# Universally Composable End-to-End Secure Messaging: A Modular Analysis[*]

Ran Canetti[†] Palak Jain[†] Marika Swanberg[†] Mayank Varia[†]

August 15, 2022

**Abstract**

We model and analyze the Signal end-to-end messaging protocol within the UC framework. In particular:

- We formulate an ideal functionality that captures end-to-end secure messaging, in a setting with PKI and an untrusted server, against an adversary that has full control over the network and can adaptively and momentarily compromise parties at any time and obtain their entire internal states. In particular our analysis captures the forward secrecy and recovery-of-security properties of Signal and the conditions under which they break.
- We model the main components of the Signal architecture (PKI and long-term keys, the backbone continuous-key-exchange or "asymmetric ratchet," epoch-level symmetric ratchets, authenticated encryption) as individual ideal functionalities that are realized and analyzed separately and then composed using the UC and Global-State UC theorems.
- We show how the ideal functionalities representing these components can be realized using standard cryptographic primitives under minimal hardness assumptions.

Our modeling introduces additional innovations that enable arguing about the security of Signal irrespective of the underlying communication medium, as well as secure composition of dynamically generated modules that share state. These features, together with the basic modularity of the UC framework, will hopefully facilitate the use of both Signal-as-a-whole and its individual components within cryptographic applications.

Two other features of our modeling are the treatment of fully adaptive corruptions, and making minimal use of random oracle abstractions. In particular, we show how to realize continuous key exchange in the plain model, while preserving security against adaptive corruptions.

[*]This article is the full version of a published article at CRYPTO 2022, © IACR 2022. This material is based upon work supported by the National Science Foundation under Grants No. 1718135, 1763786, 1801564, 1915763, and 1931714, by the DARPA SIEVE program under Agreement No. HR00112020021, by DARPA and the Naval Information Warfare Center (NIWC) under Contract No. N66001-15-C-4071, and by a Sloan Foundation Research Award.

[†]Boston University, {canetti,palakj,marikas,varia}@bu.edu.

# Contents

**1 Introduction** **4**
- 1.1 This Work 5
- 1.2 Notes on the Ideal Secure Messaging Functionality, $\mathcal{F}_{\mathsf{SM}}$ 6
- 1.3 Modelling the Components of the Signal Architecture as UC Modules 8
- 1.4 Streamlining our UC Analysis: Global Functionalities, Party Corruptions 12
- 1.5 Related Work 13

**2 Universally Composable Security** **14**
- 2.1 Universal Composability: A Primer 14
- 2.2 New Capabilities: Global Functionalities, Adversarially Provided Code 16

**3 Modelling Secure Messaging** **17**
- 3.1 Global Functionalities 18
- 3.2 The Secure Messaging Functionality, $\mathcal{F}_{\mathsf{SM}}$ 19

**4 Overview of our Modular Decomposition** **25**

**5 A Signal-style Secure Messaging Protocol: Realizing $\mathcal{F}_{\mathsf{SM}}$** **29**
- 5.1 The Epoch Key Exchange Functionality $\mathcal{F}_{\mathsf{eKE}}$ 29
- 5.2 The Forward Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ 32
- 5.3 The Signal Protocol, $\Pi_{\mathsf{SGNL}}$ 36
- 5.4 Security Analysis 39

**6 The Public Key Ratchet: Realizing Epoch Key Exchange** **43**
- 6.1 Cascaded PRF-PRG (CPRFG) 43
- 6.2 Protocol $\Pi_{\mathsf{eKE}}$ 48
- 6.3 Security Analysis 53

**7 Unidirectional Forward Secure Authenticated Channels: Realising $\mathcal{F}_{\mathsf{fs\_aead}}$** **57**
- 7.1 The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$ 58
- 7.2 The Single-Message Authenticated Encryption Functionality $\mathcal{F}_{\mathsf{aead}}$ 58
- 7.3 Protocol $\Pi_{\mathsf{fs\_aead}}$ 58

**8 The Symmetric Ratchet: Realising $\mathcal{F}_{\mathsf{mKE}}$** **63**

**9 Authenticated Encryption for Single Messages: Realising $\mathcal{F}_{\mathsf{aead}}$** **68**

**References** **73**

## List of Figures

1 Modeling and realizing secure messaging: The general subroutine structure . . . . . 9
2 The code library functionality, $\mathcal{F}_{\mathsf{lib}}$ . . . . . 17
3 The Public-Key Directory Functionality, $\mathcal{F}_{\mathsf{DIR}}$ . . . . . 18
4 The Long-Term Keys Module Functionality, $\mathcal{F}_{\mathsf{LTM}}$ . . . . . 20
5 The Programamble Random Oracle Functionality, $\mathcal{F}_{\mathsf{pRO}}$ . . . . . 21
6 The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$ . . . . . 22
7 The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$ . . . . . 23
8 The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$ . . . . . 33
9 The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$ (continued) . . . . . 34
10 The Forward-Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ . . . . . 37
11 The Signal Protocol, $\Pi_{\mathsf{SGNL}}$ . . . . . 38
12 Secure Messaging Simulator, $\mathcal{S}_{\mathsf{SM}}$ . . . . . 40
13 Internal Adversary $\mathcal{I}_{\mathsf{sm}}$ . . . . . 41
14 Cascaded PRF-PRG Security Game . . . . . 45
15 The Epoch Key Exchange Protocol $\Pi_{\mathsf{eKE}}$ . . . . . 50
16 The Epoch Key Exchange Protocol $\Pi_{\mathsf{eKE}}$ (continued) . . . . . 51
17 Internal adversarial code $\mathcal{I}_{eKE}$ . . . . . 53
18 Simulator $\mathcal{S}_{\mathsf{eKE}}$ for realizing $\mathcal{F}_{\mathsf{eKE}}$ . . . . . 55
19 Simulator $\mathcal{S}_{\mathsf{eKE}}$ (continued) . . . . . 56
20 The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$ . . . . . 59
21 The Authenticated Encryption with Associated Data Functionality, $\mathcal{F}_{\mathsf{aead}}$ . . . . . 60
22 The Forward-Secure Encryption $\Pi_{\mathsf{fs\_aead}}$ . . . . . 61
23 Forward Secure Authenticated Encryption Simulator, $\mathcal{S}_{\mathsf{fs\_aead}}$ . . . . . 64
24 Internal Adversary, $\mathcal{I}_{\mathsf{fs}}$ . . . . . 65
25 The Message Key Exchange Protocol $\Pi_{\mathsf{mKE}}$ . . . . . 66
26 Message Key Exchange Simulator, $\mathcal{S}_{\mathsf{mKE}}$ . . . . . 67
27 The Authenticated Encryption with Associated Data Protocol, $\Pi_{aead}$ . . . . . 69
28 Internal adversarial code $\mathcal{I}_{aead}$ . . . . . 70
29 Authenticated Encryption with Associated Data Simulator, $\mathcal{S}_{\mathsf{aead}}$ . . . . . 71

## 1 Introduction

Secure communication, namely allowing Alice and Bob to exchange messages securely, over an untrusted communication channel, without having to trust any intermediate component or party, is perhaps the quintessential cryptographic problem. Indeed, constructing and breaking secure communication protocols, as well as modeling security concerns and guarantees, providing a security analysis, and then breaking the modeling and analysis, has been a mainstay of cryptography since its early days.

Successful secure communication protocols have naturally been built to secure existing communication patterns. Indeed, IPSec has been designed to provide IP-layer end-to-end security for general peer-to-peer communication without the need to trust routers and other intermediaries, while SSL (which evolved into TLS) has been designed to secure client-server interactions, especially in the context of web browsing, and PGP has been designed to secure email communication.

Securing the communication over messaging applications poses a very different set of challenges, even for the case of pairwise communication (which is the focus of this work). First, the communicating parties do not typically have any direct communication connection and may not ever be online at the same time. Instead, they can communicate only via an untrusted server. Next, the communication may be intermittent and have large variability in volumes and level of interactivity. At the same time, a received message should be processed immediately and locally. Furthermore, connections may span very long periods of time, during which it is reasonable to assume that the endpoint devices would be periodically hacked or otherwise compromised – and hopefully later regain security.

The Signal protocol has been designed to give a response to these specific challenges of secure messaging, and in doing so it has revolutionized the concept of secure communication over the Internet in many ways. Built on top of predecessors like Off-The-Record [16], the Signal protocol is currently used to transmit hundreds of billions of messages per day [57].

Modeling the requirements of secure messaging in general, and analyzing the security properties of the Signal protocol in particular, has proved to be challenging and has inspired multiple analytical works [1–3, 7, 11, 12, 14, 18, 21, 29–37, 39–41, 54–56, 60–63, 65]. Some of these works directly address the Signal architecture and realization, whereas others propose new cryptographic primitives that are inspired by Signal's various modules.

**The need for composable security analysis.** It is well documented that standalone security analyses of protocols (namely, analyses that only consider an execution of the protocol "in vitro") are not always sufficient to capture the security of the protocol when used as a component within a larger system. This situation is particularly relevant to secure messaging and the Signal protocol. People typically participate concurrently in several conversations spanning several multi-platform chat services (e.g., smartphone and web), and the subtleties between a chat service and the underlying messaging protocol have led to network and systems security issues (e.g., [35, 36, 48]). For example, the Signal protocol is combined with other cryptographic protocols in WhatsApp [64] to perform abuse reporting or Status [58] and Slyo [59] to perform cryptocurrency transactions and Tor-style onion routing.

Moreover, Signal isn't always employed as a single monolithic protocol. Rather, variations and subcomponents of the Signal protocol are used within the Noise protocol family [53], file sharing services like Keybase [42] (which performs less frequent ratcheting), and videoconferencing services like Zoom [46] (which isn't concerned with asynchrony).

This state of affairs seems to call for a security analysis within a framework that allows for modular analysis and composable security guarantees. First steps in this direction were taken by the work of Jost, Maurer, and Mularczyk [41] that defines an abstract ratcheting service within the Constructive Cryptography framework [49, 50], and concurrent work by Bienstock et al. [13] that formulates an ideal functionality of the Signal protocol within the UC framework (see Section 1.5 for details). However, neither of these works give a modular decomposition of Signal into its basic components (as described in [52].)

**The apparent non-modularity of Signal.** One of the main sticking points when modeling and analyzing Signal in a composable fashion is that the protocol purposefully breaks away from the traditional structure of a short-lived "key exchange" module followed by a longer-lived module that primarily encrypts and decrypts messages using symmetric authenticated encryption. Instead, it features an intricate "continuous key exchange" module where shared keys are continually being updated, in an effort to provide forward security (i.e., preventing an attacker from learning past messages), as well as enabling the parties to quickly regain security as soon as the attacker loses access. Furthermore, Signal's process of updating the shared keys crucially depends on feedback from the "downsteam" authenticated encryption module. This creates a seemingly inherent circularity between the key exchange and the authenticated encryption modules, and gets in the way of basing the security of Signal on traditional components such as authenticated symmetric encryption, authenticated key exchange, and key-derivation functions.

**Security of Signal in face of adaptive corruptions.** Another potentially thorny aspect of the security of secure messaging protocols (Signal included) is the need to protect against an adversary that decides whom and when to corrupt, adaptively, based on all the communication seen so far. Indeed, not only is standard semantic security not known to imply security in this setting: there exist encryption schemes that are semantically secure (under reasonable intractability assumptions) but completely break in such a setting [38].

## 1.1 This Work

This work proposes a modular analysis of the Signal protocol and its components using the language of universally composable (UC) security [22, 23]. We focus on modeling Signal at the level specified in their documentation [52] (i.e., not limited to any single choice, of cipher suite), taking care to adhere to the abstractions within the specification.

We provide an ideal functionality, $\mathcal{F}_{\mathsf{SM}}$, for secure messaging along with individual ideal functionalities that capture each module within Signal's architecture. We then compose the modules to realize the top-level secure messaging functionality and demonstrate how to realize the modules in a manner consistent with the Signal specification [52]. Our instantiation achieves adaptive security against transient corruptions while making minimal use of the random oracle model. This combination of composability and modularity makes Signal and its components conveniently plug-and-play: future analyses can easily re-purpose or swap out instantiations of the modules in this work without needing to redo most of the security analysis.

In the process, we propose a new abstraction for Signal's continuous key derivation module, which we call a Cascaded PRF-PRG (CPRFG), and we show that it suffices for Signal's continuous key exchange module to achieve adaptive security. We also show how to construct CPRFGs from PRGs and puncturable PRFs. This new building block may be useful in other protocols as well.

The rest of the Introduction is organized as follows. Section 1.2 presents and motivates our formulation of $\mathcal{F}_{\mathsf{SM}}$. Section 1.3 presents and motivates the formulation of the individual modules, and describes how these modules can be realized. Section 1.4 highlights some new uses of the UC framework that might be useful elsewhere. Section 1.5 discusses related work.

## 1.2 Notes on the Ideal Secure Messaging Functionality, $\mathcal{F}_{\mathsf{SM}}$

We provide an ideal functionality $\mathcal{F}_{\mathsf{SM}}$ that captures end-to-end secure messaging, with some Signal-specific caveats. The goal here is to provide idealized security guarantees that will allow the analysis of existing protocols that use Signal, as well as enable Signal (or any protocol that realizes $\mathcal{F}_{\mathsf{SM}}$) to be readily usable as a component within other protocols in security-preserving manner.

When a party asks to encrypt a message, $\mathcal{F}_{\mathsf{SM}}$ returns a string to the party that represents the encapsulated message. When a party asks to decrypt (and provides the representative string), the functionality checks whether the provided string matches a prior encapsulation, and returns the original message in case of a match. The encapsulation string is generated via adversarially provided code that doesn't get any information about the encapsulated message, thereby guaranteeing secrecy.

**Simple user interface.** The above encapsulation and decapsulation requests are the only ways that a parent protocol interacts with $\mathcal{F}_{\mathsf{SM}}$. In particular, the parent protocol is not required to keep state related to the session, such as epoch-ids or sequence numbers. In addition to simplicity, this imparts the additional guarantee that a badly designed parent protocol cannot harm the security of a protocol realising $\mathcal{F}_{\mathsf{SM}}$.

**Abstracting away network delivery.** The fact that $\mathcal{F}_{\mathsf{SM}}$ models a secure messaging scheme as a set of local algorithms (an encapsulation algortihm and a decapsulation one) substantialy simplifies traditional UC modeling of secure communication, where the communication medium is modeled as part of the service provided by the protocol and the actual communication is abstracted away.

Furthermore, the fact that $\mathcal{F}_{\mathsf{SM}}$ returns to the parent protocol an actual string (that represents an idealized encapsulated message) allows the parent protocol to further process the string as needed, similarly to what's done in existing systems.

**Immediate decryption.** $\mathcal{F}_{\mathsf{SM}}$ guarantees that message decapsulation requests are fulfilled locally on the receiver's machine, and are not susceptible to potential network delays. Furthermore, this holds even if only a subset of the messages arrive, and arrival is out of order (as formalized in [1]). To provide this guarantee within the UC framework, we introduce a mechanism that enables $\mathcal{F}_{\mathsf{SM}}$ to execute adversarially provided code, without enabling the adversary to prevent immediate fulfillment of a decapsulation request. See more details in Section 2.

**Modeling of PKI and long term keys.** We directly model Signal's specific design for the public keys and associated secret keys that are used to identify parties across multiple sessions. Specifically, we formulate a "PKI" functionality $\mathcal{F}_{\mathsf{DIR}}$ that models a public "bulletin board," which stores the long-term, ephemeral, and one-time public keys associated with identities of parties. In addition, we model "long term private key" module $\mathcal{F}_{\mathsf{LTM}}$ for each identity. This module stores the private keys associated with the public keys of the corresponding party. Both functionalities are modeled as *global,* namely they are used as subroutines by multiple instances of $\mathcal{F}_{\mathsf{SM}}$. This modeling is what allows to tie the two participants of a session to long-term identities. Similarly to [20, 28], we treat these modules as incorruptible. It is stressed, however, that, following the Signal architecture, our realization of $\mathcal{F}_{\mathsf{SM}}$ calls the $\mathcal{F}_{\mathsf{LTM}}$ module of each party exactly once, at the beginning of the session.

**Modelling corruptions.** Resilience to recurring but transient break-ins is one of the main design goals of Signal. We facilitate the exposition of these properties as follows. First, we model corruption as an instantaneous event where the adversary learns the entire state of the corrupted party.[1]

The security guarantees for corruption and recovery are then specified as follows. When the adversary instructs $\mathcal{F}_{\mathsf{SM}}$ to corrupt a party, it is provided all the messages that have been sent to that party and were not yet received. In addition, the party is marked as compromised until a certain future point in the execution. While compromised, all the messages sent and received by the party are disclosed to the adversary, who can also instruct $\mathcal{F}_{\mathsf{SM}}$ to decapsulate ciphertexts to any plaintext of its choice. This captures the fact that as long as any one of the parties is compromised, neither party can securely authenticate incoming messages.

Forward secrecy guarantees that the adversary learns nothing about any messages that have been sent and received by the party until the point of corruption. Furthermore, the adversary obtains no information on the history of the session such as its duration or the long term identity of the peer. In $\mathcal{F}_{\mathsf{SM}}$, this is guaranteed because corruption does not provide the adversary with any messages that were previously sent and successfully received.

On the other hand, the specific point by which a compromised party regains its security is Signal-specific and described in more detail within. After this point, the adversary no longer obtains the messages the messages sent and received by the parties; furthermore, the adversary can no longer instruct $\mathcal{F}_{\mathsf{SM}}$ to decapsulate forged ciphertexts.

**Resilience to adaptive corruptions.** All the security guarantees provided by $\mathcal{F}_{\mathsf{SM}}$ hold in the presence of an adversary that has access to the entire communication among the parties and adaptively decides when and whom to corrupt based on all the communication seen so far. In particular, we do not impose any restrictions on when a party can be corrupted.

**Signal-specific limitations.** The properties discussed so far relate to the general task of secure messaging. In addition, $\mathcal{F}_{\mathsf{SM}}$ incorporates the following two relaxations that represent known weaknesses that are specific to the Signal design.

First, Signal does not give parties a way to detect whether their peers have received forged messages in their name during corruption. (Such situations may occur when either party was corrupted in the past and then recovered.) This represents a known weakness of Signal [18, 35]. Consequently, $\mathcal{F}_{\mathsf{SM}}$ exhibits a similar behavior.

Second, as remarked in the Signal documentation [52], when one of the parties is compromised, an adversary can "fork" the messaging session. That is, the adversary can create a person-in-the-middle situation where both parties believe they are talking with each other in a joint session, and yet they are actually both talking with the adversary. Furthermore, this can remain the case indefinitely, even when no party is compromised anymore. (In fact, we know this situation is inherent in an unauthenticated network with transient attacks, at least without repeated use of a long-term uncompromised public key [25].) While such a situation is mentioned in the Signal design documents, pinpointing and analyzing the conditions under which forking occurs has not been formally done before our work and the concurrent work by Bienstock et al. [13]. In our modeling, $\mathcal{F}_{\mathsf{SM}}$ forks when one of the parties is compromised, and at the same time the other party

[1] We don't directly model "Byzantine" corruptions, where the adversary is allowed to destroy or modify the state or program of the corrupted party. Indeed, when the internal state is modified, the concept of "regaining security" following a break-in becomes hard to pin down and is left out of scope for this work. We stress, however, that in our setting, where the adversary has complete control over the communication, mere knowledge of he internal state of a party suffices for full impersonation of that party to its peer.

successfully decapsulates a forged incoming message with an "epoch ID" that is different than the one used by the sender. In that case, $\mathcal{F}_{\mathsf{SM}}$ remains forked indefinitely, without any additional corruptions.

## 1.3 Modelling the Components of the Signal Architecture as UC Modules

Signal's strong forward secrecy and recovery from compromise guarantees are obtained via an intricate mechanism where shared keys are continually being updated, and each key is used to encapsulate at most a single message.

To help keep the parties in sync regarding which key to use for a given message, the conversation is logically partitioned into sending epochs, where each sending epoch is associated with one of the two parties, and consists of all the messages sent by that party from the end of its previous sending epoch until the first time this party successfully decapsulates an incoming message that belongs to the peer's latest sending epoch.

Within each sending epoch, the keys are pseudorandomly generated one after the other in a chain. The initial chaining key for each epoch is generated from a 'root chain' that ratchets forward every time a new sending epoch starts. Each ratcheting of the root chain involves a Diffie-Hellman key exchange; the resulting Diffe-Hellman secret is then used as input to the root ratchet (along with an existing chaining value). The public values of each such Diffie-Hellman exchange are piggybacked on the messages within the epoch and therefore authenticated using the same AEAD used for the data. Furthermore, these public values are used as unique identifiers of the sending epoch that each message is a part of. This mechanism allows the parties to keep in sync without storing any long-term information about the history of the session.

The Signal architecture document [52] de-composes the above mechanism into 3 main cryptographic modules, plus non-cryptographic code used to put these modules together. The modules are: (1) a symmetric authenticated encryption with associated data (AEAD) scheme that is applied to individual messages; (2) a symmetric key *ratcheting* mechanism to evolve the key between messages within an epoch; (3) an asymmetric key *ratcheting* (or "continuous key exchange") mechanism to evolve the "root chain." Since these modules are useful for applications beyond this particular protocol, we follow this partitioning and decompose Signal's protocol into similar components. (Our partitioning into components is also inspired by that of Alwen et al. [1].)

We model the security of each component as an ideal functionality within the UC framework. (These are $\mathcal{F}_{\mathsf{aead}}, \mathcal{F}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{eKE}}$, respectively.) This allows us to distill the properties provided by each module and demonstrate how they can be composed, along with the appropriate management code to obtain the desired functionality—namely to realize $\mathcal{F}_{\mathsf{SM}}$. The management code (specifically, protocols $\Pi_{\mathsf{fs\_aead}}$ and $\Pi_{\mathsf{SGNL}}$), does not directly access any keying material. Indeed, these protocols realise their respective specifications, namely $\mathcal{F}_{\mathsf{fs\_aead}}$ and $\mathcal{F}_{\mathsf{SM}}$, perfectly—see Theorems 2 and 4.

Before proceeding to describe the modules in more detail, we highlight the following apparent circularity in the security dependence between these modules: the messages in each sending epoch need to be authenticated (by the AEAD in use) using a key *k that's derived from the message itself*. Thus, modular security analysis along the above partitioning to modules might initially appear to be impossible.

The critical observation that allows us to proceed with modular decomposition is that the continuous key exchange module (which in our modeling corresponds to $\mathcal{F}_{\mathsf{eKE}}$) need not determine the authenticity of new epoch identifiers. Rather, this module is only tasked to assign a fresh pseudorandom secret key with each new epoch identifier, be it authentic or not. The determination of whether a new purported epoch identifier is authentic (or a forgery caused by an adversarially generated incoming message) is done elsewhere – specifically at the management level.

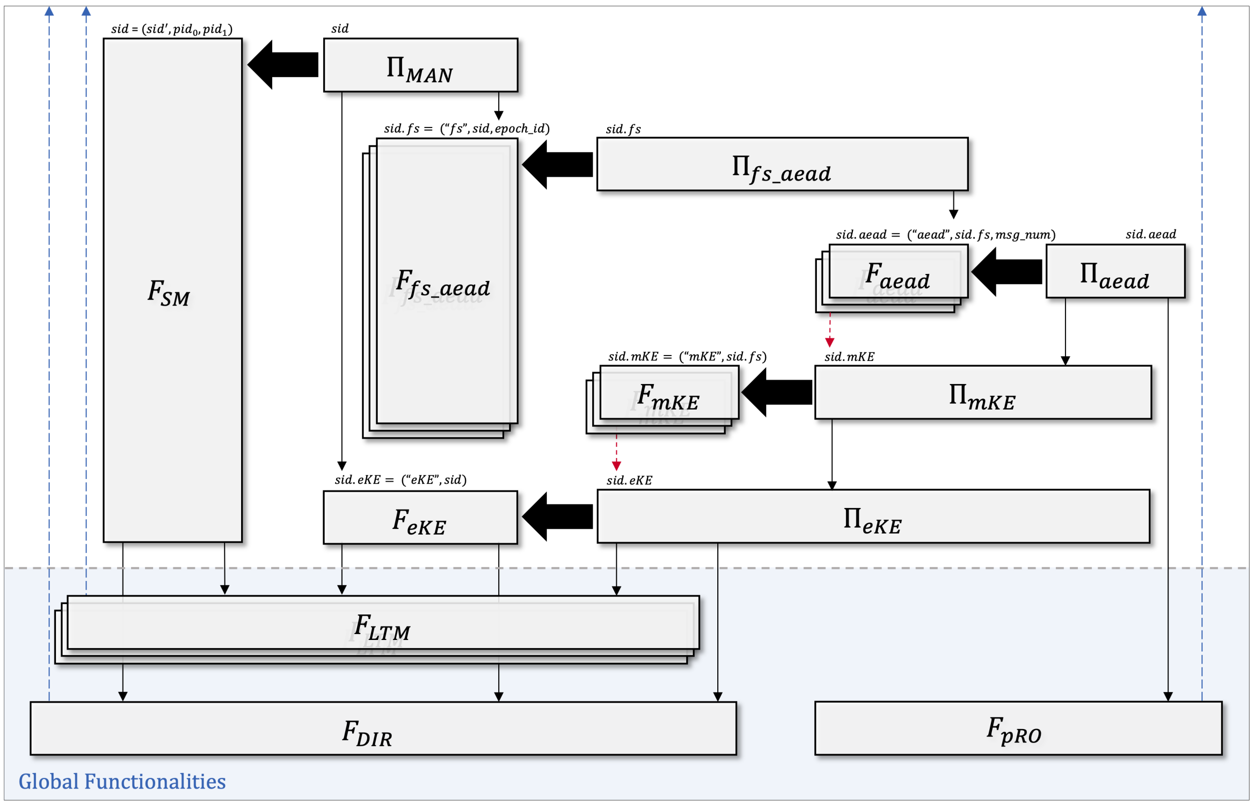

Figure 1: Modeling and realizing secure messaging: The general subroutine structure. Ideal functionalities are denoted by $F$ and protocols by $\Pi$. Thin vertical arrows denote subroutine calls, whereas thick horizontal arrows denote realization. Functionalities $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$ are global with respect to $\mathcal{F}_{\mathsf{SM}}$, whereas $\mathcal{F}_{\mathsf{eKE}}$ (and $\Pi_{\mathsf{eKE}}$) are global for $\mathcal{F}_{\mathsf{mKE}}$, and each instance of $\mathcal{F}_{\mathsf{mKE}}$ (and the corresponding instance of $\Pi_{\mathsf{mKE}}$) are global for $\mathcal{F}_{\mathsf{aead}}$.

We proceed to provide a more detailed overview of our partitioning and the general protocol logic. See also Figure 1.

**$\mathcal{F}_{\mathsf{eKE}}$.** The core component of the protocol is the epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$, which captures the generation of the initial shared secret key from the public information, as well as the continuous Diffie-Hellman protocol that generates the unique epoch identifiers and the "root chain" of secret keys. Whenever a party wishes to start a new epoch as a sender, it asks $\mathcal{F}_{\mathsf{eKE}}$ for a new epoch identifier, as well as an associated secret key. The receiving party of an epoch must present an epoch identifier, and is then given the associated secret key.

As mentioned, we allow the receiving party of a new epoch to present multiple potential epoch identifiers, and obtain a secret epoch key associated with each one of these identifiers. Furthermore, while only one of these keys is the one used by the sender for this epoch, all the keys provided by $\mathcal{F}_{\mathsf{eKE}}$ are guaranteed to appear random and independent to the adversary. In other words, $\mathcal{F}_{\mathsf{eKE}}$ leaves it to the receiver to determine which of the candidate identifiers for the new epoch is the correct one. (If $\mathcal{F}_{\mathsf{eKE}}$ recognizes, from observing the corruption activity and the generated epoch IDs, that the session has forked, then it exposes the secret keys to the adversary.) We postpone the discussion of realizing $\mathcal{F}_{\mathsf{eKE}}$ to the end of this section.

$\mathcal{F}_{\mathsf{mKE}}$. The per-epoch key chain is captured by an ideal functionality $\mathcal{F}_{\mathsf{mKE}}$ that is identified by an epoch-id, and generates, one at a time, a sequence of random symmetric keys associated with this epoch-id. The length of the chain is not a priori bounded; however, once $\mathcal{F}_{\mathsf{mKE}}$ receives an instruction to end the chain for a party, it complies. $\mathcal{F}_{\mathsf{mKE}}$ guarantees forward secrecy by making each key retrievable at most once by each party; that is, the key becomes inaccessible upon first retrieval, even for a corrupted party. However, it does not post-compromise security: once corrupted, all the future keys in the sequence are exposed to the adversary.

$\mathcal{F}_{\mathsf{mKE}}$ is realized by a protocol, $\Pi_{\mathsf{mKE}}$, that first calls $\mathcal{F}_{\mathsf{eKE}}$ with its current epoch-id, to obtain the initial chaining key associated with that epoch-id. The rest of the keys in this epoch are derived using a generic length-doubling PRG (of which Signal's typical instantiation using HKDF is a special case).

Demonstrating that $\Pi_{\mathsf{mKE}}$ realizes $\mathcal{F}_{\mathsf{mKE}}$ is relatively straightforward, except for the need to address the fact that the same instance of $\mathcal{F}_{\mathsf{eKE}}$ is used by multiple instances of $\Pi_{\mathsf{mKE}}$. using the formalism of [5], we thus show that $\Pi_{\mathsf{mKE}}$ UC-realizes $\mathcal{F}_{\mathsf{mKE}}$ in the presence of a global $\mathcal{F}_{\mathsf{eKE}}$.

$\mathcal{F}_{\mathsf{aead}}$. Authenticated encryption with associated data is captured by ideal functionality $\mathcal{F}_{\mathsf{aead}}$, which provides a one-time ideal authenticated encryption service: the encrypting party calls $\mathcal{F}_{\mathsf{aead}}$ with a plaintext and a recipient identity, and obtains an opaque ciphertext. Once the recipient presents the ciphertext, $\mathcal{F}_{\mathsf{aead}}$ returns the plaintext. (The recipient is given the plaintext only once, even when corrupted.) The "associated data," namely the public part of the authenticated message, is captured via the session identifier of $\mathcal{F}_{\mathsf{aead}}$.

$\mathcal{F}_{\mathsf{aead}}$ is realized via protocol $\Pi_{\mathsf{aead}}$, which employs an authenticated encryption algorithm using a key obtained from $\mathcal{F}_{\mathsf{mKE}}$. If we had opted to assert security against non-adaptive corruptions, any standard AEAD scheme would do. However, we strive to provide simulation-based security in the presence of fully adaptive corruptions, which is provably impossible in the plain model whenever the key is shorter than the plaintext [51]. We get around this issue by realizing $\mathcal{F}_{\mathsf{aead}}$ in the programmable random oracle model. While we provide a very simple AEAD protocol in this model, many common block cipher-based AEADs can also realize $\mathcal{F}_{\mathsf{aead}}$ provided we model the block cipher as a programmable random oracle. It is stressed however that the random oracle is used *only* in the case of short keys and adaptive corruptions. In particular, when corruptions are non-adaptive or the plaintext is sufficiently short, our protocol continues to UC-realize $\mathcal{F}_{\mathsf{aead}}$ even when the random oracle is replaced by the identity function.

Since each instance of $\mathcal{F}_{\mathsf{mKE}}$ is used by multiple instances of $\Pi_{\mathsf{aead}}$, we treat $\mathcal{F}_{\mathsf{mKE}}$ as a global functionality with respect to $\Pi_{\mathsf{aead}}$. That is, we show that $\Pi_{\mathsf{aead}}$ UC-realizes $\mathcal{F}_{\mathsf{aead}}$ in the presence of (a global) $\mathcal{F}_{\mathsf{mKE}}$.

$\mathcal{F}_{\mathsf{fs\_aead}}$. Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ is an abstraction of the management module that handles the encapsulation and decapsulation of all the messages within a single epoch. An instance of $\mathcal{F}_{\mathsf{fs\_aead}}$ is created by the main module of Signal whenever a new epoch is created, with session ID that contains the the identifier of this epoch. $\mathcal{F}_{\mathsf{fs\_aead}}$ then provides encapsulation and decapsulation services, akin to those of $\mathcal{F}_{\mathsf{aead}}$, for all the messages in its epoch. In addition, once instructed by the main module that its epoch has ended, $\mathcal{F}_{\mathsf{aead}}$ no longer allows encapsulation of new messages — even when the party is corrupted.

$\mathcal{F}_{\mathsf{fs\_aead}}$ is realized (perfectly, and in a straightforward way) by protocol $\Pi_{\mathsf{fs\_aead}}$ that calls multiple instances of $\mathcal{F}_{\mathsf{aead}}$, plus an instance of $\mathcal{F}_{\mathsf{mKE}}$ for this epoch - where, again, the session ID of $\mathcal{F}_{\mathsf{mKE}}$ contains the current epoch ID.

**$\Pi_{\mathsf{SGNL}}$.** At the highest level of abstraction, we have each of the two parties run protocol $\Pi_{\mathsf{SGNL}}$. When initiating a session, or starting a new epoch within a session, (i.e., when encapsulating the first message in an epoch), $\Pi_{\mathsf{SGNL}}$ first calls $\mathcal{F}_{\mathsf{eKE}}$ to obtain the identifier of that epoch, then creates an instance of $\mathcal{F}_{\mathsf{fs\_aead}}$ for that epoch ID and asks this instance to encapsulate the first message of the epoch. All subsequent messages of this epoch are encapsulated via the same instance of $\mathcal{F}_{\mathsf{fs\_aead}}$.

