high-level overview of our MM constructions. The constructions have two main ingredients: (i) *Public-Key Double Ratchet* PR (Section 3), which is an enhanced version of the *Double Ratchet*, strengthened with additional public key operations and (ii) A *Message Anonymizer* MA (Section 4) that takes key material from PR to wrap PR ciphertexts in an anonymized, symmetric-encryption-based layer. Intuitively, PR provides the core messaging functionality for parties in the mesh network, while MA anonymizes the ciphertexts output by PR.

Public-Key Double Ratchet The *Double Ratchet* (DR) [PM16a, CCD⁺20, ACD19, BFG⁺22, CJSV22] is an end-to-end secure messaging protocol that allows two users to asynchronously communicate with robust correctness properties. That is, any message sent from one party to another must be immediately decryptable by the receiver, despite any re-ordering or complete drops of messages in

²To avoid requiring impractical non-committing cryptographic primitives, we forbid receiver exposures as long as challenge ciphertexts sent to them were not yet received.

the network. Since in our case, the MM protocol will be the highest-level system actually sending and receiving messages, we just use the part of the DR that produces keys (that are unique for each message). Furthermore, as the DR is a well-studied protocol which we just build on top of, we only briefly describe it here and provide more details in Section 3.

Even though the DR is asynchronous and robust to ciphertext reordering and drops, it proceeds in so-called *ratchets*, roughly corresponding to continuous ping-pong round trips. When Alice starts a new ratchet, Bob agrees that this new ratchet has started as soon as he receives one such ciphertext from this ratchet (yet he still can receive ciphertexts from old ratchets, if needed). At a high-level, with every new ratchet the two parties agree on some new key material that is deterministically and symmetrically used to encrypt and decrypt each ciphertext in that ratchet.

In terms of security, the DR achieves the confidentiality guarantees mentioned above, including (weaker) FS and PCS. Additionally, in this paper we formalize and extend ideas outlined in [ACD19, CZ22] to slightly strengthen the security of the DR with additional public key operations, while maintaining the above robust correctness properties, and only decreasing efficiency slightly: We consider it a reasonable requirement that, if the receiver of a ciphertext is not corrupted, then (even if the sender was corrupted at any point before) the encapsulated key should be secure. In contrast, the original DR does not protect sent ciphertexts against earlier sender state exposures in the same ratchet, since its key material for each ratchet is symmetric. Intuitively, we add an extra layer of Public-Key Encryption which prevents the above attack. That is, only if a receiver is corrupted should the ciphertexts sent to them be decryptable by the adversary. In Section 6, we show that implementing this change only adds ≈ 32 bytes of overhead to each ciphertext when compared to the original DR.

Message Anonymizer The Public-Key Double Ratchet above does not provide anonymity on its own. In contrast, technical variables about sender and session status are contained in the publicly visible PR transcript. To solve this, we design a *Message Anonymizer* (MA) with a quite simple and intuitive construction that is sketched in Figure 1: With each ratchet of the PR, MA takes some of the established key material ck_0 as input. If this ratchet was established by party Alice, then Bob uses it for *his next ratchet* to send to Alice. For each message in Bob’s next ratchet, he will use the key material ck_{i-1} to generate via a Key Derivation Function (KDF) chain: (1) a pseudo-random tag tag_i , (2) an encryption key k_i , and (3) the new chain key ck_i . Bob will then use k_i to encrypt the PR ciphertext and label the resulting outer ciphertext with tag tag_i . Now, since Alice has the original ck_0 , for a message with some tag_i , she can derive ck_{i-1} to compute tag_i and thus the corresponding k_i , with which she obtains the PR ciphertext; Alice actually pre-populates a hash table for this, to avoid inefficient trial decryptions. For all k_i that are unknown to the adversary, the corresponding ciphertexts look pseudo-random and thus anonymous. Furthermore, since the original ck_0 for each ratchet is output by PR, our MA inherits the same FS and PCS for the subsequent ratchet. In Section 6, we show that our MA implementation only adds ≈ 44 bytes to each ciphertext for this considerable anonymity.

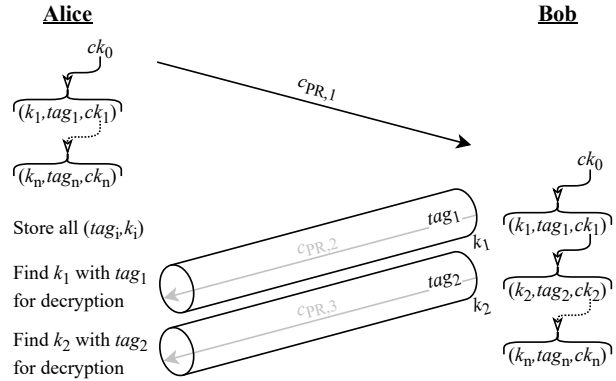


Figure 1: Sketch of Message Anonymizer construction.

ASMesh with Contact Cooperation The construction proceeds intuitively from the overviews provided above: When a Bob wants to send a message, he derives a key via the PR protocol, encrypts the message with that key, and encrypts the resulting non-anonymous metadata via the MA protocol. In this protocol, Bob also attaches the receiver’s identifier to the ciphertext. Upon receipt, Alice obtains the full PR ciphertext by processing the encrypted metadata via the MA protocol, then processes the PR ciphertext, and finally decrypts the actual payload. For updating the MA key material, Alice and Bob alternately extract fresh secrets from the PR execution and feed it into their MA states. Finally, on a higher level, if Charlie processes a ciphertext that is intended for her contact Alice with whom she has initialized a PR session, Charlie uses this session to re-encrypt the ciphertext. Since each ciphertext that arrives at Alice thus may have been re-encrypted several times, she actually has to *recursively* decrypt the ciphertext.

ASMesh protocol MM inherits the confidentiality guarantees of PR, as well as the sender anonymity guarantees of MA, including FS and PCS for both. Furthermore, due to re-encryptions using established PR sessions, we obtain contact cooperation which adds to the ciphertext the security of the PR session between the re-encryptor and the recipient. In Section 6, we show that our MM implementation only adds ≈ 145 bytes to each ciphertext for each re-encryption. Thus, tuning implementation parameters to the given setting such that number of re-encryptions is modest yields an efficient MM with considerable added security.

ASMesh with Receiver Anonymity By removing receiver identifiers from MM ciphertexts and omitting ciphertext re-encryption, our *ASMesh 2* protocol forgoes contact cooperation in favor of receiver anonymity (still without trial decryption via MA).

3 PUBLIC-KEY DOUBLE RATCHET

At the core of our mesh messaging construction, we utilize the key agreement component of the popular Signal Double Ratchet (DR) Algorithm [PM16a, CCD⁺20, ACD19, BFG⁺22, CJSV22]. As mentioned in Section 2.3, the DR allows two users to asynchronously communicate with robust correctness properties. For security, the DR achieves basic end-to-end confidentiality with PCS and FS. Additionally, in this paper we formalize and enhance an idea outlined in [ACD19] to slightly strengthen the security of the DR with an additional layer of public-key operations, while maintaining the

above robust correctness properties, and only decreasing efficiency slightly. In short, the extra security we achieve is that if the receiver of a ciphertext is not corrupted, then (even if the sender was corrupted at any point before), the underlying key should be secure.

We call our strengthened primitive the *Public-Key Double Ratchet* (PR) for which we present the formal security definition in Appendix E, but provide the basic ideas here. As discussed earlier, even though the PR is asynchronous and robust to ciphertext reorderings and drops, it proceeds in so-called *ratchets*, roughly corresponding to rounds. Based on this, the algorithms of PR work as follows:

- $\text{PR.init}(k) \rightarrow (st_A, st_B)$: Takes shared key k (exchanged with an external protocol) and outputs initial states st_A, st_B for the two parties.
- $\text{PR.snd}(st) \rightarrow_{\S} (st, rak, k_{mk}, c, h, (t, i))$: Produces a ciphertext c , associated data h , and an associated *ratchet key* rak and *message key* k_{mk} ; the former key rak is produced *only* once a new ratchet starts, and the latter is output on every send. The algorithm also outputs the ratchet number t and corresponding ciphertext count i within ratchet t .
- $\text{PR.rcv}(st, c, h) \rightarrow (rak, k_{mk}, (t, i))$: Takes a ciphertext c and associated data h , and derives corresponding message key k_{mk} and ratchet key rak ; where k_{mk} is output on every reception and rak is output *only* for new ratchets. Ratchet number t and ciphertext count i within t are output, too.

3.1 PR Building Blocks

To build our PR protocol, we first introduce its building blocks: *Continuous Key Agreement* (CKA) and *Asynchronous Continuous Key Agreement* (ACKA).

Continuous Key Agreement CKA [ACD19, BFG⁺22, DG19] is a well-established key agreement primitive, especially in the context of Secure Messaging. It corresponds to the core *public ratchet* used in the original DR protocol [PM16a]. CKA allows for two parties to *synchronously* send each other key-establishing ciphertexts in a ping-pong fashion; i.e., Alice sends, then Bob, then again Alice, etc., without any repeats. These ping-pongs will correspond exactly to the “ratchets” in our discussion of PR above. The primitive works as follows:

- $\text{CKA.init}(k) \rightarrow (cst_A, cst_B)$: Takes some shared key k (exchanged externally) and outputs initial states cst_A, cst_B for the two parties.
- $\text{CKA.snd}(cst) \rightarrow_{\S} (cs, cct)$: Produces a ciphertext cct and associated secret cs .
- $\text{CKA.rcv}(cst, cct) \rightarrow (cst, cs)$: Takes in a ciphertext cct and derives the associated secret cs .

Intuitively, the secret associated with some cct should be secure as long as the receiver is only corrupted (i) before it last sent a ciphertext or (ii) after it receives cct , corresponding to PCS and FS, respectively. We formally define this in Appendix D.1.

The original group-based instantiation of CKA used by the Double Ratchet [PM16a] is essentially just ping-pong invocations of Diffie-Hellman Key Exchange: Alice starts with exponent x_0 , and Bob with g^{x_0} . Then Bob samples and stores x_1 , sends g^{x_1} , and derives $cs_1 = g^{x_0x_1}$. Upon reception of g^{x_1} , Alice can easily compute cs_1 . She will also delete x_0 at this point, as it is no longer needed. Then, when she wants to next send, Alice samples and stores x_2 ,

sends g^{x_2} , and derives $cs_2 = g^{x_1x_2}$, which Bob can easily derive, too. This process continues for as long as the two parties desire.

For security of, e.g., cs_2 , if Bob is not corrupted after he sends g^{x_1} and before he receives g^{x_2} , then x_1 is not revealed to the adversary, which is a good start. However, if *Alice* is immediately corrupted, the adversary will obtain x_2 , and thus also cs_2 , using public g^{x_1} . Thus slightly weaker security than what we desired is obtained—one also must not corrupt the sender until they again receive a new ciphertext (and thus delete x_2). Bienstock et al. [BFG⁺22] present an improved CKA construction that solves this exact problem while retaining the same communication complexity (i.e., one group element per ciphertext). While this stronger CKA is what we actually use in our PR construction, the reader of the main body can keep in mind the (slightly) less secure instantiation for simplicity, and see Appendix D.1 for more details on the improved instantiation.

Asynchronous Continuous Key Agreement ACKA is what we use to add the extra “public-key” layer to the original Double Ratchet to build our PR construction, as discussed above. As its name suggests, one can think of it exactly as a sort of asynchronous version of the CKA primitive just presented. When Alice starts a new ratchet t , she can now send multiple ciphertexts in that ratchet instead of one, and each will establish a fresh new secret. Alice will only start a new ratchet $t + 2$ with her next send once Bob (i) decrypts one of the ratchet t ciphertexts and (ii) sends his own ratchet $t + 1$ ciphertexts, one of which (iii) Alice receives. In the meantime, each ciphertext that Alice sends is associated with a new *PCS epoch*, pc , for her. Thereby, ratchets and epochs progress independently, which leads to quicker recovery from state exposures. The ACKA algorithms ACKA.init , ACKA.snd , ACKA.rcv resemble those of CKA above, except with the above adjustments. Also, ACKA.snd outputs some bookkeeping metadata md and ACKA.rcv takes it as input.

Similarly to CKA, we expect security for a secret acs associated with some ciphertext $acct$ sent in ratchet t , as long as the following is obeyed: The receiver is only corrupted (i) before it sent the ciphertext that the sender most recently received before sending $acct$ or (ii) after it receives *every* ciphertext in ratchet t (again corresponding to PCS and FS, resp). This is formalized in Appendix D.2.

The basic group-based instantiation of ACKA follows along very similar lines as that of CKA above, with the following changes: Each time Alice sends, she will sample a fresh pair (x, g^x) , store only x as her local secret, and send g^x along with the corresponding PCS epoch, pc . Then, when Bob receives a ciphertext, he will make sure to store Alice’s latest g^x as a public key, by comparing that ciphertext’s pc epoch with those received earlier. When Bob responds, he will always compute the new secret $acs = g^{xy}$ based on the stored (and thus latest) public key g^x from Alice.

As for the basic group-based construction of CKA above, security can be broken if a sender is corrupted immediately after invoking ACKA.snd . (However, if this does not happen, the same argument can be used to show that security is reached as long as if the requirements on corruption of the receiver listed above are not violated.) We thus enhance our actual ACKA construction used in PR with the same technique from [BFG⁺22] for CKA to fix this problem. Again, the reader can for now remember the above (slightly) less secure variant for simplicity, and see Appendix D.2 for details on the improved design.

3.2 Public-Key Double Ratchet Construction

Our PR construction is presented in Figure 2. As stated above, it follows the original DR, using the “public ratchet” backbone provided by CKA and a “symmetric ratchet” provided by a KDF chain, plus our extra “public-key” layer provided by ACKA.

Both parties are initialized with shared *root key rk* and initial *chain key ck* (line 00), as well as material for instantiating CKA and ACKA (lines 02-03). Each party will maintain their view of the current ratchet number with variable t_p (line 01).

```

Proc PR.init( $k$ )
// Items in lines 00-01 used for both A, B
00 ( $rk, ck[0], k_{ACKA}, k_{CKA}$ )  $\leftarrow k$ 
01  $t_p, i, \ell_{prv} \leftarrow 0; cct, ck[\cdot] \leftarrow \perp; turn \leftarrow A$ 
02 ( $cst_A, cst_B$ )  $\leftarrow$  CKA.init( $k_{CKA}$ )
03 ( $acst_A, acst_B$ )  $\leftarrow$  ACKA.init( $k_{ACKA}$ )

Proc PR.snd( $st$ )
04  $rak \leftarrow \perp$ 
05 If  $turn = P$ :
06   ( $cs, cct$ )  $\leftarrow_{\$}$  CKA.snd( $cst$ )
07   ( $rk, ck[t_p + 1], rak$ )  $\leftarrow H_1(rk, cs)$ 
08    $t_p \leftarrow t_p + 1; i \leftarrow 0; turn \leftarrow \bar{P}$ 
09   ( $ck[t_p], drk$ )  $\leftarrow H_2(ck[t_p])$ 
10   ( $acs, acct, md$ )  $\leftarrow_{\$}$  ACKA.snd( $acst$ )
11    $h \leftarrow (md, t_p, i, cct, \ell_{prv})$ 
12    $k_{mk} \leftarrow H_3(acs, drk, (h, acct))$ 
13    $i \leftarrow i + 1$ 
14   Return ( $rak, k_{mk}, acct, h, (t_p, i - 1)$ )

Proc PR.rcv( $st, acct, h$ )
15 ( $md, t, i', cct', \ell$ )  $\leftarrow h; rak \leftarrow \perp$ 
16 If  $t > t_p$ :
17    $ck[t_p] \leftarrow \perp; \ell_{prv} \leftarrow i; t_p \leftarrow t; turn \leftarrow P$ 
18   end-ratch( $ck[t_p - 2], \ell$ )
19   ( $cst, cs$ )  $\leftarrow$  CKA.rcv( $cst, cct'$ )
20   ( $rk, ck[t], rak$ )  $\leftarrow H_1(rk, cs)$ 
21    $drk \leftarrow$  find-and-del-drk( $ck[t], t, i'$ )
22   ( $acst, acs$ )  $\leftarrow$  ACKA.rcv( $acst, acct, md$ )
23    $k_{mk} \leftarrow H_3(acs, drk, (h, acct))$ 
24   Return ( $rak, k_{mk}, (t, i)$ )

```

Figure 2: Construction of PKDR. H_1, H_2, H_3 are KDFs. We do not make states explicit for simplicity.

When a party P wishes to send a new ciphertext using PR, it first checks if it is its turn to start a new ratchet $t_p + 1$ (line 05). If so, it invokes CKA.snd to create a new CKA ciphertext and inputs the resulting secret cs with rk to a KDF H_1 to create the new root key and the initial chain key $ck[t_p + 1]$ (lines 06-07) for the new ratchet, as well as key rak . This last key rak will be output and used for initializing Message Anonymizer ratchets in Section 4. Otherwise, it will have the chain key for its current sending ratchet already stored. Next, it inputs the current ratchet’s chain key $ck[t_p]$ into another DKF H_2 to get the next double ratchet key drk and an updated chain key $ck[t_p]$ (line 09). Finally, it invokes ACKA.snd to create a new ACKA ciphertext and inputs the resulting secret

acs with the double ratchet key drk to a final KDF H_3 to output the associated key k_{mk} for this invocation of PR.snd (lines 10-12). By combining these secrets, we intuitively amalgamate their security guarantees. The PR ciphertext will include just the fresh ACKA ciphertext $acct$ (line 14), while the associated data h will include the CKA ciphertext cct for the current ratchet, and some additional bookkeeping metadata (line 11). This metadata includes the sender’s current ratchet number view t_p , the current ciphertext number i within this ratchet, and the number of ciphertexts i sent in its last sending ratchet ℓ_{prv} . Note that in our Mesh Messaging construction, this associated data will actually be anonymized by the Message Anonymizer (see Section 4).

When a party P wishes to receive a new ciphertext, it first checks if the ciphertext started a new ratchet, by comparing its current ratchet count t_p with that included in the ciphertext (line 16). If so, it invokes CKA.rcv on the included CKA ciphertext cct and inputs the resulting secret cs with rk to H_1 to create the new root key and initial chain key $ck[t]$ (lines 19-20) for the new ratchet (and also the additional key rak to be output, as above). It also uses the received ℓ in the associated data to derive the remaining double ratchet keys in its previous receiving ratchet for yet unreceived ciphertexts (i.e., up to the ℓ -th such key), and deletes the corresponding chain key $ck[t_p - 2]$, via subroutine end-ratch($ck[t_p - 2], \ell$) (line 18).

In either case, it uses $ck[t]$, and the received ciphertext number within that ratchet, i , to derive the appropriate double ratchet key drk using subroutine find-and-del-drk (line 21). If this stateful subroutine has not already derived the (t, i) -th double ratchet key, it will invoke H_2 the appropriate number of times on $ck[t]$ to do so, storing the intermediate keys derived along the way. Note that some double ratchet keys derived by subroutines end-ratch and find-and-del-drk may remain unused for some time. This, along with the fact that H_2 can be deterministically used as many times as needed to get the i -th double ratchet key of the ratchet, is in part what allows PR to be robust to out-of-order messages: these double ratchet keys will be stored until their corresponding ciphertexts are delivered (at which point they are deleted). Finally, PR.rcv invokes ACKA.rcv on the included ACKA ciphertext $acct$ and inputs the resulting secret acs with the double ratchet key drk to H_3 to output the associated key k_{mk} for this invocation of PR.rcv (lines 22-23).

Security Argument Our construction PR indeed achieves the intuitive security properties listed at the beginning of this section. First, it inherits the PCS and FS of the Double Ratchet: if the sender of a ciphertext c in ratchet t is only corrupted before ratchet t started or after sending c , and the receiver is only corrupted before starting the previous ratchet $t - 1$ or after receiving c , then we get security from that of the CKA primitive, along with the security of KDFs H_1, H_2, H_3 . Furthermore, if a party sends a ciphertext c in a ratchet t , then as long as the receiver is only corrupted (i) before it sent the last ciphertext received by the sender or (ii) after it receives every ciphertext for ratchet t , then we get security from the ACKA primitive and KDF H_3 (sender corruption not restricted in this case).

As explained in Section 5, the Simulator for our eventual Mesh Messaging construction will have to simulate ciphertexts and corrupted states *non-chronologically*; i.e., some i -th ciphertext of a ratchet t may have to be produced by the simulator before it sees any information on the preceding ciphertexts or states (possibly

even for many preceding ratchets). We can achieve this for our PR by noting that it is *history-independent*: First, since ACKA and CKA secret states and ciphertexts will just be random elements independent of their history, we can sample them accordingly at any time, even out of order. Furthermore, we can model H_1, H_2, H_3 as programmable random oracles so that we can adequately “explain” states and ciphertexts that were already simulated when we later need to simulate their preceding state and ciphertext history.

4 MESSAGE ANONYMIZER

Neither the original Double Ratchet [PM16a] nor our strengthened PR variant from Section 3 provide anonymity. In fact, several components of the ciphertexts in both constructions depend on the current session status, which essentially reveals the session itself and thus also its participants. We make these components explicit in Section 3 as the associated data h output by PR.snd. This associated data h is what will be protected (i.e., encrypted) by the generic wrapper protocol that we call *Message Anonymizer* (MA).

We first emphasize that a user who wishes to use a MA protocol has *many sender states*, one for each other user it communicates with, but *a single receiver state* that works for *multiple, independent sessions* in parallel. The reason is that the sender always knows which state to use for sending a ciphertext in a particular session. In contrast, a ciphertext should hide its sender and its session, unless the right key is used. Hence, when receiving a ciphertext, not even the receiver should know in which session to process it *before* using the right key. Therefore, algorithm MA.dec_R below has to detect the right session (among all possible ones) for successful decryption internally. With this in mind, we specify the following MA API:

- MA.up_R(st_R, usr, k) $\rightarrow st_R$: On input user usr and fresh symmetric key k , updates the receiver state st_R for the corresponding session with usr ($usr = pk$ in our MM protocol); Initially, the state is empty: $st_R = \perp$.
- MA.up_S(st_S, k) $\rightarrow st_S$: Updates sender state st_S with fresh symmetric key k ; Initially, the state is empty: $st_S = \perp$.
- MA.enc_S(st_S, m) $\rightarrow_{\S} (st_S, c)$: Turns message m into anonymous ciphertext c .
- MA.dec_R(st_R, c) $\rightarrow (st_R, m)$: Derives message m from ciphertext c .

For establishing fresh symmetric keys between senders and receivers at initialization and state updates, an MA scheme depends on an external protocol. Our mesh messaging construction, therefore, carefully intertwines PR and MA protocols. Furthermore, MA schemes are parameterized by variable fut that fixes the *tolerated* number of sequentially *dropped* ciphertexts.³ This means that a ciphertext is successfully decrypted with algorithm MA.dec_R if least one of the last fut ciphertext in the session between sender and receiver was successfully decrypted, too.

Security The intuitive security requirements for MA schemes are relatively simple: A sent ciphertext c must be indistinguishable from a random ciphertext under fresh symmetric key k , unless (1) sender state st_S was exposed between establishing k (via MA.up_S) and sending c , or (2) corresponding receiver state st_R was exposed between

establishing k (via MA.up_R) and receiving c . Our mesh messaging construction refreshes symmetric keys with every ratchet, which leads to strong FS and PCS anonymity guarantees.