On the receiver side, once $\Pi_{\mathsf{SGNL}}$ obtains an encapsulated message in a new epoch ID, it creates an instance of $\mathcal{F}_{\mathsf{fs\_aead}}$ for that epoch ID and asks this instance to decapsulate the message. It is stressed that the epoch ID on the incoming message may well be a forgery; however in this case it is guaranteed that decapsulation will fail, since the peer has encapsulated this message with respect to a different epoch ID, namely a different instance of $\mathcal{F}_{\mathsf{fs\_aead}}$. (This is where the circular dependence breaks: even though the environment may invoke $\Pi_{\mathsf{SGNL}}$ on arbitrary incoming encapsulated message, along with related epoch IDs, $\mathcal{F}_{\mathsf{fs\_aead}}$ is guaranteed to reject unless the encapsulated message uses the same epoch ID as the as actual sender. Getting under the hood, this happens since the instances of $\mathcal{F}_{\mathsf{mKE}}$ that correspond to different epoch IDs generate keys that are mutually pseudorandom.) IT is stressed that $\Pi_{\mathsf{SGNL}}$ is purely "management code" in the sense that it only handles idealized primitives and does not directly access cryptographic keying material. Commensurately, it UC-realizes $\mathcal{F}_{\mathsf{SM}}$ perfectly.

**Realizing $\mathcal{F}_{\mathsf{eKE}}$.** Recall that $\mathcal{F}_{\mathsf{eKE}}$ is tasked to generate, at the beginning of each new epoch, multiple alternative keys for that epoch – a key for each potential epoch-id for that epoch. This should be done while preserving simulatability in the presence of adaptive corruptions.

Following the Signal architecture, the main component of the protocol that realizes $\mathcal{F}_{\mathsf{eKE}}$ is a key derivation function (KDF) that combines existing secret state, with new public information (namely the public Diffie-Hellman exponents, which also double-up as an epoch-id), and a new shared key (the corresponding Diffie-Hellman secret), to obtains a new secret key associated with the given epoch-id, along with potential new local secret state for the KDF.

If the KDF is modeled as a random oracle then it is relatively straightforward to show that the resulting protocol UC-realizes $\mathcal{F}_{\mathsf{eKE}}$.

On the other extreme, it can be seen that no plain-model instantiation of the KDF module, with bounded-size local state, can possibly realize $\mathcal{F}_{\mathsf{eKE}}$ in our setting. Indeed, since the adversary can obtain unboundedly many alternative keys for a given epoch, where all keys are generated using the same bounded-size secret state, the Nielsen bound [51] applies.

We propose a middle-ground solution: we show how to instantiate the KDF via a plain-model primitive where the local state grows linearly with the number of keys requested from $\mathcal{F}_{\mathsf{eKE}}$ at the beginning of a given epoch. Once the epoch advances, the state shrinks back to its original size. Our instantiation uses standard primitives: pseudorandom generators and puncturable pseudorandom functions. We also abstract the properties of our construction into a primitive which we call *cascaded pseudorandom function and generator (CPRFG),* following a primitive of [1] that is used for a similar purpose. We stress however that technically the primitives are quite different; we elaborate in the related work section.

**Modularity with weaker adaptivity.** Some prior analyses of Signal (see Section 1.5) consider also security against adversaries which are restricted in their adaptivity. We note that the modular decomposition of Signal as presented here remains valid even when considering such adversaries. That is, as long as the building blocks withstand a certain level of adaptivity, the overall protocol does too.[2]

**The benefits of modularity** We mentioned earlier in this section that currently existing implementations of AEAD and the asymmetric ratchet could also be shown to realize $\mathcal{F}_{\mathsf{aead}}$ and $\mathcal{F}_{\mathsf{eKE}}$ if one assumes a stronger reliance on the random oracle model. Concretely, one could (1) instantiate $\mathcal{F}_{\mathsf{aead}}$ using any CTR- or CBC-based encryption scheme under the assumption that the block cipher is a random oracle, and (2) instantiate $\mathcal{F}_{\mathsf{eKE}}$ using HKDF construction under the assumption that its HMAC subroutine is a random oracle. We emphasize that the modularity and generality of our interconnected ideal functionalities allow other instantiations of $\Pi_{\mathsf{aead}}$ and $\Pi_{\mathsf{eKE}}$ to be easily plugged in.

## 1.4 Streamlining our UC Analysis: Global Functionalities, Party Corruptions

We highlight two additional modeling and analytical techniques that we used to simplify the overall analysis. We hope that these would be useful elsewhere.

**Multiple levels of global state.** Our analysis makes extensive use of universal composition with global state (UCGS) within the plain UC model, as formulated and proven in [5]. However, there is a small difficulty in using the UCGS theorem directly for our multi-layer approach; Before we describe the difficulty and our solution, we briefly summarize our multi-layer use of global functionalities.

- First, at the highest level we use UCGS to model three global modules available to all the other functionalities in our paper. (1) A single *global directory* $\mathcal{F}_{\mathsf{DIR}}$ that stores the public keys of all parties. (2) A *long term storage module* $\mathcal{F}_{\mathsf{LTM}}$ for each party that stores the private keys corresponding to that party's globally available public keys. (2) A single *adversarial code library* $\mathcal{F}_{\mathsf{lib}}$ that stores the adversarially provided code for every functionality.

- Next, we use UCGS to model the two key exchange modules in our breakdown of the Signal architecture (This allows multiple short lived functionalities to ask for keys from a single long term module): (1) For each epoch of the conversation, an instance of the *message key exchange functionality* $\mathcal{F}_{\mathsf{mKE}}$ receives a request for each message key from the encryption module for that particular message. (2) For the entire conversation, a single *epoch key exchange functionality* $\mathcal{F}_{\mathsf{eKE}}$ receives requests for each epoch key from the message key exchange module $\mathcal{F}_{\mathsf{mKE}}$ for that particular epoch.

- Finally, we use UGCS to model the *programmable random oracle* $\mathcal{F}_{\mathsf{pRO}}$ that is used in the encryption module.

To understand the difficulty, recall that the UCGS theorem allows us to demonstrate that a protocol $\rho$ UC-realizes functionality $\mathcal{F}$ in the presence of a 'global' functionality $\mathcal{G}$ (where $\mathcal{G}$ takes inputs from $\rho$, $\mathcal{F}$, and also potentially directly from the environment $\mathsf{Env}$). This theorem is all we need for most of the global functionalities used in our model. However, for the two key-exchange functionalities we would like to additionally show that $\rho$ UC-realizes functionality $\mathcal{F}$ in the presence of protocol $\Pi_G$, where $\Pi_{\mathcal{G}}$ is some protocol that UC-realizes $\mathcal{G}$. However, such implication is not true in general [6, 28].

[2]See more details on modeling levels of adaptivity of corruptions in the UC framework in [23, Section 7.1 on Page 69].

We prove a simple-but-useful lemma (Theorem 1) to get around this problem: Lemma 1 in Section 2 asserts that, if $\Pi_{\mathcal{G}}$ UC-realizes $\mathcal{G}$ via a simulator $\mathcal{S}$, then any protocol $\rho$ that UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}^{\mathcal{S}}$ also UC-realizes $\mathcal{F}$ in the presence of $\Pi_{\mathcal{G}}$, where $\mathcal{G}^{\mathcal{S}}$ is the functionality that combines $\mathcal{G}$ with $\mathcal{S}$ in the natural way. We then show that, for the protocols $\rho$ in this work, having access to $\mathcal{G}^{\mathcal{S}}$ suffices for them to realise their respective functionalities $\mathcal{F}$.

**Multiple levels of corruptions.** The UC framework allows the adversary to adaptively and individually corrupt each party in each module within a composite protocol. While this is very general, it makes the handling of party-wise corruption events (where typically the internal states of multiple modules belonging to the party are exposed together) rather complex. We thus adopt a somewhat simpler modeling of party corruption: When the environment corrupts a protocol/-functionality belonging to a party, it obtains the state of the party corresponding to that module as well as all the sub-modules used within; (1) A corrupted module forwards the corruption notice to all its subroutines. (2) Each subroutine responds with a local state for the corrupted party. (3) The module collects the local states of the subroutines together with its own and reports them to the environment. (If the corrupted module is a functionality, it asks its simulator to produce a local state corresponding to the corrupted party.) In addition to being simpler, this modeling provides a tighter correspondence between the real and ideal executions and is thus preferable whenever realizable (which is the case in this work).

### 1.5 Related Work

This section briefly surveys the state of the art for security analyses of the Signal architecture in particular and end-to-end secure messaging in general, highlighting the differences from and similarities to the present work.

There is a long line of research into the design and analysis of two-party Signal messaging, its subcomponents, and variants of the Signal architecture; this research builds upon decades of study into key exchange protocols (e.g., [9, 10, 26, 27]) and self-healing after corruption (e.g., [25, 32, 34]). Some of these secure messaging analyses purposely consider a limited notion of adaptive security in order to analyze instantiations of Signal based on standardized crypto primitives (e.g., [1, 11, 36, 40, 65]). Other works consider a strong threat model in which the adversary is malicious, fully adaptive, and can tamper with local state [4, 7, 39, 41, 54], which then intrinsically requires strong HIBE-like primitives that depart from the Signal specification. By contrast, we follow a middle ground in this work: our adversary is fully adaptive and has no restrictions on when it can corrupt a party, yet its corruptions are instantaneous and passive.

We stress that, while this work is inspired by the clear game-based modeling and analyses of Signal in works like Alwen et al. [1], our modeling differs in a number of significant ways. For one, our analysis provides a composable security guarantee. Furthermore, we directly model secrecy against a fully adaptive adversary that decides who and when to corrupt based on all the information seen so far. In contrast, Alwen et al. [1] guarantee secrecy only against a *selective* adversary that determines ahead of time who and when it will corrupt.

There are two prior works that perform composable analyses of Signal. In concurrent work to our own, Bienstock et al. [13] provide an alternative modeling of an ideal secure messaging within the UC framework and demonstrate how the Signal protocol can be modeled in a way that is shown to realize their formulation of ideal secure messaging. Like this work, they demonstrate several shortcomings of previous formulations, such as overlooking the effect of choosing keys too early or keeping them around for too long. They also propose and analyze an enhancement of the double ratchet structure that helps parties regain security faster following a compromise event. Additionally, Jost, Maurer, and Mularczyk [41] conduct an analysis in the constructive cryptography framework. Their work provides a model for message transmission as well as one for ratcheting protocols.

That said, the ideal functionalities in [13] and [41] differ from our $\mathcal{F}_{\mathsf{SM}}$ in several ways.

- *Differences between our work and both of [13, 41]*: Their modeling does not account for the session initiation process, nor the PKI and long-term key modules that are an integral part of any secure messaging application. Additionally, they include the communication medium as part of the protocol, which (a) makes it harder to argue about immediate decryption and (b) means that an instantiation of Signal would have to include an entire TCP/IP stack, which weakens modularity and inhibits the use of Signal as a sub-routine within larger functionalities.

- *Additional differences with Bienstock et al. [13]:* While the modeling of the Signal protocol in [13] follows the traditional partitioning into continuous key exchange, epoch key derivation and authenticated encryption modules, it does not formalize this partitioning within the UC framework as done in this work; commensurately, they model all key derivation modules as random oracles. Also, their modeling forces the "calling protocol" to keep track of the message IDs for the Secure Messaging functionality/protocol, and assumes uniqueness of the IDs which might create a security risk. On the other hand, [13] accounts for adversarial choice of randomness, which our modeling does not account for.

- *Additional differences with Jost, Maurer, and Mularczyk [41]:* To model ratcheting components in a modular fashion, [41] introduces a global event history defined for the entire real (or ideal) world, where a history is a list of events having happened at a module (e.g. a message being input by Alice or one having leaked to the adversary). The event history is visible to the environment, the resources, and the simulator. The security of a resource is then allowed to depend on the global event history. They make composable statements about continuous key agreement protocols in their model (a notion introduced by Alwen et al. [1]) by restricting the adversaries capabilities in the real world as a function of the global event history. Alternatively, for the case of unrestricted adversaries [41] provide a HIBE-based implementation which is quite different than that of Signal (and ours) and requires heavier cryptographic primitives. Additionally, they transform their HIBE-based protocol into one that is fully composable via a technique that requires a restriction on the number of messages a party can send before receiving a response from the other party.

## 2 Universally Composable Security

### 2.1 Universal Composability: A Primer

UC security [23] is an instantiation of the simulation-based security paradigm in which the real world execution of a protocol $\Pi$ is compared with an idealized abstraction $\mathcal{F}$. The UC security framework gives two special powers to the distinguisher (also known as the *environment* $\mathsf{Env}$) in order to provide maximum flexibility to distinguish $\Pi$ from $\mathcal{F}$: direct interaction with either the protocol or the abstract specification (or, ideal functionality) by way of providing all inputs and obtaining all outputs, as well as interaction via pre-specified adversarial channels — either directly with the protocol or else with the specification, where the interaction is mediated by a special computational component called the simulator.

A protocol $\Pi$ is deemed to be a *UC-realization* of the functionality $\mathcal{F}$ if there exists an efficient simulator $\mathcal{S}$ such that no environment can tell whether it is interacting with $\Pi$, or else with $\mathcal{F}$ and

$\mathcal{S}$, namely $\text{exec}_{\mathcal{E},\Pi} \approx \text{exec}_{\mathcal{E},\mathcal{F},\mathcal{S}}$ for all polytime environments $\mathcal{E}$.

The UC model also formulates a stylized model of execution that is sufficiently general so as to capture most realistic computational systems. In this model, machines (which are the basic computational entity) can interact by sending messages to each other. Messages take the form of either input, or output, or "side information", where the latter model either adversarial leakage of information from a machine, or adversarial influence on the behavior of the machine. Machines can create other machines dynamically during an execution of the system, and each new machine is given an identity that includes its own program and is accessible to the machine itself. When a machine sends input or output to another machine, the system lets the recipient machine know the full identity of the sender.

By convention, identities consist of two fields, called session ID ($\mathsf{sid}$) and party ID ($\mathsf{pid}$). All machines that have the same $\mathsf{sid}$ and same program $\Pi$ are called a *session* of $\Pi$. These machines are also called the *main machines* of this instance. In this paper, we use the notation $(\mathcal{F}, \mathsf{sid})$ to denote the specific instance of the machine running the code of $\mathcal{F}$ that has session id $\mathsf{sid}$; this combination uniquely identifies a single machine. Within $\mathsf{sid}$, many (but not all) of the functionalities in this work will include the $\mathsf{pid}$ of the parties that are permitted to invoke this session; this serves as a form of access control.

If machine $A$ has sent input to machine $B$ in an execution, or machine $B$ sent output to machine $A$, then we say that $B$ is a subroutine of $A$. Protocol session $B$ is a subroutine of protocol session $A$ if some machine in $B$ is a subroutine of some machine in $A$. The extended session of some protocol session $A$ in an execution of a system includes the transitive closure of all the protocol sessions under the subroutine relation starting from $A$.

Remarkably, the UC model of execution considers only a single (extended) instance of the protocol under consideration, leading to relative simplicity of the specification and analysis. Still, the UC framework provides the following generic composition theorem, called the *UC Theorem*: Suppose one proves that a protocol $\Pi$ UC-realizes $\mathcal{F}$, and there exists another "hybrid" protocol $\rho$ that makes (perhaps many) subroutine calls to functionality $\mathcal{F}$. Now, consider the protocol $\rho^{\mathcal{F}\to\Pi}$ that replaces all instances of the ideal functionality $\mathcal{F}$ with the real protocol $\Pi$. The composition guarantees that the instantiation $\rho^{\mathcal{F}\to\Pi}$ is "just as secure" as the $\rho$ itself, in the same sense defined above. In this case we say that $\rho^{\mathcal{F}\to\Pi}$ UC-emulates $\rho$.

**UC with global subroutines.** Crucially, the UC theorem requires that both $\mathcal{F}$ and $\Pi$ are *subroutine respecting.* (A protocol is subroutine respecting if the only machines in any extended session of the protocol that take input from a machine that is not part of this extended session, or provides output to a machine that is not part of this extended session, are the main machines of this protocol session.)

While this requirement is both natural and essential, it does not allow for direct, "out of the box" application of the UC theorem in prevalent situations where one wants to decompose systems where multiple protocols (or multiple sessions of the same protocol) use some common construct as subroutine. In the context of this work, examples include public-key infrastructure, a long-term memory module that is used by multiple sessions, a key generation protocol that is used in multiple epochs and multiple messages in an epoch, or a global construct modeling the random oracle.

First attempts to handle such situations involved extending the UC framework to explicitly allow for multiple sessions of protocols within the basic model of execution [24]. However, this resulted in additional complexity and incompatibility with the basic UC model. More recently, the following formalism has been shown to suffice for capturing universal composition with global subroutines within the basic UC framework [5]:

Say that protocol $\Pi$ UC-realizes functionality $\mathcal{F}$ *in the presence of global subroutine* $G$ if there exists an efficient simulator $\mathcal{S}$ such that no environment can tell whether it is interacting with $\Pi$ and $G$, or else with $\mathcal{F}$, $G$, and $\mathcal{S}$. Here $G$ can be either a single machine or an entire protocol instance, where $G$ can be a subroutine of $\Pi$ or of $\mathcal{F}$, and at the same time take inputs directly from the environment and provide outputs directly to the environment[3].

Now, consider protocol $\rho$ that makes subroutine calls to functionality $\mathcal{F}$, and additionally also calls to $G$. Then the *UC With Global Subroutines Theorem* states that the protocol $\rho^{\mathcal{F}\rightarrow\Pi}$, that is identical to $\rho$ except that all instances of the ideal functionality $\mathcal{F}$ are replaced by instances of the real protocol $\Pi$, UC emulates $\rho$ *in the presence of* $G$. Note that in both $\rho$ and in $\rho^{\mathcal{F}\rightarrow\Pi}$, $G$ may take input from and provide outputs to multiple instance of $\Pi$ (or of $\mathcal{F}$), of $\rho$, and also directly to the environment.

## 2.2 New Capabilities: Global Functionalities, Adversarially Provided Code

We describe two new modeling techniques that simplify our analysis, and may be of more general interest.

**Instantiating global functionalities.** The first technique relates to applying the UC theorem to global functionalities. Assume that we have a protocol $\rho$ that UC-realises a functionality $\mathcal{F}$ in the presence of a global functionality $\mathcal{G}$. Assume also that we have a protocol $\Pi_{\mathcal{G}}$ that UC-realises the functionality $\mathcal{G}$. It may be tempting to deduce directly that $\rho$ UC-realizes $\mathcal{F}$ in the presence of $\Pi_{\mathcal{G}}$; however, this implication is false in general [6, 28]. Still, the following implication does hold: If $\Pi_{\mathcal{G}}$ UC-realises $\mathcal{G}$, and $\rho$ UC-realizes $\mathcal{F}$ in the presence of $\Pi_{\mathcal{G}}$, then $\rho$ UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}$.

We use this fact as follows: since $\Pi_{\mathcal{G}}$ UC-realizes $\mathcal{G}$, there must exist a simulator $\mathcal{S}$ such that no environment can distinguish between an interaction with $\Pi_{\mathcal{G}}$ and an interaction with $\mathcal{G}$ and $\mathcal{S}$. Now consider the machine $\mathcal{G}^{\mathcal{S}}$ that represents the combination of $\mathcal{G}$ and $\mathcal{S}$ (the communication between $\mathcal{G}$ and $\mathcal{S}$ are now internal to the combined machine $\mathcal{G}^{\mathcal{S}}$. We observe that $\Pi_{\mathcal{G}}$ and $\mathcal{G}^{\mathcal{S}}$ UC-emulate each other, more specifically $\Pi_{\mathcal{G}}$ UC-emulates $\mathcal{G}^{\mathcal{S}}$ *and in addition* $\mathcal{G}^{\mathcal{S}}$ *UC-emulates* $\Pi_{\mathcal{G}}$. Hence, instead of demonstrating that $\rho$ UC-realizes $\mathcal{F}$ in the presence of $\Pi$, it suffices to demonstrate that $\rho$ UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}^{\mathcal{S}}$. That is:

**Lemma 1** *Let $\Pi_{\mathcal{G}}$ be a protocol that UC-realizes an ideal functionality $\mathcal{G}$, and let $\mathcal{S}$ be a simulator that demonstrates this fact, i.e $exec_{\mathcal{E},\Pi_{\mathcal{G}}} \approx exec_{\mathcal{E},\mathcal{G},\mathcal{S}}$. Then protocols $\Pi_{\mathcal{G}}$ and $\mathcal{G}^{\mathcal{S}}$ UC-emulate each other. Consequently, for any protocol $\rho$ and ideal functionality $\mathcal{F}$ we have that $\rho$ UC-realizes $\mathcal{F}$ in the presence of $\Pi_{\mathcal{G}}$ if and only if $\rho$ UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}^{\mathcal{S}}$.*

**Modeling don't-care code and immediate response.** The second modeling technique can actually be thought of as further formalization of a technique that has been used in several works for different purposes. Specifically, it is sometimes convenient to have an ideal functionality expect to obtain from the adversary a piece of code that will be later run by the functionality under some conditions. In this work we have the additional requirement that the adversarially provided code should be readily available for use by ideal functionalities immediately upon first invocation.

Specifically, the formalism proceeds as follows. We formulate an ideal functionality, $\mathcal{F}_{\mathsf{lib}}$ (see Figure 2), to be used as a global functionality in the system. The adversary can upload code to

[3]Since the standard UC model of execution only considers an interaction of the environment with a single instance of some protocol, [5] first demonstrate that, without loss of generality, an instance of $\mathcal{F}$ alongside $G$ exhibits the same behavior as an instance of a "dummy protocol" $\delta$ that simply runs $\Pi$ alongside $G$ as subroutines of $\delta$.

be used by ideal functionalities. (We call the uploaded code an $\mathcal{I}$ adversary, as it can be thought of as an adversary run internally by the functionality without direct access to the environment.) More specifically, each uploaded code is associated with an identifier. Ideal functionalities can then ask $\mathcal{F}_{\mathsf{lib}}$ for the code associated with a given identifier. $\mathcal{F}_{\mathsf{lib}}$ then either responds with the uploaded code, or else returns an error message. For further convenience, we allow uploaded code to refer and use other uploaded pieces of code, referring to these pieces of code by their identifiers. (This is akin to "code linking" in standard software packages.)

```
$\mathcal{F}_{\mathsf{lib}}$

Obtaining adversarial code: When receiving a message $(\tau, \alpha, \mathsf{linking})$ from the adversary record it.  //$\alpha$ represents the adversarial code, and $\tau$ represents the code of the target machines to obtain code $\alpha$. The linking flag lets $\mathcal{F}_{\mathsf{lib}}$ know whether the adversarial code calls adversarial code for other target machines.

Delivering adversarial code: When receiving input $\tau$ from a party, find the latest $(\tau, \alpha, \mathsf{linking})$ that has been recorded.  //This code runs for a bounded amount of time; if it exceeds its specified running time, then it outputs $\perp$.

    1. If no such $(\tau, \alpha, \mathsf{linking})$ was recorded, output $\perp$.
    2. If $\mathsf{linking} == \mathit{true}$ then:
        • Go through program $\alpha$ and link the program by doing the following for all calls to dependencies $(\tau', \mathcal{I})$:
            (a) Find the latest $(\tau', \alpha', \mathsf{linking}')$ that has been recorded.
            (b) If no such record exists for a dependency, output $\perp$.
            (c) If $\mathsf{linking}' == \mathit{true}$ then run this compilation on $\alpha'$ starting at step 2.
            (d) Inline the code for the calls to $\alpha'$.
            (e) If this is the last dependency, record $(\tau, \alpha, \mathsf{linking} = \mathit{false})$.
    3. Output $\alpha$.
```

Figure 2: The code library functionality, $\mathcal{F}_{\mathsf{lib}}$

In this work this technique is used to model the immediate encryption and immediate decryption properties of secure messaging. The uploaded internal adversarial code is specific to the protocol that realizes the functionality, and essentially it acts as the ideal-world simulator during an honest execution. This ensures that the functionality does not need to wait for the adversary to encrypt or decrypt messages that are not corrupted. In cases where the message or ciphertext is corrupted, the fully adaptive adversary is called for input (for example, asking $\mathcal{A}$ to encrypt a message or decrypt a ciphertext). The state of the static code $\mathcal{I}$ is maintained across calls in a variable $\mathsf{state}_{\mathcal{I}}$, and it is sent to the adversary upon corruption. Here the fact that $\mathcal{F}_{\mathsf{lib}}$ is global is crucial, in allowing the static code to be already defined at the time that the functionality is instantiated.

The linking feature of $\mathcal{F}_{\mathsf{lib}}$ becomes handy when writing simulators for protocols that (a) realize an ideal functionality $\mathcal{F}$ that expect adversarial code, and (b) make use of another ideal functionality $\mathcal{F}'$ that also expects adversarial code. In such situations the code that the simulator will upload to $\mathcal{F}_{\mathsf{lib}}$ will link to the code that is to be uploaded by the adversary (or simulator) for $\mathcal{F}'$.

## 3 Modelling Secure Messaging

This section presents our overall modeling of secure messaging. Relying on the overview provided in the Introduction, we dive right into the detailed descriptions of the main components: the top-level secure messaging functionality $\mathcal{F}_{\mathsf{SM}}$ along with its global functionalities $\mathcal{F}_{\mathsf{DIR}}$ (representing the directory of public keys) and $\mathcal{F}_{\mathsf{LTM}}$ (representing the long-term key storage within a party), and $\mathcal{F}_{\mathsf{pRO}}$ (representing the programmable random oracle model).

## 3.1 Global Functionalities

In Section 2.2 we described the functionality $\mathcal{F}_{\mathsf{lib}}$ that we use to model the instant encryption and instant decryption properties of secure messaging. In this section we describe three other global subroutines used in our work. The global subroutines presented here are adaptations of well-studied UC functionalities.

$\mathcal{F}_{\mathsf{DIR}}$. First, we construct a public key infrastructure functionality called *the directory* $\mathcal{F}_{\mathsf{DIR}}$ (Fig. 3). This functionality allows each party (through their long-term module $\mathcal{F}_{\mathsf{LTM}}$) to upload their long-term public identity key $\mathsf{ik}$ to the directory. The directory also supports the execution of the triple Diffie-Hellman protocol that binds the session to the identity of the two participants. Specifically, $\mathcal{F}_{\mathsf{DIR}}$ allows the initial sender in a session of secure messaging to fetch the recipient's long- and short-term keys as well as a unique one time key for the session.

**$\mathcal{F}_{\mathsf{DIR}}$**

$\mathcal{F}_{\mathsf{DIR}}$ has a fixed session ID, denoted $\mathcal{F}_{\mathsf{DIR}}$.

**RecordKeys:** On input $(\mathtt{RecordKeys}, \mathsf{pid}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}})$ from $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$ do: //For simplicity $\mathcal{F}_{\mathsf{DIR}}$ records just one public key per $\mathsf{pid}$. Multiple public keys per "user level party" can be handled by having multiple $\mathsf{pid}$'s per party.

1. Set $\mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}$ and $\mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}$ as the identity and rotating keys corresponding to $\mathsf{pid}$, respectively, and
2. Output $(\mathtt{RecordKeys}, \mathsf{pid}, \mathtt{Success})$ to the caller.

**ReplaceRotatingKey:** On input $(\mathtt{ReplaceRotatingKey}, \mathsf{pid}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}})$ from $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$, replace the rotating key corresponding to $\mathsf{pid}$ with $\mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}$ and Output $(\mathtt{ReplaceRotatingKey}, \mathsf{pid}, \mathtt{Success})$.

**StoreOnetimeKeys:** On input $(\mathtt{StoreOnetimeKeys}, \mathsf{pid}, \mathsf{ls})$ from from $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$, do:

1. If the list $\mathsf{onetime\_keys}_{\mathsf{pid}}$ corresponding to $\mathsf{pid}$ doesn't exist, then create it.
2. Append $\mathsf{ls}$ to $\mathsf{onetime\_keys}_{\mathsf{pid}}$ and Output $(\mathtt{StoreOnetimeKeys}, \mathsf{pid}, \mathtt{Success})$ to the caller.

**GetInitKeys:** On input $(\mathtt{GetInitKeys}, \mathsf{pid}_j, \mathsf{pid}_i)$:
//$\mathsf{pid}_j$ is the responder and $\mathsf{pid}_i$ is the initiator.

1. If there is no entry for $\mathsf{pid}_j$ then output $(\mathtt{GetInitKeys}, \mathtt{Fail})$ to the caller.
2. Choose the first key $\mathsf{ok}^{\mathsf{pk}}_{\mathsf{pid}_j}$ from the list $\mathsf{onetime\_keys}_{\mathsf{pid}_j}$ (If the list is empty then let $\mathsf{ok}^{\mathsf{pk}}_{\mathsf{pid}} = \perp$.)
3. Remove $\mathsf{ok}^{\mathsf{pk}}_{\mathsf{pid}}$ from the list $\mathsf{onetime\_keys}_{\mathsf{pid}}$.
4. Output $(\mathtt{GetInitKeys}, \mathsf{pid}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{ok}^{\mathsf{pk}}_{\mathsf{pid}})$ to the caller.

**GetResponseKeys:** On input $(\mathtt{GetResponseKeys}, \mathsf{pid}_i)$ from a machine with party id $\mathsf{pid}_j$: //$\mathsf{pid}_j$ is the responder.

1. Send $(\mathtt{GetResponseKeys}, \mathsf{pid}_i, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}_i})$ to the caller.

**GetRotatingKey:** On input $(\mathtt{GetRotatingKey}, \mathsf{pid})$, do: If there is no entry for $\mathsf{pid}$ then output $(\mathtt{GetRotatingKey}, \mathtt{Fail})$ to the caller. Else $(\mathtt{GetRotatingKey}, \mathsf{pid}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}})$ to the caller.

Figure 3: The Public-Key Directory Functionality, $\mathcal{F}_{\mathsf{DIR}}$

**$\mathcal{F}_{\mathsf{LTM}}$.** Next, we design a module $\mathcal{F}_{\mathsf{LTM}}$ (Fig. 4) with two responsibilities: local storage of long-term public and private cryptographic key material, and performing computations that require access to the long term private keys of a party. Intuitively, one can think of $\mathcal{F}_{\mathsf{LTM}}$ as a trusted execution enclave or secure co-processor that performs the Diffie-Hellman operations associated with the long term keys. Looking ahead, our secure messaging protocol only invokes $\mathcal{F}_{\mathsf{LTM}}$ when establishing a new session of secure messaging; it is not invoked by ongoing communications. The functionality also has several methods to support the execution of Signal's triple Diffie-Hellman protocol [47]. Concretely, each party can generate short-term rotating and one-time keys that limit the period of vulnerability if a long-term key is compromised.

**$\mathcal{F}_{\mathsf{pRO}}$.** Finally, we design a programmable random oracle module $\mathcal{F}_{\mathsf{pRO}}$ (Fig. 5) that will be used to generate one-time keys during the symmetric ratcheting step. We need a random oracle for equivocation against an adaptive attacker, given that there is no bound on the number of message keys that a party might use within an epoch. Specifically, we use the following random oracle functionality from [19]. Anyone can query the random oracle, but only the (real or ideal world) adversary has the power to program it.

### 3.2 The Secure Messaging Functionality, $\mathcal{F}_{\mathsf{SM}}$

This section presents our *secure message functionality* $\mathcal{F}_{\mathsf{SM}}$. The complete details of the functionality can be found in Figure 6. We start with high level description of the functionality and some motivation for specific choices.

Functionality $\mathcal{F}_{\mathsf{SM}}$ takes two types of inputs: `SendMessage` is used to encapsulate a message for sending to the peer, whereas `ReceiveMessage` is used to decapsulate a received message. We also have a `Corrupt` input; this is a 'modeling input' that is used to capture party corruption. In addition, $\mathcal{F}_{\mathsf{SM}}$ takes a number of 'side channel' messages from the adversary which are used to fine-tune the security guarantees.

An instance of $\mathcal{F}_{\mathsf{SM}}$ is created by some party (namely an ITM, or colloquially a machine) by way of sending the first `SendMessage` input to a machine whose code is $\mathcal{F}_{\mathsf{SM}}$ and whose session ID is $\mathsf{sid}$. (Recall that ideal functionalities have null party identifier.) The creating party encodes its own identity, as well as the identity of the desired peer for the interaction, in the session ID. That is, it is expected that $\mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1)$, where $\mathsf{pid}_0$ is the identity of the creating party (ie, the *initiator*), and $\mathsf{pid}_1$ is the identity of the other party. It is stressed that there is no special "initiation" input, namely the first `SendMessage` input already contains the message to be encapsulated.

**SendMessage.** At the first $(\mathtt{SendMessage}, m)$ activation (which is the first activation overall), $\mathcal{F}_{\mathsf{SM}}$ first verifies that the instance of $\mathcal{F}_{\mathsf{LTM}}$ that corresponds to the initiator $\mathsf{pid}_0$ exists, and that the desired peer ($\mathsf{pid}_1$) is registered with $\mathcal{F}_{\mathsf{DIR}}$. It also initiates variables that will record subsequent epoch identifiers and the numeral of each message in its epoch, as well as which party is compromised at each epoch.

Next, $\mathcal{F}_{\mathsf{SM}}$ lets the ideal-model adversary (namely, the simulator) choose the ciphertext $c$ that will correspond to $m$. If the sending party is uncompromised, then the simulator should make this choice given only the length of $m$; if the sender is currently compromised then the simulator is given $m$ in full.

Furthermore, if the local state of $\mathcal{F}_{\mathsf{SM}}$ indicates that this `SendMessage` activation is the first one in a new epoch, then $\mathcal{F}_{\mathsf{SM}}$ asks the adversary for a new epoch identifier for this epoch, and verifies that the received identifier is different than all previously used ones.

## $\mathcal{F}_{\mathsf{LTM}}$

$\mathcal{F}_{\mathsf{LTM}}$ is parameterized by a specific activator program Root, a key derivation function HKDF and key generation function keyGen(), and an algebraic group $G$. All algebraic operations are done in $G$. The local session ID is of the form $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$. Inputs from senders whose party ID is different than $\mathsf{pid}$ are ignored.

**Initialize:** On input (`Initialize`) from $(\mathsf{pid}, \text{Root})$ do: If this is not the first activation then end the activation. Else:

1. Create an empty list $\mathsf{onetime\_keys}_{\mathsf{pid}} = [\ ]$. Also, choose and record the key pairs $(\mathsf{ik}^{\mathsf{sk}}_{\mathsf{pid}}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}), (\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}) \overset{\$}{\leftarrow}$ keyGen(), which will be called the party's identity key-pair and rotating key-pair, respectively.
2. Provide input $(\texttt{RecordKeys}, \mathsf{pid}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}})$ to $\mathcal{F}_{\mathsf{DIR}}$.

**UpdateRotatingKey:** On input (`UpdateRotatingKey`) from $(\mathsf{pid}, \text{Root})$, do:

1. Replace the rotating key pair with a new key pair $(\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}) \overset{\$}{\leftarrow}$ keyGen().
2. Provide input $(\texttt{ReplaceRotatingKey}, \mathsf{pid}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}})$ to $\mathcal{F}_{\mathsf{DIR}}$.

**GenOnetimeKeys:** On input $(\texttt{GenOnetimeKeys}, \mathsf{pid}, j)$ from $(\mathsf{pid}, \text{Root})$, do:

1. Choose $j$ new key pairs $(\mathsf{ok}^{\mathsf{sk}}_1, \mathsf{ok}^{\mathsf{pk}}_1), \ldots, (\mathsf{ok}^{\mathsf{sk}}_j, \mathsf{ok}^{\mathsf{pk}}_j) \overset{\$}{\leftarrow}$ keyGen() and append them to $\mathsf{onetime\_keys}_{\mathsf{pid}}$.
2. Provide input $(\texttt{StoreOnetimeKeys}, \mathsf{pid}, \mathsf{ok}^{\mathsf{pk}}_1, \ldots, \mathsf{ok}^{\mathsf{pk}}_j)$ to $\mathcal{F}_{\mathsf{DIR}}$.

**ConfirmRegistration:** On input (`ConfirmRegistration`) from $(\mathcal{F}_{\mathsf{SM}}, \mathsf{pid})$ or $(\Pi_{MAN}, \mathsf{pid})$, do:

1. If $\mathsf{pid}$ has already called (`Initialize`), output (`ConfirmRegistration`, `Success`).
2. Otherwise output (`ConfirmRegistration`, `Fail`).

**ComputeSendingRootKey:** On input $(\texttt{ComputeSendingRootKey}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{partner}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{partner}}, \mathsf{ok}^{\mathsf{pk}}_{\mathsf{partner}})$ from a machine with PID $\mathsf{pid}$ and code $\Pi_{\mathsf{eKE}}$:

1. Choose an ephemeral key pair $(\mathsf{ek}^{\mathsf{sk}}, \mathsf{ek}^{\mathsf{pk}}) \overset{\$}{\leftarrow}$ keyGen() and compute the following:
   - $DH_1 = (\mathsf{rk}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ik}^{\mathsf{sk}}_{\mathsf{pid}}}$ //Here $(a)^b$ denotes the exponentiation operation in the respective algebraic group.
   - $DH_2 = (\mathsf{ik}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ek}^{\mathsf{sk}}}$
   - $DH_3 = (\mathsf{rk}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ek}^{\mathsf{sk}}}$
   - $DH_4 = (\mathsf{ok}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ek}^{\mathsf{sk}}}$
2. Output $(\texttt{ComputeSendingRootKey}, HKDF(DH_1||DH_2||DH_3||DH_4), \mathsf{ek}^{\mathsf{pk}})$.

**ComputeReceivingRootKey:** On input $(\texttt{ComputeReceivingRootKey}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{partner}}, \mathsf{ek}^{\mathsf{pk}}_{\mathsf{partner}}, \mathsf{ok}^{\mathsf{pk}})$ from $(\Pi_{\mathsf{eKE}}, \mathsf{pid})$ do:

1. If list $\mathsf{onetime\_keys}_{\mathsf{pid}}$ does not contain an entry $(\mathsf{ok}^{\mathsf{sk}}, \mathsf{ok}^{\mathsf{pk}})$ for the given $\mathsf{ok}^{\mathsf{pk}}$, then output an error message to $(\Pi_{\mathsf{eKE}}, \mathsf{pid})$.
2. Else, delete the one-time key pair $(\mathsf{ok}^{\mathsf{sk}}, \mathsf{ok}^{\mathsf{pk}})$ from the list and compute:
   - $DH_1 = (\mathsf{ik}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}}$
   - $DH_2 = (\mathsf{ek}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ik}^{\mathsf{sk}}_{\mathsf{pid}}}$
   - $DH_3 = (\mathsf{ek}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}}$
   - $DH_4 = (\mathsf{ek}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ok}^{\mathsf{sk}}}$
3. Output $(\texttt{ComputeReceivingRootKey}, HKDF(DH_1||DH_2||DH_3||DH_4))$.

Figure 4: The Long-Term Keys Module Functionality, $\mathcal{F}_{\mathsf{LTM}}$

$\mathcal{F}_{\mathsf{pRO}}$

On input $(\texttt{HashQuery}, m, \ell(n))$:

1. If there is a record $(m, h)$
   - If $|h| \geq \ell(n)$: let $h'$ be the first $\ell(n)$ bits of $h$. //$\mathcal{F}_{\mathsf{pRO}}$ returns prefixes of already-computed entries.
   - If $|h| < \ell(n)$: choose $h_{end} \xleftarrow{\$} \{0,1\}^{\ell(n)-|h|}$, let $h' = h||h_{end}$, and replace the record $(m, h)$ with $(m, h')$.

   Else choose $h' \xleftarrow{\$} \{0,1\}^{\ell(n)}$ and record $(m, h')$.
2. Output $(\texttt{HashQuery}, h')$ to the caller.

On message $(\texttt{Program}, m, h)$ from the adversary:

1. If there is no record $(m, h')$, then record $(m, h)$. Send $(\texttt{Program})$ to the adversary. //If $m$ has already been queried then programming fails silently.

Figure 5: The Programamble Random Oracle Functionality, $\mathcal{F}_{\mathsf{pRO}}$

Finally, $\mathcal{F}_{\mathsf{SM}}$ records $(m, c, h)$ where $h$ is the "header information" that includes the epoch identifier $\mathsf{epoch\_id}$ of the message and the message number $\mathsf{msg\_num}$ in the epoch, and outputs $(c, h)$ to $pid_0$. It is stressed that, as long as the epoch IDs are unique, no two records of encrypted messages have the same header information. Indeed, uniqueness is the only property that the epoch IDs need to satisfy.

**ReceiveMessage.** At a high level, this input (or, "function call") allows the receiving party to perform an "idealized authenticated decryption" operation, in spite of the fact that the ciphertext was generated without knowledge of the message and $\mathcal{F}_{\mathsf{SM}}$ itself has no keying material.

More specifically, on input $(\texttt{ReceiveMessage}, c, h)$ from $\mathsf{pid} \in \{pid_0, \mathsf{pid}_1\}$, where $h = (\mathsf{epoch\_id}, \mathsf{msg\_num})$, $\mathcal{F}_{\mathsf{SM}}$ proceeds as follows:

- If this is the first $\texttt{ReceiveMessage}$ activation, then $\mathcal{F}_{\mathsf{SM}}$ verifies that this request is coming from $\mathsf{pid}_1$, that the instance of $\mathcal{F}_{\mathsf{LTM}}$ that corresponds to $\mathsf{pid}_1$ exists, and that $\mathsf{pid}_0$ is registered with $\mathcal{F}_{\mathsf{DIR}}$.
- If a record $(m, c, h)$ exists then $\mathcal{F}_{\mathsf{SM}}$ returns the corresponding message $m$ to $\mathsf{pid}$ *and deletes the record* $(m, c, h)$. (That is, a message is decryptable exactly once.)
- If there exists no record $(m, c', h')$ with $h' = h$ then return $\perp$.
- If there exists a record $(m, c', h')$ where $c' \neq c$ and $h' = h$ (presumably, $c'$ is a "mauled ciphertext") then $\mathcal{F}_{\mathsf{SM}}$ gives the adversary the latitude to decide whether decryption should succeed; still, in case of successful decryption the returned value is $m$, and the record is deleted. This behavior combines the standard EU-CMA guarantee for the underlying authentication scheme, combined with one-time decryption.
- Finally, if this happens to be the first successfully received message for party $\mathsf{pid}_i$ in the newest epoch, then $\mathcal{F}_{\mathsf{SM}}$ notes that the next $\texttt{SendMessage}$ activation with sender $\mathsf{pid}$ will start a new epoch.

It is stressed that $\mathcal{F}_{\mathsf{SM}}$ only supports decrypting each header once; after a successful decryption for a header it will refuse to participate in subsequent calls with the same $h$ to ensure forward secrecy.

## $\mathcal{F}_{\mathsf{SM}}$ (Part 1)

The local session ID is parsed as $\mathsf{sid} = (sid', \mathsf{pid}_0, \mathsf{pid}_1)$. Inputs arriving from machines whose identity is neither $\mathsf{pid}_0$ nor $\mathsf{pid}_1$ are ignored. //For notational simplicity we assume some fixed interpretation of $\mathsf{pid}_0$ and $\mathsf{pid}_1$ as complete identities of the two calling machines.
It also has *internal adversary code* $\mathcal{I} = \mathcal{I}_{\mathsf{sm}}$ (Fig. 28) . We initialize the state for $\mathcal{I}$ to be $\mathsf{state}_{\mathcal{I}} = \perp$.

**Sending messages:** On receiving ($\mathtt{SendMessage}, m$) from $\mathsf{pid}$ do: //Here $\mathsf{pid}$ is an extended identity of a machine.

1. If $\mathtt{initialized}$ not set do: //initialization
   - If $\mathsf{pid} \neq \mathsf{pid}_0$, end the activation. Otherwise, send ($\mathtt{ConfirmRegistration}$) to ($\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid}$).
   - Upon output ($\mathtt{ConfirmRegistration}, t$) from $\mathcal{F}_{\mathsf{LTM}}$, if $t = \mathtt{Fail}$ end the activation. Else input ($\mathtt{GetInitKeys}$) to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon output ($\mathtt{GetInitKeys}, \mathsf{pid}_1, \mathsf{ik}_1^{\mathsf{pk}}, \mathsf{rk}_1^{\mathsf{pk}}, \mathsf{ok}_1^{\mathsf{pk}}$) from $\mathcal{F}_{\mathsf{DIR}}$: if $\mathsf{ok}_1^{\mathsf{pk}} = \perp$, end the activation. Else:
     – Set $\mathtt{initialized}$, $\mathsf{epoch\_num}_0 = 0$, $\mathsf{sent\_msgnum}_0 = 0$, $\mathsf{rcv\_msgnum}_0 = 0$,$N\_\mathsf{self}_0 = 0$, $\mathsf{diverge\_parties} = \mathit{false}$.
     – Create the dictionaries $\mathsf{advControl} = \{\}$, $\mathsf{id\_dict} = \{\}$, and $\mathsf{N\_dict} = \{\}$. Initialize $\mathsf{advControl}[\mathsf{epoch\_num}_0] = \perp$ and $\mathsf{advControl}[e] = \infty$ for all $e \geq 0$. //$\mathsf{advControl}$ will record which parties are adversarially controlled in each epoch, $\mathsf{id\_dict}$ maps epoch id's to epoch numbers, and $\mathsf{N\_dict}$ will hold the number of messages sent in each epoch.
   - Call $\mathcal{F}_{\mathsf{lib}}$ with input $\mathcal{F}_{\mathsf{SM}}$ to obtain the internal code $\mathcal{I}$.
2. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$. Increment $\mathsf{sent\_msgnum}_i$ by 1.
3. If $\mathtt{leak} \in \mathsf{advControl}[\mathsf{epoch\_num}_i]$ or $\mathsf{diverge\_parties} = \mathit{true}$: Send a backdoor message ($\mathsf{state}_{\mathcal{I}}, \mathtt{SendMessage}, \mathsf{pid}, m$) to $\mathcal{A}$.
4. Else ($\mathtt{leak} \notin \mathsf{advControl}[\mathsf{epoch\_num}_i]$ and $\mathsf{diverge\_parties} = \mathit{false}$): Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \mathtt{SendMessage}, \mathsf{pid}, |m|)$
5. Upon obtaining ($\mathsf{state}'_{\mathcal{I}}, \mathtt{SendMessage}, \mathsf{pid}, \mathsf{epoch\_id}, c$) from $\mathcal{A}$ or $\mathcal{I}$ do:
   - Update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$.
   - If $\mathsf{sent\_msgnum}_i == 1$: If $\mathsf{epoch\_id}$ equals any of the keys in the dictionary $\mathsf{id\_dict}$ then end the activation. Else record $\mathsf{id\_dict}[\mathsf{epoch\_id}] = \mathsf{epoch\_num}_i$.
   - Set $h = (\mathsf{epoch\_id}, \mathsf{sent\_msgnum}_i, N\_\mathsf{self}_i)$. //$N\_\mathsf{self}_i$ holds the # of messages sent by $\mathsf{pid}_i$ in its previous sending epoch.
   - If $\mathsf{diverge\_parties} = \mathit{false}$ then record ($\mathsf{pid}, h, c, m$). //If the parties' states have diverged, then encrypted messages are no longer recorded.
   - Output ($\mathtt{SendMessage}, \mathsf{sid}, \mathsf{pid}, h, c$) to $\mathsf{pid}$.

**Corrupt:** On receiving a ($\mathtt{Corrupt}, \mathsf{pid}_i$) request from $\mathsf{Env}$ for $\mathsf{pid}_i \in \{\mathsf{pid}_0, \mathsf{pid}_1\}$, do:

1. Append ($\mathsf{epoch\_num}_i, \mathsf{sent\_msg\_num}_i, \mathsf{received\_msg\_num}_i$) to the list $\mathsf{corruptions}_i$.
2. For all epochs $e \leq \mathsf{epoch\_num}_i$, set $\mathsf{advControl}[e] = \{\mathtt{leak}, \mathtt{inject}\}$ to allow the adversary to influence messages still in transit.
3. Create a list $\mathsf{pending\_msgs}$ with all records of the form ($\mathsf{pid}_{1-i}, h, c, m$) corresponding to headers for which there is no record ($\mathtt{Authenticate}, \mathsf{pid}_{1-i}, h, \_, 1$) (these are the messages that were not decrypted yet).
4. Send a request ($\mathsf{state}_{\mathcal{I}}, \mathtt{ReportState}, \mathsf{pid}_i, \mathsf{pending\_msgs}$) to $\mathcal{A}$.
5. On receiving a state ($\mathtt{ReportState}, \mathsf{pid}_i, S$) from $\mathcal{A}$, send $S$ to $\mathsf{Env}$.

Figure 6: The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$

## $\mathcal{F}_{\mathsf{SM}}$ (Part 2)

The local session ID is parsed as $\mathsf{sid} = (sid', \mathsf{pid}_0, \mathsf{pid}_1)$. Inputs arriving from machines whose identity is neither $\mathsf{pid}_0$ nor $\mathsf{pid}_1$ are ignored. //For notational simplicity we assume some fixed interpretation of $\mathsf{pid}_0$ and $\mathsf{pid}_1$ as complete identities of the two calling machines.
It also has *internal adversary code* $\mathcal{I} = \mathcal{I}_{\mathsf{sm}}$ (Fig. 28) . We initialize the state for $\mathcal{I}$ to be $\mathsf{state}_{\mathcal{I}} = \perp$.

**Receiving messages:** On receiving $(\mathtt{ReceiveMessage}, h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N), c)$ from $\mathsf{pid}$, do:

1. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. If this is the first $\mathtt{ReceiveMessage}$ request: If $i = 0$ then end the activation. Else ($\mathsf{pid} = \mathsf{pid}_1$), initialize the responder:
   - Send $(\mathtt{ConfirmRegistration})$ to $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$.
   - Upon receiving the output $(\mathtt{ConfirmRegistration}, t)$ from $\mathcal{F}_{\mathsf{LTM}}$: If $t = \mathtt{Fail}$ then end activation. Else provide input $(\mathtt{GetResponseKeys}, \mathsf{pid}_0, \mathsf{pid}_1)$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving output $(\mathtt{GetResponseKeys}, \mathsf{pid}_0, \mathsf{ik}_0^{\mathsf{pk}})$ from $\mathcal{F}_{\mathsf{DIR}}$, set $\mathsf{epoch\_num}_1 = 1$, $\mathsf{sent\_msgnum}_1 = 0$, and $\mathsf{rcv\_msgnum}_1 = 0$.
3. If there already was a successful $\mathtt{ReceiveMessage}$ for $h$ (i.e there is a record $(\mathtt{Authenticate}, h, c', 1)$ for some $c'$), or this ciphertext previously failed to authenticate (i.e. a record $(\mathtt{Authenticate}, h, c, 0)$ exits), output $(\mathtt{ReceiveMessage}, h, c, \mathtt{Fail})$ to $\mathsf{pid}$.
4. If $\mathsf{epoch\_id}$ appears as a key in $\mathsf{id\_dict}$, set $\mathsf{epoch\_num} = \mathsf{id\_dict}[\mathsf{epoch\_id}]$.
   Else: //this is a new epoch id that hasn't been generated within SendMessage
   - If $\mathsf{sent\_msgnum}_i = 0$, output $(\mathtt{ReceiveMessage}, h, c, \mathtt{Fail})$ to $\mathsf{pid}$. //pid is in a receiving state and hasn't sent any messages in its current sending epoch, so it should not be accepting messages with a new epoch id.
   - Otherwise set $\mathsf{epoch\_num} = \mathsf{epoch\_num}_i + 1$.
     //this temporary variable will never be made permanent if decryption is unsuccessful.
5. If $\mathsf{msg\_num} > \mathsf{N\_dict}[\mathsf{epoch\_num}]$, output $(\mathtt{ReceiveMessage}, h, c, \mathtt{Fail})$ to $\mathsf{pid}$
   //For $\mathsf{epoch\_num}$'s that are not finished yet, the $\mathsf{N\_dict}$ returns a default value of $\infty$, so this check passes automatically.
6. If ($\mathsf{diverge\_parties}$ = *false* and $\mathtt{inject} \notin \mathsf{advControl}[\mathsf{epoch\_num}]$): Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, \mathsf{pid}, h, c)$ //Honest Case
7. Else: send backdoor message $(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, \mathsf{pid}, h, c)$ to $\mathcal{A}$ // $\mathcal{F}_{\mathsf{SM}}$ is asking the adversary for advice on how to decrypt $c$.
8. On receiving $(\mathsf{state}'_{\mathcal{I}}, \mathtt{inject}, h, c, v)$ from $\mathcal{A}$ or $\mathcal{I}$ update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$ and:
   If $(\mathsf{sender}, h, c, m)$ is recorded then record $(\mathtt{Authenticate}, \mathsf{pid}, h, c, 1)$ and set $m^* = m$. Else:
   - If $v = \perp$: record $(\mathtt{Authenticate}, \mathsf{pid}, h, c, 0)$ and output $(\mathtt{ReceiveMessage}, h, c, \mathtt{Fail})$.
   - If $v \neq \perp$ and $\mathsf{diverge\_parties}$ = *false* and $\mathtt{inject} \notin \mathsf{advControl}[\mathsf{epoch\_num}]$, then:
     – If there is no record $(\mathsf{sender}, h, c^*, m)$ for header $h$, output $(\mathtt{ReceiveMessage}, h, c, \mathtt{Fail})$. //since $h$ contains $N$, this value will match the view of the sender if this check succeeds.
     – Else (there is such a record), record $(\mathtt{Authenticate}, h, c, 1)$ and set $m^* = m$. //allowing for authenticating a message with a different mac

     If $v \neq \perp$ and ($\mathsf{diverge\_parties}$ = *true* or $\mathtt{inject} \in \mathsf{advControl}[\mathsf{epoch\_num}]$)), then:
     – Record $(\mathtt{Authenticate}, h, c, 1)$, and set $m^* = v$.
     – If $\mathsf{epoch\_id}$ does not appear as a key in $\mathsf{id\_dict}$ then set $\mathsf{diverge\_parties}$ = *true*. //diverge parties is being set here.
9. If $\mathsf{epoch\_num}_i < \mathsf{epoch\_num}$, do: //we only get to this step if decryption is successful
   - Set $\mathsf{N\_dict}[\mathsf{epoch\_num} - 2] = N$, $\mathsf{epoch\_num}_i \mathrel{+}= 2$, $N\_\mathsf{self}_i = \mathsf{sent\_msgnum}_i$, and $\mathsf{sent\_msgnum}_i = 0$.
   - if $\mathsf{diverge\_parties}$ = *false* then:
     – If $\mathsf{advControl}[\mathsf{epoch\_num} - 1] = \{\mathtt{leak}, \mathtt{inject}\}$ and $\mathsf{epoch\_num}_i \notin \mathsf{corruptions}_i$, then set $\mathsf{advControl}[\mathsf{epoch\_num}] = \{\mathtt{leak}\}$. //Corruption status is changed if this is the other party's first new sending epoch that involves a fresh epoch id generated after corruption.
     – If $\mathsf{advControl}[\mathsf{epoch\_num} - 1] = \{\mathtt{leak}\}$, then set $\mathsf{advControl}[\mathsf{epoch\_num}] = \perp$.
10. Output $(\mathtt{ReceiveMessage}, h, m^*)$ to $\mathsf{pid}$.

Figure 7: The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$

In addition, $\mathcal{F}_{\mathsf{SM}}$ stores the number of messages sent in previous epochs so that honest parties can detect if an adversary has injected a message after the planned ending to a particular epoch.

**Corrupt.** The response of $\mathcal{F}_{\mathsf{SM}}$ to party corruption inputs is the core of the security guarantees it provides. The goal is to bound the effect of exposure of the local states of the parties on the loss of security and, and provide guarantees as to how soon security of the communication is restored (if at all).

Before describing the actual behavior of $\mathcal{F}_{\mathsf{SM}}$ we note two ways in which our modeling of corruptions differs from the traditional. First, corruption is captured as an instantaneous exposure event, as opposed to having separate *corrupt* and *leave* events. Second, corruptions are modeled as inputs (coming from the environment, as opposed to the traditional UC modeling corruption as message coming from the adversary. Both of these changes simplify the mechanics and yield a cleaner model while not restricting the adversary's power.

In $\mathcal{F}_{\mathsf{SM}}$ when a party is corrupted, the adversary learns nothing about all messages that have been sent and received by the party until the point of corruption. Furthermore, the adversary obtains no other information on the history of the session such as its duration. This guarantees forward secrecy of a protocol that realises $\mathcal{F}_{\mathsf{SM}}$.

The point by which a compromised party becomes uncompromised is Signal-specific, and is determined by inspecting the sequence of encapsulation and decapsulation activations of that two parties. More specifically, Signal partitions the messages sent in a session into *sending epochs,* where each sending epoch is associated with one of the two parties, and consists of all the messages sent by that party from the end of its previous sending epoch until the first time this party successfully decapsulates an incoming message that belongs to the peer's latest sending epoch. A compromised party becomes uncompromised as soon as it has started *two* new sending epochs since the last corruption event. After this point, the adversary receives no additional information about the messages sent and received by the parties. This guarantees post-compromise security of any protocol that realises $\mathcal{F}_{\mathsf{SM}}$.

Now, on input $(\texttt{Corrupt}, i)$, where $i \in \{0, 1\}$, $\mathcal{F}_{\mathsf{SM}}$ sends to the adversary the plaintexts of all the messages that have been sent by $\mathsf{pid}_{1-i}$ to $\mathsf{pid}_i$ and have not yet been received by $\mathsf{pid}_i$ at the time of corruption. In response, $\mathcal{F}_{\mathsf{SM}}$ obtains from the adversary an opaque string $S$ and reports $S$ to the corrupting entity. ($S$ represents the simulated local state of the corrupted party.) The part represents the forwards secrecy guarantees provided by $\mathcal{F}_{\mathsf{SM}}$, namely the secrecy of messages sent and received by $\mathsf{pid}_i$ prior to the corruption event.

In addition, $\mathcal{F}_{\mathsf{SM}}$ marks $\mathsf{pid}_i$ as compromised. Party $\mathsf{pid}_i$ resumes its uncompromised status once it has started a new sending epoch *for the second time* following the corruption event. This part represents the backwards secrecy guarantees, namely the event after which security is restored. (In fact, our notion of compromise is a bit more fine-tuned: in the period between the first and second new sending epochs following a corruption event, we say the party is *recovering.*)

The effect of being compromised consists of three parts. First, at any `SendMessage` that takes place where one of the parties is either compromised, or recovering, the adversary learns the message.

Second, at any `ReceiveMessage` where one of the parties is either compromised or recovering, the adversary can cause the receiving party to accept any pair of plaintext and header, as long as the header is "legitimate" (i.e., it corresponds to an actual epoch where the receiver is indeed the receiving party, the header was not part of a successfully received message, and the message number in that epoch is below a known bound).

Third, the adversary may attempt a person-in-the-middle attack; this will succeed if and only if the honest recipient accepts a new epoch created by the adversary rather than the honest sender. In more detail, if party $\mathsf{pid}_{1-i}$ calls $\mathcal{F}_{\mathsf{SM}}$ to receive an incoming ciphertext $(h, c)$, where $h$ specifies an epoch ID $\mathsf{epoch\_id}$ that corresponds to a new epoch, whereas party $\mathsf{pid}_i$ has already sent messages for that epoch but with a different epoch ID $\mathsf{epoch\_id}'$, *and in addition* $\mathsf{pid}_i$ *is compromised (but not recovering),* then the addversary can instruct $\mathcal{F}_{\mathsf{SM}}$ to successfully decrypt $(h, c)$. In that case, $\mathcal{F}_{\mathsf{SM}}$ notes that the session is *forked* — or in other words, that the parties have diverged. In this case, both parties become compromised for the rest of the session. This event represents a complete break of security of the session. On the other hand if the honest sender's message starts a new epoch, then we are on track to recovery but the messages are not yet fully equivocal; the recovery period requires 1 more epoch to achieve equivocation and return to a fully uncompromised state.

## 4 Overview of our Modular Decomposition

This work provides a modular analysis of Signal's protocol. In this section provide more details about our modular, iterative process for decomposing the Signal architecture into a collection of ideal modules that each address a specific purpose. In essence, this section summarizes the theorems that collectively serve to instantiate $\mathcal{F}_{\mathsf{SM}}$, and it serves as an organization for the rest of the paper. (While reading this section, we encourage readers to review Fig. 1 on page 9, which graphically depicts the various components and how they fit together.)

**A Signal-style realization of $\mathcal{F}_{\mathsf{SM}}$.** In our first instantiation shown at the top left of Fig. 1, we decompose the Signal protocol into two ideal modules that model the interconnected pieces of the double ratchet: (1) a long lived $\mathcal{F}_{\mathsf{eKE}}$ component that models the asymmetric ratcheted key exchange in the Signal architecture and (2) multiple copies of short lived $\mathcal{F}_{\mathsf{fs\_aead}}$ components that model unidirectional forward secure authenticated channels, each representing the combination of authenticated encryption (with associated data) and the symmetric key ratchet from the Signal architecture, for handling the messages associated with a single epoch. Protocol $\Pi_{\mathsf{SGNL}}$ uses these modules to realize $\mathcal{F}_{\mathsf{SM}}$, following the overall logic of Signal's architecture

The long lived key exchange module $\mathcal{F}_{\mathsf{eKE}}$ contributes two main features of the overall protocol: (1) healing from compromise via addition of randomness, (2) long term forward secrecy. Meanwhile, the short lived forward secure encryption modules $\mathcal{F}_{\mathsf{fs\_aead}}$ provides the overall protocol with forward secrecy at the level of messages while its dependence on $\mathcal{F}_{\mathsf{eKE}}$ in an authenticated way enables the overall protocol to realise the longterm forward secrecy and healing properties endowed by $\mathcal{F}_{\mathsf{eKE}}$. Next, we will briefly describe these modules along with our main theorem. Full descriptions as well as a rigorous proof of the theorem can be found in Section 5.

- *Signal Protocol (*$\Pi_{\mathsf{SGNL}}$*, Fig. 11)*: The protocol $\Pi_{\mathsf{SGNL}}$ (see Section 5.3) is the top-level protocol that interfaces with our ideal functionalities $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{fs\_aead}}$, as well as the global subroutines $\mathcal{F}_{\mathsf{LTM}}$, $\mathcal{F}_{\mathsf{pRO}}$, and $\mathcal{F}_{\mathsf{DIR}}$. There are three primary takeaways from the design of $\Pi_{\mathsf{SGNL}}$: it has the same input-output API as our ideal functionality $\mathcal{F}_{\mathsf{SM}}$, it displays a idealized version of the double ratchet with clearly distinct roles for the two subroutines, and finally it moves closer toward realism. Added features at this level of abstraction include key material stored within party states, explicit accounting for out-of-order messages by holding onto missed message keys, and epochs being identified directly by their $\mathsf{epoch\_id}$ rather than an idealized $\mathsf{epoch\_num}$ ordering.
- *Epoch Key Exchange (*$\mathcal{F}_{\mathsf{eKE}}$*, Fig. 8)*: The epoch key exchange functionality comprises the public key "backbone" of the secure messaging continuous key agreement. The functionality

represents the root ratchet of the Signal protocol, it is persistent during the entire session, mapping $(\mathsf{epoch\_id}_0, \mathsf{epoch\_id}_1)$ pairs to to sending and receiving chain keys for the symmetric ratchet.

- *Forward Secure Authenticated Encryption ($\mathcal{F}_{\mathsf{fs\_aead}}$, Fig. 10)*: Each instance of the forward secure authenticated encryption functionality models the encryption and decryption of messages for a single epoch of the secure messaging system. In each epoch, one of the parties may only send messages while the other may only receive messages. As the name suggests, the forward secure encryption functionality provides forward security by allowing each message to be decrypted exactly once. Additionally, the protocol $\Pi_{\mathsf{SGNL}}$ belonging to the sender (resp. receiver) for the epoch may send a `StopEncrypting` (resp. `StopDecrypting`) request to the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance at which point no more messages may be encrypted (resp. decrypted) in this epoch by anyone. This way the functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ allows for the 'expiry' of the epoch it represents once the parties move on in the conversation.

Within Section 5, we provide a rigorous specification for each of these functionalities. Then, we provide a concrete simulator and show that (together with $\mathcal{F}_{\mathsf{SM}}$) it is perfectly indistinguishable from the $\Pi_{\mathsf{SGNL}}$ hybrid world.

**Theorem 2** *Protocol $\Pi_{\mathsf{SGNL}}$ (perfectly) UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{SM}}$ in the presence of $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$.*

**Instantiating the public ratchet (realizing $\mathcal{F}_{\mathsf{eKE}}$).** Instantiating $\mathcal{F}_{\mathsf{eKE}}$ in a modular fashion is one of the most delicate parts of this work. One main challenge, as observed by Alwen et al. [1] and others, is that the key derivation module within the public ratchet must maintain security if either of the previous root key or the newly generated ephemeral keys are uncompromised. Alwen et al. formalized this guarantee by way of constructing a new primitive: a PRF-PRG.

As discussed in the Introduction, we point to an additional challenge to instantiating $\mathcal{F}_{\mathsf{eKE}}$, namely that of asserting security at the presence of adaptively chosen key exposures. In Section 6.1 we formulate a new primitive primitive called a *Cascaded PRF-PRG (CPRFG)* that extends the PRF-PRG concept to provide also the equivocation needed to handle adaptive state exposures (Def. 11). We then provide two instantiations of CPRFGs: an immediate one in the random oracle model, and another one, in the plain model, based on punctured PRFs (Thm. 12).

Using this CPRFG construction, we then provide a UC analysis of the public ratchet; see Section 6 for details.

**Theorem 3** *Assume that $KDF : \{0,1\}^n \to \{0,1\}^{2n}$ is a CPRFG, that the DDH assumption holds in the group $G$. Then protocol $\Pi_{eKE}$ UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ in the presence of global functionalities $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$.*

**Instantiating unidirectional forward secure authenticated channels (realizing $\mathcal{F}_{\mathsf{fs\_aead}}$).** In the top right part of Fig. 1, we decompose the symmetric key component of Signal into two smaller pieces: a message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$ that interfaces with the epoch key exchange to produce the symmetric chain keys, and a one-time-use authenticated encryption routine. This decomposition spans Sections 7 to 9 in the paper.