We refrain from formally defining security of MA since our overall mesh messaging security definition encodes the required anonymity guarantees already. Extracting the MA-specific components from this definition and exploring MA as a general primitive remains an interesting question for future work.

```

Proc MA.upR( $st = (ht, ST), usr, k$ )
00 If necessary, initialize  $ST[usr]$ 
01  $(t_{now}, i_{now}, i_{nxt}, ck_{now}, ck_{nxt}) \leftarrow ST[usr]$ 
02 Require  $ck_{nxt} = \perp$ 
03  $ck_{nxt} \leftarrow k$ 
04 For all  $i_{nxt} : 0 \leq i_{nxt} < fut$ :
05    $(mk, tag, ck_{nxt}) \leftarrow H(ck_{nxt})$ 
06    $ht.add(\{tag, (t_{now} + 1, i_{nxt})\}, (usr, mk))$ 
07  $ST[usr] \leftarrow (t_{now}, i_{now}, i_{nxt}, ck_{now}, ck_{nxt})$ 
08 Return  $st = (ht, ST)$ 

Proc MA.upS( $st, k$ )
09 If  $st = \perp$ :  $t, i, \ell_{prv} \leftarrow 0$ 
10 Else:  $(t, i, \ell_{prv}, ck) \leftarrow st$ ;  $t \leftarrow t + 1$ ;  $\ell_{prv} \leftarrow i - 1$ ;  $i \leftarrow 0$ 
11 Return  $st = (t, i, \ell_{prv}, k)$ 

Proc MA.encS( $st = (t, i, \ell_{prv}, ck), md$ )
12  $(mk, tag, ck) \leftarrow H(ck)$ 
13  $c' \leftarrow_{\S} AE.enc(mk, (t, i, \ell_{prv}, md), tag)$ 
14 Return  $(st = (t, i + 1, \ell_{prv}, ck), c = (tag, c'))$ 

Proc MA.decR( $st = (ht, ST), c = (tag, c')$ )
15  $(usr, mk) \leftarrow ht.acc(tag)$ 
16  $m \leftarrow AE.dec(mk, c, tag)$ 
17 Require  $\perp \neq m = (t', i', \ell'_{prv}, md)$ 
18  $ht.rem(tag)$ 
19  $(t_{now}, i_{now}, i_{nxt}, ck_{now}, ck_{nxt}) \leftarrow ST[usr]$ 
20 Require  $t' \leq t_{now} \vee (t' = t_{now} + 1 \wedge ck_{nxt} \neq \perp)$ 
21 If  $t' > t_{now}$ :
22   For all  $i : \ell_{prv} < i \leq i_{now}$ :  $ht.rem((t_{now}, i))$ 
23   For all  $i : i_{now} \leq i \leq \ell_{prv}$ :
24      $(mk, tag, ck_{now}) \leftarrow H(ck_{now})$ 
25      $ht.add(\{tag, (t_{now}, i)\}, (usr, mk))$ 
26    $t_{now} \leftarrow t'$ ;  $i_{now} \leftarrow i_{nxt}$ ;  $i_{nxt} \leftarrow 0$ 
27    $ck_{now} \leftarrow ck_{nxt}$ ;  $ck_{nxt} \leftarrow \perp$ 
28   For all  $i : i_{now} \leq i < i' + fut$ :
29      $(mk, tag, ck_{now}) \leftarrow H(ck_{now})$ 
30      $ht.add(\{tag, (t_{now}, i)\}, (usr, mk))$ 
31    $ST[usr] \leftarrow (t_{now}, i' + fut - 1, i_{nxt}, ck_{now}, ck_{nxt})$ 
32 Return  $(st = (ht, ST), md)$ 

```

Figure 3: Construction of MA, using hash table HT = (HT.init, HT.add, HT.acc, HT.rem).

4.1 Construction

The core design of our MA construction is in Figure 3, minor additional details are in Figure 19, and a sketch is in Figure 1.

³Most messengers limit “drop-tolerance”: Rösler et al. [RMS18] report $fut = 2000$ for Signal, and von Arx and Paterson [vAP22] report $fut \in \{400, 2000\}$ for different Telegram clients.

In principle, Alice establishes an initial key with Bob, from which both users derive multiple key-tag pairs; Bob then encrypts his messages to Alice with the derived keys and attaches the corresponding tags, such that Alice can use the attached tags to find the matching keys for decryption. In more detail, the protocol execution flows as follows: Receiver Alice starts by inputting established fresh key k in a session with user Bob ($usr = B$) to $MA.up_R$. This algorithm first pre-computes fut many key-tag pairs (Figure 3, lines 04-06) for the upcoming ratchet $t_{now} + 1$ by iterating through a KDF chain. Using a hash table, each derived key mk is stored as the value for two types of tags: a pseudo-random string tag and an index (t, i) for ratchet t and number i in t . Querying the hash table with either of both tags provides the corresponding value mk . This pre-computation of keys and tags in the upcoming ratchet is necessary for Alice to be prepared for detecting and decrypting anonymous ciphertext from Bob in this next ratchet. Bob uses the (same) established key k in algorithm $MA.up_S$ to actually start the next ratchet; during this, he fixes the previous ratchet's length ($\ell_{prv} \leftarrow i - 1$). To anonymize message md (which is metadata from PR in our MM construction), algorithm $MA.enc_S$ derives the next key-tag pair, encrypts md , and attaches the tag to the resulting ciphertext. Using the attached tag, Alice obtains the matching key in algorithm $MA.dec_R$ via the hash table, decrypts message md , and removes the tagged item from the hash table. If the decrypted ciphertext belongs to the next ratchet, Alice removes all unnecessarily pre-computed items from the hash table (line 22) and pre-computes both the remaining (relevant) key-tag pairs for the current ratchet (lines 23-25) and fut key-tag pairs for the new ratchet (lines 28-30).

Security Argument Our construction computes pseudo-random key-tag pairs with a forward secure KDF chain, and anonymizes input messages via encryption. By regularly re-starting the forward secure KDF chain via updates with fresh keys k , we inherit PCS guarantees from the (external) mechanism that establishes these keys k —i.e., from the PR protocol. For full FS, we note that Alice and Bob delete all key-tag pairs immediately after their actual *use*.

Broader Context By using only symmetric cryptography and avoiding trial-decryptions, our MA construction is extremely efficient. As shown in Section 5, the composition with the (Public-Key) Double Ratchet is very natural and enables anonymity with strong FS and PCS. Simultaneously, the construction is simple, which supports implementation and deployment. Therefore, we believe this sub-result is meaningful beyond our paper. For example, replacing the Sealed Sender [jlu18] in Signal by our MA construction strengthens anonymity at similar efficiency⁴. Finally, we note that limiting tolerated ciphertext drops is common in most messengers.³

5 BUILDING ASMESH

We augment the outlined description of our Mesh Messaging (MM) protocol *ASMesh* from Section 2 with technical details here. The intuition for the protocol is that it protects the confidentiality of sessions by encrypting the payload with keys derived from the Public-Key Double Ratchet PR from Section 3, and it protects anonymity by encrypting the non-anonymous parts of the resulting PR ciphertexts by encrypting them (again) with the Message Anonymizer MA

⁴The asymmetric encryption overhead in Sealed Sender may cancel out maintaining memory for our hash table.

from Section 4. In *ASMesh 1*, every node that forwards a ciphertext to one of their contacts re-encrypts this ciphertext (like normal payload) to provide contact cooperation.

For didactic reasons, we split the presentation into (1) key generation and session initialization in Figure 4, (2) message encryption, session update, and ciphertext broadcast in Figure 5, as well as (3) ciphertext processing in Figure 6. As elaborated in upcoming Section 5.2, code marked in **blue** is relevant for *ASMesh 1* with *contact cooperation* but irrelevant for *ASMesh 2* with *receiver anonymity*; the opposite holds for code marked in **green**. We start here with a description of the former, before describing the changes for the latter in Section 5.2; in short, the latter removes receiver identifiers from ciphertexts and omits all re-encryption procedures.

Parameters Depending on the particular deployment scenario (e.g., number of users, number of contacts per user, meeting frequency, etc), the delivery behavior of our MM construction can be adjusted with parameters (pt, nd, fut) : pt fixes the maximal transmission-*path length* (i.e., number of hops that sequentially forward a ciphertext aka. *lifetime*), nd fixes the transmission-*graph degree*⁵ (i.e., the number sequential meetings at which a single hop keeps forwarding the same ciphertext), and fut fixes the *tolerated* number of sequentially *dropped* ciphertexts. In summary, a ciphertext is successfully delivered if the receiver node is in the transmission graph and not more than fut ciphertext in the session between sender and receiver were dropped sequentially.

```

Proc MM.gen
00  $(sk_{me}, pk_{me}) \leftarrow_{\$} NK.gen$ 
01  $ST_{MA}[\cdot] \leftarrow \perp; st_{MA} \leftarrow \perp; ST_{PR}[\cdot] \leftarrow \perp$ 
02  $st \leftarrow (sk_{me}, pk_{me}, (ST_{MA}, st_{MA}), ST_{PR}, C = \emptyset)$ 
03 Return  $(st, pk)$ 

Proc MM.init( $st^0, pk^0, st^1, pk^1$ )
04  $(sk_{me}^\beta, pk_{me}^\beta, (ST_{MA}^\beta, st_{MA}^\beta), ST_{PR}^\beta, C^\beta) \leftarrow st^\beta$ 
05 Require  $pk^\beta = pk_{me}^\beta$ 
06  $(sk_{NK}^\beta, pk_{NK}^\beta) \leftarrow_{\$} NK.gen$ 
07  $k_{EE} \leftarrow NK.eval(sk_{NK}^0, pk_{NK}^1)$ 
08  $k_{ER} \leftarrow NK.eval(sk_{NK}^0, pk_{NK}^1)$ 
09  $k_{SE} \leftarrow NK.eval(sk_{me}^0, pk_{NK}^1)$ 
10  $(k_{PR}, k_{MA}^0, k_{MA}^1) \leftarrow G(k_{EE} \oplus k_{ER} \oplus k_{SE})$ 
11  $(st_{PR}^0, st_{PR}^1) \leftarrow PR.init(k_{PR})$ 
12  $ST_{PR}^\beta[pk^{1-\beta}] \leftarrow st_{PR}^\beta$ 
13  $st_{MA}^\beta \leftarrow MA.up_{PR}(st_{MA}^\beta, pk^{1-\beta}, k_{MA}^\beta)$ 
14  $ST_{MA}^{1-\beta}[pk^\beta] \leftarrow MA.up_S(\perp, k_{MA}^\beta)$ 
15  $st^\beta \leftarrow (sk_{me}^\beta, pk_{me}^\beta, (ST_{MA}^\beta, st_{MA}^\beta), ST_{PR}^\beta, C^\beta)$ 
16  $c^\beta \leftarrow (pk_{me}^\beta, pk_{NK}^\beta)$ 
17 Return  $(st^0, st^1, \{c^0, c^1\})$ 

```

Figure 4: Key generation and session initialization. Lines with β are executed once for each $\beta \in \{0, 1\}$.

⁵Due to sender anonymity, update-effectiveness in transmission, and hiding of the transmission path, “transmission paths” may contain circles s.t. the “transmission tree” is a *graph*.

Initialization A new user starts with generating a new public-key-state pair (Figure 4, lines 00-02). We treat the public key as a user identifier, which means that it remains unchanged throughout the protocol execution. A user can only change it by generating a fresh, independent identity. For simplicity, we instantiate the long-term public key based on Non-Interactive Key Exchange (line 00).

Using a combination of Non-Interactive Key Exchanges (NIKE), we give an example initialization which can be considered a variant of the X3DH protocol [PM16b] (lines 06-10). Alternative instantiations could, e.g., be based on forward-secure KEM [GM15, GHJL17]. Since the implementation choice depends on the deployment setting and is independent of the rest of our protocol, we kept this component simple. Using the established symmetric key (line 10), the session participants derive initial PR states for this session (lines 11-12), add this session to their MA receiver states (line 13), and derive initial MA sender states for this session (line 14).

```

Proc MM.enc( $st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C), m, pk$ )
00 Require  $ST_{PR}[pk] \neq \perp$ 
01  $(S_{MA}, ST_{PR}, c_{MA-PR-AE}, (t, i))$ 
    $\leftarrow_{\S} \text{enc}(pk_{me}, S_{MA}, ST_{PR}, M:m, pk)$ 
02  $C \stackrel{\cup}{\leftarrow} \{(nd, (pt, pk, C:c_{MA-PR-AE}))\}$ 
03 Return  $(st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C), (t, i))$ 

Proc MM.sup( $st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C), pk$ )
04 Require  $ST_{PR}[pk] \neq \perp$ 
05  $(S_{MA}, ST_{PR}, c_{MA-PR-AE}, (t, i))$ 
    $\leftarrow_{\S} \text{enc}(pk_{me}, S_{MA}, ST_{PR}, \epsilon, pk)$ 
06  $C \stackrel{\cup}{\leftarrow} \{(nd, (pt, pk, U:c_{MA-PR-AE}))\}$ 
07 Return  $(st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C), (t, i))$ 

Sub-Proc enc( $pk_{me}, S_{MA} = (ST_{MA}, st_{MA}), ST_{PR}, m, pk$ )
08  $(ST_{PR}[pk], k_{MA}, k_{AE}, c_{PR}, md, (t, i))$ 
    $\leftarrow_{\S} \text{PR.snd}(ST_{PR}[pk])$ 
09 If  $k_{MA} \neq \perp$ :  $st_{MA} \leftarrow_{\S} \text{MA.upr}(st_{MA}, pk, k_{MA})$ 
10  $(ST_{MA}[pk], c_{MA}) \leftarrow_{\S} \text{MA.encS}(ST_{MA}[pk], (pk_{me}, md))$ 
11  $c_{AE} \leftarrow_{\S} \text{AE.enc}(k_{AE}, m, (c_{MA}, c_{PR}))$ 
12  $c \leftarrow (c_{MA}, c_{PR}, c_{AE})$ 
13 Return  $(S_{MA} = (ST_{MA}, st_{MA}), ST_{PR}, c, (t, i))$ 

Proc MM.bc( $st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C)$ )
14  $C' \leftarrow \emptyset$ ;  $C'' \leftarrow \emptyset$ 
15 For all  $(nd, c) \in C$ :
16   If  $nd > 1$ :  $C' \stackrel{\cup}{\leftarrow} \{(nd - 1, c)\}$ 
17    $C'' \stackrel{\cup}{\leftarrow} \{c\}$ 
18 Return  $(st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C'), C'')$ 

```

Figure 5: Message encryption, on-demand session update, broadcast, and encryption sub-procedure.

Sending Message encryption (Figure 5, lines 00-03) and session updates (lines 04-07) follow the same principle: Sub-procedure enc computes a ciphertext that contains the payload (line 01) resp. an empty string (line 05). Combined with receiver public key pk and maximal lifetime counters for *remaining broadcasts* $nd = nd$ as well as *path length* $pt = pt$, this ciphertext is cached (lines 02,06).

The actual ciphertext computation in sub-procedure enc begins with executing the PR protocol, which provides a fresh message

key k_{AE} and, if this starts a new ratchet in this session, a fresh MA key k_{MA} (line 08). This fresh MA key k_{MA} updates the sender’s underlying MA *receiver state* (to prepare for the next ratchet in this session) (line 09). Using the underlying MA *sender state* for this session, the non-anonymous part of the PR ciphertext md is anonymized (line 10). Lastly, the actual payload message m is encrypted with an AEAD cipher (line 11).

Algorithm MM.bc broadcasts all cached ciphertexts C'' and re-caches those with *remaining broadcast* counter $nd > 1$ (lines 15-17).

Receiving When a party processes ciphertexts C'' , algorithm MM.proc in Figure 6 first uses recursive sub-procedure rec-dec to decrypt those ciphertexts that are directed to this party (lines 00-02). Next, all received ciphertexts that are not cached yet and have a *remaining path length* of $pt > 1$ are added to the cache (lines 03-04). Finally, ciphertexts are re-encrypted if the executing party maintains a session with the respective receiver (lines 05-12). For better efficiency, re-encryption aggregates all ciphertexts directed to the same receiver (line 05). Each aggregated ciphertext, computed with sub-procedure enc from Figure 5, is combined with receiver public key pk and lifetime counters (lines 11-12). We evaluate and balance delivery success, communication complexity, and computation overhead in our evaluation in Section 6. We also discuss and extend our simple policy that sets nd to the aggregated maximum (line 10) and pt to the aggregated average (line 07).

Recursive decryption reverses (re-)encryption. This begins with using the MA scheme to identify the corresponding encryptor of this encryption layer and deanonymize the corresponding PR ciphertext components (line 16). Subsequently, the PR protocol is executed to compute message key k_{AE} and, if this starts a new ratchet in the session with this layer’s sender, MA key k_{MA} (line 18). MA key k_{MA} updates the receiver’s underlying MA *sender state* for this session (to prepare for the next ratchet in the session) (line 21). The actual payload m' is AEAD decrypted with k_{AE} (line 19). If the payload consists of aggregated ciphertexts (line 25), recursive decryption is applied on each component ciphertext (lines 27-29).

5.1 Security

In addition to formalizing *existing expectations* for secure mesh messengers, our fresh look on messaging in mesh networks offers *new perspectives* on desired security goals. With our presented protocol, we achieve these goals. First and foremost, it achieves confidentiality with *stronger FS* and *PCS* guarantees due to our advanced PR protocol. Secondly, using our efficient and simple MA wrapper protocol, it provides similarly *strong anonymity* guarantees. Both of these properties are important for targeted user groups of mesh messengers. Independent of that, our new solutions also apply to centralized messaging applications (e.g., Signal or WhatsApp).

A crucial component of our protocol is the re-encryption mechanism that provides stronger confidentiality and anonymity. In particular, by *strengthening security of ciphertexts in transmission*, the weakening effect of long round-trip times and, therefore, low session update frequencies is alleviated.

We prove security for the presented construction in Appendix C.3, where we also formalize the following theorem:

THEOREM 5.1 (INFORMAL). *Taking secure instances of NIKE NK, PRG G (modeled as a random oracle), PR PR, MA MA (based on*

```

Proc MM.proc( $st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C), C''$ )
00 For all  $(pt, pk_{me}, c) \in C''$ :
01    $(S_{MA}, ST_{PR}, PKM) \leftarrow \text{rec-dec}(pk_{me}, S_{MA}, ST_{PR}, c)$ 
02   If  $PKM \neq \emptyset$ :  $C'' \leftarrow C'' \setminus \{(pt, pk_{me}, c)\}$ 
03 For all  $(pt, pk', c) \in C'' : pt > 1 \wedge (\cdot, (\cdot, pk', c)) \notin C$ :
04    $C \stackrel{\cup}{\leftarrow} \{(nd, (pt - 1, pk', c))\}$ 
05 For all  $pk' : ST_{PR}[pk'] \neq \perp \wedge \exists(\cdot, (\cdot, pk', C:\cdot)) \in C$ :
06    $nd \leftarrow 1; c \leftarrow \epsilon$ 
07    $pt \leftarrow \text{avg}(\{pt' : (nd', (pt', pk', C:c')) \in C\})$ 
08   For all  $(nd', (pt', pk', C:c')) \in C$ :
09      $C \leftarrow C \setminus \{(nd', (pt', pk', C:c'))\}$ 
10      $nd \leftarrow \max(nd, nd')$ ;  $c \leftarrow (c, C:c')$ 
11    $(S_{MA}, ST_{PR}, c_{MA-PR-AE}, (t, i))$ 
12    $\leftarrow_{\S} \text{enc}(pk_{me}, S_{MA}, ST_{PR}, R:c, pk)$ 
13    $C \stackrel{\cup}{\leftarrow} \{(nd, (pt, pk', C:c_{MA-PR-AE}))\}$ 
13 Return  $(st = (sk_{me}, pk_{me}, S_{MA}, ST_{PR}, C), PKM)$ 

Sub-Proc rec-dec( $pk_{me}, (ST_{MA}, st_{MA}), ST_{PR}, c$ )
14  $PKM \leftarrow \emptyset$ 
15 Require  $c = U:(c_{MA}, c_{PR}, c_{AE}) \vee c = C:(c_{MA}, c_{PR}, c_{AE})$ 
16  $(st_{MA}, (pk, md)) \leftarrow \text{MA.dec}_R(st_{MA}, c_{MA})$ 
17 If  $(pk, md) = \perp$ : Return  $((ST_{MA}, st_{MA}), ST_{PR}, \emptyset)$ 
18  $(ST_{PR}[pk], k_{MA}, k_{AE}, (t, i))$ 
19    $\leftarrow \text{PR.rcv}(ST_{PR}[pk], c_{PR}, md)$ 
20  $m' \leftarrow \text{AE.dec}(k_{AE}, c_{AE}, (c_{MA}, c_{PR}))$ 
21 If  $m' = \perp$ : Return  $((ST_{MA}, st_{MA}), ST_{PR}, \emptyset)$ 
22 If  $k_{MA} \neq \perp$ :  $ST_{MA}[pk] \leftarrow \text{MA.up}_S(ST_{MA}[pk], k_{MA})$ 
23  $ST \leftarrow ((ST_{MA}, st_{MA}), ST_{PR})$ 
24 If  $m' = \epsilon$ : Return  $(ST, \emptyset)$ 
25 Else if  $m' = M:m$ : Return  $(ST, \{(pk, m, (t, i))\})$ 
26 Else if  $m' = R:c = R:(C:c_{MA-PR-AE}, i)_{i \in [I]}$ :
27    $(C:c_{MA-PR-AE}, i)_{i \in [I]} \leftarrow c$ 
28   For all  $i \in [I]$ :
29      $(ST, PKM') \leftarrow \text{rec-dec}(pk_{me}, ST, C:c_{MA-PR-AE}, i)$ 
30    $PKM \stackrel{\cup}{\leftarrow} PKM'$ 
30 Return  $(ST, PKM)$ 

```

Figure 6: Ciphertext processing algorithm with recursive decryption sub-procedure.

random oracle H and AEAD AE), and AEAD AE , our ASMesh 1 MM from Figures 4, 5, and 6 is secure according to Definition 2.1.

Proof Outline The reduction to NIKE for proving that the initialization produces fresh keys is straight forward. In contrast, proving confidentiality and anonymity of session ciphertexts is more difficult. The reason is that adversary and simulator observe a non-chronological view of the (chronological) protocol execution. More concretely, recall that in the mesh network setting outlined in Section 2, the adversary can adaptively “watch” a *subset* of different parties in the network. Thus, ciphertexts from the same session might be seen by the adversary in a very different order than they were actually sent. Despite this, the simulator still has to create a sound, adaptive simulation of these *non-chronological* ciphertexts. To handle this, in our security proof, we replace components of the chronological protocol execution with indistinguishable random

counterparts one after another. Once this is done, we show that this random *chronological* protocol execution can be simulated fully *non-chronologically*.

5.2 Receiver Anonymity

For achieving *receiver anonymity* instead of *contact cooperation*, we change our main ASMesh protocol in the blue and green marked lines as follows: (1) the receiver public key is removed from ciphertexts (Figure 5, lines 02,06, Figure 6 lines 00,02-04); and (2) all re-encryption, resp., recursive decryption procedures are omitted (Figure 6 lines 05-12,25-29). It trivially follows from the proof for the base protocol, provided in Appendix C.3, that removing the public key from MM ciphertexts yields receiver anonymity. Simultaneously, omitting re-encryption by contacts abandons contact cooperation. We thus have the following theorem (with a formal variant in Appendix C.3):

THEOREM 5.2 (INFORMAL). *Taking secure instances of NIKE NK, PRG G (modeled as a random oracle), PR PR, MA MA (based on random oracle H and AEAD AE), and AEAD AE, our ASMesh 2 MM from Figures 4, 5, and 6 is secure according to Definition 2.2.*

Beyond trading these security guarantees for each other, the secondary effect is slightly changed performance: Our second ASMesh protocol needs to check for *all* received ciphertexts whether the MA decryption procedure detects a matching tag (Figure 6 lines 00-02,16-17) in order to identify those ciphertexts that were actually meant for the executing receiver. This increases the computation overhead while, due to omitting re-encryption and recursive decryption, computation overhead is decreased.

6 PERFORMANCE EVALUATION

To analyze its performance, we implement our ASMesh constructions with hash function SHA-512, AEAD AES-256-GCM, and elliptic curve 25519; see our code [BRT23c]. The ciphertext overhead of this instantiation is 145 bytes, and the average encryption-decryption time (of each layer of (re-)encryption) is about 11ms.⁶ Compared to plain Double Ratchet, the ciphertext overhead is 76 bytes larger, where 32 bytes are contributed by ACKA and 44 bytes by MA. We note that re-encryptions have the same overhead as normal encryptions. The encryption-decryption time is mostly spent on elliptic curve computations, and thus is very insensitive to the payload message size. This indicates that the MA layer adds little performance overhead, and we have competing efficiency compared to existing MM protocols based on Double Ratchet such as Bridgefy.