- *Message Key Exchange ($\mathcal{F}_{\mathsf{mKE}}$, Fig. 20)*: Each message key exchange functionality instance handles the key derivation for the symmetric ratchet for a particular epoch. Specifically, it provides $\mathsf{key\_seed}$'s to $\Pi_{\mathsf{aead}}$ instances that are then expanded to any length using the global random oracle $\mathcal{F}_{\mathsf{pRO}}$. The functionality also closes epochs at a certain message number N when instructed to by $\Pi_{\mathsf{fs\_aead}}$ by generating all $\mathsf{key\_seed}$'s up to N and later disallowing the generation of any further key seeds for its epoch. The protocol ($\Pi_{\mathsf{mKE}}$, Fig. 25) realizes $\mathcal{F}_{\mathsf{mKE}}$ by iteratively applying a length-doubling pseudorandom function to the $\mathsf{chain\_key}$ provided by $\Pi_{\mathsf{eKE}}$ to generate $\mathsf{key\_seed}$'s. If it needs to skip message key seeds (for example, if the messages arrive out of order), $\Pi_{\mathsf{eKE}}$ applies the PRG several times until reaching the correct $\mathsf{key\_seed}$, meanwhile storing intermediate $\mathsf{key\_seed}$'s. To close an epoch at a particular message number N, it generates all message key seeds up to N and then deletes the $\mathsf{chain\_key}$. The functionality (and protocol) enforces the forward security guarantee by: deleting key seeds for messages that have been retrieved, ensuring that message keys look independent and random from each other (using the PRG), and deleting the $\mathsf{chain\_key}$ when the parties have moved on to future epochs (to prevent message injection in old epochs).

- *Authenticated Encryption with Associated Data ($\mathcal{F}_{\mathsf{aead}}$, Fig. 21)*: Each authenticated encryption functionality instance handles the encryption, decryption, and authentication of a particular message for a particular epoch and hands the ciphertext or message back to $\Pi_{\mathsf{fs\_aead}}$. It gets a $\mathsf{key\_seed}$ from $\Pi_{\mathsf{mKE}}$ and then asks the adversary to provide a ciphertext $c$ (while leaking either $|m|$ or $m$ depending on whether the epoch is compromised). For decryption, if it gets the same ciphertext back, $\mathcal{F}_{\mathsf{aead}}$ returns message $m$. If it gets a different ciphertext $c' \neq c$, it asks the adversary whether it wants to inject a message. Depending on the corruption status, $\mathcal{F}_{\mathsf{aead}}$ will either return $\mathcal{A}$'s message or return $m$, or return a failure. The protocol ($\Pi_{\mathsf{aead}}$, Fig. 27) realizes $\mathcal{F}_{\mathsf{aead}}$ by querying the random oracle $\mathcal{F}_{\mathsf{pRO}}$ on the $\mathsf{key\_seed}$ to get the full $\mathsf{msg\_key}$. It then computes the ciphertext $c$ using a One Time Pad and then a secure message authentication code to authenticate $c$ as well as its $\mathsf{sid}$ (which contains information about the $\mathsf{pid}$'s, $\mathsf{epoch\_id}$, $\mathsf{msg\_num}$, and such). To decrypt, $\Pi_{\mathsf{aead}}$ similarly gets the $\mathsf{key\_seed}$ from $\mathcal{F}_{\mathsf{mKE}}$ and queries $\mathcal{F}_{\mathsf{pRO}}$ to expand it. Then, $\Pi_{\mathsf{aead}}$ decrypts the ciphertext only if the tag verifies.

Within Section 7, we rigorously define all of the functionalities described above and their associated instantiations as cryptographic protocols. We also formally prove that our hybrid world UC-realizes a real world containing only cryptographic protocols rather than functionalities (albeit still in the presence of the global subroutines).

**Theorem 4** *Protocol $\Pi_{\mathsf{fs\_aead}}$ (perfectly) UC-realizes $\mathcal{F}_{\mathsf{fs\_aead}}$, in the presence of $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$, and $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.*

**Instantiating the Symmetric Ratchet (realizing $\mathcal{F}_{\mathsf{mKE}}$).** By this point we have already described all of the functionalities in our model. As shown in Figure 1, it only remains to construct real-world protocols that realize each of them. Unlike with $\mathcal{F}_{\mathsf{eKE}}$, our instantiations of symmetric functionalities are relatively straightforward. In Section 8, we provide a simple instantiation of $\mathcal{F}_{\mathsf{mKE}}$ based on Signal's symmetric ratcheting algorithm.

**Theorem 5** *Assume that $\mathsf{PRG}$ is a secure length-doubling pseudorandom generator. Then protocol $\Pi_{\mathsf{mKE}}$ UC-realizes $\mathcal{F}_{\mathsf{mKE}}$ in the presence of $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$, and $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.*

**Instantiating authenticated encryption for single messages (realising $\mathcal{F}_{\mathsf{aead}}$).** Additionally, $\mathcal{F}_{\mathsf{aead}}$ is a slight variant of the *secure message transmission* functionality $\mathcal{F}_{\mathsf{SMT}}$ that has been analyzed in the original work of Canetti [22] that introduced the UC security framework. We show the following theorem in Section 9.

**Theorem 6** *Assuming the unforgeability of* $(\mathsf{MAC}, \mathsf{Verify})$*, protocol* $\Pi_{\mathsf{aead}}$ *UC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{aead}}$ *in the presence of* $\mathcal{F}_{\mathsf{pRO}}$*, as well as* $\mathcal{F}^{\Pi_{\mathsf{mKE}}}_{\mathsf{mKE}}$*.*
$\left(\mathcal{F}^{\Pi_{\mathsf{mKE}}}_{\mathsf{mKE}} = (\mathcal{S}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{mKE}}), \mathcal{F}^{\Pi_{\mathsf{eKE}}}_{\mathsf{eKE}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}}), \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{lib}}\right)$.

**Putting it all together (composition theorems).** It is left to put the pieces together and assert the security of the composed protocol as shown in Figure 1. We first use Theorems 3 and 5, together with Proposition 1 to deduce that $\Pi_{\mathsf{mKE}}$ UC-realizes $\mathcal{F}_{\mathsf{mKE}}$ in the presence of $\Pi_{\mathsf{eKE}}$, as well as $\mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}$:

**Corollary 7** *Assume that the KDF module used by* $\Pi_{\mathsf{eKE}}$ *is a CPRFG, that the DDH assumption holds for the group* $G$ *used in* $\mathcal{F}_{\mathsf{LTM}}$ *and* $\Pi_{\mathsf{eKE}}$*, and that the* $\mathsf{PRG}$ *used in* $\Pi_{\mathsf{mKE}}$ *is a secure length-doubling pseudorandom generator. Then protocol* $\Pi_{\mathsf{mKE}}$ *UC-realizes* $\mathcal{F}_{\mathsf{mKE}}$ *in the presence of* $\Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}$*.*

Next we use Theorem 6 together with Corollary 7 and Lemma 1 to deduce that $\Pi_{\mathsf{aead}}$ UC-realizes $\mathcal{F}_{\mathsf{aead}}$ in the presence of $\Pi_{\mathsf{mKE}}, \Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}$.

**Corollary 8** *Assume that* $(\mathsf{MAC}, \mathsf{Verify})$ *used in* $\Pi_{\mathsf{aead}}$ *is unforgeable, that the KDF module used by* $\Pi_{\mathsf{eKE}}$ *is a CPRFG, that the DDH assumption holds for the group* $G$ *used in* $\mathcal{F}_{\mathsf{LTM}}$ *and* $\Pi_{\mathsf{eKE}}$*, and that the* $\mathsf{PRG}$ *used in* $\Pi_{\mathsf{mKE}}$ *is a secure length-doubling pseudorandom generator. Then protocol* $\Pi_{\mathsf{aead}}$ *UC-realizes* $\mathcal{F}_{\mathsf{aead}}$ *in the presence of* $\Pi_{\mathsf{mKE}}, \Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{pRO}}$*.*

Now, recall that $\Pi^{\mathcal{F}_{\mathsf{mKE}} \to \Pi_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{aead}} \to \Pi_{\mathsf{aead}}}_{\mathsf{fs\_aead}}$ is the protocol that's identical to $\Pi_{\mathsf{fs\_aead}}$, except that calls to $\mathcal{F}_{\mathsf{mKE}}$ are replaced with calls to $\Pi_{\mathsf{mKE}}$ and calls to $\mathcal{F}_{\mathsf{aead}}$ are replaced with calls to $\Pi_{\mathsf{aead}}$. Then Theorem 4, together with Corollary 8 and the UC with Global Subroutines (UCGS) theorem says that:

**Corollary 9** *Assume that* $(\mathsf{MAC}, \mathsf{Verify})$ *used in* $\Pi_{\mathsf{aead}}$ *is unforgeable, that the KDF module used by* $\Pi_{\mathsf{eKE}}$ *is a CPRFG, that the DDH assumption holds for the group* $G$ *used in* $\mathcal{F}_{\mathsf{LTM}}$ *and* $\Pi_{\mathsf{eKE}}$*, and that the* $\mathsf{PRG}$ *used in* $\Pi_{\mathsf{mKE}}$ *is a secure length-doubling pseudorandom generator. Then protocol* $\Pi^{\mathcal{F}_{\mathsf{mKE}} \to \Pi_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{aead}} \to \Pi_{\mathsf{aead}}}_{\mathsf{fs\_aead}}$ *UC-realizes* $\mathcal{F}_{\mathsf{fs\_aead}}$ *in the presence of* $\Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{pRO}}$*.*

Finally, recall that $\Pi^{\mathcal{F}_{\mathsf{eKE}} \to \Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{fs\_aead}} \to \Pi^{\mathcal{F}_{\mathsf{mKE}} \to \Pi_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{aead}} \to \Pi_{\mathsf{aead}}}_{\mathsf{fs\_aead}}}_{\mathsf{SGNL}}$ is the protocol that's identical to $\Pi_{\mathsf{SGNL}}$, except that calls to $\mathcal{F}_{\mathsf{eKE}}$ are replaced with calls to $\Pi_{\mathsf{eKE}}$ and calls to $\mathcal{F}_{\mathsf{fs\_aead}}$ are replaced with calls to $\Pi^{\mathcal{F}_{\mathsf{mKE}} \to \Pi_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{aead}} \to \Pi_{\mathsf{aead}}}_{\mathsf{fs\_aead}}$. Then Theorem 2, together with Corrolary 9 and the UCGS theorem says that:

**Corollary 10** *Assume that* $(\mathsf{MAC}, \mathsf{Verify})$ *used in* $\Pi_{\mathsf{aead}}$ *is unforgeable, that the KDF module used by* $\Pi_{\mathsf{eKE}}$ *is a CPRFG, that the DDH assumption holds for the group* $G$ *used in* $\mathcal{F}_{\mathsf{LTM}}$ *and* $\Pi_{\mathsf{eKE}}$*, and that the* $\mathsf{PRG}$ *used in* $\Pi_{\mathsf{mKE}}$ *is a secure length-doubling pseudorandom generator. Then protocol* $\Pi^{\mathcal{F}_{\mathsf{eKE}} \to \Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{fs\_aead}} \to \Pi^{\mathcal{F}_{\mathsf{mKE}} \to \Pi_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{aead}} \to \Pi_{\mathsf{aead}}}_{\mathsf{fs\_aead}}}_{\mathsf{SGNL}}$ *UC-realizes* $\mathcal{F}_{\mathsf{SM}}$ *in the presence of* $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$*.*

## 5 A Signal-style Secure Messaging Protocol: Realizing $\mathcal{F}_{\mathsf{SM}}$

In this section we decompose Signal into two ideal modules ($\mathcal{F}_{\mathsf{eKE}}$, $\mathcal{F}_{\mathsf{fs\_aead}}$) and a protocol that glues them together ($\Pi_{\mathsf{SGNL}}$). We then prove that these pieces together form an instantiation of the secure messaging functionality $\mathcal{F}_{\mathsf{SM}}$. We begin by describing a long-lived component $\mathcal{F}_{\mathsf{eKE}}$ in Section 5.1 that models the public key ratchet in the Signal architecture. Next, we present a short-lived component $\mathcal{F}_{\mathsf{fs\_aead}}$ in Section 5.2 that models a unidirectional forward secure authenticated channel. (This will later, in Section 7, be instantiated using authenticated encryption and a symmetric key ratchet component in the style of the Signal architecture.) In Section 5.3 we present the protocol $\Pi_{\mathsf{SGNL}}$ that glues together a long lived $\mathcal{F}_{\mathsf{eKE}}$ and several short lived $\mathcal{F}_{\mathsf{fs\_aead}}$. Finally, in Section 5.4 we prove that the signal like protocol $\Pi_{\mathsf{SGNL}}$ realises the secure messaging functionality we presented in Section 3.2.

### 5.1 The Epoch Key Exchange Functionality $\mathcal{F}_{\mathsf{eKE}}$

The fundamental security objective of the epoch key exchange functionality (see Figure 8 on page 33) is to provide recovery from an adversarial state corruption. This requires that the parties' secret keys are updated periodically with new random values. Intrinsic to the security goal of post-compromise security and periodic updates is the concept of *epochs*: agreed-upon points in the conversation to re-randomize the secret keys. The functionality $\mathcal{F}_{\mathsf{eKE}}$ presented in this section is a long lived functionality that allows the parties to refresh their randomness every epoch. With this functionality we capture the properties provided to the Signal architecture by their public key ratchet.

An important usability feature of the public key ratchet from the Signal Architecture is that an online party can refresh their own randomness unilaterally, producing a new key that can be used for messaging without even requiring the other party to be online at the same time. Then, when the other party comes online, they are able to complete the re-randomization of the secret keys and therefore the healing from prior corruption events. To achieve this, their protocol enforces that the parties must take turns refreshing their randomness – ensuring that parties are able to assimilate the randomness introduced in the right order. When $\mathsf{pid}_i$ receives a ciphertext from $\mathsf{pid}_{1-i}$ (through $\mathcal{F}_{\mathsf{fs\_aead}}$) marked with an $\mathsf{epoch\_id}$ that it has not seen before, $\mathsf{pid}_i$ knows that $\mathsf{pid}_{1-i}$ has started a new epoch. The party $\mathsf{pid}_i$ then does a *tentative* ratcheting step using the $\mathsf{epoch\_id}$ to derive its new keys (without deleting its old keys right away). This new key must then be verified by $\mathsf{pid}_i$. If the verification succeeds, $\mathsf{pid}_i$ *confirms* the new epoch and updates its state accordingly, otherwise the party deletes any temporary variables it computed and remains in the same sending and receiving epochs. Our functionality models the exact same behavior.

Each instance of the functionality $\mathcal{F}_{\mathsf{eKE}}$ has a parties $\mathsf{pid}_0, \mathsf{pid}_1$ inherited from the overall protocol, where $\mathsf{pid}_0$ is the party that initiates the overall conversation. Additionally, each epoch within the instance has exactly one sender and one receiver, with the parties alternating between the two roles. (i.e. $\mathsf{pid}_0$ is the sender for all even numbered epochs and $\mathsf{pid}_1$ is the sender for all odd numbered epochs.) While the epoch numbers exist internally within the functionality, the epochs are only ever externally identified by the $\mathsf{epoch\_id}$ generated by the sending party. [4] An epoch id is a unique adversarially chosen value (in Signal, the $\mathsf{epoch\_id}$ is the sender's public Diffie-Hellman key). The adversarial choice of epoch id models the fact that we have no requirements from this value beyond uniquely identifying epochs.

[4]This way of identifying the epochs provides the property of revealing as little as possible about the history of the communication. The Signal protocol achieves this property by requiring that the parties store only epoch ids and never track epoch numbers at all.

At initialization, the epoch key exchange functionality checks that parties are registered in the global directory functionality $\mathcal{F}_{\mathsf{DIR}}$ and that the responding party has fresh one time keys available. It then makes a call to its long term module functionality $\mathcal{F}_{\mathsf{LTM}}$ to bind both parties' directory keys to the conversation. The directory and long term module functionalities $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$ are used by to provide the identity binding automatically assumed in $\mathcal{F}_{\mathsf{eKE}}$. Finally, $\mathcal{F}_{\mathsf{eKE}}$ makes a call to $\mathcal{F}_{\mathsf{lib}}$ to obtain the adversarial code $\mathcal{I}$ that will be run internally by the functionality to allow adversarial choice of all epoch ids without actually relinquishing control to the adversary unless a party is compromised. ($\mathcal{I}$ will also be used to allow adversarial choice of receiving keys when the receiver inputs an incorrect epoch id but neither party is compromised.) This mechanism of internally running adversarial code within our functionalities allows us to model the immediate decryption property of the Signal protocol. Overall, the epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$ models the forward secrecy and post-compromise security guarantees of the public key ratchet in the Signal protocol, but *without providing any epoch keys* at the level of $\Pi_{\mathsf{SGNL}} + \mathcal{F}_{\mathsf{fs\_aead}} + \mathcal{F}_{\mathsf{mKE}}$.

In addition to providing epoch ids, the epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$ also provides keys at the level where $\mathcal{F}_{\mathsf{fs\_aead}}$ and $\mathcal{F}_{\mathsf{mKE}}$ have been instantiated by $\Pi_{\mathsf{fs\_aead}}$ and $\Pi_{\mathsf{mKE}}$ respectively. At this level, when a $\Pi_{\mathsf{mKE}}$ protocol instance belonging to $\mathsf{pid}_i$ sends a `GetReceivingKey` request and a new $\mathsf{epoch\_id}$ to the functionality, a new receiving key for the epoch is produced for this party. The receiving key will only match the sender's key if the epoch id is the correct one. This way the functionality $\mathcal{F}_{\mathsf{eKE}}$ enables the calling party to identify that the provided receiving epoch id is not the correct one without directly communicating it. (In fact, as a sub-component of $\Pi_{\mathsf{SGNL}}$, $\mathcal{F}_{\mathsf{eKE}}$ receives such a request when the party $\mathsf{pid}_i$ is using an instantiation $\Pi_{\mathsf{fs\_aead}}+\Pi_{\mathsf{mKE}}+\mathcal{F}_{\mathsf{aead}}$ of $\mathcal{F}_{\mathsf{fs\_aead}}$ to check that the ciphertext and its associated information decrypts (and authenticates) successfully under the key produced for the received epoch id.)[5] If the verification succeeds (either by a protocol using the provided key or by a functionality directly checking whether the epoch id's match), $\mathcal{F}_{\mathsf{eKE}}$ expects a `ConfirmRcvEpoch` request directly from the party; When it receives this request, it replaces the the party's current receiving epoch id with the one that was confirmed. It then allows the adversarially provided code to choose a brand new sending epoch id for this party with which to send any future messages. The choice of id by the adversary simply represents the fact that this value is an identifier; the epoch id is able to provide the properties needed by the functionality $\mathcal{F}_{\mathsf{eKE}}$ without needing to be hidden or to be chosen by the functionality. If the verification fails, the functionality simply does not receive a `ConfirmRcvEpoch` request and does not update the relevant variables.

We now describe the input output behavior of the $\mathcal{F}_{\mathsf{eKE}}$ in detail. (The pseudocode for this functionality can be found in Fig. 8.) The epoch key exchange functionality takes inputs from and provide output to the corresponding two machines of $\Pi_{\mathsf{SGNL}}$, namely $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_0)$ and $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_1)$, (as well as $(\Pi_{\mathsf{mKE}}, \text{"}mKE\text{"}, \mathsf{sid}.fs = (\text{"}fs\_aead\text{"}, \mathsf{sid}' = (\mathsf{sid}, \mathsf{pid}_0), \mathsf{epoch\_id}))$ and $(\Pi_{\mathsf{mKE}}, \text{"}mKE\text{"}, \mathsf{sid}.fs = (\text{"}fs\_aead\text{"}, \mathsf{sid}' = (\mathsf{sid}, \mathsf{pid}_1), \mathsf{epoch\_id}))$ once $\mathcal{F}_{\mathsf{fs\_aead}}$ and its subcomponent $\mathcal{F}_{\mathsf{mKE}}$ have been instantiated by their respective protocols) and the adversary $\mathcal{A}$. Inputs coming from other sources are ignored. Recall that at first activation, $\mathcal{F}_{\mathsf{eKE}}$ checks that parties are registered in $\mathcal{F}_{\mathsf{DIR}}$, it checks that the responding party has fresh one time keys available, and it makes a call to $\mathcal{F}_{\mathsf{lib}}$ to obtain the adversarial code $\mathcal{I}$. The adversarial code $\mathcal{I}$ allows us to model the immediate decryption property of secure messaging protocols when parties have not been corrupted. In this case, the choices made by the adversary are fixed based on the protocol and are not allowed to be adaptive based on the honest keys chosen during the particular run of the architecture.

The functionality has four interfaces, which we describe next.

[5]Note that no keys are used and therefore this request is never sent to $\mathcal{F}_{\mathsf{eKE}}$ even at the level of $\Pi_{\mathsf{SGNL}}+\mathcal{F}_{\mathsf{eKE}}+(\Pi_{\mathsf{fs\_aead}}+\mathcal{F}_{\mathsf{mKE}}+\mathcal{F}_{\mathsf{aead}}.)$

**Confirm Receiving Epoch** After initialization, the epoch key exchange functionality expects an input of the form $(\texttt{ConfirmRcvEpoch}, \mathsf{epoch\_id}^*)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ only after a message has been successfully received using $\mathsf{epoch\_id}^*$ in a new receiving epoch. This input triggers the functionality to update its state according to the receiving epoch id being confirmed and provide a new adversarially chosen sending epoch id to the party. On the first activation $\mathcal{F}_{\mathsf{eKE}}$ gets the first receiving epoch id from the global directory $\mathcal{F}_{\mathsf{DIR}}$. For all other epochs, the receiving epoch id is updated to the value $\mathsf{epoch\_id}^*$ from the request. The state update carried out within the functionality depends on whether $\mathsf{epoch\_id}^*$ matches the sending epoch id of the party that $\mathsf{pid}$ is talking to.

When $\mathsf{epoch\_id}^*$ matches the epoch identifier of the other party, then the functionality runs the adversarially provided code $\mathcal{I}$ to choose a new epoch id for the party's next sending epoch. Otherwise, if $\mathsf{epoch\_id}^*$ is *not* the current sending epoch identifier for the other party (for example, if the adversary provided an alternative epoch identifier under which the ciphertext successfully authenticates) the functionality no longer provides any guarantees. To model this outcome, the variable $\mathsf{diverge\_parties}$ is set to true in this case and the adversary $\mathcal{A}$ is allowed to choose all epoch ids and chain keys for both parties for the rest of time. In the protocol, if the parties' epoch identifiers diverge in this way, their symmetric keys diverge as well and cannot be restored. Note that in the Signal protocol, the adversary can carry out a person-in-the-middle attack against both parties simultaneously forever after just a single state compromise.

**Get Sending Key** As mentioned earlier, the epoch keys generated by $\mathcal{F}_{\mathsf{eKE}}$ (and $\Pi_{\mathsf{eKE}}$) are used by the Signal protocol to derive many message keys, one for each message within the epoch (These keys don't exist at the level of $\Pi_{\mathsf{SGNL}} + \mathcal{F}_{\mathsf{eKE}} + \mathcal{F}_{\mathsf{fs\_aead}}$. Hence, this will interface never be called at this level.) However, at the level where $\mathcal{F}_{\mathsf{fs\_aead}}$ and its sub-component $\mathcal{F}_{\mathsf{mKE}}$ have both been been instantiated, the functionality $\mathcal{F}_{\mathsf{eKE}}$ expects $\texttt{GetSendingKey}$ requests from the parties' $\Pi_{\mathsf{mKE}}$ instances. For each epoch, the functionality $\mathcal{F}_{\mathsf{eKE}}$ expects one $\texttt{GetSendingKey}$ request from a $\Pi_{\mathsf{mKE}}$ instance that belongs to the sender $\mathsf{pid}_i$ for the epoch. When no parties are compromised in the corresponding epoch, the functionality responds to this request with a uniformly chosen value $\mathsf{sending\_chain\_key}_i$ which it stores in its state. Otherwise, if any party is compromised during this epoch or if divergence has occurred, $\mathcal{F}_{\mathsf{eKE}}$ allows the adversary to choose the new sending chain key.

**Get Receiving Key** Similarly to the $\texttt{GetSendingKey}$ interface, this interface will also never be called at the level of $\Pi_{\mathsf{SGNL}} + \mathcal{F}_{\mathsf{eKE}} + \mathcal{F}_{\mathsf{fs\_aead}}$. This is because the epoch keys and message keys do not exist at this level of abstraction. At the level where $\mathcal{F}_{\mathsf{fs\_aead}}$ and its sub-component $\mathcal{F}_{\mathsf{mKE}}$ have both been instantiated, the functionality $\mathcal{F}_{\mathsf{eKE}}$ expects $\texttt{GetReceivingKey}$ requests from the parties' $\Pi_{\mathsf{mKE}}$ instances. For each epoch, the functionality $\mathcal{F}_{\mathsf{eKE}}$ expects a non-zero number of $\texttt{GetReceivingKey}$ requests from $\Pi_{\mathsf{mKE}}$ instances that belong to the receiver for the epoch. When either party is compromised for the epoch or when divergence has occurred, $\mathcal{F}_{\mathsf{eKE}}$ allows the adversary $\mathcal{A}$ to choose the key that it will output. Otherwise, the output key depends on whether the epoch id of the $\Pi_{\mathsf{mKE}}$ instance matches the senders epoch id for the epoch. When the epoch ids match, $\mathcal{F}_{\mathsf{eKE}}$ responds to this request with the value $\mathsf{sending\_chain\_key}_i$ which was provided to the sender of the epoch – thus preserving the symmetry of the keys. When the epoch ids don't match, $\mathcal{F}_{\mathsf{eKE}}$ responds with a key chosen by the latest adversarial code $\mathcal{I}$ uploaded by the adversary to $\mathcal{F}_{\mathsf{lib}}$ at the time of the first activation of the $\mathcal{F}_{\mathsf{eKE}}$ instance. Note importantly that $\mathcal{I}$ will not have access to the key provided to the sender of the epoch. The functionality $\mathcal{F}_{\mathsf{eKE}}$ leaves it to the instantiation of $\mathcal{F}_{\mathsf{fs\_aead}}$ to use the key chosen by $\mathcal{I}$ to figure out that $\Pi_{\mathsf{mKE}}$'s epoch id should not be confirmed as the new receiving epoch identifier. This models the fact that in the Signal protocol, naive tampering with the sender's public exponent will cause an unsuccessful temporary ratchet upon a failure of authentication (and later reversion to the previous epoch). In traditional instantiations of the Signal architecture, if $\mathcal{A}$ corrupts the receiver after naive tampering attempts, the adversary can use the party's stored root key to compute the $\mathsf{recv\_chain\_key}$'s from these tampering events after the fact, therefore running into the Nielsen bound and being unable to realise $\mathcal{F}_{\mathsf{eKE}}$ in the plain model. Looking ahead, we show that an instantiation of the Signal architecture with a KDF built from laconic primitives can circumvent this problem. It can realise $\mathcal{F}_{\mathsf{eKE}}$ in the plain model by removing the ability to compute receiving chain keys corresponding to epoch ids that resulted in failed ratcheting attempts.

**Corrupt** On receiving a `Corrupt` notification from a party's $\Pi_{\mathsf{SGNL}}$ instance, $\mathcal{F}_{\mathsf{eKE}}$ sends a `ReportState` request to the adversary along with the $(\mathsf{epoch\_id}, \mathsf{recv\_chain\_key}')$ pairs corresponding to `GetReceivingKey` calls made so far. Additionally, at the time of making a `ReportState` call to the adversary $\mathcal{A}$, $\mathcal{F}_{\mathsf{eKE}}$ provides $\mathcal{A}$ with the corrupted party's newest receiving key if it hasn't been successfully retrieved yet. In response, the adversary sends back some state $S$ which is forwarded by $\mathcal{F}_{\mathsf{eKE}}$ to the $\Pi_{\mathsf{SGNL}}$ instance that made the corruption request. Before responding to the calling instance of $\Pi_{\mathsf{SGNL}}$, $\mathcal{F}_{\mathsf{eKE}}$ notes that the current epoch is corrupted, and adds the next 3 epochs to a list of compromised epochs until full post-compromise recovery is achieved. This is because the recovery for a single compromise goes through the following phases: fully compromised (the party's entire $\mathcal{F}_{\mathsf{eKE}}$ state is known), the sender's randomness is updated (upon starting a new epoch), both parties' randomness is updated (upon starting a new epoch). If the party corrupted is a sender, then it will be a receiver in the next epoch after corruption, at this point since the party has not added any secret randomness to its state since corruption, the adversary still knows the party's entire state. In the second epoch after corruption, the party will update its randomness as a sender and at this point the adversary is unable to tamper with the communication in a significant way but full deniability has not been restored. Finally, in the third epoch after corruption, the other party will also update its randomness and therefore deniability will be restored once the corrupted party successfully receives a message in this epoch. Before the successful receipt of a message in this third epoch, full deniability has not been restored and we model the remaining adversarial power by allowing the adversary to choose any chain keys up to the point of a successful receipt in this epoch. This is why the corruption interface must also mark this third epoch after corruption as compromised.

**Remark**

We remark here that one may easily. replace the 3 epoch compromise here and in $\mathcal{F}_{\mathsf{SM}}$ with a variable for the time before healing. This would be especially useful since a Signal-like secure messaging protocol will inherit its healing time from the epoch key exchange that it uses. Using our modular analysis, one can easily swap out one epoch key exchange protocol that heals in $r$ rounds with another that heals in a different number of rounds $r'$ and show that the resulting $\Pi_{\mathsf{SGNL}} + \Pi_{\mathsf{aead}} + \Pi'_{\mathsf{eKE}}$ system will realise an $\mathcal{F}_{\mathsf{SM}}$ that enforces healing in a number of rounds that is greater than or equal to $r'$.
A simple modification to $\Pi_{\mathsf{eKE}}$ in fact allows healing in 2 rounds instead of 3. This modification is discussed in the remark on Page 52

## 5.2 The Forward Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$

The forward secure authenticated encryption functionality (see Figure 10 on page 37) processes encryptions and decryptions for a single epoch (specified in $\mathsf{sid}.fs$) for the manager protocol $\Pi_{\mathsf{SGNL}}$.

$\mathcal{F}_{\mathsf{eKE}}$

This functionality has a session id $\mathsf{sid}.eKE$ that takes the following format: $\mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid})$. Inputs arriving from machines whose identity is neither $\mathsf{pid}_0$ nor $\mathsf{pid}_1$ are ignored. //For notational simplicity we assume some fixed interpertation of $\mathsf{pid}_0$ and $\mathsf{pid}_1$ as complete identities of the two calling machines.

The following method is also used by the sender of epoch 0 to start the conversation.

**ConfirmReceivingEpoch:** On input $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}^*)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$:

1. If this is the first activation:
   - Parse $\mathsf{sid}$ to retrieve two party ids $(\mathsf{pid}_0, \mathsf{pid}_1)$ for the initiator and responder parties and store them. If $\mathsf{pid}_0 \neq \mathsf{pid}_i$, then end the activation.
   - Provide input $(\texttt{GetInitKeys}, \mathsf{pid}_1, \mathsf{pid}_0)$ to $(\mathcal{F}_{\mathsf{DIR}})$.
   - Upon receiving output $(\texttt{GetInitKeys}, \mathsf{ik}_1^{\mathsf{pk}}, \mathsf{rk}_1^{\mathsf{pk}}, \mathsf{ok}_1^{\mathsf{pk}})$ from $(\mathcal{F}_{\mathsf{DIR}})$:
     – If $\mathsf{ok}^{\mathsf{pk}}_{\mathsf{pid}_1} = \bot$ then output $(\texttt{ConfirmReceivingEpoch}, \texttt{Fail})$. //Don't start the conversation if the one time keys belonging to the other party have run out.
     – Else, set $\mathsf{epoch\_id\_partner}_0 = \mathsf{epoch\_id\_self}_1 = \mathsf{ok}^{\mathsf{pk}}_{\mathsf{pid}_1}$, set $\mathsf{epoch\_num}_0 = -2, \mathsf{epoch\_num}_1 = -1$, initialize empty lists $\mathsf{corruptions}_0, \mathsf{corruptions}_1, \mathsf{compromised\_epochs}$, and send $(\texttt{ComputeSendingRootKey}, \mathsf{ik}_1^{\mathsf{pk}}, \mathsf{rk}_1^{\mathsf{pk}}, \mathsf{ok}_1^{\mathsf{pk}})$ to $\mathcal{F}_{\mathsf{LTM}}$.
   - On receiving $(\texttt{ComputeSendingRootKey}, k, \mathsf{ek}^{\mathsf{pk}})$, continue.
   - Call $\mathcal{F}_{\mathsf{lib}}$ with input “$eKE$” to obtain internal adversarial code $\mathcal{I}$. Initialize the state for $\mathcal{I}$ to be $\mathsf{state}_{\mathcal{I}} = \bot$.
2. If this is not the first activation, set $\mathsf{epoch\_id\_partner}_i = \mathsf{epoch\_id}^*$. //save $\mathsf{epoch\_id\_partner}$ from input if this is not the first activation.
3. If $\mathsf{epoch\_id\_partner}_i \neq \mathsf{epoch\_id\_self}_{1-i}$, then $\mathsf{diverge\_parties} = \mathit{true}$. //determine if the parties' views have diverged
4. If $\mathsf{diverge\_parties}$, send a backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch\_id\_partner}_i)$ to $\mathcal{A}$.
5. Else, run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch\_id\_partner}_i)$
6. Upon receiving $(\mathsf{state}'_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch\_id})$ from $\mathcal{A}$ or from $\mathcal{I}$, update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$ and do the following:
   - If $\mathsf{epoch\_id}$ is the same as the input to any previous invocation of $\texttt{ConfirmReceivingEpoch}$, end the activation.
   - Update $\mathsf{epoch\_num}_i \mathrel{+}= 2$. Then set $\mathsf{epoch\_id\_self}_i = \mathsf{epoch\_id}$, $\mathsf{epoch\_num\_dict}[\mathsf{epoch\_id}] = \mathsf{epoch\_num}_i$, and $\mathsf{got\_sending\_key}_i = \mathit{false}$.
7. Output $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id\_self}_i)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

**GetSendingKey:** On receiving input $(\texttt{GetSendingKey})$ from $(\Pi_{mKE}, \mathsf{sid}.mKE, \mathsf{pid})$:

1. Set $i$ such that $\mathsf{pid} = \mathsf{pid}_i$.
2. If $\texttt{ConfirmReceivingEpoch}$ has never been run successfully (i.e $\mathsf{epoch\_id\_self}_0$ hasn't been initialized) or $\mathsf{got\_sending\_key}_i = \mathit{true}$, then end the activation. //the functionality isn't initialized or the sending key for the current epoch has already been retrieved
3. Parse $\mathsf{sid}.mKE = (\text{“}mKE\text{”}, \text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$. If $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_i$ then end the activation.
4. Sample $\mathsf{sending\_chain\_key}_i \overset{\$}{\leftarrow} \mathcal{K}_{ep}$ from the key distribution. //In the honest case, the key is not known to the adversary. Otherwise the key will get overwritten in the following step.
5. If $\mathsf{diverge\_parties} = \mathit{true}$, or $\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$, then:
   - Send backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{GetSendingKey}, i)$ to $\mathcal{A}$
   - On receiving backdoor message $(\mathsf{state}'_{\mathcal{I}}, \texttt{GetSendingKey}, i, K_{\mathsf{send}})$ from $\mathcal{A}$, update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$ and set $\mathsf{sending\_chain\_key}_i = K_{\mathsf{send}}$.
6. Set $\mathsf{got\_sending\_key}_i = \mathit{true}$ and output $(\texttt{GetSendingKey}, \mathsf{sending\_chain\_key}_i)$.