Network Simulation We develop a model for simulating mesh networks [BRT23c] in order to test the performance of our constructions in various settings. In this model, there are n users moving over a 2-dimensional, $A \times A$ grid. The simulation is discrete-time, and runs for T steps. In each step, each user moves independently by a random step in $[-r, r] \times [-r, r]$. Two users establish a connection for exchanging ciphertexts when they get d -close to each other, i.e., their position difference is in $[-d, d] \times [-d, d]$. At the beginning of the simulation, we generate a social network over the n users according to the Watts-Strogatz model [WS98], which involves a degree parameter δ and a randomness parameter β . Each pair of

⁶The running time is tested on Apple M1 Pro chip, single thread.

Profile	n	r	pt	nd	#Hops	Latency	#Re-enc
Standard	600	2	10	25	5.355	5.983	0.503
					5.396	6.043	n/a
Dense	3000	10	5	5	4.550	4.550	0.372
					4.545	4.545	n/a
Sparse	100	10	10	20	3.356	5.699	0.301
					3.367	5.490	n/a

Table 2: Simulation results for standard setting, dense and sparse network, which have different user numbers and traveling speeds over the field. The performance is measured by average number of hops, average latency (in discrete steps), and average number of re-encryptions over received messages, where the upper row in a group is for ASMesh 1 and the lower row is for ASMesh 2.

users that are contacts in the social network share a mesh messaging session. In each simulation step, new messages are generated at each user, and each message is sent to a random receiver that is contact of the sender; the number of new messages is drawn from a Poisson distribution with parameter λ (and thus has expectation λ). In our simulations, we fix $A = 25$, $T = 100$, $d = 1$, $\delta = n/10$, $\beta = 1/2$, and $\lambda = 0.01$.

Parameters pt and nd of our MM protocol control the ciphertext forwarding policy in the network. Moreover, as a naive routing control, we implement a dictionary at each user that records the connection from which the user receives each ciphertext,⁷ and avoid forwarding the ciphertexts back to their incoming connections.⁸ For re-encryptions, each user also records the ciphertexts they (re-)encrypt, and those ciphertexts do not participate in future batching and re-encryptions by that user, avoiding repeated re-encryptions to happen locally at a single user. For setting parameters pt and nd, we reference the configurations of Bridgefy⁹. Since we run the simulation on a single computer instead of distributing among real nodes in a network, our available memory is limited, and thus we adopt a 1/10x down-scaling of the “hops limits” and “maximum propagations” in Bridgefy (our pt and nd, resp). Other parameters like n , r , etc. are chosen according to this scale, too.

Results In Table 2 we report the simulation results based on three settings corresponding to profiles in Bridgefy: standard setting, high density network, and sparse network. The density of the network is controlled by the different numbers of users n over the fixed $A \times A$ field. For sparse network, we assign a higher traveling speed r so that the sparsely distributed users can meet more frequently and messages are able to propagate throughout the field faster; similarly, we assign a higher traveling speed in high density network to compensate for the smaller upper bound pt on hop numbers and let messages propagate far enough; in all simulations, we have message delivery rates in the range 90%-95%. We measure the average number of hops, average latency (in discrete steps), and average number of re-encryptions as the performance of each setting. In Figure 7, we further study how the delivery rate changes

⁷For the sake of memory, the dictionary only retains records for recent ciphertexts within pt time steps.

⁸We note that in practice the connections can use short-lived identifiers to avoid user tracking; since our experiments are only simulating a short time period, we use long-term identifiers for simplicity.

⁹See propagation profiles at <https://bridgefy-1.gitbook.io/sdk/ios/usage>.

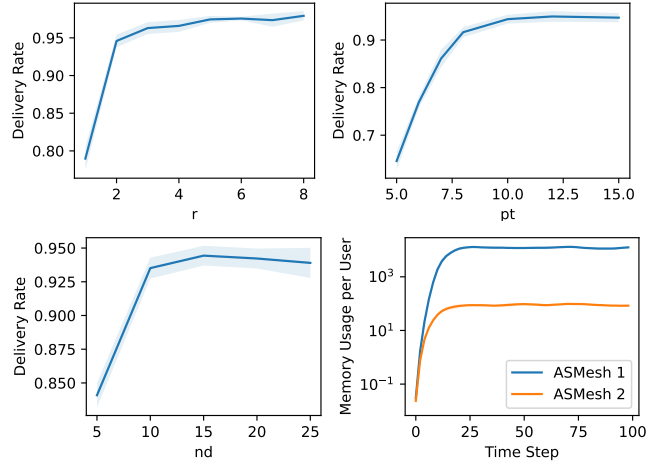


Figure 7: Top and bottom left: Delivery rate as a function of r , pt, and nd in the standard setting for ASMesh 2. (ASMesh 1 and 2 have similar delivery rates; thus, we only experiment with ASMesh 2 which involves no re-encryption and runs faster.) The shade depicts 1 standard deviation over 8 independent experiments.

Bottom right: Change of average memory usage per user along time in the standard setting. Memory usage is measured by the number of stored ciphertext and the dictionaries for recording previous hops and re-encryptions. The shade depicts 1 standard deviation over 4 independent experiments.

as a function of the parameters r , pt, and nd, in the case of the standard setting with ASMesh 2. We note that all three curves are monotonous in the intuitive direction. In the bottom right of Figure 7, we trace how the average memory usage per user changes along time, measured by number of entries in the dictionaries used. We see that ASMesh 2 is space-efficient and adds only about 10^2 storage to every user, and ASMesh 1 adds about 10^4 storage due to re-encryptions that blow up the number of messages in the network. Memory usage stabilizes after a warm-up phase.

For ASMesh 1, we observe that in high density networks, the latency is small and (almost) matches the hop number as a result of high user density—almost every message can be forwarded (in a progressive direction) at every time step. For sparse networks, all measured values are lower (partially due to the higher traveling speed), while however the relative number of re-encryptions (i.e., the ratio of number of re-encryptions to number of hops) is much higher, meaning there is a higher proportion of hops that perform re-encryptions. For all three settings, the average number of re-encryptions is at most about 0.5, which indicates the (amortized) overhead brought by re-encryption in our construction is very mild. Augmenting these results with the concrete instantiation from the beginning of this section yields a 70-byte overhead in size and a 5.5ms overhead in computation time per message. Moreover, ASMesh 2 has a very similar performance to ASMesh 1 in terms of average number of hops and latency.

7 CONCLUSION

In summary, we provide more complete security models for Mesh Messaging and accompanying secure constructions. All our building blocks use practical standard components, which supports efficiency and deployment in practice. Our overall constructions maintain robust correctness guarantees and provide strong confidentiality and anonymity guarantees. We note that most of our results are also applicable to traditional, centralized messaging. For example, recent studies on interoperable messaging [LGGR23, RS23] indicate that practical, anonymous channels are important tools to preserve and strengthen user privacy.

Limitations and Future Work Our definitions capture a cryptographic notion of anonymity that our constructions fulfill. Nevertheless, our constructions may reveal communication patterns that affect a broader sense of anonymity: For example, the lifetime counter in our ciphertexts reveals the approximate (physical) distance to the original sender in the network; or a node that takes a ciphertext as input and later outputs a different ciphertext of slightly increased size directed to the same receiver may have re-encrypted this input ciphertext, which reveals that this node is a contact of that receiver; etc.

Based on this, taking related work into account, and considering relevant use cases for mesh messaging, our work can be extended by (1) implementing non-cryptographic techniques to strengthen anonymity guarantees; (2) studying mesh messaging with group chats; (3) developing (strong) privacy-preserving authenticity mechanisms; (4) designing global state-update algorithms that avoid linear effort to recover from a state exposure; etc. Another interesting research direction is to augment Mesh Messaging protocols with smart, anonymous routing protocols.

REFERENCES

[ABJM21] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking Bridgefy. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 375–398. Springer, Heidelberg, May 2021.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, November 2020.

[AEP22] Martin R. Albrecht, Raphael Eikenberg, and Kenneth G. Paterson. Breaking bridgefy, again: Adopting libsignal is not enough. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 269–286. USENIX Association, August 2022.

[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, Heidelberg, August 2022.

[BDG⁺22] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 213–243. Springer, Heidelberg, November 2022.

[BDHK06] Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. Conditional reactive simulatability. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS 2006*, volume 4189 of *LNCS*, pages 424–443. Springer, Heidelberg, September 2006.

[BDR20] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, November 2020.

[BFG⁺22] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 784–813. Springer, Heidelberg, August 2022.

[Bri] INC. Bridgefy. Website of bridgefy. <https://bridgefy.me/>.

[BRT23a] Alexander Bienstock, Paul Rösler, and Yi Tang. Asmesh: Anonymous and secure messaging in mesh networks using stronger, anonymous double ratchet. In *CCS '23: 2023 ACM SIGSAC Conference on Computer and Communications Security 2023*. ACM, 2023.

[BRT23b] Alexander Bienstock, Paul Rösler, and Yi Tang. Asmesh: Anonymous and secure messaging in mesh networks using stronger, anonymous double ratchet. Cryptology ePrint Archive, Paper 2023, 2023. <https://eprint.iacr.org/2023>.

[BRT23c] Alexander Bienstock, Paul Rösler, and Yi Tang. Proof of concept implementation of our ASMesh protocol and mesh network simulation. <https://github.com/meshmessaging/ASMesh>, 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, Heidelberg, December 2020.

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

[CCD⁺20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020.

[CJSV22] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2022.

[CZ22] Cas Cremers and Mang Zhao. Provably post-quantum secure messaging with strong compromise resilience and immediate decryption. Cryptology ePrint Archive, Report 2022/1481, 2022. <https://eprint.iacr.org/2022/1481>.

[DG19] Nir Drucker and Shay Gueron. Continuous key agreement with reduced bandwidth. In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning - Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27-28, 2019, Proceedings*, volume 11527 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2019.

[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, Heidelberg, December 2022.

[DSKB21] Alaa Daffalla, Lucy Simko, Tadayoshi Kohno, and Alexandru G. Bardas. Defensive technology use by political activists during the sudanese revolution. In *2021 IEEE Symposium on Security and Privacy*, pages 372–390. IEEE Computer Society Press, May 2021.

[EHM17] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. Can johnny build a protocol? co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. In *European Workshop on Usable Security*, 2017.

[EM22] Ksenia Ermoshina and Francesca Musiani. *Concealing for Freedom: The Making of Encryption, Secure Messaging and Digital Liberties*. 03 2022.

[GHJL17] Felix Günther, Britta Hale, Tibor Jäger, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 519–548. Springer, Heidelberg, April / May 2017.

[GKH16] Seda Gürses, Arun Kundnani, and Joris Van Hoboken. Crypto and empire: the contradictions of counter-surveillance advocacy. *Media, Culture & Society*, 38(4):576–590, 2016.

[GM15] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE Computer Society Press, May 2015.

[jlu18] jlund. Technology preview: Sealed sender for signal. <https://signal.org/blog/sealed-sender/>, 10 2018.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. A unified and composable take on ratcheting. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 180–210. Springer, Heidelberg, December 2019.

[Koe19] John Koetsier. Hong kong protestors using mesh messaging app china can't block: Usage up 3685%. <https://www.forbes.com/sites/johnkoetsier/2019/09/02/hong-kong-protestors-using-mesh-messaging-app-china-cant-block-usage-up-3685/>, 09 2019.

[LGGR23] Julia Len, Esha Ghosh, Paul Grubbs, and Paul Rösler. Interoperability in end-to-end encrypted messaging. Cryptology ePrint Archive, Paper 2023/386, 2023. <https://eprint.iacr.org/2023/386>.

- [PHE⁺17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 1199–1216. USENIX Association, August 2017.
- [PJW⁺22] Amogh Pradeep, Hira Javaid, Ryan Williams, Antoine Rault, David R. Choffnes, Stevens Le Blond, and Bryan Ford. Moby: A blackout-resistant anonymity network for mobile devices. *PoPETS*, 2022(3):247–267, July 2022.
- [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.
- [PR18] 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, Heidelberg, August 2018.
- [PSEB22] Neil Perry, Bruce Spang, Saba Eskandarian, and Dan Boneh. Strong anonymity for mesh messaging, 2022.
- [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 European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 415–429. IEEE, 2018.
- [RS23] Paul Rösler and Jörg Schwenk. Interoperability between messaging services secure – implementation of encryption. Study for the Federal Network Agency, 2023.
- [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, Heidelberg, April 2023.
- [Sta21] Reuters Staff. Encrypted messaging app signal stops working in china. <https://www.reuters.com/article/us-china-tech-signal/encrypted-messaging-app-signal-stops-working-in-china-idUSKBN2B8094>, 03 2021.
- [vAP22] Theo von Arx and Kenneth G. Paterson. On the cryptographic fragility of the telegram ecosystem. Cryptology ePrint Archive, Report 2022/595, 2022. <https://eprint.iacr.org/2022/595>.
- [WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.
- [Yee22] Vivian Yee. Despite iran’s efforts to block internet, technology has helped fuel outrage. <https://www.nytimes.com/2022/09/29/world/middleeast/iran-internet-censorship.html>, 09 2022.

A UNIVERSAL COMPOSABILITY

We model security and correctness of Mesh Messaging in the Universal Composability (UC) framework [Can01].

Physical Communication In UC, parties typically communicate to one another via a network which is controlled by the adversary so that, in particular, the adversary sees all (possibly encrypted) messages sent between parties. However, in mesh messaging, parties do not communicate over centralized, standard networks, and instead can only communicate when in close proximity to one another (using, e.g., bluetooth). Thus an adversary will often not be close enough to eavesdrop and will therefore *only sometimes* learn information about messages sent between parties. So, the standard UC modelling of communication does not fit. We instead choose to model communication in mesh networks via an additional ideal functionality $\mathcal{F}_{\text{mesh}}$ which directly transmits messages from sender to receiver and only leaks them to the adversary if the adversary explicitly commands it. We expand on this in the next paragraph.

Corruption Model We utilize the recent corruption models of (transient, passive) state exposures formalized in [ACJM20, AJM22, BFG⁺22, CJSV22]. In a nutshell, this corruption model allows the adversary to repeatedly expose the states of parties by sending them a corruption message *Expose*. Upon receipt of such a message, the corresponding party first notifies the environment that it has been corrupted and then sends its current state to the adversary (once per message).

In the ideal world, corruptions are routed through ideal functionalities. That is, the simulator announces corruptions to the ideal functionality, who then notifies the corresponding (dummy) party.

We also allow for another type of corruption which models adversarial eavesdropping on parties’ close-proximity communication. Namely, the adversary can repeatedly send them *Watch* and *Unwatch* messages, and the parties respond by simply notifying the environment. Note that although this on its own does not allow for any adversarial behavior (and in fact we assume that the adversary cannot block messages from being delivered), it enables the adversary to eavesdrop on parties. Indeed, the following is the interface of our mesh network communication functionality $\mathcal{F}_{\text{mesh}}$ interacting with simulator \mathcal{S} :

- **Initialization:** Set $WXP = \emptyset$.
- **On input** (*Send*, m , P') **from** \mathcal{P} : If $\{P, P'\} \cap WXP \neq \emptyset$, send (P, P', m) to \mathcal{S} and wait for it to return OK. Then send m to P and P' . Else, send right away.¹⁰
- **On input** (*Watch*, P) **from** \mathcal{S} : $WXP \stackrel{\cup}{\leftarrow} \{P\}$.
- **On input** (*Unwatch*, P) **from** \mathcal{S} : $WXP \setminus \{P\}$.

Note that as above, the ideal functionality will forward all *Watch* and *Unwatch* messages to the (dummy) parties.

Restricted Environments In order to avoid the so-called commitment problem caused by fully adaptive adversaries (e.g., when an adversary corrupts a receiver while a secure ciphertext is in-transit), we use a technique of [ACJM20, AJM22] (based on prior works [BDHK06, JMM19]): In particular, we consider a weakened variant of UC security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment \mathcal{Z} is admissible or not is defined as part of the ideal functionality \mathcal{F} : The functionality can enforce certain boolean conditions by using keyword *Require*, and \mathcal{Z} is then called admissible (for \mathcal{F}), if it has negligible probability of violating any such condition when interacting with \mathcal{F} .

B FORMAL MESH MESSAGING DEFINITION

We provide a formal pseudo-code definition of our ideal functionality in Figures 8, 9, 10, and 11. For comprehensibility, we emphasize the most relevant lines of code in these Figures. These emphasized code lines trace the progression in a session with respect to state updates, which directly influences the encoded forward-security and post-compromise security requirements. As elaborated in upcoming Appendix B.1, code marked in blue is relevant for security with *contact cooperation* but irrelevant for security with *receiver anonymity*. We start here with a description of the former first, before describing the changes for the latter in Appendix B.1; in short, the latter removes receiver identifiers from ciphertexts and omits re-encryptions.

Our ideal functionality models that a party P can generate fresh key material, initialize a new session with some party P' (both in Figure 8), encrypt a message m to some party P' , update their local secrets for a particular session with some party P' (both in Figure 9), and meet with another party P' (in Figure 10). When two parties

¹⁰We send m also to P as a modelling artifact, so that it knows \mathcal{S} indeed allowed the message to be delivered as intended.

meet, they are considered to broadcast all their cached ciphertexts and process the ciphertexts broadcast by their counterpart.

An adversary \mathcal{A} (resp. simulator \mathcal{S}) can expose the state of a party P (in Figure 11). This reveals all secrets stored by P locally at the moment of exposure. Furthermore, \mathcal{A} (resp. \mathcal{S}) can watch and unwatch a party P (also in Figure 11). As long as P is watched, \mathcal{A} (resp. \mathcal{S}) sees all ciphertexts exchanged during their meetings. If neither of two meeting parties is watched, their ciphertext exchange remains hidden from the adversary (resp. simulator).

To avoid that our definition induces impractical requirements for constructions (e.g., non-committing encryption), we carefully trace the internal state of every party P during each executed operation. Based on this tracing, we restrict state exposures that trivially break challenge ciphertexts. More precisely, in case a challenge ciphertext c was seen by the adversary, we forbid the exposure of the respective receiver state until c was received.

State Progression, FS, and PCS With our ideal functionality definition, we require immediate forward secrecy (FS) as well as strong post-compromise security (PCS) with respect to confidentiality and anonymity. This means that all received ciphertexts that were sent in Alice’s sessions remain fully secure even if her state is exposed anytime later (FS); beyond this, future ciphertexts become fully secure again after an exposure of Alice’s state (PCS). To specify how far confidentiality and anonymity of ciphertexts are affected by state exposures more precisely, we consider the state progression within a session.

The communication in a session between Alice and Bob naturally contains a continuous “ping-pong” pattern that starts at session initialization, which we consider the first “ping”. As soon as Alice receives the first ciphertext from Bob, this is the first “pong”. When Bob receives the first ciphertext that was sent by Alice after this “pong”—no matter which ciphertext after this “pong” it is—, we consider this as another “ping”, and so on. The period from a ping until the next pong (or from a pong until the next ping) is called a **ratchet**.

Independent of ratchets, each ciphertext sent by Alice (resp. Bob) can be considered a response to the newest ciphertext received from Bob (resp. Alice) before. Thus, each newest ciphertext received from Bob (resp. Alice) determines an **epoch** for Alice (resp. Bob) that lasts until a newer ciphertext is received.

Finally, we number all ciphertext sent within a ratchet consecutively with an **index**. This means, each ciphertext in a session has a unique **ratchet-epoch-index triple** (t, l, i) .

Setup The three components provided in Figure 8 basically initialize variables. The only non-trivial part is that, on input (Init, P') from P , the two parties are defined to communicate their newest key material. This is captured by exchanging the perspective on their respective partner’s newest *epoch* via array PC (lines 10-11). After this, each party increments their epoch counter. Array SP determines who the sending and receiving party in this session is, respectively.

Sending Ideal message encryption and session update in Figure 9 are identical except that message encryption internally traces the encrypted message. For both operations, we first trace if the operation starts a new ratchet in this session (lines 01-02,14-15). This is based on whether it is the encrypting, resp., updating party’s turn

Ideal Functionality Initialization:

```

00  $G[\cdot] \leftarrow 0; IN[\cdot] \leftarrow 0; CT[\cdot] \leftarrow \perp$ 
01  $n \leftarrow 0; AM[\cdot] \leftarrow \emptyset; SM[\cdot] \leftarrow \emptyset; CH \leftarrow \emptyset$ 
02  $PC[\cdot][\cdot] \leftarrow 0; FS[\cdot][\cdot][\cdot] \leftarrow [\infty]$ 
03  $WXP \leftarrow \emptyset; PXP[\cdot] \leftarrow \emptyset; AXP[\cdot][\cdot], TXP[\cdot][\cdot] \leftarrow \emptyset$ 
04  $I[\cdot][\cdot], T[\cdot][\cdot], SP[\cdot][\cdot], PC2T[\cdot][\cdot][\cdot] \leftarrow 0$ 

On input Gen from party P:
05  $G[P] \leftarrow 1$ 
06 Send (Gen, P) to  $\mathcal{S}$ 

On Input (Init,  $P'$ ) from party P:
07 Require  $G[P] = G[P'] = 1$ 
08 Require  $IN[\{P, P'\}] = 0$ 
09  $IN[\{P, P'\}] \leftarrow 1$ 
10  $PC[P][P'] \leftarrow PC[P'][P']$ 
11  $PC[P'][P] \leftarrow PC[P][P]$ 
12  $PC[P'][P'] \leftarrow PC[P'][P'] + 1$ 
13  $PC[P][P] \leftarrow PC[P][P] + 1$ 
14  $SP[P][P'] \leftarrow 1; SP[P'][P] \leftarrow 0$ 

```

Figure 8: Setup and interfaces for key generation and session initialization of our ideal functionality $\mathcal{F}_{MM}^{pt,nd,eoh}$. The ideal functionality is parameterized by pt , the path length for ciphertexts in the mesh network, nd , the number of meetings in which a node broadcasts a ciphertext, and eoh , the length overhead of the corresponding encryption scheme. $uniq : \{0, 1\}^\lambda \rightarrow \{0, 1\}^\lambda$ is a random permutation.

to do so. More concretely, if, from the perspective of party P , P' most recently started a new ratchet, then P will start a new ratchet now. Array T keeps track of the current ratchet in each session from each partner’s perspective.

After this, we assemble all variables that describe the encryption status of the produced ciphertext (lines 03-05,16-18). Most importantly, variable enc stores epoch, ratchet, and index in the session from P ’s perspective at the time of encryption. (See the explanations for *epoch*, *ratchet*, and *index* in above paragraph *State Progression, FS, and PCS*.) In addition to this, we attach technical details about the ciphertext in variable \widehat{cvar} that is described in following paragraph *Leaked Technical Variables*. Variable upc stores party P ’s current epoch, which models that P updates their own state and attaches a corresponding key update to the ciphertext when encrypting the message, resp., performing a session update.

We add the new ciphertext to the set of *active ciphertexts* AM that will be broadcast by party P during the next meeting, and to the set SM of *ciphertexts seen* by party P (lines 07-08,20-21). Finally, we increment the index counter for the session between P and P' and we increment global epoch counter of P .

Meeting Interface Meet in Figure 10 models that two users exchange their cached ciphertexts. More concretely, both users first *broadcast* the ciphertexts from their own cache and then *process* the ciphertexts that were broadcast by the respective counterpart. If either of the two participants is currently watched by the adversary (see following paragraph *Corruption*), this interface forwards information about the exchanged ciphertexts as well as the identifiers

```

On input (Enc, m, P') from P:
00 Require  $IN[\{P, P'\}] = 1$ 
01 If  $T[P][P'] \bmod 2 \neq SP[P][P']$ :
02    $T[P][P'] \leftarrow T[P][P'] + 1$ ;  $I[P][P'] \leftarrow 0$ 
03  $fw \leftarrow (pt, nd)$ 
04  $enc \leftarrow (PC[P][P'], T[P][P'], I[P][P'], \widehat{cvar})$ 
05  $upc \leftarrow PC[P][P]$ 
06  $CT[n] \leftarrow (P, P', enc, upc, m, 0)$ 
07  $AM[P] \stackrel{\cup}{\leftarrow} \{(fw, n)\}$ 
08  $SM[P] \stackrel{\cup}{\leftarrow} \{n\}$ 
09  $PC2T[P][P'] [PC[P][P]] \leftarrow T[P][P']$ 
10  $n \leftarrow n + 1$ 
11  $I[P][P'] \leftarrow I[P][P'] + 1$ 
12  $PC[P][P] \leftarrow PC[P][P] + 1$ 

On Input (Up, P') from P:
13 Require  $IN[\{P, P'\}] = 1$ 
14 If  $T[P][P'] \bmod 2 \neq SP[P][P']$ :
15    $T[P][P'] \leftarrow T[P][P'] + 1$ ;  $I[P][P'] \leftarrow 0$ 
16  $fw \leftarrow (pt, nd)$ 
17  $enc \leftarrow (PC[P][P'], T[P][P'], I[P][P'], \widehat{cvar})$ 
18  $upc \leftarrow PC[P][P]$ 
19  $CT[n] \leftarrow (P, P', enc, upc, \epsilon, 0)$ 
20  $AM[P] \stackrel{\cup}{\leftarrow} \{(fw, n)\}$ 
21  $SM[P] \stackrel{\cup}{\leftarrow} \{n\}$ 
22  $PC2T[P][P'] [PC[P][P]] \leftarrow T[P][P']$ 
23  $n \leftarrow n + 1$ 
24  $I[P][P'] \leftarrow I[P][P'] + 1$ 
25  $PC[P][P] \leftarrow PC[P][P] + 1$ 

```

Figure 9: Interfaces for message encryption and session update of our ideal functionality $\mathcal{F}_{MM}^{pt, nd, eoh}$. Parameters (pt, nd, eoh) are explained in Figure 8. Variable \widehat{cvar} is described in paragraph *Leaked Technical Variables*.

of the participants to the simulator (lines 34-35). Such forwarded ciphertexts are then treated as challenges (line 36).

The interface begins with assembling the set of broadcast ciphertexts (lines 02,13-15). We describe helper procedure SimCtxt below. Additionally, it internally assembles the set of messages that are decrypted by both participants due to receiving ciphertexts during the meeting (lines 05-06). We also describe helper procedure DecCtxt below. For all received ciphertexts that are *not* decrypted, the interface updates the receivers' ciphertext caches (lines 08-12). In particular, the sets of active and seen ciphertexts are updated.

In the second step, the interface models re-encryption of those ciphertexts that are directed to session partners of the respective processing party (lines 16-33). In principle, this re-encryption resembles ordinary encryption from Figure 9. The core difference is that all ciphertexts to the same receiver are aggregated before re-encryption (line 17,28,30).

Helper procedure SimCtxt assembles the information leaked per exchanged ciphertext. More precisely, it recursively unwraps the re-encryption of a ciphertext (lines 42-46, particularly line 43) and then leaks its overall length. This length is determined by the lengths

of the contained payloads as well as the overhead of the used encryption scheme, which is defined via parameter eoh. In addition to this, starting from the outermost (re-)encryption layer, it unwraps all non-confidential ciphertext layers until the first confidential one is reached. From those layers, it assembles the technical data about each unwrapped encryption as well as the potentially contained payload. Depending on whether the identified *confidential* ciphertexts are also *anonymous*, the leaked technical data differs. Anonymous layers (line 51) only reveal their length, their receiver, and their forwarding status (i.e., remaining path length and local re-broadcasts). In addition to that, non-anonymous, confidential layers (line 53) reveal encrypting (aka. sender) party and technical encryption information \widehat{cvar} . Variable \widehat{cvar} is described in paragraph *Leaked Technical Variables*. Non-confidential layers (line 57) reveal either the set of contained ciphertexts or their payload.

Helper procedure DecCtxt derives the contained payload from a ciphertext and adds the sender identifier as well as the payload identifier to the output (lines 62-64). For ciphertexts that contain a set of (re-encrypted) ciphertexts, DecCtxt recursively calls itself (lines 65-67). Finally, based on processing the decryption, it updates the receiver's view of the ratchet, epoch, and index counters as well as the set of seen ciphertexts (lines 68-71). For this, set FS traces all received ciphertexts.

Corruption The simulator can query exposure of a party's local state and it can start and stop watching a party, which we formalize in Figure 11.

At state exposure, the interface first performs a *dummy exposure* (lines 03-18). Based on the dummy outcome, it decides whether the exposure is permitted or not (line 19). We permit an exposure if none of the existing challenge ciphertexts is *solved due to the exposure* based on our FS and PCS requirements. By being solved, we mean that the exposed receiver state can be used to break confidentiality and/or anonymity of the challenge ciphertext.

The dummy exposure processes one session of the exposed party after another. It first considers all ratchets in which the exposed party is a receiver (lines 04-06). The exposed party uses/used the key material for this ratchet to decrypt responses *from* the session partner. The ciphertexts not yet received in these ratchets are thus marked non-confidential (line 05) and non-anonymous (line 06). If the current ratchet in the session was started by the exposed party (line 07), then all following indexes of the current ratchet are marked *possibly* non-confidential (line 08). They are considered *ultimately* non-confidential if the session partner's epoch with which they are associated is exposed, too (lines 17-18). Additionally, all indexes of the next ratchet are marked *possibly* non-confidential (line 09). These indexes become *ultimately* non-confidential if the exposed party's epoch with which they are eventually associated is exposed, too (lines 17-18). Finally, all following indexes of the current ratchet, and all indexes of the next two ratchets are marked non-anonymous (lines 10-12). If, in contrast, the current ratchet was started by the session partner, all indexes of the next ratchet are marked non-anonymous (lines 15). Beyond that, all epochs that were started in ratchets for which their are still (potentially, if the ratchet after was the session partner's latest sending ratchet) undelivered ciphertexts are marked ineffective (lines 17-18).

<pre> On Input (Meet, P₁) from P₀: 00 Require G[P₀] = G[P₁] = 1 01 M₀, M₁, C₀, C₁, CH' ← ∅ 02 For all ((pt, nd), i) ∈ AM[P_j] : j ∈ {0, 1}: 03 (P_s, P_r, enc, upc, m, ct) ← CT[i] 04 AM[P_j] ← AM[P_j] \ {(pt, nd), i} 05 If pt ≥ 0 ∧ P_r = P_{1-j}: 06 M_{1-j} ←[∪] DecCtxt(P_{1-j}, i) 07 Else: 08 If nd > 0: 09 AM[P_j] ←[∪] {(pt, nd - 1), i} 10 If i ∉ SM[P_{1-j}] ∧ pt > 0: 11 AM[P_{1-j}] ←[∪] {(pt - 1, nd), i} 12 SM[P_{1-j}] ←[∪] {i} 13 (·, C'_i, CH'_i) ← SimCtxt(i, pt, ⊥) 14 C_j ←[∪] {C'_i} 15 CH' ←[∪] CH'_i 16 For all P_r : IN[{\P_j, P_r}] = 1, j ∈ {0, 1}: 17 FWI ← {(fw, i) ∈ AM[P_j] CT[i] = (P', P_r, 18 ·, ·, m, ·) ∧ m ≠ ε} 19 AM[P_j] ← AM[P_j] \ FWI 20 If FWI ≠ ∅: 21 If T[P_j][P_r] mod 2 ≠ SP[P_j][P_r]: 22 T[P_j][P_r] ← T[P_j][P_r] + 1 23 I[P_j][P_r] ← 0 24 enc ← (PC[P_j][P_r], T[P_j][P_r], I[P_j][P_r], \widehat{cvar}) 25 upc ← PC[P_j][P_r] 26 PC2T[P_j][P_r][PC[P_j][P_r]] ← T[P_j][P_r] 27 PC[P_j][P_r] ← PC[P_j][P_r] + 1 28 I[P_j][P_r] ← I[P_j][P_r] + 1 29 ct ← {i (fw, i) ∈ FWI} 30 CT[n] ← (P_j, P_r, enc, upc, ⊥, ct) 31 pt ← avg{pt_i ((pt_i, nd_i), i) ∈ FWI} 32 AM[P_j] ←[∪] {(pt, nd), n} 33 SM[P_j] ←[∪] {n} 34 n ← n + 1 35 If {P₀, P₁} ∩ WXP ≠ ∅: 36 Send (P₀, P₁, C₀, C₁) to S and wait for it 37 to return OK 38 CH ←[∪] CH' 39 Send M₀ to P₀ and M₁ to P₁ </pre>	<pre> Proc SimCtxt(i, pt, nd): 38 SC ← ∅ 39 ℓ ← 0 40 CH' ← ∅ 41 (P_s, P_r, enc, upc, m, ct) ← CT[i] 42 For all i' ∈ ct: 43 (ℓ_{i'}, C'_{i'}, CH'_{i'}) ← SimCtxt(i', ⊥, ⊥) 44 ℓ ← ℓ + ℓ_{i'} 45 SC ←[∪] {C'_{i'}} 46 CH' ←[∪] CH'_{i'} 47 (pc, t, l, \widehat{cvar}) ← enc 48 If pc ∉ PXP[P_r] ∨ l ∉ TXP[{\P_s, P_r}] [t]: 49 CH' ← {i} 50 If l ∉ AXP[{\P_s, P_r}] [t]: 51 C ← (uniq(i), ℓ + m + eoh, ∅, ⊥, P_r, pt, nd, ⊥, ⊥) 52 Else: 53 C ← (uniq(i), ℓ + m + eoh, ∅, P_s, P_r, pt, nd, \widehat{cvar}, ⊥) 54 Else: 55 If ct = ∅: m' ← m 56 Else: m' ← ⊥ 57 C ← (uniq(i), ℓ + m + eoh, SC, P_s, P_r, pt, nd, \widehat{cvar}, m') 58 Return (ℓ + m + eoh, C, CH') Proc DecCtxt(P, i): 59 M ← ∅ 60 (P_s, P_r, enc, upc, m, ct) ← CT[i] 61 (·, t_s, l_s, ·) ← enc 62 If ct = ∅ ∧ m ≠ ε: 63 If i ∉ SM[P]: 64 M ←[∪] {(P_s, (t_s, l_s), m)} 65 Else if ct ≠ ∅: 66 For all i' ∈ ct: 67 M ←[∪] DecCtxt(P, i') 68 T[P][P_s] ← max{T[P][P_s], t_s} 69 FS[P_s][P][t_s] ← FS[P_s][P][t_s] \ {t_s} 70 PC[P][P_s] ← max(PC[P][P_s], upc) 71 SM[P] ←[∪] {i} 72 Return M </pre>
---	--

Figure 10: Meeting interface of our ideal functionality $\mathcal{F}_{MM}^{\text{pt,nd,eoh}}$ with helper procedures. Parameters (pt, nd, eoh) and permutation uniq are explained in Figure 8. Variable \widehat{cvar} is described in paragraph *Leaked Technical Variables*.

Based on these variables, our definition permits exposure (line 19) if for every challenge:

- Either the corresponding epoch is still effective (i.e., the epoch was not exposed; $pc \notin PXP[P_r]$)
- Or the ciphertext is marked fully confidential (i.e., not even possibly non-confidential; $l \notin TXP[{\P_s, P_r}] [t]$)
- And the ciphertext is considered anonymous ($l \notin AXP[{\P_s, P_r}] [t]$).

If the dummy exposure is permitted, it is turned into a real exposure (line 20). Then, information about the set of cached ciphertexts is assembled and, based on this, the set of challenge ciphertexts is extended (lines 21-24). Finally, information about the set of session secrets is assembled (lines 25-28). We refer the reader to following paragraph *Leaked Technical Variables* for a description of variable \widehat{svar} that specifies technical information about the session status.

```

On input (Expose, P) from S:
00 Require  $G[P] = 1$ 
01  $PXP' \leftarrow PXP; TXP' \leftarrow TXP; AXP' \leftarrow TXP$ 
02  $C \leftarrow \emptyset$ 
03 For all  $P' : IN[\{P, P'\}] = 1$ :
04   For all  $t : t \leq T[P][P'] \wedge t \bmod 2 \neq SP[P][P']$ :
05      $TXP'[\{P, P'\}][t] \stackrel{u}{\leftarrow} FS[P'][P][t]$ 
06      $AXP'[\{P, P'\}][t] \stackrel{u}{\leftarrow} FS[P'][P][t]$ 
07   If  $T[P][P'] \bmod 2 = SP[P][P']$ :
08      $TXP'[\{P, P'\}][T[P][P']] \stackrel{u}{\leftarrow} [\infty] \setminus [I[P][P']]$ 
09      $TXP'[\{P, P'\}][T[P][P'] + 1] \leftarrow [\infty]$ 
10      $AXP'[\{P, P'\}][T[P][P']] \stackrel{u}{\leftarrow} [\infty] \setminus [I[P][P']]$ 
11      $AXP'[\{P, P'\}][T[P][P'] + 1] \leftarrow [\infty]$ 
12      $AXP'[\{P, P'\}][T[P][P'] + 2] \leftarrow [\infty]$ 
13      $t_{\text{last-rcv}} \leftarrow T[P][P'] - 1$ 
14   Else:
15      $AXP'[\{P, P'\}][T[P][P'] + 1] \leftarrow [\infty]$ 
16      $t_{\text{last-rcv}} \leftarrow T[P][P']$ 
17   For all  $pc : PC2T[P][P'][pc] + 1 = t_{\text{last-rcv}} \vee \exists i, enc :$ 
     $(CT[i] = (P', P, enc, \cdot, \cdot) \wedge i \notin SM[P] \wedge$ 
     $enc = (PC2T[P][P'][pc] + 1, \cdot, \cdot, \cdot))$ :
18      $PXP'[P] \stackrel{u}{\leftarrow} \{pc\}$ 
19   Require  $\forall i \in CH : (pc \notin PXP'[P_r]$ 
     $\vee l \notin TXP'[\{P_s, P_r\}][t] \wedge l \notin AXP'[\{P_s, P_r\}][t],$ 
     $(P_s, P_r, enc, \cdot, \cdot) = CT[i],$ 
     $(pc, t, l, \cdot) = enc$ 
20  $PXP \leftarrow PXP'; TXP \leftarrow TXP'; AXP \leftarrow AXP'$ 
21 For all  $((pt, nd), i) \in AM[P]$ :
22    $(\cdot, C'_i, CH'_i) \leftarrow \text{SimCtxt}(i, pt, nd)$ 
23    $C \stackrel{u}{\leftarrow} \{C'_i\}$ 
24    $CH \stackrel{u}{\leftarrow} CH'_i$ 
25  $SL \leftarrow \emptyset$ 
26 For all  $P' : IN[\{P, P'\}] = 1$ :
27    $afs \leftarrow (t^*, FS[P'][P][t^*])_{t^* \leq T[P][P']}$ 
28    $SL \stackrel{u}{\leftarrow} \{(P', T[P][P'], I[P][P'], PC[P][P'], afs, \widehat{svar})\}$ 
29 Send  $(P, C, PC[P][P], SL)$  to S

On input (Watch, P) from S:
30 Require  $G[P] = 1$ 
31  $WXP \stackrel{u}{\leftarrow} \{P\}$ ;

On input (Unwatch, P) from S:
32 Require  $G[P] = 1$ 
33  $WXP \leftarrow WXP \setminus \{P\}$ ;

```

Figure 11: Interfaces for corrupting and watching parties in our ideal functionality $\mathcal{F}_{MM}^{\text{pt, nd, eoh}}$. Parameters (pt, nd, eoh) are explained in Figure 8. Variable \widehat{svar} is described in paragraph *Leaked Technical Variables*.

The interfaces for starting and stopping to watch a party are self-explaining.

Leaked Technical Variables Our MM construction from Section 5 leaks further, non-critical, technical variables through ciphertexts and session states. Hence, for being able to prove security

of this construction, we leak corresponding variables also in our ideal functionality, too. It is important to note that our construction uses these variables for better efficiency, but leaking these variables does not harm security. Since we consider these technical variables construction-specific, we refrain from detailing them in the pseudo-code description of our ideal functionality fully formally.

For *non-anonymous* ciphertexts, we leak variable \widehat{cvar} that contains the following sub-variables. The variable values are fixed at the time of encryption, where party P conducts the encryption in the session with partner P' .

- \widehat{rtc} : P's ratchet counter in this session ($T[P][P']$)
- \widehat{ixc} : P's index counter in this session ($I[P][P']$)
- \widehat{epc} : P's own epoch counter ($PC[P][P]$)
- \widehat{pcp} : P's view of P' 's epoch counter ($PC[P][P']$)
- \widehat{rpc} : P's view of P' 's epoch counter ($PC[P][P']$) fixed at the moment when P started their most recent ratchet
- \widehat{npr} : Number of sent ciphertexts in P's previous ratchet

Each exposed session state of party P with session partner P' leaks a variable \widehat{svar} . This variable contains the following sub-variables.

- \widehat{rtc} : P's ratchet counter in this session ($T[P][P']$)
- \widehat{ixc} : P's index counter in this session ($I[P][P']$)
- \widehat{epc} : P's own epoch counter ($PC[P][P]$)
- \widehat{pcp} : P's view of P' 's epoch counter ($PC[P][P']$)
- \widehat{rpc} : P's view of P' 's epoch counter ($PC[P][P']$) fixed at the moment when P started their most recent ratchet
- \widehat{pppc} : P's view of P's epoch counter ($PC[P'][P]$) fixed at the moment when P' started their most recent ratchet (that P has seen)
- $\widehat{pc2ratch}$: A dictionary that maps PCS epoch pc to corresponding ratchet t in which it was initiated (using array $PC2T$)
- \widehat{npr} : Number of sent ciphertexts in P's previous ratchet

Confidentiality and Anonymity To give a summarized intuition for our ideal functionality, we require confidentiality and anonymity in the following sense: a *secure* ciphertext must not reveal anything besides the length of the contained message(s), the identity of the receiver, and the remaining lifetime (i.e., the number of remaining hops on the transmission path).

Due to state exposures, a ciphertext can become entirely insecure, or provide only confidentiality but not anonymity. Beyond the data revealed by a secure ciphertext, an *entirely insecure* ciphertext reveals the contained message itself, the sender identity, as well as technical details about the encryption (e.g., position within the session, number of past round-trips in the session, etc). Except for the contained message, the same data is revealed by a *confidential, non-anonymous* ciphertext.

Re-Encryption We observe that a special feature of mesh networks is that honest users can cooperate to increase the security of transmitted ciphertexts. Intuitively, each hop on the transmission path of a ciphertext who shares a session with the respective receiver can re-encrypt this ciphertext before forwarding it to the next hop. Thereby, an insecure ciphertext can be turned into a secure one without interaction with sender or receiver.

A consequence of this is that a transmitted ciphertext can consist of multiple encryption layers, some of which can be insecure, confidential but non-anonymous, or fully secure. The outermost secure (i.e., at least confidential) layer protects all underlying layers. Thus, our ideal functionality traces the layers of every ciphertext, and the level of security of each of these layers. For efficiency reasons, a re-encryption can aggregate all ciphertexts directed to the same receiver.

B.1 Receiver Anonymity

For defining *receiver anonymity* instead of *contact cooperation*, we change our main ideal functionality in the blue marked lines as follows: (1) Receiver identifier is not included in the information revealed to the simulator for ciphertexts deemed anonymous (Figure 10, line 51); and (2) all procedures and details pertaining to re-encryption, resp., recursive decryption are omitted (Figure 9, lines 06,19, Figure 10, lines 03,16-33,38,39,41-46,51,53,55-58,60,62,65-67). Thus, any protocol that realizes this modified ideal functionality should additionally have *fully anonymous* ciphertexts (with PCS and FS guarantees), but is not required to have contact cooperation.

B.2 Game-Based Attempts

Initially, we tried to define security with a *game-based* approach. These attempts were unsatisfactory and eventually unsuccessful because the preliminary definitions were too complex, incomprehensible, and forced us to choose between unnatural modeling options and unrealistic restrictions of the adversary. We illustrate this with an example that our ultimate *simulation-based* definition in Appendix B solves naturally: Consider a ciphertext c_0 that is confidential and anonymous. The typical approach to model anonymity of c_0 is that either this *real ciphertext* c_0 or a *randomly sampled ciphertext* c_1 is given to the adversary. An adversary in our setting can obtain a (real or random) ciphertext c_b , $b \in \{0, 1\}$ by either watching a meeting at which c_b is exchanged or by exposing a party’s state that contains c_b in the local cache. However, if the definition has to replace c_0 by c_1 in a party’s exposed state to model anonymity of that ciphertext, this means that the definition needs to know how c_0 is embedded in this state. More concretely, the structure of the exposed state is construction-dependent, and so the location of cached ciphertext c_0 in this state also depends on the particular analyzed construction. Hence, in order to model anonymity of ciphertext with this traditional game-based approach, we would be forced to introduce construction-dependent artifacts in the definition. Alternatively, one could forbid exposures of states that contain an anonymous ciphertext c_b . We considered both options unnatural and undesirable. In contrast, the ideal functionality of our ultimate simulation-based definition models this case by specifying the information that an anonymous ciphertext would reveal more generically.

C MESH MESSAGING SECURITY ANALYSIS

To prove security of our MM construction from Section 5 with respect to our ideal functionality from Appendix B, we design a simulator in Figure 12. As elaborated upon below, code marked in

blue is relevant for the Simulator of *ASMesh 1* with *contact cooperation* but irrelevant for the Simulator of *ASMesh 2* with *receiver anonymity*.

C.1 Simulator

Our simulator translates the ideal functionality’s output on three operations into real (looking) counterparts. These operations are *key generation*, *broadcast* during a meeting, and *state exposure*. Simulating the key generation is straight forward.

Broadcast To simulate a broadcast, we have to simulate each broadcast ciphertext individually. For this, helper procedure BuildCtxt distinguishes between anonymous and non-anonymous ciphertexts. The former are simply sampled at random.

For the latter, we first use the *history-independence PKDR simulator* from Figure 18. This simulator creates ciphertexts together with the keys that are established by them on demand. By *on demand*, we mean that a ciphertext-key pair from a session can be created even if it was actually sent in the middle of this session and the ideal functionality didn’t provide any information on earlier communication in this session yet. This on-demand, history-independent simulation is necessary due to the way adversaries can start and stop watching parties meetings. (Note that early and late communication in a session can remain hidden entirely, if the adversary only watches a short period of meetings; for simulating the ciphertexts and keys exchanged during these watched meetings, the simulator has to work on-demand and history-independent.)

If the ciphertext is confidential (but non-anonymous), we sample the contained message m and the key for its encryption k_{AE} at random. Otherwise, if the ciphertext is neither confidential nor anonymous but contains a set of (re-encrypted) ciphertexts, we compile this set of ciphertext by recursively calling BuildCtxt. Using our message-anonymizer simulator from Appendix C.2, we encrypt the actual payload—which is either a re-encrypted set of ciphertexts or a message—and return the final, composed ciphertext, prepended with remaining path length pt and receiver public key.

The Simulator for *ASMesh 2* is changed in the blue marked lines to (1) omit procedures pertaining to re-encryption and (2) omit receiver public keys from all simulated ciphertexts (Note: non-anonymous ciphertexts are still built in the same way, otherwise).

State Exposure To simulate an exposed state, we begin with simulating the ciphertexts contained in the state with helper function BuildCtxt (also prepending with remaining graph forwarding degree nd). Then, for each session of the exposed party, we obtain the PKDR session state via the *history-independence PKDR simulator* from Figure 18. Furthermore, for every such session, we simulate the message-anonymizer *sender* state (see Appendix C.2). Finally, we build the party’s combined message-anonymizer *receiver* state (see Appendix C.2).