(The rest of this functionality is in Fig. 9 on Page 34)

Figure 8: The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$

**$\mathcal{F}_{\mathsf{eKE}}$ continued...**

(This functionality begins in Fig. 8 on Page 33)

**GetReceivingKey:** On receiving input $(\texttt{GetReceivingKey}, \mathsf{epoch\_id})$ from $(\Pi_{mKE}, \mathsf{sid}.mKE, \mathsf{pid})$:

1. If $\mathsf{pid} \notin \{\mathsf{pid}_0, \mathsf{pid}_1\}$ then end this activation. Otherwise, set $i$ such that $\mathsf{pid} = \mathsf{pid}_i$.
2. If $\texttt{ConfirmReceivingEpoch}$ has never been run successfully (i.e $\mathsf{epoch\_id\_self}_0$ hasn't been initialized) or $\mathsf{sending\_chain\_key}_{1-i}$ has been deleted then end the activation.
3. Parse $\mathsf{epoch\_id} = (\mathsf{epoch\_id}', \mathsf{ek}^{\mathsf{pk}}_j, \mathsf{ok}^{\mathsf{pk}}_{i\leftarrow j})$ and $\mathsf{sid}.mKE = (\text{“}mKE\text{”}, \text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}\text{”})$. If $\mathsf{epoch\_id}' \neq \mathsf{epoch\_id}$” end the activation.
4. If this is the first call to $\texttt{GetReceivingKey}$:
   - If $\mathsf{pid}_1 \neq \mathsf{pid}_i$, then end the activation.
   - Send $(\texttt{GetResponseKeys}, \mathsf{pid}_{1-i})$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving $(\texttt{GetResponseKeys}, \mathsf{ik}^{\mathsf{pk}}_j)$, send input $(\texttt{ComputeReceivingRootKey}, \mathsf{ik}^{\mathsf{pk}}_j, \mathsf{ek}^{\mathsf{pk}}_j, \mathsf{ok}^{\mathsf{pk}}_{i\leftarrow j})$ to $\mathcal{F}_{\mathsf{LTM}}$.
5. If $\mathsf{diverge\_parties} = \textit{true}$ or $\mathsf{epoch\_num}_i + 1 \in \mathsf{compromised\_epochs}$: //Let $\mathcal{A}$ choose key
   - Send $(\mathsf{state}_{\mathcal{I}}, \texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$ to $\mathcal{A}$.
   - Upon receiving $(\mathsf{state}'_{\mathcal{I}}, \texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*)$ from $\mathcal{A}$.
   - Output $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key}^*)$.
6. Otherwise ($\mathsf{diverge\_parties} = \textit{false}$, $\mathsf{epoch\_num}_i + 1 \notin \mathsf{compromised\_epochs}$) if $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_{1-i}$:
   - Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$.
   - Upon obtaining output $(\mathsf{state}'_{\mathcal{I}}, \texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*)$, update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$.
   - Add $\mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*$ to $\mathsf{receive\_attempts}[\mathsf{epoch\_num}]$.
   - Output $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key}^*)$.
7. Else ($\mathsf{diverge\_parties} = \textit{false}$ and $\mathsf{epoch\_id} = \mathsf{epoch\_id\_self}_{1-i}$), output $(\texttt{GetReceivingKey}, \mathsf{sending\_chain\_key}_{1-i})$. //Expected case

**Corrupt:** On receiving a $(\texttt{Corrupt})$ request from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$ for $i \in \{0, 1\}$ do:

- Add $\mathsf{epoch\_id\_self}_i$ to the list $\mathsf{corruptions}_i$.
- Add $\mathsf{epoch\_num}_i, \mathsf{epoch\_num}_i + 1, \mathsf{epoch\_num}_i + 2, \mathsf{epoch\_num}_i + 3$ to the list $\mathsf{compromised\_epochs}$. //We need the compromise to go through the following stages: fully compromised, sender randomness updated, both parties' randomness updated.
- Initialize an empty list $\mathsf{leak} = []$ and a variable $\mathsf{recv\_chain\_key} = \perp$.
- If $\mathsf{epoch\_num}_{1-i} > \mathsf{epoch\_num}_i$:
  - Set $\mathsf{recv\_chain\_key} = \mathsf{sending\_chain\_key}_{1-i}$.
  - If $\mathsf{epoch\_num}_{1-i} \in \mathsf{receive\_attempts}.\textit{keys}$ then set $\mathsf{leak} = \mathsf{receive\_attempts}[\mathsf{epoch\_num}_{1-i}]$
- Send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}_i, \mathsf{leak})$ to $\mathcal{A}$.
- Upon receiving $(\texttt{ReportState}, i, S)$ from $\mathcal{A}$, output $(\texttt{Corrupt}, S)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

Figure 9: The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$ (continued)

As the name suggests, $\mathcal{F}_{\mathsf{fs\_aead}}$ enforces the forward security property for messages encrypted within an epoch. That is, on a state compromise of the receiver for the epoch, the adversary only gets $(c, m)$ pairs for messages that are *in transit* from the sender to receiver, and it gets the power to replace ciphertexts in transit with authentic-looking ones. Note that once an epoch has been compromised, there is no recovery within the epoch; that is, the adversary retains the power to tamper with in-transit ciphertexts from the epoch until all have arrived at the receiver. Any ciphertexts that the receiver decrypted prior to state compromise are not available to $\mathcal{A}$; this models the forward security property of Signal's symmetric chain.

**Encryption** On receiving an encryption request for a message $m$, it sends $N$ (the number of messages sent in the previous sending epoch) and leaks either $|m|$ or $m$ to $\mathcal{A}$ (depending on whether the epoch is compromised) and gets a ciphertext $c$ in return, which it records along with $m$, $\mathsf{msg\_num}$, $N$, and the leakage. Note that in the real protocol, the $\mathsf{msg\_num}$, $\mathsf{epoch\_id}$, as well as $N$ are authenticated but sent in the clear with each ciphertext. $\mathcal{F}_{\mathsf{fs\_aead}}$ then sends the ciphertext up to the manager $\Pi_{\mathsf{SGNL}}$.

**Decryption** When receiving a decrypt request for ciphertext $c$ and message number $\mathsf{msg\_num}$, $\mathcal{F}_{\mathsf{fs\_aead}}$ checks whether the receiver has already successfully decrypted this $\mathsf{msg\_num}$; if so, the $\mathsf{msg\_num}$ was set to inaccessible and $\mathcal{F}_{\mathsf{fs\_aead}}$ will return a failure message to $\Pi_{\mathsf{SGNL}}$. Next, $\mathcal{F}_{\mathsf{fs\_aead}}$ checks whether the ciphertext $c$ previously failed authentication for $\mathsf{msg\_num}$; in this case, the functionality also outputs a failure message to $\Pi_{\mathsf{SGNL}}$. If the decryption has not failed from the previous two cases, the functionality sends an `inject` message to $\mathcal{A}$.

If the state of $\mathcal{F}_{\mathsf{fs\_aead}}$ is not compromised, then $\mathcal{A}$ should only be able to `inject` the true message $m$ that was encrypted for $\mathsf{msg\_num}$. In the honest setting (no state corruption), if the adversary returns $\perp$ or there is no record of an encryption for $\mathsf{msg\_num}$, $\mathcal{F}_{\mathsf{fs\_aead}}$ returns a failure; otherwise, regardless of which message $v$ the adversary returns, $\mathcal{F}_{\mathsf{fs\_aead}}$ sends $m$ to $\Pi_{\mathsf{SGNL}}$. This models the fact that without compromising a party, the real world adversary should not be able to produce ciphertexts that authenticate.

In the case that a state compromise has occurred, if $\mathcal{A}$ returns some $v \neq \perp$, $\mathcal{F}_{\mathsf{fs\_aead}}$ marks $\mathsf{msg\_num}$ as unavailable and sends $v$ up to $\Pi_{\mathsf{SGNL}}$. This models the power that the adversary has after a state compromise (of either party) to tamper with the sender's ciphertexts to produce authentic-looking ciphertexts. Note that $\mathcal{F}_{\mathsf{fs\_aead}}$ never recovers from a state compromise; thus, the adversary maintains the power to tamper with the ciphertexts for the epoch as long as there are messages from the epoch in transit.

In all of the above cases, if a decryption was successful, $\mathcal{F}_{\mathsf{fs\_aead}}$ marks that $\mathsf{msg\_num}$ as unavailable for future decryption attempts. This models the forward security property of the symmetric ratchet in the Signal protocol. Note that $\mathcal{F}_{\mathsf{fs\_aead}}$ doesn't mark messages as inaccessible upon unsuccessful decryption. We chose this to prevent denial-of-service attacks that would prevent honest parties from decrypting messages sent to them.

**Stop Encrypting** When receiving a `StopEncrypting` request from the sender for the epoch, $\mathcal{F}_{\mathsf{fs\_aead}}$ notes this and blocks all future encryptions for the epoch. This prevents an adversary from compromising the sender and injecting additional messages after the sender has moved to a new epoch.

**Stop Decrypting** When receivng a (`StopDecrypting`, $\mathsf{msg\_num}^*$) request from the receiver, $\mathcal{F}_{\mathsf{fs\_aead}}$ marks all message numbers larger than $\mathsf{msg\_num}^*$ as inaccessible, thereby preventing their decryption. This prevents an adversary from compromising the receiver of the epoch and injecting additional messages in the epoch after the receiver has advanced to a new epoch.

**Corruption** On receiving a state compromise notification from above (specifically, $\Pi_{\mathsf{SGNL}}$), if the receiver is the corrupted party, $\mathcal{F}_{\mathsf{fs\_aead}}$ leaks to the adversary all message/ciphertext pairs that are *in transit* from the sender to receiver. This models the fact that in the Signal protocol, the receiver has keys for out-of-order messages stored in its state until they have all arrived. On the other hand, if the sender is corrupted, no sent messages are leaked to the adversary, since in the Signal protocol the sender does not store any keys for message it has already sent. (Either way, the adversary will be able to read all messages sent after corruption.) The adversary (or simulator) then returns a constructed state for the party: the message keys for messages in transit, along with the chain key if it has not been deleted. This state is passed back up to $\Pi_{\mathsf{SGNL}}$.

## 5.3 The Signal Protocol, $\Pi_{\mathsf{SGNL}}$

Protocol $\Pi_{\mathsf{SGNL}}$ (see Figure 11 on page 38) is the top-level protocol that takes input commands from a party, communicates with $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{fs\_aead}}$, and returns outputs to the party to coordinate the encryption and decryption of messages for the duration of the conversation session between two parties. The ciphertexts are transferred between parties via the environment, which has full control over the network. Next we describe the three interfaces to $\Pi_{\mathsf{SGNL}}$ (which have identical API's as $\mathcal{F}_{\mathsf{SM}}$).

**Send Message** When receiving a `SendMessage` request from party $\mathsf{pid}$, $\Pi_{\mathsf{SGNL}}$ first initializes $\mathcal{F}_{\mathsf{eKE}}$ if necessary to get the first $\mathsf{epoch\_id}$, and sends an `Encrypt` request with message $m$ and $N_{last}$ (the number of messages that were sent in the party's previous sending epoch) to the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance for the current epoch. On receiving a ciphertext $c$ from $\mathcal{F}_{\mathsf{fs\_aead}}$, the manager deletes $m$, increments the number of messages sent, and outputs $c$ along with a header $h = (\mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{sent\_msg\_num}, N_{last})$ to $\mathsf{pid}$.

**Receive Message** On receiving a `ReceiveMessage` command with ciphertext $c$ and header $h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N)$ from $\mathsf{pid}$, the manager first initializes the receiver's state if necessary. It then sends a `Decrypt` request for ciphertext $c$ to the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance corresponding to the $\mathsf{epoch\_id}$ listed in header $h$. On receiving a response from $\mathcal{F}_{\mathsf{fs\_aead}}$, if decryption failed, $\Pi_{\mathsf{SGNL}}$ outputs a failure message. Otherwise, the manager updates the list of $\mathsf{msg\_num}$'s that were skipped in the epoch for $\mathsf{epoch\_id}$. If the $\mathsf{epoch\_id}$ is new, then $\Pi_{\mathsf{SGNL}}$ closes its partner's previous sending epoch by sending a (`StopDecrypting`, N) request to the appropriate $\mathcal{F}_{\mathsf{fs\_aead}}$ instance. The manager then updates $\mathsf{epoch\_id}_{\mathsf{partner}}$ with the new value it received; it sends a `StopEncrypting` request to the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance for its sending epoch. Lastly, $\Pi_{\mathsf{SGNL}}$ sends (`ConfirmReceivingEpoch`, $\mathsf{epoch\_id}$) to $\mathcal{F}_{\mathsf{eKE}}$ to ratchet forward and receive a new $\mathsf{epoch\_id}^*$ for its next sending epoch. Finally, $\Pi_{\mathsf{SGNL}}$ deletes the decrypted message $v$ returned by $\mathcal{F}_{\mathsf{fs\_aead}}$ and outputs the ciphertext $c$, message $v$, and header $h$ to $\mathsf{pid}$.

**Corruption** The manager has one additional interface, a `Corrupt` interface that is accessible only to $\mathsf{Env}$. This interface is not part of the real protocol, but is included only for UC-modelling purposes. On a corruption from the environment, $\Pi_{\mathsf{SGNL}}$ sends `Corrupt` notifications to $\mathcal{F}_{\mathsf{eKE}}$ and to every $\mathcal{F}_{\mathsf{fs\_aead}}$ instance that has messages in transit. These sub-functionalities report their internal states to $\Pi_{\mathsf{SGNL}}$ who forwards the union of their states up to $\mathsf{Env}$.

$\mathcal{F}_{\mathsf{fs\_aead}}$

This functionality processes encryptions and decryptions for a *single* epoch and has session id $\mathsf{sid}.fs$ that takes the following format: $\mathsf{sid}.fs = (\text{"}fs\_aead\text{"}, \mathsf{sid} = (\mathsf{sid}', (\mathsf{pid}_0, \mathsf{pid}_1)), \mathsf{epoch\_id})$. Inputs arriving from machines whose identity is neither $\mathsf{pid}_0$ nor $\mathsf{pid}_1$ are ignored. //For notational simplicity we assume some fixed interpertation of $\mathsf{pid}_0$ and $\mathsf{pid}_1$ as complete identities of the two calling machines. It also has *internal adversary code* $\mathcal{I} = \mathcal{I}_{\mathsf{fs}}$ (??) .

**Encrypt:** On receiving input $(\mathtt{Encrypt}, m, N)$ from $(\Pi_{MAN}, \mathsf{sid}, \mathsf{pid})$ do:

1. If this is the first activation then:
   - Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$ and initialize $\mathsf{msg\_num} = 0$ and $\mathsf{state}_{\mathcal{I}}$ to empty, sender $\mathsf{sender} = i$.
   - Call $\mathcal{F}_{\mathsf{lib}}$ with input $\mathcal{F}_{\mathsf{fs\_aead}}$ to obtain the internal code $\mathcal{I}$.
2. Verify that $\mathsf{sid}$ matches the one in the local state and $\mathsf{pid} = \mathsf{pid}_b$, otherwise end the activation.
3. If the sender has deleted the ability to encrypt messages, then end the activation.
4. Increment $\mathsf{msg\_num} = \mathsf{msg\_num} + 1$.
5. If $\mathtt{IsCorrupt?} = \textit{false}$: Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \mathtt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, |m|)$. Obtain the updated state $\mathsf{state}_{\mathcal{I}}$ and the output $(\mathtt{Encrypt}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$.
6. If $\mathtt{IsCorrupt?} = \textit{true}$: Send a backdoor message $(\mathsf{state}_{\mathcal{I}}, \mathtt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, m)$ to $\mathcal{A}$. Upon receiving a response $(\mathsf{state}_{\mathcal{I}}, \mathtt{Encrypt}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$, record the updated state $\mathsf{state}_{\mathcal{I}}$.
7. Record $(m, c, \mathsf{msg\_num}, N)$ and output $(\mathtt{Encrypt}, c)$ to $(\Pi_{MAN}, \mathsf{sid}, \mathsf{pid})$.

**Decrypt:** On receiving $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ do:

1. Verify that $\mathsf{sid}$ matches the one in the local state and $\mathsf{pid} = \mathsf{pid}_{1-\mathsf{sender}}$, otherwise end the activation.
   //end the activation if the decrypt request is not from the receiving party
2. If $\mathsf{msg\_num}$ is set as inaccessible, or there is a record $(\mathtt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$, then output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, \mathtt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
3. If $\mathtt{IsCorrupt?} = \textit{false}$:
   - Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \mathtt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ and obtain updated state $\mathsf{state}_{\mathcal{I}}$ and output $(\mathtt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, v)$.
   - If $v = \perp$, then record $(\mathtt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$ and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, \mathtt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
   - Otherwise, mark $\mathsf{msg\_num}$ as inaccessible and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, m)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
4. Else $(\mathtt{IsCorrupt?} = \textit{true})$:
   - Send $(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ to $\mathcal{A}$.
   - On receiving the updated $\mathsf{state}_{\mathcal{I}}$ and $(\mathtt{inject}, v)$ from $\mathcal{A}$, do:
     – If $v = \perp$, record $(\mathtt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$ and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, \mathtt{Fail})$.
     – Else, then mark $\mathsf{msg\_num}$ as inaccessible and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, v)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

**StopEncrypting:** On receiving $(\mathtt{StopEncrypting})$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ do:

1. If $\mathsf{sid}$ doesn't match the one in the local state, if $\mathsf{pid} \neq \mathsf{pid}_{\mathsf{sender}}$, or if this is the first activation: end the activation.
2. Otherwise, note that $\mathsf{pid}_i$ has deleted the ability to encrypt future messages. Output $(\mathtt{StopEncrypting}, \mathtt{Success})$.

**StopDecrypting:** On receiving $(\mathtt{StopDecrypting}, \mathsf{msg\_num}^*)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ do:

1. If $\mathsf{sid}$ doesn't match the one in the local state, $\mathsf{pid} \neq \mathsf{pid}_{1-\mathsf{sender}}$, or no messages have been successfully decrypted by $\mathsf{pid}_i$: end the activation.
2. Mark all $\mathsf{msg\_num} > \mathsf{msg\_num}^*$ as inaccessible, and output $(\mathtt{StopDecrypting}, \mathtt{Success})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

**Corrupt:** On receiving $(\mathtt{Corrupt}, \mathsf{pid})$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$:

1. Record $(\mathtt{Corrupt}, \mathsf{pid})$ and set $\mathtt{IsCorrupt?} = \textit{true}$.
2. If $\mathsf{pid} = \mathsf{pid}_{1-\mathsf{sender}}$ ($\mathsf{pid}$ is the receiver), let $\mathsf{leak} = \{(\mathsf{pid}_{\mathsf{sender}}, h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N), c, m)\}$ be the set of all messages sent by $\mathsf{pid}_{\mathsf{sender}}$ which are not marked as inaccessible.
3. Otherwise ($\mathsf{pid}$ is the sender), set $\mathsf{leak} = \emptyset$
4. Send $(\mathtt{ReportState}, \mathsf{state}_{\mathcal{I}}, \mathsf{pid}, \mathsf{leak})$ to $\mathcal{A}$.
5. Upon receiving a response $(\mathtt{ReportState}, \mathsf{state}_{\mathcal{I}}, \mathsf{pid}, S)$ from $\mathcal{A}$, send $S$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

Figure 10: The Forward-Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$

**$\Pi_{\mathsf{SGNL}}$**

**SendMessage:** Upon receiving input $(\texttt{SendMessage}, m)$ from $\mathsf{pid}$, do:

1. If this is the first activation do: //initialization for the initiator of the session
   - Parse the local session id $\mathsf{sid}$ to retrieve the party identifiers $(\mathsf{pid}_0, \mathsf{pid}_1)$ for the initiator and responder. If $\mathsf{pid}_0$ is different from either the local party identifier $\mathsf{pid}$, or the party identifier of $\mathsf{pid}$, end the activation.
   - Initialize $\mathsf{epoch\_id}_{\mathsf{self}} = \perp$, $\mathsf{epoch\_id}_{\mathsf{partner}} = \perp$, $\mathsf{sent\_msg\_num} = 0$, $N_{\mathsf{last}} = 0$.
   - Provide input $(\texttt{ConfirmReceivingEpoch}, \perp)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
   - On receiving $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$, set $\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}$.
   - Initialize a list $\mathsf{receiving\_epochs} = []$.
2. Provide input $(\texttt{Encrypt}, m, N_{\mathsf{last}})$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$, where $\mathsf{sid}.fs = (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{self}})$. //$\mathcal{F}_{\mathsf{fs\_aead}}$ already knows $\mathsf{epoch\_id}$ and $\mathsf{msg\_num}$
3. On receiving $(\texttt{Encrypt}, c, N_{\mathsf{last}})$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$, delete $m$, increment $\mathsf{sent\_msg\_num} \mathrel{+}= 1$, output $(\texttt{SendMessage}, \mathsf{sid}, h, c)$ to $\mathsf{pid}$, where $h = (\mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{sent\_msg\_num}, N_{\mathsf{last}})$.

**ReceiveMessage:** Upon receiving $(\texttt{ReceiveMessage}, h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N), c)$ from $\mathsf{pid}$:

1. If this is the first activation then do: //initialization for the responder of the session
   - Parse the local session identifier $\mathsf{sid}$ to retrieve the party identifiers $(\mathsf{pid}_0, \mathsf{pid}_1)$ for the initiator and responder. If $\mathsf{pid}_1$ is different from either the local party identifier, or the party identifier for $\mathsf{pid}$, then end the activation.
   - Initialize $\mathsf{epoch\_id}_{\mathsf{self}} = \perp$, $\mathsf{epoch\_id}_{\mathsf{partner}} = \perp$, $\mathsf{sent\_msg\_num} = 0$ and $N_{\mathsf{last}} = 0$, $\mathsf{received\_msg\_num} = 0$.
   - Initialize a dictionary $\mathsf{missed\_msgs} = \{\}$ and a list $\mathsf{receiving\_epochs} = []$.
2. Provide input $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs = (\mathsf{sid}, \mathsf{epoch\_id}))$.
3. Upon receiving $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N, v)$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$: if $v = \texttt{Fail}$ then send $(\texttt{ReceiveMessage}, h, \mathsf{ad}, \texttt{Fail})$ to $\mathsf{pid}$. //Otherwise, $v$ is the decrypted message
4. While $\mathsf{msg\_num} > \mathsf{received\_msg\_num}$:
   //note down any expected messages
   - Append $\mathsf{received\_msg\_num}$ to the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$.
   - Increment $\mathsf{received\_msg\_num} += 1$.
5. If $\mathsf{msg\_num}$ is in the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$:
   - remove it from the list.
   - If the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$ is now an empty list then remove $\mathsf{epoch\_id}$ from $\mathsf{missed\_msgs}.keys$.
6. Else ($\mathsf{msg\_num} \notin \mathsf{missed\_msgs}[\mathsf{epoch\_id}]$):
   - If $\mathsf{epoch\_id} = \mathsf{epoch\_id}_{\mathsf{partner}}$ or $\mathsf{sent\_msg\_num} = 0$, output $(\texttt{ReceiveMessage}, h, c, \perp)$.
     Otherwise continue. //Starting new epoch–ratchet forward
   - Append the numbers $\mathsf{received\_msg\_num}, \ldots, N$ to the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$.
   - Send $(\texttt{StopDecrypting}, N)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{partner}}))$. //'Closing' the $\mathcal{F}_{\mathsf{fs\_aead}}$ for the last epoch.
   - On receiving $(\texttt{StopDecrypting}, \texttt{Success})$, update $\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}$, and send $(\texttt{StopEncrypting})$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{self}}))$.
   - On receiving $(\texttt{StopEncrypting}, \texttt{Success})$, send $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
   - On receiving $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}^*)$, update $\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}^*$, $N_{\mathsf{last}} = \mathsf{sent\_msg\_num}$, and $\mathsf{sent\_msg\_num} = 0$.
7. Output $(\texttt{ReceiveMessage}, h, c, v)$ to $\mathsf{pid}$ while deleting the decrypted message $v$.

**Corruption:** Upon receiving $(\texttt{Corrupt}, \mathsf{pid})$ from $\mathsf{Env}$:
//Note that the Corrupt interface is not part of the "real" protocol; it is only included for modelling purposes.

1. Initialize a list $S$ and send $(\texttt{Corrupt})$ as input to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE = \text{“}eKE\text{”}, \mathsf{sid})$.
2. On receiving $(\texttt{Corrupt}, S_{eKE})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE = \text{“}eKE\text{”}, \mathsf{sid})$, add it to $S$ and continue. //now corrupt individual $\mathcal{F}_{\mathsf{fs\_aead}}$ instances.
3. For $\mathsf{epoch\_id} \in \mathsf{missed\_msgs}.keys$ do:
   - Send $(\texttt{Corrupt})$ as input to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs = (\text{“}fs\_aead''}, \mathsf{sid}, \mathsf{epoch\_id}))$.
   - On receiving $S_{\mathsf{epoch\_id}}$, add it to $S$.
4. Output $(\texttt{Corrupt}, \mathsf{pid}_i, S)$ to $\mathsf{Env}$.

## 5.4 Security Analysis

In this section we prove that $\Pi_{\mathsf{SGNL}}$, $\mathcal{F}_{\mathsf{eKE}}$, and $\mathcal{F}_{\mathsf{fs\_aead}}$ together UC-realize $\mathcal{F}_{\mathsf{SM}}$. We refer readers to Section 2 for a primer on the universally composable security framework. As a reminder, the claim that "$A$ UC-realizes $B$ in the presence of $C$" means that the environment's views are indistinguishable when interacting with $A$ or $B$, together with their respective adversaries and a global subroutine $C$.

**Theorem 2** *Protocol* $\Pi_{\mathsf{SGNL}}$ *(perfectly) UC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{SM}}$ *in the presence of* $\mathcal{F}_{\mathsf{DIR}}$ *and* $\mathcal{F}_{\mathsf{LTM}}$.

**Proof:**We construct an ideal-process adversary $\mathcal{S}_{\mathsf{SM}}$ in Fig. 12 that interacts with functionality $\mathcal{F}_{\mathsf{SM}}$. The objective of $\mathcal{S}_{\mathsf{SM}}$ is to simulate the interactions that would take place between the environment and the manager protocol $\Pi_{\mathsf{SGNL}}$ (together with its subroutine functionalities $\mathcal{F}_{\mathsf{fs\_aead}}$ and $\mathcal{F}_{\mathsf{eKE}}$), so that the views of the environment $\mathsf{Env}$ are perfectly identical in the real and ideal scenarios.

Observe that the APIs of $\mathcal{F}_{\mathsf{SM}}$ and $\Pi_{\mathsf{SGNL}}$ are identical: they each have 3 methods. We summarize the actions undertaken by $\mathcal{S}_{\mathsf{SM}}$ when invoked within each method.

- Within `SendMessage`, as long as initialization has been properly performed then $\mathcal{F}_{\mathsf{SM}}$ will send a direct message $(\texttt{SendMessage}, \mathsf{pid}, \ell)$ to $\mathcal{S}_{\mathsf{SM}}$. The simulator responds by using $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{fs\_aead}}$ to form an epoch identifier $\mathsf{epoch\_id}$ and ciphertext $c$, respectively. The sending routine always uses the newest epoch ids generated for the sending party.
- Within `ReceiveMessage`, after performing several input validation checks (e.g., that the epoch/message header hasn't been used before) $\mathcal{F}_{\mathsf{SM}}$ will send a direct message `inject` to $\mathcal{S}_{\mathsf{SM}}$ that (in limited circumstances depending on corruption, divergence, and MAC tag status) allows the simulator to decide whether the message is authentic or even run a rushing attack to change the message conents. The simulator $\mathcal{S}_{\mathsf{SM}}$ also keeps track of these state variables to determine whether it is allowed to make an adversarial injection. If so, $\mathcal{S}_{\mathsf{SM}}$ uses $\mathcal{F}_{\mathsf{fs\_aead}}$ to determine the appropriate value to inject and whether this injection will be successful (i.e., whether it is caught by any subsequent validation checks). Finally, if this is the first successfully received message of a new epoch, then $\mathcal{S}_{\mathsf{SM}}$ also interacts with the environment to perform the public ratchet within $\mathcal{F}_{\mathsf{eKE}}$ and generate a new $\mathsf{epoch\_id}$ for the recipient party to use when it next sends a message.
- If a corruption is requested from the environment to $\mathcal{F}_{\mathsf{SM}}$, then the simulator uses the `ReportState` methods within $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{fs\_aead}}$ to recover the corrupted party's chain key and pending messages in transit, respectively.

In the remainder of this proof, we describe why the simulator's actions maintain the property that the view of the environment $\mathsf{Env}$ is identically distributed (when treated as a random variable) when interacting with either the real protocol $\Pi_{\mathsf{SGNL}}$ or with the ideal functionality $\mathcal{F}_{\mathsf{SM}}$ together with the simulator $\mathcal{S}_{\mathsf{SM}}$. Without loss of generality, we restrict our attention to a deterministic environment $\mathsf{Env}$ and we only consider a dummy adversary $\mathcal{A}$. As a consequence, there are only two sources of randomness in the entire execution: first the choice of $\mathsf{ik}$ and $\mathsf{rk}$ within $\mathcal{F}_{\mathsf{DIR}}$, which influences the epoch key generated in $\mathcal{F}_{\mathsf{eKE}}$, and second the simulator's random sampling of an epoch key $\mathsf{recv\_chain\_key}$ within the view of a corrupted receiver.