C.2 Message Anonymizer Simulator

To keep our overall simulation compact and comprehensible, we capture the cryptographic simulation of the message-anonymizer as well as the actual encryption of messages in a separate simulator. This simulator provides three interfaces: one for creating both a message-anonymizer ciphertext and a payload ciphertext (MA.enc^{sim}), one for composing a message-anonymizer *sender* state

<p>Simulator Initialization:</p> <pre> 00 $ST[\cdot] \leftarrow \perp$; $CT[\cdot] \leftarrow \perp$ On input (Gen, P) from IF \mathcal{F}_{MM}: 01 $(pk_{NK}, sk_{NK}) \leftarrow_{\\$} NK.gen$ 02 $ST_{MA}[\cdot] \leftarrow \perp$; $st_{MA} \leftarrow \perp$ 03 $ST[P] \leftarrow (sk_{NK}, pk_{NK})$ 04 Send pk_{NK} to Adversary \mathcal{D} On Input ($P_0, P_1, \tilde{C}_0, \tilde{C}_1$) from IF \mathcal{F}_{MM}: 05 $C_0, C_1 \leftarrow \emptyset$ 06 For all $\tilde{c} \in \tilde{C}_j, j \in \{0, 1\}$: 07 $C_j \xleftarrow{U} BuildCtxt(\tilde{c}, M, 1)$ 08 Send (C_0, C_1) to Adversary \mathcal{A} 09 Send OK to Adversary \mathcal{A} On Input (P, \tilde{C}, PC, SL) from IF \mathcal{F}_{MM}: 10 $(sk_{NK}, pk_{NK}) \leftarrow ST[P]$ 11 $C \leftarrow \emptyset$ 12 For all $\tilde{c} \in \tilde{C}$: 13 $C \xleftarrow{U} BuildCtxt(\tilde{c}, ST, 1)$ 14 $P^* \leftarrow \emptyset$; $K_{MA}^*[\cdot] \leftarrow \perp$ 15 For all $P' : \exists(P', t, i, pc, afs, \widehat{svar}) \in SL$: 16 If $\mathcal{S}_{PR}[\{P, P'\}] = \perp$: $\mathcal{S}_{PR}[\{P, P'\}](Init)$ 17 $(sk'_{NK}, pk'_{NK}) \leftarrow ST[P']$ 18 $(ST_{PR}[pk'_{NK}], K_{MA})$ 19 $\leftarrow \mathcal{S}_{PR}[\{P, P'\}](sim-st, P, \widehat{svar}, afs)$ 20 $ST_{MA}[P'] \leftarrow MA.sts^{sim}(P, P', \widehat{svar}, K_{MA})$ 21 $P^* \xleftarrow{U} \{P'\}$; $K_{MA}^*[P'] \leftarrow K_{MA}$ 22 $st_{MA} \leftarrow MA.str^{sim}(P, P^*, t, i, afs, \widehat{svar}, K_{MA}^*)$ 23 $st \leftarrow (sk_{NK}, pk_{NK}, (ST_{MA}, st_{MA}), ST_{PR}, C)$ 23 Send st to Adversary \mathcal{A} </pre>	<pre> Proc BuildCtxt($\tilde{c}, type, top$): 24 $(id, l, SC, P_s, P_r, pt, nd, \widehat{cvar}, m) \leftarrow \tilde{c}$ 25 If $CT[id] = \perp$: 26 If $P_s = \perp$: 27 $c \leftarrow_{\\$} C^{l+eoh}$ 28 $CT[id] \leftarrow c$ 29 Else: 30 $(c_{PR}, md, k_{AE}, k_{MA})$ 31 $\leftarrow_{\\$} \mathcal{S}_{PR}[\{P_s, P_r\}](sim-ct, P_s, P_r, \widehat{cvar})$ 32 If $m = \perp \wedge SC = \emptyset$: 33 $m \leftarrow 0^l$ 34 $k_{AE} \leftarrow_{\\$} \mathcal{K}$ 35 Else if $SC \neq \emptyset$: 36 $m \leftarrow \emptyset$ 37 For all $\tilde{c} \in SC$: 38 $m \xleftarrow{U} BuildCtxt(\tilde{c}, M, 0)$ 39 (c_{MA}, c_{AE}) 40 $\leftarrow_{\\$} MA.enc^{sim}(P_s, P_r, t, i, c_{PR}, md, k_{AE}, k_{MA}, l, m)$ 41 $CT[id] \leftarrow (c_{MA}, c_{PR}, c_{AE})$ 42 $(sk'_{NK}, pk'_{NK}) \leftarrow ST[P_r]$ 43 If $top = 0$: Return $CT[id]$ 44 Else if $type = M$: Return $(pt, pk'_{NK}, CT[id])$ 45 Else: Return $(nd, (pt, pk'_{NK}, CT[id]))$ </pre>
--	---

Figure 12: Simulator $\mathcal{S}^{pt, nd, eoh}$ for proving security of our mesh messaging MM construction. The simulator is parameterized by construction-dependent variables pt , nd , and eoh .

($MA.sts^{sim}$), and one for composing a message-anonymizer receiver state ($MA.str^{sim}$). We describe and explain the simulation of these three interfaces below.

- $MA.enc^{sim}(P_s, P_r, t, i, c_{PR}, md, k_{AE}, k_{MA}, l, m)$:
 - (1) If the initial message-anonymizer key of ratchet t in the session of $\{P_s, P_r\}$ is not yet set, set it to k_{MA}
Note: k_{MA} is the key that initializes the message-anonymizer key derivation chain that is *used* in ratchet t ; however, k_{MA} is *established* by the PKDR protocol in ratchet $t - 1$
 - (2) Derive message-anonymizer key $k_{MA,i}$ and tag tag_i for index i in ratchet t via i chained evaluations of the random oracle from k_{MA}
 - (3) Use $k_{MA,i}$ and tag_i for computing the message-anonymizer ciphertext via $(tag_i, AE.enc(k_{MA,i}, (t, i, l, md), tag_i))$
 - (4) Encrypt the message via $c_{AE} \leftarrow_{\$} AE.enc(k_{AE}, m, (c_{MA}, c_{PR}))$

For clarity, we neglect the prefix markers M; C; U; and R: with which our construction differentiates between payload encryption, session update ciphertext, and ciphertext re-encryption

- (5) Return(c_{MA}, c_{AE})
- $MA.sts^{sim}(P, P', t, i, \widehat{svar}, K_{MA})$:
 - (1) In our MM construction, we implement an independent message-anonymizer for each direction. Since the sender (and receiver) of one of these message-anonymizers only increments the ratchet-counter with every second ratchet, t is adjusted as follows:
If t was started by P' : $t \leftarrow t + 1$
 - (2) Get $k_{MA} \leftarrow K_{MA}[t]$ and derive key $k_{MA,i}$ for index i in ratchet t via i chained evaluations of the random oracle from it
 - (3) Extract \widehat{npr} from \widehat{svar}
 - (4) To set t consistently with the 2-ratchet update frequency within each direction's message-anonymizer, it is adjusted to $t \leftarrow t/2$
 - (5) Return $(t, i, l, k) = (t, i, \widehat{npr}, k_{MA,i})$
- $MA.str^{sim}(P, P^*, t, i, afs, \widehat{svar}, K_{MA}^*)$:

For each session with $P' \in P^*$ in which P started ratchet t :

- (1) Following the observation that ratchet counters in each direction's message anonymizer are incremented

with halved frequency, we set:

- $t_{\text{now}} \leftarrow (t - 1)/2$
- (2) i_{now} is the maximal index i' in ratchet $t - 1$ plus $\text{fut} - 1$, where i' is derived from afs
- (3) $i_{\text{next}} \leftarrow \text{fut} - 1$
- (4) ck_{now} is the output of the i_{now} -th evaluation of the random oracle chain that started with input $K_{\text{MA}}^*[P'] [t - 1]$
- (5) ck_{next} is the output of the i_{next} -th evaluation of the random oracle chain that started with input $K_{\text{MA}}^*[P'] [t + 1]$

For each session with $P' \in P^*$ in which P' started ratchet t :

- (1) $t_{\text{now}} \leftarrow t/2$
- (2) i_{now} is the maximal index i' in ratchet t plus $\text{fut} - 1$, where i' is derived from afs
- (3) $i_{\text{next}} \leftarrow 0$
- (4) ck_{now} is the output of the i_{now} -th evaluation of the random oracle chain that started with input $K_{\text{MA}}^*[P'] [t]$
- (5) $ck_{\text{next}} \leftarrow \perp$

For each tuple (t', i') with session partner $P' \in P^*$ that is encoded in afs , i.e., $(t', I') \in \text{afs}$ and $i' \in I'$:

- (1) Obtain the key and tag outputs $(k_{\text{MA}}^*, \text{tag}^*)$ of the i' -th evaluation of the random oracle chain that started with input $K_{\text{MA}}^*[P'] [t']$
- (2) Add $(k_{\text{MA}}^*, \text{tag}^*)$ with message-anonymizer index-tuple $(t'/2, i')$ to hash table ht

Finally, return $st = (ht, ST)$, where $ST[pk'_{\text{NK}}] = (t_{\text{now}}, i_{\text{now}}, i_{\text{next}}, ck_{\text{now}}, ck_{\text{next}})$ is the message-anonymizer receiver session state for each partner P'

C.3 Proof by Reduction

THEOREM C.1. *Assume that NK is a secure NIKE scheme according to Definition F.2, PR is a secure PKDR scheme according to Definition E.1, AE is a secure AEAD scheme with random ciphertexts according to Definition F.3, and H and G are modeled as random oracles. Then base protocol MM 1 UC-realizes base functionality $\mathcal{F}_{\text{MM}}^{\text{pt,nd,eoh}}$ in the $\mathcal{F}_{\text{mesh}}$ -hybrid model.*

Our proof begins with a real, chronological execution of protocol MM against adversary \mathcal{A} . Step by step, we replace components of this real protocol execution with indistinguishable random counterparts. In the last step, we show that this random *chronological* protocol execution can be simulated fully *non-chronologically*. This final step is crucial since the adversary against the MM construction actually doesn't interact with this construction chronologically. In contrast, it sees ciphertexts non-chronologically based on when they are observed during a meeting between watched parties. Since ciphertexts travel through the network on different paths, and since the adversary adaptively can start and stop watching parties in the network, the order in which ciphertexts are seen—and, hence, in which they are simulated—is independent of the order in which they are originally sent in a session.

In total, this allows us to prove that our simulation from Appendices C.1 and C.2, which is based on the interaction with ideal functionality $\mathcal{F}_{\text{MM}}^{\text{pt,nd,eoh}}$ from Appendix B, is indistinguishable from the real, chronological execution of protocol MM. The indistinguishability of the modified simulation for *ASMesh 2*, based on the

interaction with the modified ideal functionality discussed in Appendix B.1, from the real, chronological protocol execution follows immediately:

THEOREM C.2. *Assume that NK is a secure NIKE scheme according to Definition F.2, PR is a secure PKDR scheme according to Definition E.1, AE is a secure AEAD scheme with random ciphertexts according to Definition F.3, and H and G are modeled as random oracles. Then alternative protocol MM 2 UC-realizes alternative functionality $\mathcal{F}_{\text{MM}}^{\text{pt,nd,eoh}}$ in the $\mathcal{F}_{\text{mesh}}$ -hybrid model.*

Game 1: Initialization with NIKE In the first game, we replace all keys k_{EE} in session initialization MM.init by random keys k_{EE}^* (Figure 4, line 07). Distinguishing this change is trivially reduced to the security of NIKE scheme NK. The two remaining NIKE keys exchanged during each session initialization are irrelevant for our security notion against passive adversaries.

Using random oracle G, we also replace keys k_{PR} , k_{MA}^0 , and k_{MA}^1 in each session initialization (Figure 4, line 10), which is detectable with negligible probability.

Game 2: Random Keys for Encryption and Message-Anonymizer with PKDR In our ideal functionality for MM, we carefully align the effect of state exposures and the resulting *security requirements for an MM protocol* on the one side with the *security guarantees provided by our PKDR security definition* on the other side. As a result, we can now replace both all *secure PKDR message keys* as well as the *secure initial MA keys for each ratchet* with random keys, respectively.

However, in this game hop, we already have to regard the non-chronological interaction between adversary and (simulated) MM protocol execution. The reason is that, at the moment at which a ciphertext is sent in the real, chronological protocol execution, it is unclear whether this ciphertext will provide any security guarantees. More concretely, when sending a PKDR ciphertext, it is unclear whether this ciphertext will provide confidentiality (and anonymity) or whether a future exposure of the receiver's state will render this ciphertext fully insecure. Interestingly, this is not an issue in the non-chronological interaction between MM simulation and MM adversary: whenever a ciphertext is actually *given* to the adversary in this (non-chronological) interaction, our ideal functionality fixes the provided security guarantees irreversibly (i.e., if it is insecure, confidential, or confidential *and* anonymous). The slightly misleading conclusion is that our chronological MM protocol execution would have to wait until the non-chronological adversary sees an MM ciphertext in order to learn the required security guarantees of that MM ciphertext. This is misleading because the non-chronological adversary may want to see a later ciphertext c_2 in a session earlier than an early ciphertext c_1 in that same session. Waiting until c_1 is seen by the adversary would stall the chronological protocol execution infinitely.

To close this apparent gap between chronological *protocol execution* and non-chronological *adversary interaction*, we split the execution of the PKDR protocol from the remaining components of our MM protocol—which are: payload encryption and MA protocol execution. While the execution of the PKDR protocol remains

unchanged, we simulate payload encryption and MA protocol execution *lazily*. This means that we only calculate the payload ciphertext and the MA ciphertext as soon as the adversary sees them (non-chronologically). This simulation behavior is undetectable because the actual computations remain the same.

Since we know the required security guarantees of an MM ciphertext at the moment at which we give it to the (non-chronological) adversary, our lazy simulation can now take advantage of this information as follows. For all confidential ciphertexts, we now replace the used encryption key k_{AE} by a random key k_{AE}^* (Figure 5, lines 08, 11; note that the same replacement is done on the receiver side). Furthermore, for all confidential ciphertexts, we now replace the initial MA key k_{MA} that is input to algorithm $MA.up_R$ by a random key k_{MA}^* (Figure 5, line 09; again, the same replacement is done on the receiver side).

Note that the time of establishing keys k_{AE} and k_{MA} via the PKDR protocol differs from the time our simulation lazily replaces them for the computation of payload encryption and MA receiver-state update. Therefore, we now explain how we can simulate payload encryption and MA protocol execution non-chronologically, on demand. Since payload encryption is stateless, its on-demand simulation is trivial. Yet, for MA simulation, we have to make sure that all message-anonymization encryptions in the next ratchet $t + 1$ can be simulated *even before* a secure ciphertext from ratchet t is given (non-chronologically) to the adversary. For this, we use a random oracle that is programmed lazily: For every ciphertext, given to the adversary, it programs the used MA key tuple (mk, tag, ck_{now}) (Figure 3, line 12) as output to the random oracle. To maintain consistency for the entire ratchet $t + 1$, all random oracle queries for the corresponding MA ratchet are pre-programmed. This means, chaining each output ck to the subsequent input until the last index in that ratchet is computed. Only the very first random oracle input $ck = k_{MA}$ in ratchet $t + 1$ is not pre-programmed. Instead, k_{MA} is programmed as soon as an exposed session state or a ciphertext sent in ratchet t fixes it. For the case that a *secure* ciphertext in ratchet t fixes it, we describe its replacement by a random key above.

An adversary that distinguishes the conducted change is reduced to an adversary that breaks the security of the PKDR protocol. The reduction simulates the entire PKDR protocol (chronologically) with the oracles specified in Figure 16. For all secure MM ciphertexts given to the (non-chronological) MM adversary, the established PKDR keys are obtained by (adaptively and retrospectively) querying oracle $Chall$. For all insecure MM ciphertexts given to the (non-chronological) MM adversary, the established PKDR keys are obtained by (adaptively and retrospectively) querying oracle $Reveal$.

Game 3: Random Payload Encryption with AEAD Using the randomized keys k_{AE}^* for payload encryption, we now replace the messages that are encrypted in secure ciphertexts. That means, for all confidential ciphertexts, we now replace the encrypted message m by an equally long constant bit string $0^{|m|}$ (Figure 5, line 11). Detecting this change is negligible, as a successful distinguisher is directly reduced to breaking AEAD security of AE.

Game 4: Random Message-Anonymizer Keys with ROM Our *almost* lazy simulation of the MA key derivation chain from Game 3 is now turned *fully* lazily. That means, we only compute MA key

tuple (mk, tag, ck_{now}) (Figure 3, line 12) as output to the random oracle as soon as our simulation uses either of these values or as soon as the adversary queries the random oracle on the matching input itself. This change is undetectable as it only shifts the time of computation.

As a second step, we abort the simulation as soon as the adversary queries the random oracle on an input whose output contains a key mk that is used for MA encryption of an *anonymous* ciphertext. Since a ciphertext is only required to provide anonymity if the initial MA key k_{MA} in that ratchet was established by a confidential ciphertext in the prior ratchet, we can show that the probability of aborting is negligible. More concretely, due to Game 2, secure initial MA keys k_{MA} are replaced by random keys k_{MA}^* . Hence, querying the random oracle on any intermediate ck of the key derivation ratchet that is started by k_{MA}^* is bounded by the birthday paradox. The only exception to this is that an intermediate ck is leaked by an exposed session state. In this case, however, subsequent ciphertexts in the corresponding MA ratchet are not required to be anonymous anymore.

Game 5: Random MA Encryption with IND $\$$ Since, by Game 4, none of the keys for anonymous ciphertexts was ever given to the adversary, and all of these keys are random bit strings, we can now change all real anonymous ciphertexts to random ciphertexts. More concretely, we replace the MA ciphertexts c' (Figure 3, line 13) of all anonymous PKDR ciphertexts by random MA ciphertexts $c^* \leftarrow_{\$} C^{|c'|}$. Detecting this modification is reduced to breaking the IND $\$$ security of AEAD scheme AE.

Game 6: Non-Chronological Simulation All parts of the MM protocol except for the PKDR are now simulated on demand, based on the non-chronological interaction between simulator and adversary. To show that our simulation can run fully non-chronological, without knowing anything about the protocol execution except for the ideal functionality's inputs, we also have to turn the PKDR simulation non-chronological. For this, we use the history-independence simulator from Figure 18. That is, instead of computing PKDR ciphertexts, PKDR session states, and (insecure) PKDR keys with a real, chronological PKDR protocol execution, we obtain these values on demand by querying this simulator non-chronologically.

In total, this simulation equals our simulator from Appendices C.1 and C.2. Showing that the probability of distinguishing a real MM protocol execution from our simulator is negligible concludes our proof. \square

D ADDITIONAL PRIMITIVES

D.1 CKA

We now provide the formal security definition, detailed (more secure) construction, and corresponding security analysis for the CKA primitive from Section 3.1 used in our PR construction.

Security In Figure 13, we give a formal game-based security definition for the CKA primitive introduced in Section 3.1. At a high-level parties in CKA send key-establishing ciphertexts to one another in a ping-pong order. The key of a ciphertext should be secure as long as if the receiver is not corrupted (i) after it sent its previous ciphertext or (ii) before it receives this ciphertext.

This security game is for *selective-security*, based on challenge sender and ratchet P^* , t^* specified by the adversary. In the security game, parties are first initialized with some shared random key (line 02). The adversary can invoke oracle $\text{Snd}(P)$ when it is P 's turn to send to the other party (line 07). The oracle then invokes CKA.snd (line 09), from which the corresponding key k and ciphertext c is returned.

The adversary can also invoke oracle $\text{Rcv}(P)$ when it is P 's turn to receive from the other party (line 19). The oracle then invokes CKA.rcv (line 20), and the output key k is checked with that which was output by \bar{P} 's most recent send (lines 22, 17, and 10) for correctness.

The adversary can also invoke oracle $\text{Expose}(P)$ to expose the state of one of the parties, after checking the following requirements. If P is the challenge sender, then this exposure is always allowed; otherwise, if P is the receiver, then it must not have just sent the ciphertext *before* the challenge ratchet, for otherwise the adversary could trivially win (line 12).

Finally, the adversary can invoke $\text{Chall}(P)$ to send a challenge ciphertext. First, the game checks if this challenge is for the correct party and ratchet (line 14). Then, it invokes CKA.snd (line 16) from which it returns *only* the ciphertext.

The goal of the adversary in this game is to *recover* the key k^* corresponding to the challenge ciphertext. We remark that our PR passes the CKA output through a random oracle, and thus recovery security suffices for proving security of PR.

Definition D.1. A CKA scheme is *selectively-secure against key-recovery attacks* if for any adversary \mathcal{A} in the security game of Figure 13, the probability it guesses challenge key k^* is negligible.

History-Independence In order to prove security of our final Mesh Messaging construction MM, we need our PR construction to be *history-independent*. We thus also require the same property for the CKA from which we build PR. For CKA, *history-independence* intuitively means each CKA state and ciphertext is independent of, and thus can be simulated without, knowledge of the history of the execution. We give a formal definition below.

First, let $\text{seq} = (\text{op}_1, \text{op}_2, \dots, \text{op}_\ell)$ be some sequence of CKA operations. Recall that these will be invocation of CKA.snd and CKA.rcv in ping-pong order. Now consider the execution of some CKA protocol on seq . Also, let l be a counter initialized to 0, and \mathcal{S}_{CKA} be some stateful algorithm that is run given (possibly out-of-order) information from the execution of seq through two possible inputs:

- On input $(\text{sim-ct}, t)$, \mathcal{S}_{CKA} outputs simulated ciphertext c for ratchet t , and current dictionary of all known keys, K . Let $\text{sim-info}[l] \leftarrow (c, K)$ and $\text{exec-info}[l]$ be the actual ciphertext of ratchet t , along with the real CKA keys for ratchets t in which $K[t] \neq \perp$. Increment $l = l + 1$.
- On input $(\text{sim-st}, P, t)$, \mathcal{S}_{CKA} outputs some simulated ratchet t state st for P , and current dictionary of all known shared keys, K . Let $\text{sim-info}[l] \leftarrow (st, K)$ and $\text{exec-info}[l]$ be the actual state of party P at this point in the execution of seq , along with the real CKA keys for ratchets t in which $K[t] \neq \perp$. Increment $l = l + 1$.

Definition D.2. CKA is *history-independent* if for every seq , the distributions sim-info and exec-info are computationally indistinguishable (over the randomness sampled by the algorithms of CKA and \mathcal{S}_{CKA}).

Construction Our simple group-based construction presented below is from [BFG⁺22]. It is secure and history-independent if H is modelled as a programmable random oracle and the Computational Diffie-Hellman (CDH) assumption holds in the group G ; i.e., G is *CDH-secure*.

- $\text{CKA.init}(x_0)$ takes exponent x_0 , sets $\text{cst}_A \leftarrow g^{x_0}$ for the first sender and $\text{cst}_B \leftarrow x_0$ for the first receiver.
- $\text{CKA.snd}(\text{cst})$ takes the current state $\text{cst} = h$ and proceeds as follows:
 - (1) Samples random exponent x , sets key $k \leftarrow h^x$, and ciphertext $c \leftarrow g^x$.
 - (2) Then sets new state $\text{cst} \leftarrow x \cdot H(k)$.
 - (3) Returns (cst, k, c)
- $\text{CKA.rcv}(\text{cst}, h)$ takes the current state $\text{cst} = x$ and message h and proceeds as follows:
 - (1) Computes key $k = h^x$ and sets new state $\text{cst} \leftarrow h^{H(k)}$.
 - (2) Returns (cst, k) .

Security Analysis We now prove that our CKA scheme is *secure* and *history-independent*.

THEOREM D.3. *If H is modelled as a random oracle and G is CDH-secure according to Definition [?], then CKA is selectively-secure against key-recovery attacks.*

PROOF. Let (g^a, g^b) be the CDH challenge. The reduction simulates the CKA protocol in a straight-forward way, except for when embedding the CDH challenge. I.e., for every ratchet $t \notin \{t^* - 1, t^*, t^* + 1\}$, it proceeds as normally in the protocol. Then:

- In ratchet $t^* - 1$, it sets $\text{cct}_{t^*-1} \leftarrow g^a$, $k_{t^*-1} \leftarrow g^{a \cdot x \cdot H(k_{t^*-2})}$ and $\text{cst}_{\bar{P}^*} \leftarrow \perp$, where x is the exponent used to simulate $\text{cct}_{t^*-2} = g^x$.
- In ratchet t^* , it sets $\text{cct}_{t^*} \leftarrow g^b$, $k_{t^*} \leftarrow g^{ab \cdot H(k_{t^*-1})}$ (implicitly), $y \leftarrow_{\mathcal{S}} X$, and $\text{cst}_{P^*} \leftarrow y$.
- In ratchet $t^* + 1$, it randomly samples x' as in the protocol, and sets $\text{cct}_{t^*+1} \leftarrow g^{x'}$, $k_{t^*+1} \leftarrow g^{yx'}$.