Our proof proceeds by induction over the steps of the simulator, in order to show that each individual action maintains the property that the environment's view is identical in the real and

## $\mathcal{S}_{\mathsf{SM}}$

**At first activation:** Send $(\mathcal{F}_{\mathsf{SM}}, \mathcal{I})$ to $\mathcal{F}_{\mathsf{lib}}$.

**SendMessage:** On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{SendMessage}, \mathsf{pid}, m)$ from $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$ do:

1. Set $i$ such that $\mathsf{pid} = \mathsf{pid}_i$. //this is the identity of the sender
2. If this is the first invocation of $\texttt{SendMessage}$ do:
   - Initialize $\mathsf{diverge\_parties} = \mathit{false}$, $\mathsf{injectable} = \mathit{false}$, $\mathsf{corrupted\_party} = \bot$, $\mathsf{sent\_msg\_num}_0 = 0$, and $\mathsf{sent\_msg\_num}_1 = 0$.
   - Create empty stacks $\mathsf{sent\_ids}_0 = []$ and $\mathsf{sent\_ids}_1 = []$.
   - Create empty sets $\mathsf{injectable\_ids}_0 = \emptyset$ and $\mathsf{injectable\_ids}_1 = \emptyset$.
   - Parse $\mathsf{state}.eKE$ from $\mathsf{state}_{\mathcal{I}}$
   - Send $(\mathsf{state}.eKE, \texttt{GenEpochId}, i, \bot)$ to $\mathsf{Env}$ (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$).
   - Upon receiving $(\mathsf{state}, \texttt{GenEpochId}, i, \mathsf{epoch\_id})$ from $\mathsf{Env}$, update $\mathsf{state}.eKE \leftarrow \mathsf{state}$ and push $\mathsf{epoch\_id}$ onto the stack $\mathsf{sent\_ids}_0$.
3. Set $\mathsf{epoch\_id}$ equal to the top of the stack $\mathsf{sent\_ids}_i$, and increment $\mathsf{sent\_msg\_num}_i \mathrel{+}= 1$.
4. Parse $\mathsf{state}.fs.\mathsf{epoch\_id}$ from $\mathsf{state}_{\mathcal{I}}$.
5. Send $(\mathsf{state}.fs.\mathsf{epoch\_id}, \texttt{Encrypt}, \mathsf{pid}, \mathsf{sent\_msg\_num}_i, m)$ to $\mathsf{Env}$ (in the name of $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$, where $\mathsf{sid}.fs = (\textit{“fs\_aead”}, \mathsf{sid}, \mathsf{epoch\_id}))$
6. On receiving $(\mathsf{state}, \texttt{Encrypt}, \mathsf{pid}, \mathsf{sent\_msg\_num}, c)$ from $\mathsf{Env}$:
   - Update $\mathsf{state}.fs.\mathsf{epoch\_id} \leftarrow \mathsf{state}$
   - Add $(h, c) \in \mathsf{sentheaders}$.
   - Update $\mathsf{sentheaders}$ and $\mathsf{sent\_ids}_0, \mathsf{sent\_ids}_1$ in $\mathsf{state}_{\mathcal{I}}$
   - Send $(\mathsf{state}_{\mathcal{I}}, \texttt{SendMessage}, \mathsf{pid}, \mathsf{epoch\_id}, c)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$

**Inject:** On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \mathsf{pid}, h, c)$ from $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$ do:

1. Set $i$ such that $\mathsf{pid} = \mathsf{pid}_i$. //this is the identity of the attempted sender, and $\mathsf{pid}_{1-i}$ is the receiver
2. Parse $h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N)$.
3. Parse $\mathsf{state}.fs.\mathsf{epoch\_id}$ from $\mathsf{state}_{\mathcal{I}}$.
4. Send $(\mathsf{state}.fs.\mathsf{epoch\_id}, \texttt{inject}, \mathsf{pid}, \mathsf{msg\_num}, c)$ to $\mathsf{Env}$ (in the name of $(\mathcal{F}_{\mathsf{fs\_aead}}$, $\mathsf{sid}.\mathsf{fs})$ where $\mathsf{sid}.fs = (\textit{“fs\_aead”}, \mathsf{sid}, \mathsf{epoch\_id}))$.
5. On receiving $(\mathsf{state}, \texttt{inject}, v)$ from $\mathsf{Env}$, update $\mathsf{state}.fs.\mathsf{epoch\_id} \leftarrow \mathsf{state}$.
6. If $(h, c) \notin \mathsf{sentheaders} \wedge v = \bot$: output $(\texttt{inject}, h, c, \bot)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$. //in this case $\mathcal{F}_{\mathsf{SM}}$ should output $\texttt{Fail}$, otherwise decryption succeeds
7. If $\mathsf{epoch\_id} \neq \mathsf{sent\_ids}_i$ then set $\mathsf{diverge\_parties} = \mathit{true}$. //this epoch id was generated by the adversary rather than the sender, and it caused a divergence
8. If this is the first successfully received message with this $\mathsf{epoch\_id}$ do: //received first message from the sender's newest epoch
   - Parse $\mathsf{state}.eKE$ from $\mathsf{state}_{\mathcal{I}}$
   - Send a message $(\mathsf{state}.eKE, \texttt{GenEpochId}, i, \mathsf{epoch\_id})$ to $\mathsf{Env}$ (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$)
   - On receiving $(\mathsf{state}, \texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$, update $\mathsf{state}.eKE \leftarrow \mathsf{state}$ and add $\mathsf{epoch\_id}^*$ to the stack $\mathsf{sent\_ids}_{1-i}$ and add $\mathsf{epoch\_id}^*$ to $\mathsf{state}_{\mathcal{I}}$. //this will be party $i$'s next $\mathsf{epoch\_id}$ when it next sends a message
   - If $\mathsf{epoch\_id} = \mathsf{sent\_ids}_i.\mathit{top}$ and $\mathsf{injectable} = \mathit{true}$ and $\mathsf{diverge\_parties} = \mathit{false}$ and $\mathsf{corrupted\_party} = i$ then: set $\mathsf{injectable} = \mathit{false}$ and $\mathsf{corrupted\_party} = \bot$. //if party $i$ succeeds in establishing a new sending epoch, the adversary can no longer inject
9. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, h, c, v)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$.

**ReportState:** On receiving $(\texttt{ReportState}, \mathsf{pid}, \mathsf{pending\_msgs})$ from $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$ do:

1. Set $i$ such that $\mathsf{pid} = \mathsf{pid}_i$. //this is the identity of the corrupted party
2. Set $\mathsf{corrupted\_party} = i$, $\mathsf{injectable\_ids}_0 = \mathsf{sent\_ids}_0$, $\mathsf{injectable\_ids}_1 = \mathsf{sent\_ids}_1$, and initialize an empty list $S_i$.
3. Set $\mathsf{recv\_chain\_key} \overset{\$}{\leftarrow} \mathcal{K}_{ep}$ from the key distribution.
4. Send $(\mathsf{state}.eKE, \texttt{ReportState}, i, \mathsf{recv\_chain\_key})$ to $\mathsf{Env}$ (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$).
5. Upon receiving $(\texttt{ReportState}, i, S^*)$ from $\mathsf{Env}$, add $S^*$ to $S_i$.

**$\mathcal{I}_{\mathsf{sm}}$**

This internal adversary is only called when no parties are compromised.
**SendMessage:** On receiving $(\mathsf{state}_{\mathcal{I}}, \mathtt{SendMessage}, \mathsf{pid}, |m|)$ from $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$ do:

1. Set $i$ such that $\mathsf{pid} = \mathsf{pid}_i$. //this is the identity of the sender
2. If this is the first invocation of $\mathtt{SendMessage}$ do:
   - Initialize $\mathsf{sent\_msg\_num}_0 = 0$, and $\mathsf{sent\_msg\_num}_1 = 0$.
   - Create empty stacks $\mathsf{sent\_ids}_0 = []$ and $\mathsf{sent\_ids}_1 = []$.
   - Create empty sets $\mathsf{injectable\_ids}_0 = \emptyset$ and $\mathsf{injectable\_ids}_1 = \emptyset$.
   - Parse $\mathsf{state}.eKE$ from $\mathsf{state}_{\mathcal{I}}$
   - Run $\mathcal{I}_{\mathsf{eke}}(\mathsf{state}.eKE, \mathtt{GenEpochId}, i, \perp)$
   - Upon receiving $(\mathsf{state}', \mathtt{GenEpochId}, i, \mathsf{epoch\_id})$ from $\mathsf{Env}$, update $\mathsf{state}.eKE \leftarrow \mathsf{state}'$ and push $\mathsf{epoch\_id}$ onto the stack $\mathsf{sent\_ids}_0$.
3. Set $\mathsf{epoch\_id}$ equal to the top of the stack $\mathsf{sent\_ids}_i$, and increment $\mathsf{sent\_msg\_num}_i \mathrel{+}= 1$.
4. Parse $\mathsf{state}.fs.\mathsf{epoch\_id}$ from $\mathsf{state}_{\mathcal{I}}$.
5. Run $\mathcal{I}_{fs}(\mathsf{state}.fs.\mathsf{epoch\_id}, \mathtt{Encrypt}, \mathsf{pid}, \mathsf{sent\_msg\_num}_i, |m|)$ for specifically $(\mathcal{I}_{fs}, \mathsf{sid}.fs)$, where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}))$.
6. On receiving $(\mathsf{state}, \mathtt{Encrypt}, \mathsf{pid}, \mathsf{sent\_msg\_num}, c)$ from $\mathcal{I}_{fs}$:
   - Update $\mathsf{state}.fs.\mathsf{epoch\_id} \leftarrow \mathsf{state}$
   - Add $(h, c) \in \mathsf{sentheaders}$
   - Update $\mathsf{sentheaders}$ and $\mathsf{sent\_ids}_0, \mathsf{sent\_ids}_1$ in $\mathsf{state}_{\mathcal{I}}$
   - Send $(\mathsf{state}_{\mathcal{I}}, \mathtt{SendMessage}, \mathsf{pid}, \mathsf{epoch\_id}, c)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$

**Inject:** On receiving $(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, \mathsf{pid}, h, c)$ from $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$ do:

1. Set $i$ such that $\mathsf{pid} = \mathsf{pid}_i$. //this is the identity of the attempted sender, and $\mathsf{pid}_{1-i}$ is the receiver
2. Parse $h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N)$, read in $\mathsf{sentheaders}$ and $\mathsf{sent\_ids}_0, \mathsf{sent\_ids}_1$ from $\mathsf{state}_{\mathcal{I}}$, and read in $\mathsf{state}.fs.\mathsf{epoch\_id}$ from $\mathsf{state}_{\mathcal{I}}$
3. Run $\mathcal{I}_{\mathsf{fs}}(\mathsf{state}.fs.\mathsf{epoch\_id}, \mathtt{inject}, \mathsf{pid}, \mathsf{msg\_num}, c)$ specifically for $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.\text{fs})$ where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}))$.
4. On receiving $(\mathsf{state}, \mathtt{inject}, v)$ from $\mathcal{I}$, update $\mathsf{state}.fs.\mathsf{epoch\_id} \leftarrow \mathsf{state}$.
5. If $(h, c) \notin \mathsf{sentheaders} \wedge (v = \perp \vee (\nexists c^* \text{ s.t. } (h, c^*) \in \mathsf{sentheaders}))$: output $(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, h, c, \perp)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$. //in this case $\mathcal{F}_{\mathsf{SM}}$ should output $\mathtt{Fail}$, otherwise decryption succeeds
6. If $\mathsf{epoch\_id} \neq \mathsf{sent\_ids}_i$ then set $\mathsf{diverge\_parties} = \textit{true}$. //this epoch id was generated by the adversary rather than the sender, and it caused a divergence
7. If this is the first successfully received message with this $\mathsf{epoch\_id}$ do: //received first message from the sender's newest epoch
   - Parse $\mathsf{state}.eKE$ from $\mathsf{state}_{\mathcal{I}}$
   - Run $\mathcal{I}_{\mathsf{eke}}(\mathsf{state}.eKE, \mathtt{GenEpochId}, i, \mathsf{epoch\_id})$
   - On receiving $(\mathsf{state}', \mathtt{GenEpochId}, i, \mathsf{epoch\_id}^*)$, update $\mathsf{state}.eKE \leftarrow \mathsf{state}'$, add $\mathsf{epoch\_id}^*$ to the stack $\mathsf{sent\_ids}_{1-i}$ and add $\mathsf{epoch\_id}^*$ to $\mathsf{state}_{\mathcal{I}}$. //this will be party $i$'s next $\mathsf{epoch\_id}$ when it next sends a message
8. Output $(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, h, c, v)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$.

Figure 13: Internal Adversary $\mathcal{I}_{\mathsf{sm}}$

ideal world executions. We split our proof into two cases based on whether a corruption or divergence has occurred. We also observe that the simulator $\mathcal{S}_{\mathsf{SM}}$ may assume that the first call made by the environment is to `SendMessage` and that all subsequent calls to `ReceiveMessage` use $(\mathsf{epoch\_id}, \mathsf{msg\_num})$ headers that haven't been used before and that have valid $\mathsf{epoch\_id}$. If these constraints do not hold, then we observe by inspection that both $\mathcal{F}_{\mathsf{SM}}$ and $\Pi_{\mathsf{SGNL}}$ terminate before even invoking the ideal-world adversary $\mathcal{S}_{\mathsf{SM}}$ or the real-world adversary $\mathcal{A}$, respectively.

**Case 1: neither party is compromised.** The functionality $\mathcal{F}_{\mathsf{SM}}$ notifies the simulator when any message is sent or received, as long as it passes the input validation checks described above. During `ReceiveMessage`, the simulator is given a message header $(\mathsf{epoch\_id}, \mathsf{msg\_num}, N)$ together with a ciphertext $c$, and it is permitted to attempt to inject a message. In the analogous `ReceiveMessage` routine in the real world, the protocol $\Pi_{\mathsf{SGNL}}$ invokes the specific instance of $\mathcal{F}_{\mathsf{fs\_aead}}$ corresponding to $\mathsf{epoch\_id}$, which then sends a backdoor message asking for the desired message to inject. In the ideal world, the simulator $\mathcal{S}_{\mathsf{SM}}$ sends this exact backdoor message and retrieves a value $v$. It is straightforward to confirm by inspection that $\mathcal{S}_{\mathsf{SM}}$ matches the input-validation logic used within the real world: $v$ is ignored if $c$ is a valid ciphertext created by a previous invocation to `SendMessage`, and otherwise the message is authenticated if and only if $v \neq \perp$. Finally, if this is the first successfully received message of a new epoch, then the real world protocol $\Pi_{\mathsf{SGNL}}$ invokes $\mathcal{F}_{\mathsf{eKE}}$ to perform a public ratchet; the simulator $\mathcal{F}_{\mathsf{SM}}$ emulates the backdoor message required in order to receive the new $\mathsf{epoch\_id}$ from the environment. Hence, the ideal and emulated worlds move to a new epoch in lockstep. The simulator also records all $\mathsf{epoch_id}$s for later use during `SendMessage`, as described below. It is straightforward to check that the code of $\mathcal{F}_{\mathsf{SM}}$ and $\Pi_{\mathsf{SGNL}}$ identically perform other checks that do not involve the (ideal or real world) adversary at all, such as refusing to decrypt a message whose header $h$ has already been used or that is invalid because its $\mathsf{msg\_num}$ is larger than expected. As a result, the views of the environment in both scenarios contains the same backdoor messages, as well as the same outputs since they are deterministically derived from the environment's own responses to the backdoor messages.

During `SendMessage`, the simulator receives as input the identity of the sending party $\mathsf{pid}$ and the length of the desired message $\ell = |m|$. In the ideal world, by the time that $\mathcal{F}_{\mathsf{SM}}$ has invoked $\mathcal{S}_{\mathsf{SM}}$, it has properly initialized `SendMessage` and is awaiting a ciphertext from $\mathcal{S}_{\mathsf{SM}}$ so it can complete the message transmission. For the corresponding call to `SendMessage` in the real world, $\Pi_{\mathsf{SGNL}}$ performs the same initialization and then invokes the specific instance of $\mathcal{F}_{\mathsf{fs\_aead}}$ corresponding to the latest $\mathsf{epoch\_id}$, which in turn sends a backdoor message requesting a ciphertext. Because it knows $\ell$ and the newest $\mathsf{epoch\_id}$ (from previous calls to `ReceiveMessage`), our simulator $\mathcal{S}_{\mathsf{SM}}$ can also send this backdoor message on behalf of the appropriate instance of $\mathcal{F}_{\mathsf{fs\_aead}}$, and it receives back the desired ciphertext. The views of the environment in both scenarios are the same: this one backdoor message communication, together with a sent message with its desired ciphertext. It is also simple to observe by inspection that internal state variables like $\mathsf{msg\_num}$ and $N$ remain in sync as well.

**Case 2: one or both parties are compromised.** During the interval where the adversary can tamper with the communication (i.e., `inject` $\in \mathsf{advControl}$) or if the parties' root chains have diverged (i.e., $\mathsf{diverge\_parties} = \mathit{true}$), the simulator works slightly differently than described in Case 1 above. The only difference during `SendMessage` is that the simulator $\mathcal{S}_{\mathsf{SM}}$ receives the party's desired message $\ell = m$ as input, and it provides the message to the environment in its backdoor message. The only change during `ReceiveMessage` is that the simulator $\mathcal{S}_{\mathsf{SM}}$ must compute a different input validation predicate, because both $\mathcal{F}_{\mathsf{SM}}$ and $\Pi_{\mathsf{SGNL}}$ allow the adversary to inject any message of its choice unless (i) decryption uses the exact ciphertext $c$ that was generated by a previous `Encrypt` call, in which case decryption must succeed with the message used during encryption, or (ii) the adversary chooses $v = \perp$ to indicate that it wants the `ReceiveMessage` routine to `Fail`.

To complete the proof, it only remains to show that corruption and uncorruption happen at the same times in the real and ideal worlds and that the environment receives the same state in response to a `ReportState` command. Corruption is initiated by the environment itself; both $\Pi_{\mathsf{SGNL}}$ and $\mathcal{F}_{\mathsf{SM}}$ (with $\mathcal{S}_{\mathsf{SM}}$) respond immediately to this request by corrupting parties and reporting state back to the environment. In response to a `ReportState` command sent to $\mathcal{F}_{\mathsf{SM}}$, the simulator $\mathcal{S}_{\mathsf{SM}}$ constructs the corrupted party's view by using $\mathcal{F}_{\mathsf{eKE}}$ and all pertinent $\mathcal{F}_{\mathsf{fs\_aead}}$ in exactly the same way as $\Pi_{\mathsf{SGNL}}$ does, with only one exception. Because $\Pi_{\mathsf{SGNL}}$ does not expose to the environment the tcommands `GetSendingKey` and `GetReceivingKey` within its $\mathcal{F}_{\mathsf{eKE}}$ subroutine, it suffices for the simulator to sample $\mathsf{recv\_chain\_key}$ independently (from the same distribution as $\mathcal{F}_{\mathsf{eKE}}$ does) because $\mathsf{Env}$ cannot make the queries needed to test consistency with the underlying state held within $\mathcal{F}_{\mathsf{eKE}}$. Additionally, it is simple to observe by inspection that $\mathcal{F}_{\mathsf{SM}}$ and $\Pi_{\mathsf{SGNL}}$ uncorrupt a party at the same time (when the corrupted party successfully begins a new sending epoch and has its message received by the honest recipient), and that $\mathcal{S}_{\mathsf{SM}}$ appropriately tracks when this event occurs using the $\mathsf{injectable}$ flag. This flag governs whether the simulator responds according to Case 1 or Case 2.

**Case 3: the parties' views of the conversation have (potentially) diverged.** Finally, divergence is even easier to analyze. In both $\mathcal{F}_{\mathsf{SM}}$ and $\Pi_{\mathsf{SGNL}}$, it occurs when a party receives a message with an $\mathsf{epoch\_id}$ that its partner never sent, the code of $\mathcal{S}_{\mathsf{SM}}$ properly tracks this event using the $\mathsf{diverge\_parties}$ flag and moves to Case 2, and the parties never recover from a divergence in either $\mathcal{F}_{\mathsf{SM}}$ or $\Pi_{\mathsf{SGNL}}$. In this case, the adversary has full visibility and control to perform a person-in-the-middle attack in both $\mathcal{F}_{\mathsf{SM}}$ and $\Pi_{\mathsf{SGNL}}$. □

## 6 The Public Key Ratchet: Realizing Epoch Key Exchange

This section describes a protocol $\Pi_{\mathsf{eKE}}$ (figure 15) that UC-realizes $\mathcal{F}_{\mathsf{eKE}}$ (figure 8) (without using the random oracle abstraction) in the presence of the global functionalities $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$. Protocol $\Pi_{\mathsf{eKE}}$ mirrors the public key ratchet from the Signal protocol – in particular it uses values from a continuous Diffie-Hellman ratchet as inputs to a KDF[6] chain. We capture the properties of a KDF chain by a new primitive, Cascaded PRF-PRG (CPRFG), that we describe in section 6.1. We then present $\Pi_{\mathsf{eKE}}$ in Section 6.2 and analyze it in Section 6.3.

### 6.1 Cascaded PRF-PRG (CPRFG)

The protocol $\Pi_{\mathsf{eKE}}$ (fully specified later in Fig. 15) uses a stateful key derivation function (KDF). This function is started off at epoch 0 a random initial state $\mathsf{root\_key}_0$ and provides the following functionality: given a randomizer $\mathsf{root\_input}$, the KDF can either

- use $\mathsf{root\_input}$ to generate a chaining key $\mathsf{chain\_key}$ for the current epoch, without advancing to this epoch. (In the Signal protocol, the $\mathsf{chain\_key}$ is used to derive message keys for the current epoch.)

[6] key derivation function

- use $\mathsf{root\_input}$ to update the state and advance to the corresponding current epoch, then advance to the next epoch.

In this subsection we define a new primitive that captures the security requirements from this key derivation function – Cascaded PRF-PRG (CPRFG). Our CPRFG primitive is inspired by the PRF-PRNG model from Alwen et al. [1]. However, our primitive directly models a KDF-chain under adaptive security in contrast to the PRF-PRNG in [1] that models a single instance of a KDF. This means that while the PRF-PRNG requirements from [1] have to be stringent enough to provide the right adaptive guarantees when used in a chain, our primitive directly requires the exact adaptive properties needed from a KDF-chain as specified in the Signal architecture [52]. We remark upfront that the HKDF function [45] used in the Signal application can also meet our definition, albeit in the random oracle model.

Concretely, a KDF consists of two deterministic algorithms, **Compute** and **Advance**. An execution of the KDF involves a state variable $\mathsf{root\_key}$ that is initialized to some value. The algorithm $\mathbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) = (\mathsf{chain\_key}, \mathsf{root\_key}')$ is given the current state $\mathsf{root\_key}$ and randomizer $\mathsf{root\_input}$, and computes a chaining key $\mathsf{chain\_key}$ and an updated state $\mathsf{root\_key}'$. (It is stressed that this operation does not advance the epoch; still, the local state is updated and the old one is discarded.) Algorithm $\mathbf{Advance}(\mathsf{root\_key}, \mathsf{root\_input}) = \mathsf{root\_key}'$ is given a state $\mathsf{root\_key}$ and randomizer $\mathsf{root\_input}$, and updates the state for a new epoch.

At high level, we require that the KDF provide the following guarantees:

1. As long as the state is random and secret and the epoch is not advanced, the KDF should behave like a random function with $\mathsf{root\_input}$ as input: there should be a fresh random $\mathsf{chain\_key}$ associated with each value of $\mathsf{root\_input}$. This is because an adversary that asks a party to decapsulate multiple bogus ciphertexts that purport to be a first message in a new epoch can cause the receiver to generate multiple new candidate chaining keys for the same epoch.

2. Once the epoch is advanced, the KDF should behave like a fresh random function, independently from all previous ones. This should hold as long as either (a) the current state of the KDF is random and secret, or (b) the randomizer $\mathsf{root\_input}$ is random and secret. Notice that condition (a) prevents an adversary that breaks into a party from re-computing keys generated by the party in previous epochs, and condition (b) allows parties to recover from break-ins as soon as they agree on a new secret randomizer $\mathsf{root\_input}$.

3. To demonstrate that adaptive break-ins do not compromise the overall security of the protocol, we would like to demonstrate that the process whereby the adversary first obtains the externally-visible outputs of the KDF when executed within $\Pi_{\mathsf{eKE}}$, and then obtains the current internal state of the KDF, does not give the adversary any computational advantage whatsoever. To do that, we would like there to exist a simulator that takes as input a sequence of pairs $(\mathsf{root\_input}_1, \mathsf{chain\_key}_1), ..., (\mathsf{root\_input}_k, \mathsf{chain\_key}_k)$, representing the inputs and resulting chaining keys that were generated by the KDF since the last change of root key, and generates a simulated local state that is consistent with the given sequence.

The above security requirements are formalized by requiring that there exists a simulator such that no adversary can distinguish between a game where the adversary interacts with a stateful oracle that represents an execution of an actual $KDF$, and a game where the oracle represents an execution of an ideal system that exhibits the properties described above, along with a simulator that captures the allowable leakage upon exposure of the internal state.

In each one of the two executions, the adversary can repeatedly make one of three queries: either (Compute, root_input), (R-Advance), or (Expose-Advance).

In the real execution, first a random secret state root_key is chosen. Next, an adversarial query (Compute, root_input) simply returns chain_key $\leftarrow$ **Compute**(root_input, root_key) and updates root_key accordingly. Query R-Advance runs **Advance**(root_input) for a randomly chosen root_input and updates root_key accordingly. Query (Expose-Advance) returns the current value of root_key, and then runs **Advance**(root_input) for a randomly chosen root_input.

In the ideal execution, (Compute, root_input) applies a random function to root_input, R-Advance switches to a new random function, and Expose-Advance returns a value $s'$ that is generated by the simulator given the list of queries made by the adversary since the last R-Advance and the corresponding responses and then switches to a new random function. A more formal presentation follows.

**Cascaded PRF-PRG Security Game**

The Cascaded PRF-PRG security game for a KDF (Compute, Advance) with domain $\{0,1\}^n$ for the initial secret state, domain $\{0,1\}^{m(n)}$ for the chaining keys and domain $\{R_n\}_{n\in N}$ for the randomizer, and a simulator $\mathcal{S}$, proceeds as follows:

**Real game:**

- Oracle $O$ is initialized with random state $s \leftarrow \{0,1\}^n$.
- On input (Compute, root_input): $O$ runs **Compute**($s$, root_input) = (chain_key, root_key$'$), outputs chain_key and changes state $s$ = root_key$'$.
- On input (R-Advance): $O$ chooses root_input$'$ $\leftarrow R_n$ at random, runs **Advance**($s$, root_input$'$) = root_key$'$, and changes state $s$ = root_key$'$.
- On input (Expose-Advance): $O$ outputs the old state $s$, chooses root_input$'$ $\leftarrow R_n$ at random, computes **Advance**($s$, root_input$'$) = root_key$'$, and changes state $s$ = root_key$'$.

**Ideal game:**

- Oracle $O$ is initialized with a state consisting of a random function $F : R_n \rightarrow \{0,1\}^m$.
- On input (Compute, root_input): $O$ outputs $F$(root_input) = chain_key.
- On input (R-Advance): $O$ updates its state to a new random function $F : R_n \rightarrow \{0,1\}^m$.
- On input (Expose-Advance): $O$ outputs $\mathcal{S}$((root_input$_1$, $F$(root_input$_1$)), ..., (root_input$_k$, $F$(root_input$_k$)), where root_input$_1$, ..., root_input$_k$ are all the queries made by $\mathcal{A}$ since the last Advance query and $F$ is the currently used random function. Finally $O$ updates its state to a new random function $F : R_n \rightarrow \{0,1\}^m$.

Figure 14: Cascaded PRF-PRG Security Game

**Definition 11 (Cascaded PRF-PRG (CPRFG))** *Let $m$ be a polynomial, $n \in \mathbb{N}$. A module $KDF$ = (**Compute**, **Advance**) with secret state $s \in \{0,1\}^n$, input* root_input $\in \{0,1\}^n$*, and output length $m(n)$ is a* cascaded PRF-PRG (CPRFG) *if there exist polytime algorithm $\mathcal{S}$ such that any polytime oracle machine $\mathcal{A}$ can distinguish between the real and ideal interactions described above only with advantage that's negligible in $n$.*

**Constructing CPRFGs.** We first note that constructing a CPRFG in the programmable random oracle model is straightforward. Simply set:

$$\begin{aligned}\textbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) &= (H_c(\mathsf{root\_key}, \mathsf{root\_input}), \mathsf{root\_key}) \\ \textbf{Advance}(\mathsf{root\_key}, \mathsf{root\_input}) &= H_s(\mathsf{root\_key}, \mathsf{root\_input})\end{aligned}$$

where $H_c, H_s$ are two random functions. That is, **Compute** matches a random net chaining key with each $(\mathsf{root\_key}, \mathsf{root\_input})$ pair and does not change the local state. **Advance** simply provides a new random state for each $(\mathsf{root\_key}, \mathsf{root\_input})$ pair. The simulator will just provide a random value for $\mathsf{root\_key}$ and program $H_c, H_s$ in the right locations to match. In this way, the HKDF function [45] also meets our definition of a CPFRG, if the underlying HMAC is modeled as a random oracle.

We also present a construction of a CPRFG from standard cryptographic primitives, specifically puncturable PRFs and PRGs. We describe the construction in stages:

As a first approximation, can have $\mathsf{root\_key} = \mathsf{root\_key}_1, \mathsf{root\_key}_2$ and:

$$\begin{aligned}\textbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) &= (f_{\mathsf{root\_key}_1}(\mathsf{root\_input}), \mathsf{root\_key})\\ \textbf{Advance}(\mathsf{root\_key}, \mathsf{root\_input}) &= g_{\mathsf{root\_key}_2}(\mathsf{root\_input})\end{aligned}$$

where $\{f\}$ and $\{g\}$ are any two pseudorandom function families with the appropriate output lengths. Indeed, as long as no $\mathsf{Expose\text{-}Advance}$ queries are made, this construction would suffice, as demonstrated in [8]. (The use of two separate function families is for sake of exposition only; it is not essential for security.)

As a second approximation, we allow $\mathsf{Expose\text{-}Advance}$ queries, but only immediately after an $\mathtt{R}\text{-}\mathsf{Advance}$ query. In this case, plain PRFs may not suffice since the new root key may no longer be pseudorandom when the PRF key is exposed. However, as in Alwen et al. [1], this can be easily fixed by making sure that the pseudorandom function family $\{g\}$ is a PRF-PRG, namely $g_a(\cdot)$ is a pseudorandom number generator for a random and public $a$.

Finally, assume that $\mathsf{Expose\text{-}Advance}$ queries can be made even immediately after $\mathsf{Compute}$ queries. Now the simulator appears to be in a bind. Let $k$ be the number of $\mathsf{Compute}$ queries made since the last $\mathtt{R}\text{-}\mathsf{Advance}$ or $\mathsf{Expose\text{-}Advance}$ query. In the real game, the adversary obtains a value $\mathsf{root\_key}$ that is consistent with the response $\mathsf{chain\_key}_i$ provided to each query $(\mathsf{Compute}, \mathsf{root\_input}_i)$ made by the adversary since the last **Advance** operation (i.e., $f_{\mathsf{root\_key}}(\mathsf{root\_input}_i) = \mathsf{chain\_key}_i$ holds for $i = 1, \ldots, k$). In contrast, in the ideal game the simulator is given the set of keys $(\mathsf{root\_input}_1, \mathsf{chain\_key}_1), \ldots, (\mathsf{root\_input}_k, \mathsf{chain\_key}_k)$ and is expected to come up with $\mathsf{root\_key}$ such that $\textbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input}_i) = \mathsf{chain\_key}_i$ for all $i \in \{1, \ldots, k\}$. Clearly, finding an appropriate value for $\mathsf{root\_key}$ is hard. If $k$ is large enough, then such values would not exist at all with high probability.

We get around this difficulty by having the real **Compute** operation, at each generation of an output value $f_{\mathsf{root\_key}}(\mathsf{root\_input})$, update the local state so that the new state consists of two separate values: the value $f_{\mathsf{root\_key}}(\mathsf{root\_input})$, plus a residual value $\mathsf{root\_key}'$ that allows evaluating $f_{\mathsf{root\_key}}$ at all points except $\mathsf{root\_input}$, and is computationally independent from the value $f_s(\mathsf{root\_input})$. Specifically, we let $f$ be a *puncturable* PRF (as in [15, 17, 43]), and have $\textbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) = (\mathsf{chain\_key}, (\mathsf{root\_key}', (\mathsf{root\_input}, \mathsf{chain\_key}))$ where $\mathsf{chain\_key} = f_{\mathsf{root\_key}}(\mathsf{root\_input})$ and $\mathsf{root\_key}'$ is the result of puncturing $\mathsf{root\_key}$ at point $\mathsf{root\_input}$.

So, in sum, our final construction has state $\mathsf{root\_key} = (\mathsf{root\_key}_1, \mathsf{root\_key}_2)$, and:

$$\begin{aligned}\textbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) &= (\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input}), \mathsf{root\_key}')\\ \textbf{Advance}(\mathsf{root\_key}, \mathsf{root\_input}) &= g_{\mathsf{root\_key}_2}(\mathsf{root\_input})\end{aligned}$$

where $\{g\}$ is a PRF-PRG, and $\mathsf{root\_key}' = (\mathsf{root\_key}'_1, \mathsf{root\_key}_2))$. The values $\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input})$ and $\mathsf{root\_key}'_1$ are computed as follows: Parse $\mathsf{root\_key}_1 = (\widehat{\mathsf{root\_key}}_1, (a_1, b_1), ..., (a_k, b_k))$. Then, if $\mathsf{root\_input} = a_i$ for some $i$ then let $\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input}) = b_i$ and $\mathsf{root\_key}'_1 = \mathsf{root\_key}_1$. Else, let

$\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input}) = f_{\widehat{\mathsf{root\_key}}_1}(\mathsf{root\_input})$, and let

$$\mathsf{root\_key}_1' = (\widehat{\mathsf{root\_key}}_1', (a_1, b_1), ..., (a_k, b_k), (\mathsf{root\_input}, f_{\widehat{\mathsf{root\_key}}_1}(\mathsf{root\_input})),$$

where $\widehat{\mathsf{root\_key}}_1'$ is the result of puncturing $\widehat{\mathsf{root\_key}}_1$ at point $\mathsf{root\_input}$. Furthermore, once a key is advanced, the old key is erased.