The reduction guesses whether and when the adversary queries the random oracle on k_{t^*} . If it guesses that it does, it exponentiates the corresponding input by $H(k_{t^*-1})^{-1}$ and forwards this to its challenger. Otherwise, it does the same for the adversary's guess of k' . Note that the adversary can only distinguish randomly sampled y from $b \cdot H(k_{t^*})$ if it indeed queries the random oracle on k^* . If it does before the reduction guesses so, the reduction will simply forward a random group element to its challenger. Security thus follows from that of the CDH-security of group G .

Correctness of the scheme easily follows, too. \square

THEOREM D.4. *CKA is history-independent if H is modelled as a programmable random oracle.*

PROOF. We first specify \mathcal{S}_{CKA} . It initializes $X[\cdot], S[\cdot], K[\cdot] \leftarrow \perp$. For every input $(\text{sim-ct}, t)$, it samples random exponent $X[t]$. If $S[t-1] = \perp$, it samples it randomly. Then it sets $K[t] \leftarrow g^{X[t] \cdot S[t-1]}$. Let $t' \leftarrow t$. It now runs the following function, which we call

<p>Game $\text{REC}_{\text{CKA}}^{P^*, t^*}(\mathcal{A})$</p> <p>00 $t_A, t_B \leftarrow 0$</p> <p>01 $k \leftarrow_{\mathcal{S}} \mathcal{K}$</p> <p>02 $(st_A, st_B) \leftarrow \text{CKA.init}(k)$</p> <p>03 $S \leftarrow A; R \leftarrow \perp$</p> <p>04 $k_{\text{last}} \leftarrow \perp$</p> <p>05 $k' \leftarrow_{\mathcal{S}} \mathcal{A}$</p> <p>06 Stop with $k' \stackrel{?}{=} k^*$</p> <p>Oracle Snd(P)</p> <p>07 Require $S = P$</p> <p>08 $t_P \leftarrow t_P + 1; S \leftarrow \perp; R \leftarrow \bar{P}$</p> <p>09 $(stp, k, c) \leftarrow_{\mathcal{S}} \text{CKA.snd}(stp)$</p> <p>10 $k_{\text{last}} \leftarrow k$</p> <p>11 Return (k, c)</p> <p>Oracle Expose(P)</p> <p>12 Require $P = P^* \vee t_P \neq t^* - 1$</p> <p>13 Return stp</p>	<p>Oracle Chall(P)</p> <p>14 Require $S = P = P^* \wedge t_P = t^* - 1$</p> <p>15 $t_P \leftarrow t_P + 1; S \leftarrow \perp; R \leftarrow \bar{P}$</p> <p>16 $(stp, k^*, c) \leftarrow_{\mathcal{S}} \text{CKA.snd}(stp)$</p> <p>17 $k_{\text{last}} \leftarrow k^*$</p> <p>18 Return c</p> <p>Oracle Rcv(P)</p> <p>19 Require $R = P$</p> <p>20 $t_P \leftarrow t_P + 1; R \leftarrow \perp; S \leftarrow P$</p> <p>21 $(stp, k) \leftarrow_{\mathcal{S}} \text{CKA.rcv}(stp, c)$</p> <p>22 If $k \neq k_{\text{last}}$: WIN</p> <p>23 Return</p>
---	---

Figure 13: Game REC for defining security of CKA.

$\text{prog-loop}(t')$: While $S[t'] = \perp \wedge X[t' + 1] \neq \perp$; it sets $S[t'] \leftarrow X[t'] \cdot H(K[t'])$ and $K[t' + 1] \leftarrow g^{S[t'] \cdot X[t' + 1]}$, and increments $t' \leftarrow t' + 1$. Once the loop exits, if $S[t'] \neq \perp$, it programs $H(K[t']) := S[t'] / X[t']$; otherwise, it sets $S[t'] \leftarrow X[t'] \cdot H(K[t'])$. After the above execution of $\text{prog-loop}(t')$, it returns $(g^{X[t]}, K)$.

For every input $(\text{sim-st}, P, t)$, if P sent in ratchet t , \mathcal{S}_{CKA} checks if $S[t] = \perp$. If so, it samples random exponent $S[t]$, and if also $X[t + 1] \neq \perp$, it sets $K[t + 1] \leftarrow g^{S[t] \cdot X[t + 1]}$ and $t' \leftarrow t + 1$, then runs $\text{prog-loop}(t')$. Finally, it returns $(S[t], K)$.

If P received in ratchet t , \mathcal{S}_{CKA} checks if $S[t] \neq \perp$. If so, it returns $(g^{S[t]}, K)$; otherwise, it samples random $S[t]$, and if also $X[t + 1] \neq \perp$, it sets $K[t + 1] \leftarrow g^{S[t] \cdot X[t + 1]}$ and $t' \leftarrow t + 1$, then runs $\text{prog-loop}(t')$. Finally, it returns $g^{S[t]}$.

Now that we have specified \mathcal{S}_{CKA} , we argue that it provides history-independence. Indeed, we can easily see that the distributions sim-info and exec-info are computationally-indistinguishable. This is because in both, ciphertexts are always sampled randomly. Also, states are always computed in an identical manner, except for the case in which \mathcal{S}_{CKA} does not have sufficient information to compute one and instead samples it randomly. H is modelled as a programmable random oracle, so this is still an identical distribution because (i) if there is always insufficient information to compute it; it is an output of H , so it is also randomly distributed in exec-info , and (ii) if at some point there is sufficient information to compute it; we can program H , so we still get an identical view as in exec-info . A similar argument holds for the keys K output by the simulator. (There is only some negligible probability that the distinguisher succeeds due to the fact that (with some negligible probability), the random oracle can be queried on values before they are programmed) \square

D.2 Asynchronous CKA

We now provide the formal security definition, detailed (more secure) construction, and corresponding security analysis for the ACKA primitive from Section 3.1 used in our PR construction.

Security In Figure 14, we give a formal game-based security definition for the ACKA primitive introduced in Section 3.1. At a high-level parties in ACKA send key-establishing ciphertexts to one another *asynchronously* in *ratchets*. Each new send establishes a new PCS epoch pc for the sender. The key of a ciphertext in ratchet t should be secure as long as if the receiver is not corrupted (i) after it sent the newest ciphertext which the sender of the challenge ciphertext had received before sending the challenge or (ii) before it receives *all* ciphertexts of ratchet t .

This security game is for *selective-security*, based on challenge sender, ratchet, and pcs epoch: P^*, t^*, pc^* , respectively, specified by the adversary. In the security game, parties are first initialized with a shared random key (line 06). The adversary can query oracle Snd(P) to send a new ciphertext from P . First, the oracle checks if P should start a new ratchet, and if so, does it (lines 10-11). Then, it invokes ACKA.snd on behalf of P (line 12), from which it returns the corresponding k, c, md .

The adversary can also query oracle Expose(P) to expose the state of one of the parties. The oracle first checks that if P is not the challenge sender, that a challenge is currently not in progress (line 16). If so, it adds to the set XP_P of exposed PCS epochs for P : Those epochs between \bar{P} 's current view of P 's PCS latest PCS epoch, and P 's actual PCS epoch (line 17).

The adversary can also query oracle Chall(P) which sends a challenge ciphertext. First, the oracle checks if P should start a new ratchet, and if so, does it (lines 20-21). Then it checks (i) if P is the challenge party P^* , and if their current ratchet and pcs epoch are t^*, pc^* ; and (ii) that the epoch which P views as \bar{P} 's current epoch is not exposed (line 22). If these checks pass, then it sets that a challenge is currently in progress (line 24). It then invokes ACKA.snd

(line 25) and returns the corresponding challenge ciphertext and metadata.

Finally, the adversary can query oracle $\text{Rcv}(P, c, md)$, which instructs P to receive ciphertext and metadata c, md . First, the oracle checks that this ciphertext, metadata pair is indeed in-transit from \bar{P} . (line 29). If so, it invokes ACKA.rcv on this pair (line 30). Then it checks that the key and corresponding ratchet that ACKA.rcv outputs is indeed correct (line 32). Finally, the oracle sets that a challenge is no longer in progress if every ciphertext from the challenge ratchet has been received (line 36).

The goal of the adversary in this game is to *recover* the key k^* corresponding to the challenge ciphertext. We remark that our PR passes the ACKA output through a random oracle, and thus recovery security suffices for proving security of PR.

Definition D.5. An ACKA scheme is *selectively-secure against key-recovery attacks* if for any adversary \mathcal{A} in the security game of Figure 14, the probability it guesses challenge key k^* is negligible.

History-Independence In order to prove security of our final Mesh Messaging construction MM, we need our PR construction to be *history-independent*. We thus also require the same property for the ACKA from which we build PR. For ACKA, *history-independence* intuitively means each ACKA state and ciphertext is independent of, and thus can be simulated without, knowledge of the history of the execution. We give a formal definition below.

First, let $\text{seq} = (\text{op}_1, \text{op}_2, \dots, \text{op}_\ell)$ be some sequence of ACKA operations. We will take advantage of the bookkeeping in the security game of Figure 14, and thus will specify this sequence in terms of oracle calls to Snd, Rcv (as in Section E.2). Now consider the execution of some ACKA protocol on seq . Also, let l be a counter initialized to 0, and $\mathcal{S}_{\text{ACKA}}$ be some stateful algorithm that is run given (possibly out-of-order) information from the execution of seq through two possible inputs:

- On input $(\text{sim-ct}, P, t, i, pc, pcp, pcp_{\text{init}}, \ell_{\text{prv}})$, $\mathcal{S}_{\text{ACKA}}$ simulates the i -th ciphertext c and key k of ratchet t on behalf of P , with P 's PCS epoch pc and \bar{P} 's PCS epoch pcp . Additionally pcp_{init} is P 's view of \bar{P} 's PCS epoch at the start of ratchet t and ℓ_{prv} is the number of ciphertexts sent in ratchet $t - 2$. Let $\text{sim-info}[l] \leftarrow (c, k)$ and $\text{exec-info}[l]$ be the actual i -th ciphertext and key of ratchet t produced during the execution of seq . Increment $l = l + 1$.
- Let (t, pc, pcp) be the respective values of $(t_p, PC_P[P], PC_P[\bar{P}])$ at some point in the execution of seq . Also let i be the number of times a ciphertext was sent in t_p at this point in the execution, and $pc2ratch$ be a dictionary mapping pc values to the ratchets t in which they were initiated at this point in the execution (i.e., the ratchet t in which $PC_P[P]$ was equal to pc). Let all-rcvd be an array storing 1 for index t if all messages of that ratchet have been received by P ; 0 otherwise. Finally, let $pc_{\text{prv}}, pcp_{\text{init}}, \ell_{\text{prv}}$ be as above. On input $(\text{sim-st}, P, t, i, pc, pcp, pc_{\text{prv}}, pcp_{\text{init}}, \ell_{\text{prv}}, pc2ratch, \text{all-rcvd})$, $\mathcal{S}_{\text{ACKA}}$ outputs a simulated st . Let $\text{sim-info}[l] \leftarrow st$ and $\text{exec-info}[l]$ be the actual state of party P at this point in the execution of seq . Increment $l = l + 1$.

Definition D.6. ACKA is *history-independent* if for every seq , the distributions sim-info and exec-info are computationally indistinguishable (over the randomness sampled by the algorithms of ACKA and $\mathcal{S}_{\text{ACKA}}$).

Construction Our simple group-based construction presented in Figure 15 is very similar to our CKA construction and relies on techniques from [BFG⁺22]. It is secure and history-independent if H is modelled as a programmable random oracle and the Computational Diffie-Hellman (CDH) assumptions holds in the group G ; i.e., G is *CDH-secure*.

The parties are first initialized with their own starting secret exponent $x[0]$, and their counterpart's starting public key $g^{x[0]}$ (lines 03-04).

When one party P wants to send, it first checks if it should start a new ratchet, and if so, does it (lines 07-08). It then samples new exponent x , computes the secret key acs by exponentiating its current view of \bar{P} 's public key h with x , and stores a new secret exponent $x \cdot H(acs)$ (lines 10-12).

When a party P wants to receive a ciphertext h_c, md , it first checks if the ciphertext establishes a new ratchet (line 20). If so, it advances to the new ratchet, and uses subroutine end-ratch to ensure that once all of the ciphertexts for the previous receiving ratchet have been received, the secret keys which may have been encrypted to during that ratchet are all deleted (lines 20-24). Next, it derives the secret key acs by exponentiating the h_c part of the ciphertext with the stored secret exponent corresponding to the PCS epoch pcp' of P which \bar{P} indicated it sent to in md (line 25). Finally, P checks if it needs to update its current PCS epoch view of \bar{P} , and if so, updates the public key for \bar{P} by exponentiating the h_c part of the ciphertext by $H(acs)$ just derived (lines 26-28).

Security Analysis We now prove that our ACKA scheme is *secure* and *history-independent*.

THEOREM D.7. *If H is modelled as a random oracle and G is CDH-secure according to Definition F.1, then ACKA in Figure 15 is selectively-secure against key-recovery attacks.*

PROOF. Let (g^a, g^b) be the CDH challenge. The reduction simulates the ACKA protocol in a straight-forward way, except for when embedding the CDH challenge. I.e., for every ratchet $t \notin \{t^* - 1, t^*, t^* + 1\}$, it proceeds as normal in the protocol. Then:

- In ratchet $t^* - 1$, it guesses pcs epoch pc to be that which P^* will use to establish the challenge key; if it guesses wrong, it returns a random group element to the challenger. It is easy to see that the reduction can make this guess with only non-negligible loss in success probability. For the iteration of ACKA.snd corresponding to pc , let pc' be the corresponding pcs epoch of P^* for which a key is established. The reduction sets $acct \leftarrow g^a$, $acs \leftarrow g^{a \cdot x_{pc'}[pc']}$ and $x_{\bar{P}^*}[pc] \leftarrow \perp$. For all other iterations, it will proceed as in the protocol.
- In ratchet t^* , for all $pc \neq pc^*$, it will proceed as in the protocol. For pc^* , it will set $acct \leftarrow g^b$, $acs^* \leftarrow g^{ab \cdot H(acs)}$ (implicitly), $y \leftarrow_{\mathcal{S}} X$ and $x_{P^*}[pc^*] \leftarrow y$.
- In ratchet $t^* + 1$, when \bar{P} establishes keys with pcs epoch pc^* of P^* and its own pcs epoch pc'' , it randomly samples

<p>Game $\text{REC}^{\mathcal{P}^*, t^*, pc^*}(\mathcal{A})$</p> <p>00 $AC_A[\cdot], AC_B[\cdot] \leftarrow \perp$</p> <p>01 $PC_A[\cdot], PC_B[\cdot] \leftarrow 0$</p> <p>02 $t_A, t_B \leftarrow 0$</p> <p>03 $ch \leftarrow 0$</p> <p>04 $XP_A, XP_B \leftarrow \emptyset$</p> <p>05 $k \leftarrow_{\mathcal{S}} \mathcal{K}$</p> <p>06 $(st_A, st_B) \leftarrow \text{ACKA.init}(k)$</p> <p>07 $snd\text{-}par_A \leftarrow 1; snd\text{-}par_B \leftarrow 0$</p> <p>08 $k' \leftarrow_{\mathcal{S}} \mathcal{A}$</p> <p>09 Stop with $k' \stackrel{?}{=} k^*$</p> <p>Oracle $\text{Snd}(\mathcal{P})$</p> <p>10 If $t_{\mathcal{P}} \bmod 2 \neq snd\text{-}par_{\mathcal{P}}$:</p> <p>11 $t_{\mathcal{P}} \leftarrow t_{\mathcal{P}} + 1$</p> <p>12 $(stp, k, c, md) \leftarrow_{\mathcal{S}} \text{ACKA.snd}(stp)$</p> <p>13 $AC_{\mathcal{P}}[(c, md)] \leftarrow (t_{\mathcal{P}}, PC_{\mathcal{P}}[\mathcal{P}], k)$</p> <p>14 $PC_{\mathcal{P}}[\mathcal{P}] \leftarrow PC_{\mathcal{P}}[\mathcal{P}] + 1$</p> <p>15 Return (k, c, md)</p> <p>Oracle $\text{Expose}(\mathcal{P})$</p> <p>16 Require $\mathcal{P} = \mathcal{P}^* \vee ch = 0$</p> <p>17 $XP_{\mathcal{P}} \leftarrow \bigcup \{pc : PC_{\overline{\mathcal{P}}}[\mathcal{P}] \leq pc < PC_{\mathcal{P}}[\mathcal{P}]\}$</p> <p>18 Return stp</p>	<p>Oracle $\text{Chall}(\mathcal{P})$</p> <p>19 $t'_{\mathcal{P}} \leftarrow t_{\mathcal{P}}$</p> <p>20 If $t'_{\mathcal{P}} \bmod 2 \neq snd\text{-}par_{\mathcal{P}}$:</p> <p>21 $t'_{\mathcal{P}} \leftarrow t_{\mathcal{P}} + 1$</p> <p>22 Require $\mathcal{P} = \mathcal{P}^* \wedge t'_{\mathcal{P}} = t^* \wedge$ $PC_{\mathcal{P}}[\mathcal{P}] = pc^* \wedge PC_{\mathcal{P}}[\overline{\mathcal{P}}] \notin XP_{\overline{\mathcal{P}}}$</p> <p>23 $t_{\mathcal{P}} \leftarrow t'_{\mathcal{P}}$</p> <p>24 $ch \leftarrow 1$</p> <p>25 $(stp, k^*, c, md) \leftarrow_{\mathcal{S}} \text{ACKA.snd}(stp)$</p> <p>26 $AC_{\mathcal{P}}[(c, md)] \leftarrow (t_{\mathcal{P}}, PC_{\mathcal{P}}[\mathcal{P}], k^*)$</p> <p>27 $PC_{\mathcal{P}}[\mathcal{P}] \leftarrow PC_{\mathcal{P}}[\mathcal{P}] + 1$</p> <p>28 Return (c, md)</p> <p>Oracle $\text{Rcv}(\mathcal{P}, c, md)$</p> <p>29 Require $AC_{\overline{\mathcal{P}}}[(c, md)] \neq \perp$</p> <p>30 $(stp, k, (t, \ell)) \leftarrow_{\mathcal{S}} \text{ACKA.rcv}(stp, c, md)$</p> <p>31 $(t', pc', k') \leftarrow AC_{\overline{\mathcal{P}}}[(c, md)]$</p> <p>32 If $(t', k') \neq (k, t)$: WIN</p> <p>33 $t_{\mathcal{P}} \leftarrow \max\{t_{\mathcal{P}}, t\}$</p> <p>34 $PC_{\mathcal{P}}[\overline{\mathcal{P}}] \leftarrow \max\{PC_{\mathcal{P}}[\overline{\mathcal{P}}], pc'\}$</p> <p>35 $AC_{\overline{\mathcal{P}}}[(c, md)] \leftarrow \perp$</p> <p>36 If $ch = 1 \wedge \overline{\mathcal{P}} = \mathcal{P}^* \wedge t_{\mathcal{P}} \geq t^* + 2 \wedge$ $\nexists (c', md') : AC_{\overline{\mathcal{P}}}[(c', md')] = (t^*, \cdot, \cdot)$:</p> <p>37 $ch \leftarrow 0$</p> <p>38 Return</p>
--	---

Figure 14: Game REC for defining security of ACKA.

$x_{\overline{\mathcal{P}^*}}[pc'']$ as in the protocol, and sets $acct \leftarrow g^{x_{\overline{\mathcal{P}^*}}[pc'']}$,
 $acs' \leftarrow g^{x_{\mathcal{P}^*}[pc^*] \cdot x_{\overline{\mathcal{P}^*}}[pc'']}$.

The reduction guesses whether and when the adversary queries the random oracle on acs^* . If it guesses that it does, it exponentiates the corresponding input by $H(acs)^{-1}$ and forwards this to its challenger. Otherwise, it does the same for the adversary's guess of acs^* . Note that the adversary can only distinguish randomly sampled $x_{\mathcal{P}^*}[pc^*]$ from $b \cdot H(acs^*)$ if it indeed queries the random oracle on acs^* . If it does before the reduction guesses so, the reduction will simply forward a random group element to its challenger. Thus security follows.

Correctness of the scheme easily follows, too. \square

THEOREM D.8. *ACKA is history-independent if H is modelled as a programmable random oracle.*

PROOF. We first specify $\mathcal{S}_{\text{ACKA}}$. It initializes $X_A[\cdot][\cdot], S_A[\cdot], X_B[\cdot][\cdot], S_B[\cdot], PC_A[\cdot][\cdot], PC_B[\cdot][\cdot], K_A[\cdot], K_B[\cdot] \leftarrow \perp$. For every input (sim-ct, $\mathcal{P}, t, i, pc, pcp, pcp_{\text{init}}, \ell_{\text{priv}}$):

- Samples random exponent $X_{\mathcal{P}}[t][i]$ and sets $PC_{\mathcal{P}}[t][i] \leftarrow (pc, pcp)$.
- If $S_{\overline{\mathcal{P}}}[pcp] = \perp$: samples random $S_{\overline{\mathcal{P}}}[pcp]$.
- Sets $K_{\mathcal{P}}[pc] \leftarrow g^{S_{\overline{\mathcal{P}}}[pcp] \cdot X_{\mathcal{P}}[t][i]}$
- If $S_{\mathcal{P}}[pc] = \perp$:
 - Sets $S_{\mathcal{P}}[pc] \leftarrow X_{\mathcal{P}}[t][i] \cdot H(K_{\mathcal{P}}[pc])$.

- Sets $pc' \leftarrow pc, t' \leftarrow t + 1, i' \leftarrow \min\{i^* : PC_{\overline{\mathcal{P}}}[t' + 1][i^*][1] = pc\}$,¹¹ $\mathcal{P}' \leftarrow \overline{\mathcal{P}}$
- Runs prog-loop($pc', t', i', \mathcal{P}'$) (defined below).
- Otherwise, it programs $H(K_{\mathcal{P}}[pc]) := S_{\mathcal{P}}[pc]/X_{\mathcal{P}}[t][i]$.
- Finally, it returns $(g^{X_{\mathcal{P}}[t][i]}, (t, pc, pcp, pcp_{\text{init}}, \ell_{\text{priv}}), K_{\mathcal{P}}[pc])$.

prog-loop($pc', t', i', \mathcal{P}'$) is defined as follows:

- While $PC[t'][i'][1] = pc'$:
 - (1) If $X_{\mathcal{P}'}[t'][i'] \neq \perp$: sets $K_{\mathcal{P}'}[PC[t'][i'][0]] \leftarrow g^{S_{\overline{\mathcal{P}'}}[pc'] \cdot X_{\mathcal{P}'}[t'][i']}$; otherwise skips to step 4.
 - (2) Then, if $S_{\mathcal{P}'}[PC[t'][i'][0]] = \perp$: it sets $S_{\mathcal{P}'}[PC[t'][i'][0]] \leftarrow X_{\mathcal{P}'}[t'][i'] \cdot H(K_{\mathcal{P}'}[PC[t'][i'][0]])$ and $pc' \leftarrow PC[t'][i'][0], t' \leftarrow t' + 1, j' \leftarrow \min\{j^* : PC_{\overline{\mathcal{P}'}}[t' + 1][j^*][1] = pc'\}$, $\mathcal{P}' \leftarrow \overline{\mathcal{P}'}$, then runs prog-loop($pc', t', j', \mathcal{P}'$).
 - (3) Otherwise, it programs $H(K_{\mathcal{P}'}[PC[t'][i'][0]]) := S_{\mathcal{P}'}[PC[t'][i'][0]]/X_{\mathcal{P}'}[t'][i']$.
 - (4) Finally, increments $i' \leftarrow i' + 1$.