**Theorem 12** *Assume that $\{f\}$ is a puncturable PRF and $\{g\}$ is a PRF-PRG. Then the above KDF is a cascaded PRF-PRG.*

**Proof:** Consider the following simulator $\mathcal{S}$: Given $((\mathsf{root\_input}_1, r_1), ..., (\mathsf{root\_input}_k, r_k))$, $\mathcal{S}$ chooses random $n$-bit keys $s_0$ and $s_2$, sets $\widehat{s}_1 = s_0$, and for $i = 2, \ldots, k$ sets $\widehat{s}_i$ as the result of puncturing $\widehat{s}_{i-1}$ at point $\mathsf{root\_input}_{i-1}$. Finally, $\mathcal{S}$ outputs the simulated state $s = (s_1, s_2)$ where $s_1 = (\widehat{s}_k, (\mathsf{root\_input}_1, r_1), ..., (\mathsf{root\_input}_k, r_k))$.

We argue that an adversary $\mathcal{A}$ that distinguishes between the real and ideal games can be used to break either the fact that $\{f\}$ is a puncturable PRF, or the fact that $\{g\}$ is a PRF-PRG.

The proof uses the following hybrid argument. Let the $i$-th hybrid experiment $H_i$ consist of the real CPRFG game until the $i$-th Expose–Advance event, at which point the remainder of the interaction happens in the ideal game. Fix a CPRFG adversary $\mathcal{A}$. Suppose $\mathcal{A}$ can distinguish $H_i$ from $H_{i+1}$. We will reduce this to either the puncturable PRF property of $\{f\}$ or the PRF-PRG property of $\{g\}$.

There are two cases to consider. First, suppose the $i$-th Expose–Advance occurs before any root_input's have been computed for the current root_key (i.e., the expose is occurring at the very beginning of the epoch, before any attempts have been made). In this case, we will construct the following adversary $\mathcal{A}_{PRFG}$ against the PRF-PRG property of $\{g\}$. On input $r$ that is either truly random or the output of a PRG-PRG:

- Emulate the real CPRFG game to $\mathcal{A}$ up until the $i$-th Expose–Advance event.
- On the $i$-th Expose–Advance event, set $\mathsf{chain\_key} \leftarrow r$, the PRF-PRG game randomness.
- Emulate the ideal CPRFG game to $\mathcal{A}$ for the remainder of the game.

First, note that the case where the chain_key in the $i$-th event is generated by a PRF-PRG corresponds to a perfect emulation of hybrid $H_i$ to $\mathcal{A}$, and similarly the case where the chain_key in the $i$-th event is truly random corresponds to a perfect emulation of hybrid $H_{i+1}$ to $\mathcal{A}$, for the simulator $\mathcal{S}$ given.

Since $\mathcal{A}$ can distinguish between $H_i$ and $H_{i+1}$ the distinguishing advantage of $\mathcal{A}_{PRFG}$ is identical to the distinguishing advantage of $\mathcal{A}$. Thus, under the assumption that $\{g\}$ is a PRF-PRG, they are both negligible in the security parameter.

For the second case, suppose the $i$-th Expose–Advance occurs *after* some chain_keys have been computed for the current root_key. We will construct an adversary $\mathcal{A}_{PPRF}$ against the puncturable PRF property of $\{f\}$.

- Emulate the real CPRFG game to $\mathcal{A}$ up until the $i$-th Expose–Advance event.
- On the $i$-th Expose–Advance event, puncture $f$ with the root_input's of all **Compute** queries that $\mathcal{A}$ has made for the current epoch.
- Emulate the ideal CPRFG game to $\mathcal{A}$ for the remainder of the game.

Again, note that the emulation of either hybrid $H_i$ or $H_{i+1}$ (corresponding to whether $f$ is truly random or pseudorandom) to $\mathcal{A}$ are *perfect*, for the simulator $\mathcal{S}$ given.

Since $\mathcal{A}$ can distinguish between $H_i$ and $H_{i+1}$, the distinguishing advantages of $\mathcal{A}$ and $\mathcal{A}_{PPRF}$ are *identical*. Thus, by the assumption that $\{f\}$ is a puncturable PRF, both distinguishers have a negligible probability of success.

Lastly, since the CPRFG adversary $\mathcal{A}$ runs in polynomial time in the security parameter, there can be at most a polynomial number of Expose-Advance events–and thus, hybrids. Overall, $\mathcal{A}$'s distinguishing advantage over all hybrids remains negligible.

□

Observe that the size of the state of above CPRFG grows roughly linearly with the number of applications of **Compute** between two consecutive applications of **Advance**, which may in principle lend to a denial of service attack on the protocol. However, we argue that a linear growth in space is unavoidable in plain model constructions – not only for the notion of CPRFG, but also for realizing $\mathcal{F}_{\mathsf{eKE}}$ in the presence of security against adaptive corruptions [51]. Furthermore, it may be reasonable to mitigate such denial of service attacks by imposing an a priori bound on the number of failed attempts to advance an epoch before the protocol raises an alarm to the user.

## 6.2 Protocol $\Pi_{\mathsf{eKE}}$

The epoch key exchange protocol (see Figure 15) mirrors the public key ratchet from the Signal architecture. It persists for the entire duration of the secure messaging session between two parties. The rest of this section describes how it instantiates the four methods from $\mathcal{F}_{\mathsf{eKE}}$ using a Cascaded PRF-PRG protocol $\Pi_{KDF}$.

Recall that each of the two parties will run their own copy of epoch key-exchange protocol $\Pi_{\mathsf{eKE}}$. This is different from the ideal world where the parties share a copy of the epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$ that they both interact with. One cosmetic impact of this is that each party's protocol must be initialized on first activation. The functionality $\mathcal{F}_{\mathsf{eKE}}$ is initialized just once, on the first call to `ConfirmReceivingEpoch` by the initiator of the conversation. Since there are two copies of the protocol, each must be initialized separately. The initiator's instance of the protocol $\Pi_{\mathsf{eKE}}$ is also initialized on the first call to `ConfirmReceivingEpoch` by the initiator. However, the responder's instance of $\Pi_{\mathsf{eKE}}$ is initialized on the first call to `GetReceivingKey` by the responder since this is the first action the responder must take.

The protocol has two basic components: (1) A continuous Diffie-Hellman ratchet between the two parties which provides both forward secrecy and healing from compromise. This is the place where parties can add fresh randomness into the system. (2) A KDF-chain that provides immediate decryption for the secure messaging system. This chain leverages the healing property of the Diffie-Hellman ratchet and extends the forward secrecy property. We briefly describe each of the methods of $\Pi_{\mathsf{eKE}}$ below. The full details can be found in Figure 15 on Page 50.

Each of the parties involved in the protocol execution run an instance of $\Pi_{\mathsf{eKE}}$. Just like in $\mathcal{F}_{\mathsf{eKE}}$ the parties take turns starting epochs in which they will send messages. They achieve this interleaving by updating their sending randomness (via a new Diffie-Hellman pair) as soon as they get confirmation of a new Diffie-Hellman pair being used by the other party. This means that most of the work happens when the parties successfully receive a message in the other party's newest sending epoch. In that vein, let's start by briefly describing the `GetReceivingKey` method.

`GetReceivingKey` When a party receives a new public Diffie-Hellman value $\mathsf{epoch\_id}$ from its partner, it runs the method $(\texttt{GetRotatingKey}, \mathsf{epoch\_id})$ to produce a tentative $\mathsf{recv\_chain\_key}$.

This key is then confirmed to be the correct value (or confirmed to be wrong) by the party out of band of the epoch key exchange protocol.

`ConfirmReceivingEpoch` If a new public Diffie-Hellman value $\mathsf{epoch\_id}$ produces a key that the party confirms to be correct (with this confirmation being out of band of the epoch key exchange protocol), then the party will run the `ConfirmReceivingEpoch` method to update the both the Diffie-Hellman ratchet and the KDF-chain accordingly. (This update enatils the overwriting of the partner's old epoch id with the new one and the overwriting of the old $\mathsf{root\_key}$ with the new one output by $\Pi_{KDF}$.) The party also knows at this point that it must update its own sending randomness – so this method additionally updates both the Diffie-Hellman ratchet and the KDF chain again according to a randomly chosen Diffie-Hellman pair $(\mathsf{epoch\_id}', \mathsf{epoch\_key}')$, this update also produces a new $\mathsf{sending\_chain\_key}$ value (output by $\Pi_{KDF}$) which is stored temporarily till the party retrieves it using the `GetSendingKey` method. (This second update also entails overwriting of old value. This time, the party's old Diffie-Hellman pair is overwritten along with the old $\mathsf{root\_key}$.)

`GetSendingKey` This method simply outputs the stored $\mathsf{sending\_chain\_key}$ value and then deletes it. If the value has already been deleted then it does nothing.

Now we briefly discuss the initialization of the protocol for each party and the corrupt method that exists only for record keeping purposes.

**Initiator Initialization** The first time the initiator runs the `ConfirmReceivingEpoch` method of $\Pi_{\mathsf{eKE}}$, the method must initialize the KDF-chain and the Diffie-Hellman ratchet. It then updates both using a randomly chosen Diffie-Hellman pair $(\mathsf{epoch\_id}', \mathsf{epoch\_key}')$, and temporarily store the produced $\mathsf{sending\_chain\_key}$ value till the party retrieves it using the `GetSendingKey` method. To initialize the Diffie-Hellman Ratchet, the party retrieves the responder's keys from directory functionality $\mathcal{F}_{\mathsf{DIR}}$ using the `GetInitKeys` method. It can then use the `ComputeSendingRootKey` method of its long-term memory functionality $\mathcal{F}_{\mathsf{LTM}}$ to run a triple Diffie-Hellman on both parties' keys. This initializes the Diffie-Hellman ratchet and KDF-chain, it also binds the conversation to the longterm identity keys of the two parties.

**Responder Initialization** The first time the responder runs the `GetReceivingKey` method of $\Pi_{\mathsf{eKE}}$, the method must initialize the KDF-chain and Diffie-Hellman ratchet of the responder in much the same way as the initialization of the Initiator that happens on the its first call to `ConfirmReceivingEpoch`. After this, the steps of the `GetReceivingKey` method are run like they will be for the rest of the conversation.

**Corrupt** The corrupt method in Figure 15 defines the model of corruption we are considering. On corruption, $\Pi_{\mathsf{eKE}}$ returns its internal state containing: $(\mathsf{epoch\_key}, \mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{partner}}, \mathsf{root\_key})$. Note that, as with the other protocols, the `Corrupt` method is not a "real" interface, but is only record keeping for the purposes of the model.

**$\Pi_{\mathsf{eKE}}$**

This protocol has a party id $\mathsf{pid}$ and session id $\mathsf{sid}.eKE$ of the form: $\mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid})$ where $\mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1)$.
This protocol uses protocols keyGen and $\Pi_{KDF}$ as subroutines: (1) keyGen chooses a random Diffie-Hellman exponent $\mathsf{epoch\_key} \overset{\$}{\leftarrow} |G|$ for a known group $G$ and sets $\mathsf{epoch\_id} = g^{\mathsf{epoch\_key}}$. It then outputs $(\mathsf{epoch\_key}, \mathsf{epoch\_id})$. (2) The protocol $\Pi_{KDF}$ is a cascaded PRF-PRG.

//The method `ConfirmReceivingEpoch` is also used by the initiator of the conversation to start the first sending epoch (epoch 0) before it has any receiving epochs to confirm.

**ConfirmReceivingEpoch:** On input $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}^*)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}')$:

1. If $\mathsf{pid}' \neq \mathsf{pid}$, then end the activation. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathsf{temp\_epoch\_id\_partner}_i = \mathsf{epoch\_id}^*$.
3. If this is the first activation:
   - Initialize state variables $\mathsf{root\_key}, \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Send $(\texttt{GetInitKeys}, \mathsf{pid}_{1-i}, \mathsf{pid}_i)$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving $(\texttt{GetInitKeys}, \mathsf{ik}^{\mathsf{pk}}_j, \mathsf{rk}^{\mathsf{pk}}_j, \mathsf{ok}^{\mathsf{pk}}_{j\leftarrow i})$, send input $(\texttt{ComputeSendingRootKey}, \mathsf{ik}^{\mathsf{pk}}_j, \mathsf{rk}^{\mathsf{pk}}_j, \mathsf{ok}^{\mathsf{pk}}_{j\leftarrow i})$ to $\mathcal{F}_{\mathsf{LTM}}$.
   - Upon receiving $(\texttt{ComputeSendingRootKey}, k, \mathsf{ek}^{\mathsf{pk}}_i)$, set $\mathsf{root\_key} = k$.
   - Run the subroutine **Compute Sending Chain Key**.
   - Erase $\mathsf{ek}^{\mathsf{pk}}_i$ and output $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}_{\mathsf{self}}||\mathsf{ek}^{\mathsf{pk}}_i||\mathsf{ok}^{\mathsf{pk}}_{j\leftarrow i})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$
4. Else (this is not the first activation):
   - Run the steps in **Compute Sending Chain Key**.
   - Output $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}_{\mathsf{self}})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

**GetSendingKey:** On receiving input $(\texttt{GetSendingKey})$ from $(\Pi_{mKE}, \mathsf{sid}.mKE, \mathsf{pid}')$:

1. If $\mathsf{pid}' \neq \mathsf{pid}$, or if $\mathsf{sending\_chain\_key}$ has already been erased, end the activation.
2. Output $(\texttt{GetSendingKey}, \mathsf{sending\_chain\_key})$ and erase $\mathsf{sending\_chain\_key}$.

**GetReceivingKey:** On receiving input $(\texttt{GetReceivingKey}, \mathsf{epoch\_id})$ from $(\Pi_{mKE}, \mathsf{sid}, \mathsf{pid}')$:

1. If $\mathsf{pid}' \neq \mathsf{pid}$, then end the activation. Otherwise, let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathsf{temp\_epoch\_id\_partner} = \mathsf{epoch\_id}$.
3. If this is the first activation:
   - Initialize state variables $\mathsf{root\_key}, \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Parse $\mathsf{epoch\_id} = (\mathsf{epoch\_id}', \mathsf{ek}^{\mathsf{pk}}_j, \mathsf{ok}^{\mathsf{pk}}_{i\leftarrow j})$ and set $\mathsf{temp\_epoch\_id\_partner} = \mathsf{epoch\_id}'$
   - Send $(\texttt{GetResponseKeys}, \mathsf{pid}_{1-i})$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving $(\texttt{GetResponseKeys}, \mathsf{ik}^{\mathsf{pk}}_j)$, send input $(\texttt{ComputeReceivingRootKey}, \mathsf{ik}^{\mathsf{pk}}_j, \mathsf{ek}^{\mathsf{pk}}_j, \mathsf{ok}^{\mathsf{pk}}_{i\leftarrow j})$ to $\mathcal{F}_{\mathsf{LTM}}$.
   - Upon receiving $(\texttt{ComputeReceivingRootKey}, k)$, set $\mathsf{root\_key} = k$.
4. **Compute Receiving Chain Key.**
5. Output $(\texttt{GetReceivingKey}, \mathsf{temp\_recv\_chain\_key})$ and erase $\mathsf{temp\_recv\_chain\_key}$.

Figure 15: The Epoch Key Exchange Protocol $\Pi_{\mathsf{eKE}}$

**$\Pi_{\mathsf{eKE}}$ continued...**

**Corrupt:** On receiving (`Corrupt`) from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$: return $(\mathsf{epoch\_key}, \mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{partner}}, \mathsf{root\_key})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$
//Note that the `Corrupt` interface is not part of the "real" protocol; it simply serves to formalize what the adversary sees on corruption.

//Below are subroutines used in the interfaces above. This is where the cascaded PRF-PRG protocol $\Pi_{KDF}$ is used.

**Compute Sending Chain Key:**

1. If this is the first activation start at step 5.
2. Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathsf{epoch\_key}_{\mathsf{self}})$.
3. Compute $(\mathsf{root\_key}) = KDF.\textbf{Advance}(\mathsf{root\_key}, \mathsf{root\_input})$.
4. Generate a key pair $(\mathsf{epoch\_key}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{self}}) \leftarrow \text{keyGen}()$.
5. Compute the next input $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathsf{epoch\_key}_{\mathsf{self}})$.
6. Compute $(\mathsf{root\_key}, \mathsf{sending\_chain\_key}) = KDF.\textbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input})$.
7. Finally, advance $(\mathsf{root\_key}) = KDF.\textbf{Advance}(\mathsf{root\_key}, \mathsf{root\_input})$
8. Erase $\mathsf{root\_input}$. //The old root key is overwritten and therefore erased. The old sending chain key was already erased.

**Compute Receiving Chain Key:**

1. Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathsf{epoch\_key}_{\mathsf{self}})$.
2. Compute $(\mathsf{root\_key}, \mathsf{temp\_recv\_chain\_key}) = KDF.\textbf{Compute}(\mathsf{root\_key}, \mathsf{root\_input})$.
3. Erase $\mathsf{root\_input}$. //The old root key is overwritten and therefore already erased.

Figure 16: The Epoch Key Exchange Protocol $\Pi_{\mathsf{eKE}}$ (continued)

**Remark**

Here we discuss a simple modification to $\Pi_{\mathsf{eKE}}$, also mentioned in [1], that allows for the protocol to heal even faster with a small increase in communication. This modified protocol can realize a functionality $\mathcal{F}_{\mathsf{eKE}}$ that heals in 2 rounds instead of 3. Such a protocol can easily be substituted in place of the current $\Pi_{\mathsf{eKE}}$ to show that the resulting $\Pi_{\mathsf{SGNL}} + \Pi_{\mathsf{aead}} + \Pi'_{\mathsf{eKE}}$ system will realise functionality $\mathcal{F}_{\mathsf{SM}}$ that heals from corruption in $\geq 2$ rounds. This modification has each party use two Diffie-Hellman pairs when an epoch turns over, one for the party's new sending epoch, and one for the other party's next sending epoch when it responds. This allows both parties to delete the Diffie-Hellman exponents corresponding to their sending epochs as soon as they begin.

## 6.3 Security Analysis

In this subsection, we prove Theorem 3 that $\Pi_{\mathsf{eKE}}$ UC-realizes $\mathcal{F}_{\mathsf{eKE}}$.

**Theorem 3** *Assume that $KDF : \{0,1\}^n \to \{0,1\}^{2n}$ is a CPRFG, that the DDH assumption holds in the group $G$. Then protocol $\Pi_{eKE}$ UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ in the presence of global functionalities $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$.*

To do so, we construct an ideal-process adversary $\mathcal{S}_{eKE}$, and adversarial code $\mathcal{I}_{eKE}$. We show that, under the above assumptions, no polynomial-time environment $\mathsf{Env}$ can distinguish between an interaction with ($\mathcal{F}_{\mathsf{DIR}}$, $\mathcal{F}_{\mathsf{LTM}}$, $\mathcal{F}_{\mathsf{eKE}}$, $\mathcal{S}_{eKE}$, $\mathcal{I}_{eKE}$) and an interaction with ($\mathcal{F}_{\mathsf{DIR}}$, $\mathcal{F}_{\mathsf{LTM}}$, $\Pi_{eKE}$).

The detailed simulator with all the computations can be found in Figs. 18 to 19. We first describe the high level intuition behind the simulator and then follow up with a hybrid argument to complete our proof.

When neither party has ever been corrupted, the functionality runs the following internal code $\mathcal{I} = \mathcal{I}_{eKE}$.

**$\mathcal{I}_{eKE}$**

The keyGen($\cdot$) operation is the same one as from $\Pi_{\mathsf{eKE}}$. It chooses a random Diffie-Hellman exponent $\mathsf{epoch\_key} \xleftarrow{\$} |G|$ and sets $\mathsf{epoch\_id} = g^{\mathsf{epoch\_key}}$. It then outputs ($\mathsf{epoch\_key}, \mathsf{epoch\_id}$).

**GenEpochId:** On receiving ($\mathsf{state}_{\mathcal{I}}, \mathtt{GenEpochId}, i, \mathsf{epoch\_id}^*$) from ($\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE$) do:

1. Store $\mathsf{old\_epoch\_id}_i = \mathsf{epoch\_id}_i$ and $\mathsf{old\_epoch\_key}_i = \mathsf{epoch\_key}_i$ in $\mathsf{state}_{\mathcal{I}}$.
2. Sample a new pair ($\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i$) $\xleftarrow{\$}$ keyGen() and record $\mathsf{epoch\_id}_i$ in $\mathsf{state}_{\mathcal{I}}$.
3. Output ($\mathsf{state}_{\mathcal{I}}, \mathtt{GenEpochId}, i, \mathsf{epoch\_id}_i$) to ($\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE$).

Figure 17: Internal adversarial code $\mathcal{I}_{eKE}$

When a corruption occurs, the simulator must return the corrupted party's state to simulate a passive corruption. The simulator must then: (1) provide $\mathsf{sending\_chain\_keys}$ and $\mathsf{recv\_chain\_keys}$ for the parties to $\mathcal{F}_{\mathsf{eKE}}$ until a compromise of the real protocol would end (maybe never if the adversary chooses to person-in-the-middle the parties forever!), and (2) continue to return snapshots of the current party's state on new corruptions in a way that seems consistent with the previously produced values, even if the party is still compromised from a previous corruption.

To understand how the simulator goes about this, imagine that at corruption, the simulator creates dummy parties $\mathcal{P}_0, \mathcal{P}_1$. These dummy parties run the code of $\Pi_{\mathsf{eKE}}$ based on the state that the simulator provides them. If the simulator can produce a 'convincing state' for the corrupted party at the time of corruption, then simply running the parties as in the honest protocol and updating them based on the inputs provided by the environment will look indistinguishable from how an honest corruption would go. To flesh out this intuition, let's start by discussing how the simulator will handle a `ReportState` request from $\Pi_{\mathsf{eKE}}$.

**ReportState:** When the functionality $\mathcal{F}_{\mathsf{eKE}}$ receives a corruption notification from its calling protocol, it will send a request of the form ($\mathtt{ReportState}, i, \mathsf{recv\_chain\_key}$) to the simulator. The simulator must return 'the state of $\mathsf{pid}_i$'; this state will be returned to the calling protocol and in turn to the adversary.

The only variables stored in a party's state in $\Pi_{\mathsf{eKE}}$ are $\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{1-i}, \mathsf{root\_key}$. Note that $\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i$, and $\mathsf{epoch\_id}_{1-i}$ were chosen by the adversarial code $\mathcal{I}_{eKE}$ just like they would be in the real protocol and can be provided as-is. So now we need to figure out what $\mathsf{root\_key}$ the simulator should provide. Remember that any sending or receiving chain key not yet provided by $\mathcal{F}_{\mathsf{eKE}}$ will be chosen by the simulator in the future using these dummy parties who simply run the honest protocol on this provided state. So, the $\mathsf{root\_key}$ produced here only needs to take account past chain keys. Fortunately, because the output $KDF(\mathsf{input}, \mathsf{key})$ of a $KDF$ is indistinguishable from random if the $\mathsf{key}$ is hidden and random, and because the output gives away no information about the hidden $\mathsf{key}$ and $\mathsf{input}$, the current $\mathsf{root\_key}$ in the state of a party will be unrelated to all previous keys provided by $\mathcal{F}_{\mathsf{eKE}}$ in most cases. The only case where $\mathsf{root\_key}$ is related to a previous output of $\mathcal{F}_{\mathsf{eKE}}$ is when the manager for $\mathsf{pid}_{1-i}$ has already started a sending epoch whose chain key $\mathsf{recv\_chain\_key}$ has not yet been retrieved from $\mathcal{F}_{\mathsf{eKE}}$ by the receiving manager $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$. In this case, we need the provided $\mathsf{root\_key}$ to satisfy $KDF(\mathsf{epoch\_id}_{1-i}^{\mathsf{epoch\_key}_i}, \mathsf{root\_key}) = \mathsf{recv\_chain\_key}$. In these cases only, the functionality will provide $\mathsf{recv\_chain\_key}$ to the simulator at the time of making the request.

After $\mathsf{pid}_i$ is corrupted, the simulator will be able to provide all the keys for the parties by running the instructions for $\Pi_{\mathsf{eKE}}$ within the dummy parties and using the $\mathsf{epoch\_id}^*$ provided in the `GenEpochId` requests to know how to ratchet forward to the receiving epochs for each party. Once the functionality 'heals' from the corruption, it will stop asking the simulator for keys unless the parties have diverged. If divergence doesn't occur, the simulator must end the execution of the dummy parties and go back to the simulation instructions for the case of no corruptions. However, in the case of divergence, it must continue to run the dummy parties to provide chain keys.

The simulator can detect that the adversary has diverged the keys of the parties via the $\mathsf{epoch\_id}^*$'s provided in the `GenEpochId` requests, it can use this information along with the provided $\mathsf{epoch\_id}^*$'s diverge the keys of the parties in the functionality accordingly (in the way that the environment and adversary expect) using the `GetReceivingKey` and `GetSendingKey` requests by simply running the honest protocol instructions from $\Pi_{\mathsf{eKE}}$ for each party.

**Analysis of $\mathcal{S}_{\mathsf{eKE}}$.**

We demonstrate the validity of $\mathcal{S}_{\mathsf{eKE}}$ via the following hybrid executions that bridge the gap between a real-world execution (namely an execution with $\Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$) and an ideal execution (namely an execution with $\mathcal{F}_{\mathsf{eKE}}, \mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$).

### 6.3.1 Hybrid $H_0$ (Ideal World)

This is the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ in Fig. 8 along with the simulator $\mathcal{S}_{\mathsf{eKE}}$ in Fig. 18.

### 6.3.2 Hybrid $H_1$ (Env's view is identically distributed in this hybrid)

This hybrid sets up the format for transitioning from the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ to the use of a KDF chain while parties are not compromised and have not diverged. We replace the random choice of chain keys in `GetSendingKey` and `GetReceivingKey` with a random choice of function for every epoch, and randomly chosen inputs for these functions.

Note that when an epoch is compromised or the parties have diverged, all output values are provided to $\mathcal{F}_{\mathsf{eKE}}$ by the simulator who simply creates two emulated parties who run the protocol of $\Pi_{\mathsf{eKE}}$ as specified. (At initial compromise, the state of the emulated parties is chosen specifically so that it is indistinguishable from a state compromise in the real world. To do this $\mathcal{S}_{\mathsf{eKE}}$ take advantage of the simulator used to prove that the KDF protocol is a secure CPRFG. So, all

## Simulator $\mathcal{S}_{\mathsf{eKE}}$ for realizing $\mathcal{F}_{\mathsf{eKE}}$

$\mathcal{S}_{\mathsf{eKE}}$ only runs if the parties are diverged, or if the current epoch is compromised and we're within the quarantine period for the compromise.

The keyGen$(\cdot)$ and $\Pi_{KDF}$ components are the same ones from $\Pi_{\mathsf{eKE}}$: (1) keyGen chooses a random Diffie-Hellman exponent $\mathsf{epoch\_key} \xleftarrow{\$} |G|$ for a known group $G$ and sets $\mathsf{epoch\_id} = g^{\mathsf{epoch\_key}}$. It then outputs $(\mathsf{epoch\_key}, \mathsf{epoch\_id})$. (2) The protocol $\Pi_{KDF}$ is a cascaded PRF-PRG.

**At first activation:** Send $(\mathcal{F}_{\mathsf{eKE}}, \mathcal{I})$ to $\mathcal{F}_{\mathsf{lib}}$.

//`GetSendingKey` and `GetReceivingKey` are used as subroutines to answer `ReportState` and `GenEpochId` requests.
**GetSendingKey:** On receiving $(\texttt{GetSendingKey}, i)$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:

1. If $\mathcal{V}iew_i.\mathsf{sending\_chain\_key}$ exists then output $(\texttt{GetSendingKey}, i, \mathcal{V}iew_i.\mathsf{sending\_chain\_key})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ and delete $\mathcal{V}iew_i.\mathsf{sending\_chain\_key}$. Otherwise end the activation.

**GetReceivingKey:** On receiving $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:

1. Set $\mathcal{V}iew_i.\mathsf{temp\_epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}$
2. Compute $\mathcal{V}iew_i.\mathsf{root\_input} = \mathsf{Exp}(\mathcal{V}iew_i.\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}})$.
3. Compute $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key} = \mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
4. Erase $\mathcal{V}iew_i.\mathsf{root\_input}$ and $\mathcal{V}iew_i.\mathsf{temp\_epoch\_id}_{\mathsf{partner}}$.
5. Output $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ and delete $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key}$.

**ReportState:** On receiving $(\texttt{ReportState}, \mathsf{pid}_i, \mathsf{recv\_chain\_key}, \mathsf{leak})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:
//This method is run every time $\mathcal{F}_{\mathsf{eKE}}$ is informed that a party has been corrupted.

1. Add $\mathsf{epoch\_num}_i, \mathsf{epoch\_num}_i + 1, \mathsf{epoch\_num}_i + 2, \mathsf{epoch\_num}_i + 3$ to the list $\mathsf{compromised\_epochs}$ in $\mathsf{state}_{\mathcal{I}}$.
2. If $\mathcal{V}iew_0, \mathcal{V}iew_1$ already exist, output $(\texttt{ReportState}, \mathsf{pid}_i, \mathcal{V}iew_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$, otherwise continue to create them.
3. If $\mathsf{recv\_chain\_key} = \perp$, choose $\mathcal{V}iew_i.\mathsf{root\_key} \xleftarrow{\$} \{0,1\}^n$.
   //$\mathsf{pid}_i$ is in a sending state and $\mathsf{pid}_{1-i}$ has not yet started a newer sending epoch.
4. Otherwise, $\mathsf{recv\_chain\_key} \neq \perp$:
   //choose the root_key such that $\mathbf{Compute}(\mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i), \mathsf{root\_key}) = (\mathsf{root\_key}^*, \mathsf{recv\_chain\_key})$
   - Compute $\mathsf{leak}' = \{\mathsf{Exp}(\mathsf{epoch\_id}, \mathsf{epoch\_key}_i) \mid \mathsf{epoch\_id} \in \mathsf{leak}\}$.
   - Let $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$ and set $T = \{\mathsf{leak}', (\mathsf{root\_input}, \mathsf{recv\_chain\_key})\}$.
   - Finally, set $\mathsf{root\_key} = KDF\_Sim(T)$.
5. Create two dictionaries $\mathcal{V}iew_0, \mathcal{V}iew_1$ and set the following values in these objects:
   - If $\mathsf{epoch\_num}_i > \mathsf{epoch\_num}_{1-i}$: //$\mathsf{pid}_i$ is ahead
     - $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}_{1-i}$
     - $\mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{old\_epoch\_id}_i$
   - Else $(\mathsf{epoch\_num}_i < \mathsf{epoch\_num}_{1-i})$:
     - $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{old\_epoch\_id}_{1-i}$
     - $\mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}_i$
   - $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}_i$, $\mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}} = \mathsf{epoch\_key}_i$
   - $\mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}_{1-i}$, $\mathcal{V}iew_{1-i}.\mathsf{epoch\_key}_{\mathsf{self}} = \mathsf{epoch\_key}_{1-i}$.
   - $\mathcal{V}iew_i.\mathsf{root\_key} = \mathsf{root\_key}$, $\mathcal{V}iew_{1-i}.\mathsf{root\_key} = \mathsf{root\_key}$.
6. Output $(\texttt{ReportState}, \mathsf{pid}_i, \mathcal{V}iew_i = \{\mathsf{epoch\_key}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{partner}}, \mathsf{root\_key}\})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

//**Continued in Figure 19 on Page 56**

Figure 18: Simulator $\mathcal{S}_{\mathsf{eKE}}$ for realizing $\mathcal{F}_{\mathsf{eKE}}$

**Simulator $\mathcal{S}_{\mathsf{eKE}}$ continued...**

//This simulator starts in Figure 18 on Page 55

```
GenEpochId: On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:
//All the variables used and created here are within $\mathsf{state}_{\mathcal{I}}$.    //The variable $i$ denotes the party.

    1. Set $\mathsf{epoch\_num}_i$ += 2.
    2. Set $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}^*$.
    3. Compute $\mathcal{V}iew_i.\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}^*, \mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}})$.
    4. Set $\mathsf{root\_key} = \mathbf{Advance}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
       //Next, we check if the states have diverged.
    5. If $\mathcal{V}iew_i.\mathsf{root\_key} \neq \mathcal{V}iew_{1-i}.\mathsf{root\_key}$ then set $\mathsf{diverge\_parties} = true$.   //divergence occurs here
    6. If $\mathsf{epoch\_num}_i \notin \mathsf{compromised\_epochs}$ and $\mathsf{diverge\_parties} \neq true$):
       //If the views are in sync and this is the first uncompromised epoch after corruption.
       //Delete the view objects and generate a new epoch id as is done in the internal code.
        • Set $\mathsf{state}_{\mathcal{I}}[\mathsf{epoch\_id}_i] = \mathcal{V}iew_i[\mathsf{epoch\_id}_{\mathsf{self}}]$ and $\mathsf{state}_{\mathcal{I}}[\mathsf{epoch\_id}_{1-i}] = \mathcal{V}iew_{1-i}[\mathsf{epoch\_id}_{\mathsf{self}}]$.
        • Set $\mathsf{state}_{\mathcal{I}}[\mathsf{epoch\_key}_i] = \mathcal{V}iew_i[\mathsf{epoch\_key}_{\mathsf{self}}]$ and $\mathsf{state}_{\mathcal{I}}[\mathsf{epoch\_key}_{1-i}] = \mathcal{V}iew_{1-i}[\mathsf{epoch\_key}_{\mathsf{self}}]$.
        • Delete the objects $\mathcal{V}iew_0, \mathcal{V}iew_1$
        • Output $(\mathsf{state}_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch\_id}_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
    7. Otherwise ($\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$ or $\mathsf{diverge\_parties} = true$):
       //If the parties are still compromised or the views have diverged.
       //Continue to update $\mathcal{V}iew_i$ according to $\Pi_{\mathsf{eKE}}$.
        • Sample $(\mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}}, \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}}) \overset{\$}{\leftarrow} \mathrm{keyGen}()$.
        • Compute $\mathcal{V}iew_i.\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}^*, \mathcal{V}iew_i.\mathsf{epoch\_key}_s elf)$.
        • Set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key} = \mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
        • Set $\mathcal{V}iew_i.\mathsf{root\_key} = \mathbf{Advance}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
        • Erase $\mathcal{V}iew_i.\mathsf{root\_input}$
        • Output $(\texttt{GenEpochId}, i, \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
```

Figure 19: Simulator $\mathcal{S}_{\mathsf{eKE}}$ (continued)

we need to do is argue the indistinguishability of choices made within the functionality without consulting the simulator.)