For every input (sim-st, $\mathcal{P}, t, i, pc, pcp, pcp_{\text{priv}}, pcp_{\text{init}}, \ell_{\text{priv}}, pc2ratch, all\text{-}rcvd$), $\mathcal{S}_{\text{ACKA}}$ executes:

- Sets $x_{\mathcal{P}}[\cdot] \leftarrow \perp$
- For every $pc' \leq pc : \neg all\text{-}rcvd(pc2ratch[pc'])$:
 - If $S_{\mathcal{P}}[pc'] = \perp$: samples random $S_{\mathcal{P}}[pc']$ and sets $pc'' \leftarrow pc', t' \leftarrow pc2ratch[pc'] + 1, i' \leftarrow \min\{i^* :$

¹¹ $i' \leftarrow \infty$ if there is no such i^*

<pre> Proc ACKA.init(k) 00 $pc, pcp, pc_{prv}, pcp_{init}, t, i, \ell_{prv} \leftarrow 0$ 01 $(x_A[0], x_B[0]) \leftarrow k$ 02 $snd-par_A \leftarrow 1; snd-par_B \leftarrow 0$ 03 $st_A \leftarrow (snd-par_A, pc, pcp, pc_{prv},$ $pcp_{init}, t, i, \ell_{prv}, x_A, g^{x_B[0]})$ 04 $st_B \leftarrow (snd-par_B, pc, pcp, pc_{prv},$ $pcp_{init}, t, i, \ell_{prv}, x_B, g^{x_A[0]})$ 05 Return (st_A, st_B) Proc ACKA.snd(st) 06 $(snd-par_P, pc, pcp, pc_{prv},$ $pcp_{init}, t, i, \ell_{prv}, x_P, h) \leftarrow st$ 07 If $t_P \bmod 2 \neq snd-par_P$: 08 $t_P \leftarrow t_P + 1; i \leftarrow 0$ 09 $pcp_{init} \leftarrow pcp$ 10 $x \leftarrow_{\mathcal{S}} \mathcal{X}$ 11 $acs \leftarrow h^x$ 12 $x_P[pc + 1] \leftarrow x \cdot H(acs)$ 13 $pc \leftarrow pc + 1$ 14 $i \leftarrow i + 1$ 15 $st \leftarrow (snd-par_P, pc, pcp, pc_{prv},$ $pcp_{init}, t, i, \ell_{prv}, x_P, h)$ 16 $md \leftarrow (t_P, pc, pcp, pcp_{init}, \ell_{prv})$ 17 Return (st, acs, g^x, md) </pre>	<pre> Proc ACKA.rcv(st, h_c, md) 18 $(snd-par_P, pc, pcp, pc_{prv},$ $pcp_{init}, t, i, \ell_{prv}, x_P, h) \leftarrow st$ 19 $(t, pc', pcp', pcp'_{init}, \ell) \leftarrow md$ 20 If $t > t_P$: 21 $\ell_{prv} \leftarrow i$ 22 end-ratch($x_P, pc_{prv}, pcp'_{init} - 1, \ell$) 23 $pc_{prv} \leftarrow pcp'_{init}$ 24 $t_P \leftarrow t$ 25 $acs \leftarrow h_c^{x_P[pcp']}$ 26 If $pc' > pcp$: 27 $pcp \leftarrow pc'$ 28 $h \leftarrow h_c^{H(acs)}$ 29 $st \leftarrow (snd-par_P, pc, pcp, pc_{prv},$ $pcp_{init}, t, i, \ell_{prv}, x_P, h)$ 30 Return ($st, acs, (t_P, \ell)$) </pre>
---	--

Figure 15: Construction of ACKA. `end-ratch($X, strt, end, \ell$)` remembers ℓ internally such that once $X[y]$ for $y \in [strt, end]$ have collectively been used to receive ℓ keys, they are each deleted. The construction includes in metadata md components which will appear duplicated in the metadata of our PR construction. We thus remove these duplicates in PR in our ultimate implementation.

- $PC_{\bar{P}}[t'][i^*] = pc'$, and $P' \leftarrow \bar{P}$, then runs `prog-loop(pc'', t', i', P')`.
- Sets $x_P[pc'] \leftarrow S_P[pc']$
 - If $S_{\bar{P}}[pcp] = \perp$:
 - Samples random $S_{\bar{P}}[pcp]$ and sets $t' \leftarrow t + 1$ if P received in epoch t , or $t' \leftarrow t$ if P sent in epoch t , $i' \leftarrow \min\{i^* : PC_P[t'][i^*] = pcp\}$, and $P' \leftarrow P$, then runs `prog-loop(pc', t', i', P')`.
 - Returns $(snd-par_P, pc, pcp, pc_{prv}, pcp_{init}, t, i, \ell_{prv}, x_P, g^{S_{\bar{P}}[pcp]})$

Now that we have specified \mathcal{S}_{ACKA} , we argue that it provides history-independence. Indeed, we can easily see that the distributions `sim-info` and `exec-info` are computationally-indistinguishable. This is because in both, ciphertexts are always sampled randomly. Also, states are always computed in an identical manner, except for the case in which \mathcal{S}_{ACKA} does not have sufficient information to compute one and instead samples it randomly. H is modelled as a programmable random oracle, so this is still an identical distribution because (i) if there is always insufficient information to compute it; it is an output of H , so it is also randomly distributed in `exec-info`, and (ii) if at some point there is sufficient information to compute it; we can program H , so we still get an identical view as in `exec-info`. (There is only some negligible probability that the distinguisher succeeds due to the fact that (with some negligible

probability), the random oracle can be queried on values before they are programmed) \square

E PUBLIC-KEY DOUBLE RATCHET FORMAL DETAILS

We now provide the formal security definition for the Public-Key Double Ratchet (PR) and the security analysis for our construction from Section 3.

E.1 Security

In Figure 16, we give a formal game-based security definition for the PR. The game first initializes the parties with some shared key material (line 08).

The adversary can query oracle `Snd(P)` for a send from party P. The oracle first checks if P should start a new ratchet, and accounts for it if so (lines 12-13). It then invokes `PR.snd` (line 14), from which it returns ciphertext and metadata (c, md).

The adversary can also query oracle `Reveal(t, i)`, which reveals the key generated for index i of ratchet t . The oracle first checks that the corresponding ciphertext has been sent already (line 44) and that (t, i) has not already been challenged (line 46). It then returns the key.

The adversary can query oracle `Rcv(P, c, md)` which instructs party P to receive ciphertext (c, md). The oracle first checks that

this ciphertext is indeed in-transit (line 21). Then, it invokes PR.rcv on (c, md) (line 22). The game then checks that the output keys rak and k , as well as index i within ratchet t for this ciphertext are correct (line 25). Finally, the oracle checks for each old receiving ratchet whether all ciphertexts for that ratchet have been received; if so, it sets that the PCS epochs for this ratchet are no longer challenged (lines 29-30).

The adversary can also query oracle $\text{Expose}(P)$ which exposes the state of party P . First, the oracle checks that none of P 's PCS epochs are currently being challenged and there are no challenge ciphertexts in-transit (33). It then sets all unreceived double ratchet keys as exposed (lines 34-39). It then sets as exposed all PCS epochs that are used for decrypting ciphertexts in ratchets where there are still in-transit ciphertexts (lines 41-42).

Finally, the adversary can query oracle $\text{Chall}(t, i)$ which gives it a real-or-random key corresponding to the i -th index of ratchet t . For this, it first checks that (i) the corresponding ciphertext has already been sent, (ii) the key has not already been revealed, and (iii) either the corresponding pcs epoch or double ratchet key is secure (lines 50-57). Then, if security relies on the pcs epoch, it sets as such for the ratchet (lines 58-59). Finally, depending on challenge bit b , it returns either random challenge key or the actual key computed for the (t, i) ciphertext (lines 61-64).

The goal of the adversary is to successfully guess the challenge bit b .

Definition E.1. A PR scheme is *secure* if for any adversary \mathcal{A} in the security game of Figure 16, the probability it guesses challenge bit b is at most $1/2 + \text{negligible}$.

E.2 History-Independence

For our Mesh Messaging construction, we require an additional property from our PR construction that we name *history-independence*. Intuitively, we require that future ciphertexts and secret values in future states look pseudorandom even given current ciphertexts and secret values in current states. Now we give a more formal definition.

First, let $\text{seq} = (\text{op}_1, \text{op}_2, \dots, \text{op}_\ell)$ be some sequence of PR operations. We will take advantage of the bookkeeping in the security game of Figure 16, and thus will specify this sequence in terms of oracle calls. Namely, each op_i is of the form $\text{Snd}(P)$ or $\text{Rcv}(P, c, md)$, for $P \in \{A, B\}$. Now consider the execution of some PR protocol on seq , where first lines 00-09 of Figure 16 are executed:

- For each op_i of the form $(\text{PR.snd}, P)$, the code of oracle call $\text{Snd}(P)$ is executed.
- For each op_i of the form $(\text{PR.rcv}, P, c, md)$, the code of oracle call $\text{Rcv}(P, c, md)$ is executed.

Now, let l be a counter initialized to 0, and \mathcal{S}_{PR} be some stateful algorithm that is run given information from the execution of seq through two possible inputs:

- On input $(\text{sim-ct}, P_s, P_r, t, i, pc, pcp, pcp_{\text{init}}, \ell_{\text{prv}})$, \mathcal{S}_{PR} simulates (i) ciphertext c with (ii) associated data h and (iii) corresponding message key k_{mk} from P_s to P_r in ratchet t at index i with P_s 's PCS epoch pc and P_r 's PCS epoch pcp , and (iv) ratchet key rak output by P_r 's most recent ratchet. Additionally pcp_{init} is P_s 's view of P_r 's PCS epoch

at the start of ratchet t and ℓ_{prv} is the number of ciphertexts sent in ratchet $t - 2$. Let $\text{sim-info}[l] \leftarrow (c, h, k_{\text{mk}}, rak)$ and $\text{exec-info}[l]$ be the actual ciphertext, associated data, message key, and ratchet key output by the $\text{Snd}(P)$ oracle corresponding to the operation $\text{op}_j \in \text{seq}$ at the beginning of which $t_p = t$ and $i_p = i$. Increment $l = l + 1$.

- Let (t, i, pc, afs) be the respective values of $(t_p, i_p, PC_P[\bar{P}], FS_P)$ at some point in the execution of seq . Also let i be the number of times a ciphertext was sent in t_p at this point in the execution, and $pc2ratch$ be a dictionary mapping pc values to the ratchets t in which they were initiated at this point in the execution (i.e., the ratchet t in which $PC_P[P]$ was set to pc). Let all-rcvd be an array storing 1 for index t if all messages of that ratchet have been received by P ; 0 otherwise. Finally, let $pc_{\text{prv}}, pcp_{\text{init}}, \ell_{\text{prv}}$ be as above. On input $(\text{sim-st}, P, t, i, pc, pcp, pc_{\text{prv}}, pcp_{\text{init}}, \ell_{\text{prv}}, afs, pc2ratch, \text{all-rcvd})$, \mathcal{S}_{PR} outputs (i) a simulated state st for P with these state variables and (ii) array of ratchet keys rak generated with ratchets $t, t + 1$ and all previous t' for which $\text{all-rcvd}[t'] = 0$. Let $\text{sim-info}[l] \leftarrow (st, rak)$ and $\text{exec-info}[l]$ be the actual state of party P along with the corresponding ratchet key array rak at this point in the execution of seq . Increment $l = l + 1$.

Definition E.2. PR is *history-independent* if for every seq , the distributions sim-info and exec-info are computationally indistinguishable (over the randomness sampled by the algorithms of PR and \mathcal{S}_{PR}).

E.3 PR Full Construction and Comparison to ACD19

We provide the full version of our PR construction in Figure 17. We simply make the states explicit and write out subroutine find-and-del-drk in full.

We here also compare the added public-key layer of security to the original DR which our PR provides to that of [ACD19]. First, their construction provides a public-key layer of authenticity, via digital signatures, which we do not attempt to provide at all. Their construction also provides a public-key layer of confidentiality (like ours). Yet, their confidentiality guarantees are weaker than ours. Their construction only has a user publish a new public key for each new ratchet, while our construction has a user publish a new public key for *each new message sent*, which is clearly more secure.

E.4 PR Security Analysis

THEOREM E.3. *If CKA is a secure CKA, ACKA is a secure ACKA, H_1 and H_3 are modelled as random oracles, and H_2 is a PRG, then PR in Figure 2 is secure.*

For simplicity, we first consider those adversaries that only query the challenge oracle once. Security against those adversaries which query the challenge oracle an arbitrary polynomially-many times follows by a standard hybrid argument.

Now, we consider two types of adversaries:

- (1) Adversaries who query the challenge oracle for ratchet and index t^*, i^* such that $t^* \notin \text{TXP}[t^*]$
- (2) All other adversaries.

<p>Game $\text{IND}_{\text{PR}}^b(\mathcal{A})$</p> <p>00 $AC_A[\cdot], AC_B[\cdot], K[\cdot][\cdot][\cdot] \leftarrow \perp$</p> <p>01 $CH, REV \leftarrow \emptyset$</p> <p>02 $PC_A[\cdot], PC_B[\cdot], PCCH[\cdot] \leftarrow 0$</p> <p>03 $PC2T_A[\cdot], PC2T_B[\cdot] \leftarrow 0$</p> <p>04 $t_A, t_B \leftarrow 0; i_A, i_B \leftarrow 0;$</p> <p>05 $FS_A[\cdot], FS_B[\cdot] \leftarrow [\infty]$</p> <p>06 $PXP_A, PXP_B, TXP[\cdot] \leftarrow \emptyset$</p> <p>07 $k \leftarrow_{\mathcal{S}} \mathcal{K}$</p> <p>08 $(st_A, st_B) \leftarrow \text{PR.init}(k)$</p> <p>09 $snd\text{-}par_A \leftarrow 1; snd\text{-}par_B \leftarrow 0$</p> <p>10 $b' \leftarrow_{\mathcal{S}} \mathcal{A}$</p> <p>11 Stop with b'</p> <p>Oracle Snd(P)</p> <p>12 If $t_P \bmod 2 \neq snd\text{-}par_P$:</p> <p>13 $t_P \leftarrow t_P + 1; i_P \leftarrow 0$</p> <p>14 $(stp, rak, k, c, md) \leftarrow_{\mathcal{S}} \text{PR.snd}(stp)$</p> <p>15 $AC_P[(c, md)] \leftarrow (t_P, i_P, PC_P[P], rak, k)$</p> <p>16 $K[t_P][i_P][PC_P[\bar{P}]] \leftarrow (rak, k)$</p> <p>17 $PC2T_P[PC_P[P]] \leftarrow t_P$</p> <p>18 $PC_P[P] \leftarrow PC_P[P] + 1$</p> <p>19 $i_P \leftarrow i_P + 1$</p> <p>20 Return (c, md)</p> <p>Oracle Rcv(P, c, md)</p> <p>21 Require $AC_{\bar{P}}[(c, md)] \neq \perp$</p> <p>22 $(stp, rak, k, (t, i)) \leftarrow_{\mathcal{S}} \text{PR.rcv}(stp, c, md)$</p> <p>23 $t_P \leftarrow \max\{t_P, t\}$</p> <p>24 $(t', i', pc', rak', k') \leftarrow AC_{\bar{P}}[(c, md)]$</p> <p>25 If $(rak', k', t', i') \neq (rak, k, t, i)$: WIN</p> <p>26 $FS_P[t] \leftarrow FS_P[t] \setminus \{i\}$</p> <p>27 $PC_P[\bar{P}] \leftarrow \max\{PC_P[\bar{P}], pc'\}$</p> <p>28 $AC_{\bar{P}}[(c, md)] \leftarrow \perp$</p> <p>29 For all $t \leq t_P - 2 : t \bmod 2 \neq snd\text{-}par_P$:</p> <p>30 If $\nexists(c', md') : AC_{\bar{P}}[(c', md')] = (t, \cdot, \cdot, \cdot)$:</p> <p>31 $PCCH[t] \leftarrow 0$</p> <p>32 Return</p>	<p>Oracle Expose(P)</p> <p>33 Require $(\forall t : t \bmod 2 \neq snd\text{-}par_P :$</p> <p style="padding-left: 20px;">$PCCH[t] = 0 \wedge \nexists(t, i) \in CH :$</p> <p style="padding-left: 20px;">$\exists(c, md) : AC_{\bar{P}}[(c, md)] = (t, i, \cdot, \cdot)$)</p> <p>34 For all $t : t \leq t_P \wedge t \bmod 2 \neq snd\text{-}par_P$:</p> <p>35 $TXP[t] \xleftarrow{\cup} FS_P[t]$</p> <p>36 $t_{\text{last-rcv}} \leftarrow t_P$</p> <p>37 If $t_P \bmod 2 = snd\text{-}par_P$:</p> <p>38 $TXP[t_P] \xleftarrow{\cup} [\infty] \setminus [i_P]$</p> <p>39 $TXP[t_P + 1] \leftarrow [\infty]$</p> <p>40 $t_{\text{last-rcv}} \leftarrow t_P - 1$</p> <p>41 For all $pc : PC2T_P[pc] + 1 = t_{\text{last-rcv}} \vee$</p> <p style="padding-left: 20px;">$\exists(c, md) : AC_{\bar{P}}[(c, md)] = (PC2T_P[pc] + 1, \cdot, \cdot, \cdot) :$</p> <p>42 $PXP_P \xleftarrow{\cup} \{pc\}$</p> <p>43 Return stp</p> <p>Oracle Reveal(t, i)</p> <p>44 $P \leftarrow P' : t \bmod 2 = snd\text{-}par_{P'}$</p> <p>45 Require $t_P \geq t \wedge i_P > i$</p> <p>46 Require $(t, i) \notin CH$</p> <p>47 $REV \xleftarrow{\cup} \{(t, i)\}$</p> <p>48 $pc \leftarrow pc' : K[t][i][pc'] \neq \perp$</p> <p>49 Return $K[t][i][pc]$</p> <p>Oracle Chall(t, i)</p> <p>50 $P \leftarrow P' : t \bmod 2 = snd\text{-}par_{P'}$</p> <p>51 Require $t_P \geq t \wedge i_P > i$</p> <p>52 Require $(t, i) \notin REV$</p> <p>53 $pc \leftarrow pc' : K[t][i][pc'] \neq \perp$</p> <p>54 If $i = 0$:</p> <p>55 Require $i \notin TXP[t]$</p> <p>56 Else:</p> <p>57 Require $pc \notin PXP_{\bar{P}} \vee i \notin TXP[t]$</p> <p>58 If $i \in TXP[t] \wedge (t_{\bar{P}} < t + 2 \vee \exists(c, md) :$</p> <p style="padding-left: 20px;">$AC_P[(c, md)] = (t, \cdot, \cdot, \cdot)$):</p> <p>59 $PCCH[t] \leftarrow 1$</p> <p>60 $CH \xleftarrow{\cup} \{(t, i)\}$</p> <p>61 If $b = 1$:</p> <p>62 $k \leftarrow_{\mathcal{S}} \mathcal{K}; rak \leftarrow \perp$</p> <p>63 If $i = 0$: $rak \leftarrow_{\mathcal{S}} \mathcal{RAK}$</p> <p>64 Else: $(rak, k) \leftarrow K[t][i][pc]$</p> <p>65 Return (rak, k)</p>
---	--

Figure 16: Game IND for defining security of PKDR.

First, we handle Type 1 adversaries.

LEMMA E.4. *If CKA is a secure CKA, H_1, H_3 are random oracles, and H_2 is a PRG, then PR in Figure 2 is secure against Type 1 adversaries.*

We first prove security with respect to a weakened PR security notion. Namely, at the beginning of the game, the adversary specifies P, t^*, i^*, t_L^P , and $t_L^{\bar{P}}$, where P is the party that creates the challenge key, t^*, i^* are the ratchet and index within that ratchet, respectively, of the challenge key, and $t_L^P, t_L^{\bar{P}}$ are the last ratchets

in which P, \bar{P} (according to their view of the latest ratchet) will be corrupted before t^* . The Chall oracle is augmented to have an additional “Require” statement that mandates queried $(t, i) = (t^*, i^*)$. The Reveal oracle is correspondingly augmented to have an additional “Require” statement that mandates queried $(t, i) \neq (t^*, i^*)$. Also, if for any query to Expose(P), $t_P \in \{t_L^P + 1, \dots, t^* - 1\}$, the attacker immediately loses the game. Similarly, if for any query to Expose(\bar{P}), $t_{\bar{P}} \in \{t_L^{\bar{P}} + 1, \dots, t^* - 2\}$, the attacker immediately loses the game.

<pre> Proc PR.init(k) 00 ($rk, ck[0], k_{ACKA}, k_{CKA}$) $\leftarrow k$ 01 $t, i, i_{rcv}, \ell_{prv} \leftarrow 0$; $cst, ck[\cdot], drk[\cdot][\cdot] \leftarrow \perp$ 02 $turn \leftarrow A$ 03 (cst_A, cst_B) \leftarrow CKA.init(k_{CKA}) 04 ($acst_A, acst_B$) \leftarrow ACKA.init(k_{ACKA}) 05 $st_A \leftarrow (rk, cst_A, cct, acst_A, ck, drk, turn, t, i, i_{rcv}, \ell_{prv})$ 06 $st_B \leftarrow (rk, cst_B, cct, acst_B, ck, drk, turn, t, i, i_{rcv}, \ell_{prv})$ Proc PR.snd(st) 07 ($rk, cst, cct, acst, ck, drk, turn, tp, i, i_{rcv}, \ell_{prv}$) $\leftarrow st$ 08 $rak \leftarrow \perp$ 09 If $turn = P$: 10 (cs, cct) \leftarrow_{\S} CKA.snd(cst) 11 ($rk, ck[tp+1], rak$) $\leftarrow H_1(rk, cs)$ 12 $tp \leftarrow tp+1$; $i \leftarrow 0$; $turn \leftarrow \bar{P}$ 13 ($ck[tp], drk$) $\leftarrow H_2(ck[tp])$ 14 ($acs, acct, md$) \leftarrow_{\S} ACKA.snd($acst$) 15 $h \leftarrow (md, tp, i, cct, \ell_{prv})$ 16 $k_{mk} \leftarrow H_3(acs, drk, (h, acct))$ 17 $i \leftarrow i+1$ 18 $st \leftarrow (rk, cst, cct, acst, ck, drk, turn, tp, i, i_{rcv}, \ell_{prv})$ 19 Return ($st, rak, k_{mk}, acct, h, (tp, i-1)$) </pre>	<pre> Proc PR.rcv($st, acct, h$) 20 ($rk, cst, cct, acst, ck, drk, turn, tp, i, i_{rcv}, \ell_{prv}$) $\leftarrow st$ 21 (md, t, i, cct', ℓ) $\leftarrow h$ 22 $rak \leftarrow \perp$ 23 If $t > tp$: 24 $ck[tp] \leftarrow \perp$; $\ell_{prv} \leftarrow i$; $i_{rcv} \leftarrow 0$ 25 $tp \leftarrow t$; $turn \leftarrow P$ 26 end-ratch($ck[tp-2], \ell$) 27 (cst, cs) \leftarrow CKA.rcv(cst, cct') 28 ($rk, ck[t], rak$) $\leftarrow H_1(rk, cs)$ 29 $k \leftarrow drk[t][i]$ 30 $drk[t][i] \leftarrow \perp$ 31 If $k = \perp \wedge (t \leq tp-2 \vee i < i_{rcv})$: 32 Return ($st, \perp$) 33 Else if $k = \perp \wedge t > tp-2 \wedge i \geq i_{rcv}$: 34 While $i_{rcv} < i$: 35 ($ck[t], drk[t][i_{rcv}]$) $\leftarrow H_2(ck[t])$ 36 $i_{rcv} \leftarrow i_{rcv} + 1$ 37 ($ck[t], k$) $\leftarrow H_2(ck[t])$ 38 $i_{rcv} \leftarrow i + 1$ 39 ($acst, acs$) \leftarrow ACKA.rcv($acst, acct, md$) 40 $k_{mk} \leftarrow H_3(acs, drk, (h, acct))$ 41 $st \leftarrow (rk, cst, cct, acst, ck, drk, turn, tp, i, i_{rcv}, \ell_{prv})$ 42 Return ($st, rak, k_{mk}, (t, i)$) </pre>
--	---

Figure 17: Full version of PR construction from Section 3.

First we show that if a PR is secure with respect to the weakened notion, then it is secure with respect to our standard notion (with some security loss).

LEMMA E.5. *If PR is secure with respect to the weakened security notion described above, then it is secure with respect to the game of Figure 16, both against a Type 1 adversary.*

PROOF. The reduction will first guess $t_L^P, t_L^{\bar{P}}, t^*, i^*$ such that $t_L^P < t^*$ and $t_L^{\bar{P}} < t^* - 1$. It can easily be seen that the reduction can make these guesses with only non-negligible security loss. Indeed, if the adversary queries Chall for $(tp, ip) \neq (t^*, i^*)$, or Reveal for (t^*, i^*) , the reduction simply sends a random guess bit b to its challenger. Moreover, if the adversary queries Expose(P) for $tp > t_L^P$ or Expose(\bar{P}) for $tp > t_L^{\bar{P}}$, the reduction does the same.

For all other queries from the adversary, the reduction simply forwards them to its challenger, and outputs the response back to the adversary. This can easily be seen to simulate the stronger game properly. The reduction then simply forwards the guess bit b' from the adversary to its challenger. \square

Now, we show that PR is secure with respect to the weakened notion against Type 1 adversaries via a sequence of hybrids.

Hybrid \mathcal{H}_0 . This is the original weakened security game with challenge bit $b = 0$.

Hybrid \mathcal{H}_1 . Denote by $(rk_1, ck[1]), (rk_2, ck[2]), \dots$ the outputs of H_1 for each ratchet, and let $t_L \leftarrow \max\{t_L^P, t_L^{\bar{P}}\}$. If the sender of t_L is indeed corrupted, then let $t_{\text{heal}} \leftarrow t_L + 2$; otherwise, let $t_{\text{heal}} \leftarrow t_L + 1$.

\mathcal{H}_1 works as \mathcal{H}_0 , except that $(rk_t, ck[t])$ are replaced with uniformly random values for all $t_{\text{heal}} \leq t \leq t^*$

LEMMA E.6. *If CKA is a secure CKA and H_1 is modelled as a random oracle, then hybrids \mathcal{H}_0 and \mathcal{H}_1 are indistinguishable.*

PROOF. We proceed by a simple reduction to the security of CKA. The reduction initializes state as normal, except that it implicitly initializes the state of CKA via the challenger, with challenge ratchet t_{heal} . To simulate all Snd queries except for the first one in ratchet t_{heal} , and all Rcv queries, the reduction performs all steps as normal, except it queries the respective oracles of the CKA game and uses its outputs accordingly. For the first Snd query in ratchet t_{heal} , the reduction performs all steps as normal, except it queries its CKA Chall oracle and uses its output c along with its own randomly sampled $(rk_{t_{\text{heal}}}, ck[t_{\text{heal}}])$. For Reveal queries, the reduction outputs the key it computed for ratchet t , index i . For the adversary's Chall query, the reduction outputs the key it computed for ratchet t^* , index i^* . To simulate Expose queries, the reduction will query the Expose oracle of the CKA game to obtain cst and return the entire state as normal.

It is easy to see that the reduction simulates both \mathcal{H}_0 and \mathcal{H}_1 correctly and the adversary can only attempt to successfully find out otherwise if it queries the random oracle on cs^* , the challenge CKA key. Indeed, for corruptions, due to t_{heal} 's relation to t_L^P and $t_L^{\bar{P}}$, corruptions will be allowed by the CKA game. So the reduction guesses which random oracle query corresponds to the above, and forwards its input to the challenger. \square

Hybrid \mathcal{H}_3 . \mathcal{H}_3 works as \mathcal{H}_2 , except k_{mk} output by \mathcal{H}_3 for index i^* of ratchet t^* is replaced by uniformly random.

LEMMA E.7. *If \mathcal{H}_2 is a PRG and \mathcal{H}_3 is modelled as a random oracle, then Hybrids \mathcal{H}_2 and \mathcal{H}_3 are indistinguishable.*

PROOF. Denote by $(ck_1, k_1), (ck_2, k_2), \dots$ the outputs of \mathcal{H}_2 for all indices i of ratchet t^* . This lemma follows by a straightforward hybrid argument to the PRG security of \mathcal{H}_2 , for each $i \leq i^*$. Namely, for each such i we use the fact that ck_{i-1} is uniformly random to argue that (ck_i, k_i) is indistinguishable from uniformly random. Based on k_{i^*} being uniformly random, so too is k_{mk} . \square

PROOF OF LEMMA E.4. Immediately follows from Lemmas E.5 through E.7. \square

Now, we handle Type 2 adversaries.

LEMMA E.8. *If ACKA is a secure ACKA and \mathcal{H}_3 is modelled as a random oracle, then PR in Figure 2 is secure against Type 2 adversaries.*

PROOF. This proof proceeds by a straight forward reduction to the security of ACKA. First, the reduction guesses for which party, ratchet, and pcs epoch the adversary will query the challenge oracle. It can easily be seen that the reduction can make this guess with only non-negligible security loss. It then initializes the ACKA challenger with this guess.

Now, to simulate the PR security game, the reduction first initializes all parts of the state of both parties using freshly sampled randomness, except for ACKA, which is implicitly initialized by the challenger, with corresponding challenge party, ratchet, and pcs epoch. To simulate Snd queries except for the challenge one, and Rcv queries, the reduction will perform all steps as normal, except it will query the respective oracles of the ACKA game, and use the outputs $acs, acct, md$ and $acs, (t, \ell)$, respectively. To simulate Expose queries, the reduction will query the Expose oracle of the ACKA game to obtain $acst$ and return the entire state as normal. Finally, for the Snd query corresponding to the challenge, the reduction will perform all steps as normal, except it will query the Chall oracle of the ACKA game, and use the output $acct, md$, along with its own randomly sampled k_{mk} . For Reveal queries, the reduction outputs the key it computed for ratchet t , index i . For the adversary's Chall query, the reduction outputs the key it computes for ratchet t^* , index i^* .

The adversary can only distinguish between the actual security game and the simulation if it queries the random oracle on acs^* . So, the reduction guesses which query corresponds to this guess, and forwards this input to its challenger. It is fairly easy to see that this is a proper simulation of the PR security game otherwise. Indeed, all outputs of Snd, Rcv, Reveal are easily seen to be proper. Furthermore, because of our assumption that the adversary is Type 2, and thus when it queries the challenge oracle, challenge epoch $pc^* \notin PXP_{\bar{P}}$, the same will hold for $PC_P[\bar{P}] = pc^*$ in the reduction's query to the ACKA game. For similar reasons, all Expose queries are properly simulated. \square

PROOF OF THEOREM E.3. This easily follows from Lemmas E.4 and E.8. \square

THEOREM E.9. *PR of Figure 2 is history-independent if ACKA is history-independent, CKA is history-independent, and $\mathcal{H}_1, \mathcal{H}_2$ are modelled as programmable random oracles.*

PROOF. In Figure 18, we first define a history-independence simulator S_{PR} for the PR in Figure 2.

The proof follows in a series of hybrids. Hybrid \mathcal{H}_0 is the view of the real execution. Hybrid \mathcal{H}_1 is the real world, except ACKA ciphertexts and states are produced by $\mathcal{S}_{\text{ACKA}}$. It is easy to see that these two hybrids are computationally indistinguishable, from the fact that ACKA is history-independent.

Finally, hybrid \mathcal{H}_2 is the view of the simulated execution produced by S_{PR} . To argue that \mathcal{H}_1 and \mathcal{H}_2 we reduce to the history-independence of CKA, with access to programmable random oracles modelling $\mathcal{H}_1, \mathcal{H}_2$. This reduction uses outputs from the CKA history-independence game so that CKA ciphertexts, states, and keys are output according to either a real CKA execution, or one produced by the simulator \mathcal{S}_{CKA} . Now, PR ciphertexts are completely determined by the ACKA and CKA (and deterministic information known by the reduction), the latter of which, as above, will determine which hybrid we are in. We next argue that all components of states, besides those produced by the CKA, are distributed the same in both \mathcal{H}_1 and \mathcal{H}_2 due to the programmable random oracles. Indeed, when sufficient information is known, all root keys, chain keys, and message keys are computed exactly the same in both hybrids. When insufficient information is known, the reduction picks random keys. This is a proper simulation, since if there is never sufficient information, the keys are all outputs of random oracles, so they are properly simulated. If sufficient information does become known, we simply program the random oracle to make things consistent. Moreover, the distinguisher is poly-time, so they will only query the random oracles on these values before they are programmed with negligible probability.

Thus \mathcal{H}_1 and \mathcal{H}_2 are computationally indistinguishable, so we conclude that \mathcal{H}_0 and \mathcal{H}_2 are computationally indistinguishable, and thus PR is history-independent. \square

F STANDARD DEFINITIONS

Definition F.1 (CDH Problem). The advantage of an adversary against the *Computational Diffie-Hellman* problem in group G is

$$\text{Adv}_G^{\text{cdh}} := \Pr[\mathcal{A}(G, g^a, g^b) \rightarrow_{\$} g^{ab} \mid g^a, g^b \leftarrow_{\$} G].$$

The CDH problem is considered hard if the advantage is negligible.

Definition F.2 (Secure NIKI). The advantage of an adversary against a *Non-Interactive Key Exchange* protocol $\text{NK} = (\text{NK.gen}, \text{NK.eval})$ is

$$\begin{aligned} \text{Adv}_{\text{NK}}^{\text{ind}} := & \left| \Pr[\mathcal{A}(pk_A, pk_B, k) \rightarrow_{\$} 1 \mid (sk_A, pk_A) \leftarrow_{\$} \text{NK.gen}, \right. \\ & (sk_B, pk_B) \leftarrow_{\$} \text{NK.gen}, k \leftarrow \text{NK.eval}(sk_A, pk_B)] \\ & \left. - \Pr[\mathcal{A}(pk_A, pk_B, k) \rightarrow_{\$} 1 \mid (sk_A, pk_A) \leftarrow_{\$} \text{NK.gen}, \right. \right. \\ & \left. \left. (sk_B, pk_B) \leftarrow_{\$} \text{NK.gen}, k \leftarrow_{\$} \mathcal{K}_{\text{NK}}] \right|. \end{aligned}$$

A NIKI construction is considered secure if the advantage is negligible.

Definition F.3 (Secure AEAD). Let $\text{AE.enc}(k, \cdot, \cdot)$ be an oracle that on input of variables (m, ad) outputs $\text{AE.enc}(k, m, ad)$ and $\$(\cdot, \cdot)$ an oracle that on input of variables (m, ad) outputs a random string $c \in \{0, 1\}^{|\text{AE.enc}(k, m, ad)|}$. The advantage of an adversary against an *Authenticated Encryption scheme with Associated Data* $\text{AE} = (\text{AE.enc}, \text{AE.dec})$ is

$$\text{Adv}_{\text{AE}}^{\text{ind\$-cpa}} := |\Pr[\mathcal{A}^{\text{AE.enc}(k, \cdot, \cdot)} \rightarrow_{\$} 1 \mid k \leftarrow_{\$} \mathcal{K}] - \Pr[\mathcal{A}^{\$(\cdot, \cdot)} \rightarrow_{\$} 1]|.$$

An AEAD construction is considered (passively) secure if the advantage is negligible.

<p>S_{PR} Init</p> <pre> 00 $cct[\cdot], cs[\cdot], rak[\cdot], ck[\cdot][\cdot], drk[\cdot][\cdot], rk[\cdot] \leftarrow \perp$ 01 $snd-par_A \leftarrow 1; snd-par_B \leftarrow 0$ 02 $st \leftarrow (rk, cct, cs, rak, ck, drk, snd-par_A, snd-par_B)$ 03 Initialize $\mathcal{S}_{CKA}, \mathcal{S}_{ACKA}$ 04 Return st On input $(sim-ct, P_s, P_r, t, i, pc, pcp, pcp_{init}, \ell_{priv})$ 05 $(rk, cct, cs, rak, ck, drk, snd-par_A, snd-par_B) \leftarrow st$ 06 If $rak[t-1] = \perp$: 07 $rak[t-1] \leftarrow_{\mathcal{S}} \mathcal{RAK}$ 08 If $rak[t-1] = \perp$: $rak[t-1] \leftarrow_{\mathcal{S}} \mathcal{K}$ 09 root-prog(t) 10 If $cct[t] = \perp$: 11 $(cs[t], cct[t]) \leftarrow \mathcal{S}_{CKA}(sim-ct, t)$ 12 If $rak[t-1] \neq \perp$: root-prog(t) 13 If $drk[t][i] = \perp$: $drk[t][i] \leftarrow_{\mathcal{S}} \mathcal{K}$ 14 $(acct, md, acs) \leftarrow \mathcal{S}_{ACKA}(sim-ct, P_s, t, i, pc, pcp, pcp_{init}, \ell_{priv})$ 15 $h \leftarrow (md, t, i, cct[t], \ell_{priv})$ 16 $k_{mk} \leftarrow H_3(ac, drk[t][i], (h, acct))$ 17 Return $(acct, h, k_{mk}, rak[t-1])$ Proc root-prog(t') 18 While $rak[t'] = \perp \wedge cs[t'] \neq \perp$: 19 $(rk[t'], ck[t'][0], rak[t']) \leftarrow H_1(rak[t'-1], cs[t'])$ 20 chain-prog($t', 1$) 21 $t' \leftarrow t' + 1$ 22 If $rak[t'] \neq \perp \wedge cs[t'] \neq \perp$: 23 Program $H_1(rak[t'-1], cs[t']) := (rk[t'], ck[t'][0], rak[t'])$ Proc chain-prog(t', i') 24 While $ck[t'][i'] = \perp \wedge \exists i^* \geq i'$: 25 $(ck[t'][i^*] \neq \perp \vee drk[t'][i^*] \neq \perp)$: 26 $(ck[t][i'], drk[t][i']) \leftarrow H_2(ck[t][i'-1])$ 27 $i' \leftarrow i' + 1$ 28 If $(ck[t'][i'] \vee drk[t'][i']) \neq \perp$: 29 Program $H_2(ck[t'][i'-1]) := (ck[t][i'], drk[t][i'])$ Proc chain-comp(t', i', l') 29 While $l' < i'$: 30 $(ck[t'][l'+1], drk[t][l'+1]) \leftarrow H_2(ck[t][l'])$ 31 $l' \leftarrow l' + 1$ </pre>	<p>On input $(sim-st, P, t, i, pc, pcp, pc_{priv}, pcp_{init}, \ell_{priv}, afs, pc2ratch, all-rcvd)$</p> <pre> 32 $(rk, cct, cs, rak, ck, drk, snd-par_A, snd-par_B) \leftarrow st$ 33 $(cst, cs[t]) \leftarrow \mathcal{S}_{CKA}(sim-st, P, t)$ 34 $t' \leftarrow (t \bmod 2 = snd-par_P ? t : t-1)$ 35 If $cct[t'] = \perp$: 36 $(cs[t'], cct[t']) \leftarrow \mathcal{S}_{CKA}(sim-ct, t')$ 37 If $rak[t'-1] \neq \perp$: root-prog(t') 38 $cct' \leftarrow cct[t']$ 39 $rak'[\cdot] \leftarrow \perp$ 40 For $t^* \in \{t, t+1\}$: 41 If $rak[t^*] = \perp$: 42 $rak[t^*] \leftarrow_{\mathcal{S}} \mathcal{RAK}$ 43 If $rak[t^*] = \perp$: $rak[t^*] \leftarrow_{\mathcal{S}} \mathcal{K}$ 44 root-prog($t^* + 1$) 45 $rak'[t^*] \leftarrow rak[t^*]$ 46 $ck'[\cdot] \leftarrow \perp$ 47 $t' \leftarrow (t \bmod 2 = snd-par_P ? t-1 : t)$ 48 $j^{**} \leftarrow \max\{j : j \notin afs[t']\}$ 49 $l^* \leftarrow \max\{l \leq j^{**} : ck[t'][l] \neq \perp\}^a$ 50 If $l^* \geq 0$: chain-comp(t', l^*, j^{**}) 51 Else: 52 $ck[t'][j^*] \leftarrow_{\mathcal{S}} \mathcal{K}$ 53 chain-prog($t', j^* + 1$) 54 If $t \bmod 2 = snd-par_P$: 55 $l^* \leftarrow \max\{l \leq i : ck[t][l] \neq \perp\}$ 56 If $l^* \geq 0$: chain-comp(t, i, l^*) 57 Else: 58 $ck[t][i] \leftarrow_{\mathcal{S}} \mathcal{K}$ 59 chain-prog($t, i + 1$) 60 $ck'[t-1] \leftarrow ck[t][j^*]; ck'[t] \leftarrow ck[t][i]$ 61 Else: $ck'[t] \leftarrow ck[t][j^*]$ 62 $drk'[\cdot][\cdot] \leftarrow \perp$ 63 For $t^* : t^* \leq t \wedge t^* \bmod 2 \neq snd-par_P$: 64 If $all-rcvd[t^*] = 1$: continue 65 If $rak[t^*] = \perp$: 66 $rak[t^*] \leftarrow_{\mathcal{S}} \mathcal{RAK}$ 67 If $rak[t^*] = \perp$: $rak[t^*] \leftarrow_{\mathcal{S}} \mathcal{K}$ 68 root-prog(t^*) 69 $rak'[t^*] \leftarrow rak[t^*]$ 70 $j^* \leftarrow \max\{j : j \notin afs[t^*]\}$ 71 $l^* \leftarrow \max\{l < j^* : ck[t^*][l] \neq \perp\}$ 72 If $l^* \geq 0$: chain-comp(t^*, l^*, j^*) 73 Else: 74 For $j \leq j^* : j \in afs[t^*]$: $drk[t^*][j] \leftarrow_{\mathcal{S}} \mathcal{K}$ 75 For $j \leq j^* : j \in afs[t^*]$: 76 $drk'[t^*][j] \leftarrow drk[t^*][j]$ 77 $acst \leftarrow \mathcal{S}_{ACKA}(sim-st, P, t, i, pc, pcp, pc_{priv}, pcp_{init}, \ell_{priv}, pc2ratch, all-rcvd)$ 78 Return $((rk[t], cst, cct', acst, ck', drk', snd-par_P, t, i, j^{**}, \ell_{priv}, rak')$ </pre> <p>a: max returns -1 if there is no such l.</p>
--	---

Figure 18: History-independence simulator S_{PR} for the PR of Figure 2, where hash functions H_1, H_2, H_3 are modelled as programmable random oracles.

<pre> Proc MA.enc_S(<i>st</i>, <i>md</i>) 00 (<i>t</i>, <i>i</i>, <i>l</i>, <i>ck</i>) ← <i>st</i> 01 (<i>mk</i>, <i>tag</i>, <i>ck</i>) ← H(<i>ck</i>) 02 <i>c</i>' ←_§ AE.enc(<i>mk</i>, (<i>t</i>, <i>i</i>, <i>l</i>, <i>md</i>), <i>tag</i>) 03 <i>c</i> ← (<i>tag</i>, <i>c</i>') 04 <i>i</i> ← <i>i</i> + 1 05 <i>st</i> ← (<i>t</i>, <i>i</i>, <i>l</i>, <i>ck</i>) 06 Return (<i>st</i>, <i>c</i>) Proc MA.dec_R(<i>st</i>, <i>c</i>) 07 (<i>ht</i>, <i>ST</i>) ← <i>st</i> 08 (<i>tag</i>, <i>c</i>') ← <i>c</i> 09 (<i>usr</i>, <i>mk</i>) ← <i>ht</i>.acc(<i>tag</i>) 10 <i>m</i> ← AE.dec(<i>mk</i>, <i>c</i>, <i>tag</i>) 11 Require ⊥ ≠ <i>m</i> = (<i>t</i>', <i>i</i>', <i>l</i>', <i>md</i>) 12 <i>ht</i>.rem(<i>tag</i>) 13 (<i>t</i>_{now}, <i>i</i>_{now}, <i>i</i>_{nxt}, <i>ck</i>_{now}, <i>ck</i>_{nxt}) ← <i>ST</i>[<i>usr</i>] 14 Require <i>t</i>' ≤ <i>t</i>_{now} ∨ (<i>t</i>' = <i>t</i>_{now} + 1 ∧ <i>ck</i>_{nxt} ≠ ⊥) 15 If <i>t</i>' > <i>t</i>_{now}: 16 For all <i>i</i> : <i>l</i> < <i>i</i> ≤ <i>i</i>_{now}: 17 <i>ht</i>.rem((<i>t</i>_{now}, <i>i</i>)) 18 For all <i>i</i> : <i>i</i>_{now} ≤ <i>i</i> ≤ <i>l</i>: 19 (<i>mk</i>, <i>tag</i>, <i>ck</i>_{now}) ← H(<i>ck</i>_{now}) 20 <i>ht</i>.add({<i>tag</i>, (<i>t</i>_{now}, <i>i</i>)}, (<i>usr</i>, <i>mk</i>)) 21 <i>t</i>_{now} ← <i>t</i>' 22 <i>i</i>_{now} ← <i>i</i>_{nxt} 23 <i>i</i>_{nxt} ← 0 24 <i>ck</i>_{now} ← <i>ck</i>_{nxt} 25 <i>ck</i>_{nxt} ← ⊥ 26 For all <i>i</i> : <i>i</i>_{now} ≤ <i>i</i> < <i>i</i>' + fut: 27 (<i>mk</i>, <i>tag</i>, <i>ck</i>_{now}) ← H(<i>ck</i>_{now}) 28 <i>ht</i>.add({<i>tag</i>, (<i>t</i>_{now}, <i>i</i>)}, (<i>usr</i>, <i>mk</i>)) 29 <i>i</i>_{now} ← <i>i</i>' + fut - 1 30 <i>ST</i>[<i>usr</i>] ← (<i>t</i>_{now}, <i>i</i>_{now}, <i>i</i>_{nxt}, <i>ck</i>_{now}, <i>ck</i>_{nxt}) 31 <i>st</i> ← (<i>ht</i>, <i>ST</i>) 32 Return (<i>st</i>, <i>md</i>) </pre>	<pre> Proc MA.up_R(<i>st</i>, <i>usr</i>, <i>k</i>) 33 If <i>st</i> = ⊥: 34 <i>ht</i> ← HT.init() 35 <i>ST</i>[·] ← ⊥ 36 Else: (<i>ht</i>, <i>ST</i>) ← <i>st</i> 37 If <i>ST</i>[<i>usr</i>] = ⊥: 38 <i>t</i>_{now} ←_§ -1 39 <i>i</i>_{now}, <i>i</i>_{nxt} ← 0 40 <i>ck</i>_{now}, <i>ck</i>_{nxt} ← ⊥ 41 Else: (<i>t</i>_{now}, <i>i</i>_{now}, <i>i</i>_{nxt}, <i>ck</i>_{now}, <i>ck</i>_{nxt}) ← <i>ST</i>[<i>usr</i>] 42 Require <i>ck</i>_{nxt} = ⊥ 43 <i>ck</i>_{nxt} ← <i>k</i> 44 For all <i>i</i> : <i>i</i>_{nxt} ≤ <i>i</i> < fut: 45 (<i>mk</i>, <i>tag</i>, <i>ck</i>_{nxt}) ← H(<i>ck</i>_{nxt}) 46 <i>ht</i>.add({<i>tag</i>, (<i>t</i>_{now} + 1, <i>i</i>)}, (<i>usr</i>, <i>mk</i>)) 47 <i>i</i>_{nxt} ← fut - 1 48 <i>ST</i>[<i>usr</i>] ← (<i>t</i>_{now}, <i>i</i>_{now}, <i>i</i>_{nxt}, <i>ck</i>_{now}, <i>ck</i>_{nxt}) 49 <i>st</i> ← (<i>ht</i>, <i>ST</i>) 50 Return <i>st</i> Proc MA.up_S(<i>st</i>, <i>k</i>) 51 If <i>st</i> = ⊥: 52 <i>t</i>, <i>i</i>, <i>l</i> ← 0 53 Else: 54 (<i>t</i>, <i>i</i>, <i>l</i>, <i>ck</i>) ← <i>st</i> 55 <i>t</i> ← <i>t</i> + 1 56 <i>l</i> ← <i>i</i> - 1 57 <i>i</i> ← 0 58 <i>st</i> ← (<i>t</i>, <i>i</i>, <i>l</i>, <i>k</i>) 59 Return <i>st</i> </pre>
---	--

Figure 19: Full construction of MA, using a hash table HT = (HT.init, HT.add, HT.acc, HT.rem). The first parameter of algorithm HT.add takes a set of tags from which one suffices to access (via HT.acc) or remove (via HT.rem) the value (i.e., the second parameter). Constant fut specifies the number of pre-computed tags for which anonymous decryption of ciphertexts is successful.