### 6.3.3 Hybrid $H_2$

This hybrid transitions from using a version of the ideal CPRFG game to using the actual protocol. In particular, a random shared $\mathsf{root\_key}$ is chosen for the parties at first activation. Then, each ratchet forward chooses random inputs using the code of the simulator (like in the previous hybrid). Unlike the previous hybrid, we now use the **Compute** and **Advance** methods of the KDM to compute the chain keys and update the $\mathsf{root\_key}$ for each party instead of choosing a new random function. This hybrid is indistinguishable from the previous one because our KDM is a secure CPRFG.

### 6.3.4 Hybrid $H_3$

This hybrid replaces the randomly chosen $\mathsf{root\_input}$'s with the actual Diffie-Hellman keys. This hybrid will be computationally indistinguishable from the previous one because we assume that the $DDH$ assumption holds for group $G$.

### 6.3.5 Hybrid $H_4$ (Real World)

In this hybrid we will replace the initial root keys with root keys computed using $\mathcal{F}_{\mathsf{DIR}}$, $\mathcal{F}_{\mathsf{LTM}}$ as in $\Pi_{\mathsf{eKE}}$ instead of being chosen randomly. While the functionality $\mathcal{F}_{\mathsf{eKE}}$ automatically provides a binding between party id's and epoch keys up to the first compromise, the protocol $\Pi_{\mathsf{eKE}}$ realizes this binding via access to $\mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}$. Also, this initial root key is indistinguishable from a random choice because $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$ is not corruptible and it hides all Diffie-Hellman exponents belonging to $\mathsf{pid}$. In summary, this hybrid is indistinguishable from the previous one because:

- $\mathcal{F}_{\mathsf{LTM}}$ hides the exponents for all the group elements it chooses.
- The DDH property applies to the group that $\mathcal{F}_{\mathsf{LTM}}$ chooses elements from.
- $\mathcal{F}_{\mathsf{DIR}}$ reliably provide parties with each others' public diffie hellman halves $\mathsf{ik}, \mathsf{rk}$.
- $\mathcal{F}_{\mathsf{DIR}}$ reliably provides pairs of parties with the same public diffie hellman half $\mathsf{ok}$.
- The provided $\mathsf{ok}$ is unique to a pair of parties.

Note that this hybrid is the real protocol.

## 7 Unidirectional Forward Secure Authenticated Channels: Realising $\mathcal{F}_{\mathsf{fs\_aead}}$

The symmetric key chain guarantees forward security within an epoch while also supporting immediate decryption for out-of-order message arrivals. Because we wanted to allow for messages of any length, our message key computation proceeds in two steps: first, $\Pi_{\mathsf{mKE}}$ provides a short message $\mathsf{key\_seed}$ to $\Pi_{\mathsf{aead}}$ that is generated by applying a $\mathsf{PRG}$ iteratively to the $\mathsf{chain\_key}$; then, $\Pi_{\mathsf{aead}}$ expands this $\mathsf{key\_seed}$ to a $\mathsf{msg\_key}$ of the correct length by querying the global random oracle $\mathcal{F}_{\mathsf{pRO}}$. $\mathcal{F}_{\mathsf{aead}}$ proceeds with authenticated encryption and decryption as usual with this $\mathsf{msg\_key}$.

In this section, we show how protocol $\Pi_{\mathsf{fs\_aead}}$ (which uses ideal functionalities $\mathcal{F}_{\mathsf{mKE}}$ and $\mathcal{F}_{\mathsf{aead}}$ as subroutines) UC-realizes $\mathcal{F}_{\mathsf{fs\_aead}}$. We begin by defining these functionalities. In Section 7.1, we describe the message key exchange functionality, $\mathcal{F}_{\mathsf{mKE}}$. Next, we present the single-message authenticated encryption functionality in Section 7.2. In Section 7.3 we present protocol $\Pi_{\mathsf{fs\_aead}}$ and show that it UC-realizes $\mathcal{F}_{\mathsf{fs\_aead}}$, in the presence of $\mathcal{F}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$.

## 7.1 The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$

Below we describe the ideal message key exchange functionality, $\mathcal{F}_{\mathsf{mKE}}$. This functionality (See Figure 20 on page 59) provides message key management for a single epoch, mainly by keeping track of which message keys have been generated and retrieved and which message keys have been exposed during a state compromise.

## 7.2 The Single-Message Authenticated Encryption Functionality $\mathcal{F}_{\mathsf{aead}}$

Below we define an ideal single-message encryption functionality (See Figure 21 on page 60). On receiving an encryption request, $\mathcal{F}_{\mathsf{aead}}$ simply stores the message and adversarially-generated ciphertext in a table and on receiving a decryption request, it looks up the message. $\mathcal{F}_{\mathsf{aead}}$ consults the message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$ associated with the epoch to decide whether to allow or disallow encryption or decryption.

The ideal encryption's session id, $\mathsf{sid}.aead = (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{S}}, \mathsf{epoch\_id}_{\mathsf{R}}, b, \mathsf{msg\_num})$ contains information about the party id's of each party, the public $\mathsf{epoch\_id}$ for each party, which party is the sender (indicated by bit $b$), as well as the message number for the message. If it receives any requests with incorrect $\mathsf{sid}.aead$, the functionality ignores the request and the requester is activated.

## 7.3 Protocol $\Pi_{\mathsf{fs\_aead}}$

As outlined in Section 4, protocol $\Pi_{\mathsf{fs\_aead}}$ makes straightforward use of multiple instances of $\mathcal{F}_{\mathsf{aead}}$, along with $\mathcal{F}_{\mathsf{mKE}}$. It is presented in Figure 20 on page 59.

**Theorem 4** *Protocol $\Pi_{\mathsf{fs\_aead}}$ (perfectly) UC-realizes $\mathcal{F}_{\mathsf{fs\_aead}}$, in the presence of $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$, and $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.*

**Proof:**

We construct an ideal-process adversary $\mathcal{S}_{\mathsf{fs\_aead}}$ (See Figure 23 on page 64) that interacts with functionality $\mathcal{F}_{\mathsf{fs\_aead}}$. The objective of $\mathcal{S}_{\mathsf{fs\_aead}}$ is to simulate the interactions that would take place between the environment and the protocol $\Pi_{\mathsf{fs\_aead}}$ (together with its subroutine functionalities $\mathcal{F}_{\mathsf{mKE}}$ and $\mathcal{F}_{\mathsf{aead}}$), so that the views of the environment $\mathsf{Env}$ are perfectly identical in the real and ideal scenarios. When the simulator $\mathcal{S}_{\mathsf{fs\_aead}}$ receives messages from $\mathcal{F}_{\mathsf{fs\_aead}}$ in the ideal world, it takes the actions specified in the pseudocode on Fig. 23.

Observe that the APIs of $\mathcal{F}_{\mathsf{fs\_aead}}$ and $\Pi_{\mathsf{fs\_aead}}$ are identical: they each have 5 methods. In the remainder of the proof, we describe the actions taken by $\Pi_{\mathsf{fs\_aead}}$ in the real world and $\mathcal{F}_{\mathsf{fs\_aead}}$ together with $\mathcal{S}_{\mathsf{fs\_aead}}$ and explain why they maintain the property that the view of the real environment $\mathsf{Env}$ is identical when interacting with either the ideal functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ or the real protocol $\Pi_{\mathsf{fs\_aead}}$.

Without loss of generality, we restrict our attention to a dummy adversary $\mathcal{A}$ and a deterministic environment $\mathsf{Env}$. As a consequence, the entire execution is deterministic, since the message keys themselves are never used for encryption in $\mathcal{F}_{\mathsf{aead}}$. The proof proceeds by induction over the steps of the simulator.

**$\mathcal{F}_{\mathsf{mKE}}$**

This functionality has a session id $\mathsf{sid}.mKE$ that takes the following format: $\mathsf{sid}.mKE = (\text{“}mKE\text{”}, sid.fs)$. Where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$. The local session ID is parsed as $\mathsf{sid} = (sid', \mathsf{pid}_0, \mathsf{pid}_1)$. Inputs arriving from machines whose identity is neither $\mathsf{pid}_0$ nor $\mathsf{pid}_1$ are ignored.

This functionality is parametrized by a seed length $\lambda$

**RetrieveKey:** On receiving $(\texttt{RetrieveKey}, \mathsf{pid})$ from $(\Pi_{\mathsf{aead}}, \mathsf{sid}.aead)$, where $\mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num})$, or $\mathcal{F}_{\mathsf{aead}}$ if $\texttt{IsCorrupt?} = true$:

1. If this is the first activation,
   - Initialize dictionary $\mathsf{key\_dict}$ and variables $\texttt{IsCorrupt?} = \mathit{false}$, $\mathsf{msg\_num}_0, \mathsf{msg\_num}_1 = 0$.
   - Parse $\mathsf{sid}$ to recover the two party ids $(\mathsf{pid}_0, \mathsf{pid}_1)$.
2. If $\mathsf{pid} \notin \{\mathsf{pid}_0, \mathsf{pid}_1\}$ then end this activation.
3. End the activation if there is record $(\texttt{Retrieved}, i, \mathsf{msg\_num})$ or a record $(\texttt{StopKeys}, i, N)$ for $N < \mathsf{msg\_num}$.
4. If $\texttt{IsCorrupt?} = \mathit{false}$:
   - If $\mathsf{msg\_num} \in \mathsf{key\_dict}.keys$, set $k = \mathsf{key\_dict}[\mathsf{msg\_num}]$.
   - Else $(\mathsf{msg\_num} \notin \mathsf{key\_dict}.keys)$, set $k \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$.
5. Else $(\texttt{IsCorrupt?} = true)$:
   - Send $(\texttt{RetrieveKey}, \mathsf{pid}, \mathsf{msg\_num})$ to the the adversary.
   - Upon receiving $(\texttt{RetrieveKey}, \mathsf{pid}, k)$ from the adversary, continue.
6. Store $\mathsf{key\_dict}[\mathsf{msg\_num}] = k$.
7. If $\mathsf{msg\_num} > \mathsf{msg\_num}_i$, set $\mathsf{msg\_num}_i = \mathsf{msg\_num}$. //msg_num$_i$ is the largest successfully retrieved message by party $i$.
8. Record $(\texttt{Retrieved}, i, \mathsf{msg\_num})$ and output $(\texttt{RetrieveKey}, \mathsf{pid}, k)$ to $(\Pi_{\mathsf{aead}}, \mathsf{sid}.aead)$.

**StopKeys:** On receiving $(\texttt{StopKeys}, N)$ from $(\Pi_{fs\_aead}, \mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}, b), \mathsf{pid})$,

- Run steps 3-7 of $\texttt{RetrieveKey}$ for all $\mathsf{msg\_num}$ such that $\mathsf{msg\_num}_i < \mathsf{msg\_num} \leq N$.
- Record $(\texttt{StopKeys}, i, N)$ and output $(\texttt{StopKeys}, \texttt{Success})$.

**Corruption:** On receiving $(\texttt{Corrupt})$ from $(\Pi_{fs\_aead}, \mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}, b), \mathsf{pid})$:

1. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\texttt{IsCorrupt?} = true$, create empty lists $\mathsf{keys\_in\_transit}, \mathsf{pending\_msgs}$, and initialize $\mathsf{chain\_key} \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$.
3. If $\mathsf{msg\_num}_i = 0$ send $(\texttt{GetReceivingKey}, \mathsf{epoch\_id})$ to $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE, \mathsf{pid})$.
   //The chain key is selected at random unless the receiver is corrupted before retrieving any keys for the epoch, this is because later chain keys should be unrelated to the initial one due to the $PRG$ property. If the receiver has not retrieved any keys, we get the chain key from $\Pi_{\mathsf{eKE}}$ to provide to the simulator so that it matches the real world.
   //Also note that we only get corrupted if the sender has already initialized this box $\implies$ the sender's $\mathsf{msg\_num}$ will never be 0.
4. On receiving $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key})$ set $\mathsf{chain\_key} = \mathsf{recv\_chain\_key}$.
5. For all $\mathsf{msg\_num} \in \mathsf{key\_dict}.keys$, if there is no record $(\texttt{Retrieved}, i, \mathsf{msg\_num})$ then append $(\mathsf{msg\_num}, \mathsf{key\_dict}[\mathsf{msg\_num}])$ to $\mathsf{keys\_in\_transit}$ and append $\mathsf{msg\_num}$ to $\mathsf{pending\_msgs}$.
6. If there is a record $(\texttt{StopKeys}, i, N)$ then let $\mathsf{chain\_key} = \perp$.
7. Send $(\texttt{ReportState}, i, \mathsf{keys\_in\_transit}, \mathsf{msg\_num}_i, \mathsf{chain\_key})$ to $\mathcal{A}$.
8. On receiving a response $(\texttt{ReportState}, i, S)$ from $\mathcal{A}$:
   - Output $(\texttt{Corrupt}, \mathsf{pending\_msgs}, S)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid}_i)$.

Figure 20: The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$

$\mathcal{F}_{\mathsf{aead}}$

This functionality has a session id $\mathsf{sid}.aead = (\text{“}aead'', \mathsf{sid}.fs, \mathsf{msg\_num})$ where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1), \mathsf{epoch\_id})$. It also has *internal adversary code* $\mathcal{I} = \mathcal{I}_{\mathsf{aead}}$ (Fig. 28) .

We initialize the state for $\mathcal{I}$ to be $\mathsf{state}_{\mathcal{I}} = \perp$.

**Encryption:** On receiving $(\texttt{Encrypt}, m, N)$ from $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$:

1. If this is not the first encryption request or $\mathsf{pid} \notin (\mathsf{pid}_0, \mathsf{pid}_1)$, end the activation. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Provide input $(\texttt{RetrieveKey}, \mathsf{pid})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$
3. Upon receiving output $(\texttt{RetrieveKey}, \mathsf{pid}, k)$ from $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$:
   - If $k = \perp$ then end the activation. //The key is not available.
   - Else if $\texttt{IsCorrupt?} = \mathit{false}$:
     - Call $\mathcal{F}_{\mathsf{lib}}$ with input $\mathcal{F}_{\mathsf{aead}}$ to obtain the internal code $\mathcal{I}$.
     - Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, |m|, N)$, obtain the updated state $\mathsf{state}_{\mathcal{I}}$, and the output $(\texttt{Encrypt}, \mathsf{pid}, c)$.
     - Record the tuple $(c, k, m, N, 1)$, record $i$ as the sender, and set $\mathsf{ready2decrypt} = \mathit{true}$.
   - Else ($k \neq \perp$ and $\texttt{IsCorrupt?} = \mathit{true}$):
     - Send a backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, m, N)$ to $\mathcal{A}$.
     - Upon receiving a response $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, c)$, record the updated state $\mathsf{state}_{\mathcal{I}}$, record the tuple $(c, k, m, N, 1)$, record $i$ as the sender, and set $\mathsf{ready2decrypt} = \mathit{true}$.
   - Output $(\texttt{Encrypt}, c)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.

**Decryption:** On receiving $(\texttt{Decrypt}, c, N)$ from $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$:

1. If there hasn't been a successful encryption request, if $\mathsf{pid} \neq \mathsf{pid}_{1-i}$, or if $\mathsf{ready2decrypt} = \mathit{false}$ then output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
2. Provide input $(\texttt{RetrieveKey}, \mathsf{pid})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$.
3. Upon obtaining a response $(\texttt{RetrieveKey}, \mathsf{pid}, k)$ from $\mathcal{F}_{\mathsf{mKE}}$: If $k = \perp$ then output $(\texttt{Decrypt}, \texttt{Fail})$. //Failure of decryption can occur for an honest receiver so we need an explicit failure notification.
4. If there is a record $(c, k, m, N, 1)$, note $\mathsf{ready2decrypt} = \mathit{false}$ and output $(\texttt{Decrypt}, m)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
5. If there is a record $(c, N, 0)$, output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
6. If $\texttt{IsCorrupt?} = \mathit{false}$
   - Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, N)$ and obtain updated state $\mathsf{state}_{\mathcal{I}}$ and a value $v$ from $\mathcal{I}$.
   - If $v = \perp$ then record $(c, N, 0)$, and output $(\texttt{Decrypt}, \texttt{Fail})$.
   - Otherwise, note $\mathsf{ready2decrypt} = \mathit{false}$, and output $(\texttt{Decrypt}, m)$.
7. Else $(\texttt{IsCorrupt?} = \mathit{true})$
   - Send backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \mathsf{pid}, c, N)$ to $\mathcal{A}$.
   - Upon receiving response $(\texttt{inject}, \mathsf{pid}, c, N, v)$ from $\mathcal{A}$ continue.
   - If $v = \perp$ then record $(c, N, 0)$, and output $(\texttt{Decrypt}, \texttt{Fail})$.
   - Else, note $\mathsf{ready2decrypt} = \mathit{false}$, and output $(\texttt{Decrypt}, v)$.

**Corruption:** On receiving $(\texttt{Corrupt})$ from $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$:

1. End the activation if $\mathsf{pid} \notin (\mathsf{pid}_0, \mathsf{pid}_1)$. Otherwise, let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$. Else, set $\texttt{IsCorrupt?}$ to $\mathit{true}$, and send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}})$ to $\mathcal{A}$.
2. Upon receiving a response $(\texttt{ReportState}, \mathsf{pid}, S)$ from $\mathcal{A}$, send $(\texttt{Corrupt}, S)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

Figure 21: The Authenticated Encryption with Associated Data Functionality, $\mathcal{F}_{\mathsf{aead}}$

## $\Pi_{\mathsf{fs\_aead}}$

This protocol is active during a *single* epoch and has session id $\mathsf{sid}.fs$ that takes the following format: $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$.

**Encrypt:** On receiving input $(\mathtt{Encrypt}, m, N)$ from $(\Pi_{MAN}, \mathsf{sid}, \mathsf{pid})$ do:

1. Check that $\mathsf{sid}$ matches the one in the local session ID, and that $\mathsf{pid}$ matches the local party id, else abort.
2. If this is the first activation then initialize $\mathsf{curr\_msg\_num} = 0$.
3. Increment $\mathsf{curr\_msg\_num} += 1$.
4. Send $(\mathtt{Encrypt}, m, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$ and delete $m$.
5. Upon receiving $(\mathtt{Encrypt}, c)$, output $(\mathtt{Encrypt}, c)$ to $(\Pi_{MAN}, \mathsf{sid}, \mathsf{pid})$.

**Decrypt:** On receiving $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ do:

1. Check that $\mathsf{sid}$ matches the one in the local session ID, and that $\mathsf{pid}$ matches the local party id, else abort.
2. If this is the first activation then initialize $\mathsf{curr\_msg\_num} = 0$, $\mathsf{pending\_msgs} = []$.
3. Send $(\mathtt{Decrypt}, c, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
4. Upon receiving $(\mathtt{Decrypt}, v)$, if $v = \mathtt{Fail}$ then output $(\mathtt{Decrypt}, \mathtt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
5. Otherwise $(v \neq \mathtt{Fail})$:
   - While $\mathsf{curr\_msg\_num} < \mathsf{msg\_num}$:
     - Increment $\mathsf{curr\_msg\_num} += 1$.
     - Add $\mathsf{curr\_msg\_num}$ to $\mathsf{pending\_msgs}$.
   - Remove $\mathsf{msg\_num}$ from $\mathsf{pending\_msgs}$ and output $(\mathtt{Decrypt}, v)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

**StopEncrypting:** On receiving $(\mathtt{StopEncrypting})$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ do:

1. Check that $\mathsf{sid}$ matches the one in the local session ID, and that $\mathsf{pid}$ matches the local party id, else abort.
2. Send $(\mathtt{StopKeys}, \mathsf{msg\_num})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE)$.
3. On receiving $(\mathtt{StopKeys}, \mathtt{Success})$, output $(\mathtt{StopEncrypting}, \mathtt{Success})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

**StopDecrypting:** On receiving $(\mathtt{StopDecrypting}, \mathsf{msg\_num}^*)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ do:

1. Check that $\mathsf{sid}$ matches the one in the local session ID, and that $\mathsf{pid}$ matches the local party id, else abort.
2. Send $(\mathtt{StopKeys}, \mathsf{msg\_num}^*)$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE)$.
3. On receiving $(\mathtt{StopKeys}, \mathtt{Success})$:
   - While $\mathsf{curr\_msg\_num} < \mathsf{msg\_num}^*$:
     - Increment $\mathsf{curr\_msg\_num} += 1$.
     - Add $\mathsf{curr\_msg\_num}$ to $\mathsf{pending\_msgs}$.
   - Output $(\mathtt{StopDecrypting}, \mathtt{Success})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

**Corruption:** On receiving $(\mathtt{Corrupt})$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$:
//Note that the `Corrupt` interface is not part of the “real” protocol; it is only included for UC-modelling purposes.

1. Check that $\mathsf{sid}$ matches the one in the local session ID, and that $\mathsf{pid}$ matches the local party id, else abort.
2. Initialize a state object $S$ and add $\mathsf{curr\_msg\_num}, \mathsf{pending\_msgs}$ to it.
3. Send $(\mathtt{Corrupt})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE'', \mathsf{sid}.fs))$.
4. On receiving $(\mathtt{Corrupt}, S_{mKE})$, add it to $S$ and do the following.
5. For each record $\mathsf{msg\_num} \in \mathsf{pending\_msgs}$:
   - Send $(\mathtt{Corrupt}, \mathsf{pid}_i)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead'', \mathsf{sid}.fs, \mathsf{msg\_num}))$
   - On receiving a response $(\mathtt{Corrupt}, \mathsf{pid}_i, S_{\mathsf{msg\_num}})$, add $S_{\mathsf{msg\_num}}$ to $S$.
6. Output $(\mathtt{Corrupt}, S)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

Figure 22: The Forward-Secure Encryption $\Pi_{\mathsf{fs\_aead}}$

**Encrypt** In the real world, $\Pi_{\mathsf{fs\_aead}}$ outsources the encryption to separate $\mathcal{F}_{\mathsf{aead}}$ instances for each new message. Functionality $\mathcal{F}_{\mathsf{aead}}$ sends a `RetrieveKey` request to $\Pi_{\mathsf{mKE}}$ to check whether the message key seed is available, then $\mathcal{F}_{\mathsf{aead}}$ leaks either the message length or the full message (depending on whether the epoch is compromised) to the real world adversary $\mathcal{A}$ and returns the ciphertext provided by $\mathcal{A}$. In the ideal world, $\mathcal{F}_{\mathsf{fs\_aead}}$ sends the encrypt message to $\mathcal{S}_{\mathsf{fs\_aead}}$ (leaking either the message length or full message). $\mathcal{S}_{\mathsf{fs\_aead}}$ simulates the real world actions of $\mathcal{F}_{\mathsf{aead}}$ by retrieving the sending_chain_key from $\Pi_{\mathsf{eKE}}$ on behalf of $\Pi_{\mathsf{mKE}}$ on the first activation. It advances the sending_chain_key honestly using the same PRG as $\Pi_{\mathsf{mKE}}$ (and it ignores the message seeds that it generates). Finally, $\mathcal{S}_{\mathsf{fs\_aead}}$ sends an `Encrypt` request (with appropriate leakage) to the real world adversary $\mathcal{A}$, just as $\mathcal{F}_{\mathsf{aead}}$ does. It returns the ciphertext produced by $\mathcal{A}$, and $\mathcal{F}_{\mathsf{fs\_aead}}$ returns this ciphertext as well.

**Decrypt** In the real world, $\Pi_{\mathsf{fs\_aead}}$ again uses the appropriate $\mathcal{F}_{\mathsf{aead}}$ instance for decrypting ciphertexts. $\mathcal{F}_{\mathsf{aead}}$ then sends a `RetrieveKey` request to $\Pi_{\mathsf{mKE}}$ to check whether the message key seed is still available. If $\Pi_{\mathsf{mKE}}$ says that the key is available and if the ciphertext $c$ is exactly the one that $\mathcal{F}_{\mathsf{aead}}$ sent on encryption, then $\mathcal{F}_{\mathsf{aead}}$ outputs the message $m$ to $\Pi_{\mathsf{fs\_aead}}$. If, on the other hand, the ciphertext is different, then $\mathcal{F}_{\mathsf{aead}}$ asks the real world adversary $\mathcal{A}$ whether it wants to inject a message. On receiving the message $v$ from $\mathcal{A}$, if the epoch is compromised then $\mathcal{F}_{\mathsf{aead}}$ passes $v$ to $\Pi_{\mathsf{fs\_aead}}$ (since $\mathcal{A}$ should be able to forge this information after compromise). On the other hand, if the epoch is not compromised, $\mathcal{F}_{\mathsf{aead}}$ passes $m$ to $\Pi_{\mathsf{fs\_aead}}$ instead.

In the ideal world, $\mathcal{F}_{\mathsf{fs\_aead}}$ sends an `inject` request to $\mathcal{S}_{\mathsf{fs\_aead}}$. The simulator emulates what $\mathcal{F}_{\mathsf{aead}}$ does on receiving a `Decrypt` request, including making a `RetrieveKey` request to $\mathcal{F}_{\mathsf{mKE}}$. It emulates the `RetrieveKey` request by sending a `GetReceivingKey` to $\Pi_{\mathsf{eKE}}$ and later applying the PRG as in the real world execution to get the correct message key. The simulator then emulates $\mathcal{F}_{\mathsf{aead}}$ asking $\mathcal{A}$ whether it wants to inject a message if the ciphertext does not match the one it encrypted. Lastly, $\mathcal{S}_{\mathsf{fs\_aead}}$ iterates the message key seed forward and stores missed message key seeds and outputs the appropriate message to $\mathcal{F}_{\mathsf{fs\_aead}}$.

**Stop Encrypting** In the real world protocol, a `StopEncrypting` request causes $\Pi_{\mathsf{fs\_aead}}$ to send a `StopKeys` request to $\mathcal{F}_{\mathsf{mKE}}$ for the current message number. Then $\mathcal{F}_{\mathsf{mKE}}$ will note that the sending epoch is closed, thereby disallowing the sender (or adversary) from being able to encrypt more messages in the epoch.

In the ideal world functionality, $\mathcal{F}_{\mathsf{fs\_aead}}$ simply notes that the sender has deleted the ability to encrypt any more messages for the epoch, also preventing the sender from encrypting more messages for the epoch.

**Stop Decrypting** When $\Pi_{\mathsf{fs\_aead}}$ receives a (`StopDecrypting`, N) request from above, it sends (`StopKeys`, N) to the $\mathcal{F}_{\mathsf{mKE}}$ instance for the epoch. Then, $\mathcal{F}_{\mathsf{mKE}}$ will generate all message key seeds up to message number N and make a note that the epoch is closed for the receiver. This prevents the receiver (or adversary) from decrypting any messages past N for the epoch. The protocol $\Pi_{\mathsf{fs\_aead}}$ then stores all remaining message numbers for the epoch in its pending_msgs list.

The ideal world functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ simply marks all messages beyond N as inaccessible and returns control to $\Pi_{\mathsf{SGNL}}$.

**Corruption** The protocol $\Pi_{\mathsf{fs\_aead}}$, on receiving a `Corrupt` request from $\Pi_{\mathsf{SGNL}}$, sends `Corrupt` to $\mathcal{F}_{\mathsf{mKE}}$, which leaks to $\mathcal{A}$ a list of message seed keys in transit as well as the current message number for the corrupted party and the chain key. On recieving a state $S_{mKE}$ from $\mathcal{A}$, $\mathcal{F}_{\mathsf{mKE}}$ reports a list of all messages still in transit and $S_{mKE}$. Then, $\Pi_{\mathsf{fs\_aead}}$ corrupts each $\mathcal{F}_{\mathsf{aead}}$ instance that has a pending message, each of which leaks its message and key seed to $\mathcal{A}$. When $\mathcal{F}_{\mathsf{aead}}$ gets a state $S_{\mathsf{msg\_num}}$ back from $\mathcal{A}$, it reports this to $\Pi_{\mathsf{fs\_aead}}$. $\Pi_{\mathsf{fs\_aead}}$ then reports all of the states $S_{mKE}$ and $S_{\mathsf{msg\_num}}$'s along with the list of pending messages and current message number to $\Pi_{\mathsf{SGNL}}$.

The ideal functionality, on receiving `Corrupt`, sends a `ReportState` request to $\mathcal{S}_{\mathsf{fs\_aead}}$ and includes a list of the ciphertexts and messages that are in transit. The simulator $\mathcal{S}_{\mathsf{fs\_aead}}$ then emulates the corruptions of $\mathcal{F}_{\mathsf{mKE}}$ and $\mathcal{F}_{\mathsf{aead}}$ instances exactly as in the real world protocol using the honestly generated state from the other activations.

In conclusion, $\mathcal{S}_{\mathsf{fs\_aead}}$ gives an *exact* emulation of the real world for every action taken by Env and $\mathcal{A}$ in the ideal world, so the views of Env and $\mathcal{A}$ is identical to the real world. □

## 8 The Symmetric Ratchet: Realising $\mathcal{F}_{\mathsf{mKE}}$

As outlined in Section 4, protocol $\Pi_{\mathsf{mKE}}$ for realizing $\mathcal{F}_{\mathsf{mKE}}$ implements the symmetric keychain for the epoch specified in its session ID. This is done by obtaining the base key for the chain from $\mathcal{F}_{\mathsf{eKE}}$ and then extending the chain as needed. Importantly, the keys on the main symmeric chain are never directly given as output; rather the outputs are keys derived from the chain keys. This structure allows the key derivation function to only be a (length doubling) pseudorandom number generator. Protocol $\Pi_{\mathsf{mKE}}$ is presented in Figure 20 on page 59.

**Theorem 5** *Assume that* PRG *is a secure length-doubling pseudorandom generator. Then protocol* $\Pi_{\mathsf{mKE}}$ *UC-realizes* $\mathcal{F}_{\mathsf{mKE}}$ *in the presence of* $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$, *and* $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.

**Proof:**We construct an ideal-process adversary $\mathcal{S}_{\mathsf{mKE}}$ in the figure on Fig. 26 that interacts with functionality $\mathcal{F}_{\mathsf{mKE}}$. The objective of $\mathcal{S}_{\mathsf{mKE}}$ is to simulate the interactions that would take place between the environment and the protocol $\Pi_{\mathsf{mKE}}$ (in the presence of $\mathcal{F}_{\mathsf{eKE}} + \mathcal{S}_{\mathsf{eKE}}$), so that the views of the environment Env are computationally indistinguishable in the real and ideal scenarios. When the simulator $\mathcal{S}_{\mathsf{mKE}}$ receives messages from $\mathcal{F}_{\mathsf{mKE}}$ in the ideal world, it takes the actions specified in the pseudocode on Fig. 26.

Note that the real world adversary sees no keys before a compromise. If the epoch has been compromised, the real world adversary gets all key seeds for messages that are in transit. The real world adversary also gains the ability to compute all future key seeds available to the party (if `StopKeys` has not been called).

At such a compromise, the ideal world functionality $\mathcal{F}_{\mathsf{mKE}}$ provides the simulator $\mathcal{S}_{\mathsf{mKE}}$ with all the message numbers for which pending keys should be stored in the party's state, the party's current message number, as well as a chain key. The provided chain key is chosen at random in most cases, except for the case in which the party has not retrieved a single key in the epoch. In this case, the functionality provides $\mathcal{S}_{\mathsf{mKE}}$ with the initial chain key that a party would retrieve from the combined $\mathcal{F}_{\mathsf{eKE}} + \mathcal{S}_{\mathsf{eKE}}$ for this epoch to provide indistinguishability from the real world. The simulator $\mathcal{S}_{\mathsf{mKE}}$ then produces the keyseeds that are "in transit" by sampling them uniformly at random; it computes the future key seeds honestly using the chain_key provided to it during compromise. Since $\mathcal{S}_{\mathsf{mKE}}$ generates future keys just like $\Pi_{\mathsf{mKE}}$ would and otherwise uses the keys produced at compromise, if the state produced by the simulator at compromise is indistinguishable from the real world, then this proof is complete.

Note that if a uniformly random input is run through a PRG and then the output of the PRG run through the PRG again – and chained like this polynomially many times, the tuple containing all the outputs is still pseudorandom and therefore indistinguishable from outputs chosen independently

$\mathcal{S}_{\mathsf{fs\_aead}}$

**At first activation:** Send $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathcal{I})$ to $\mathcal{F}_{\mathsf{lib}}$.

**Encrypt:** On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, m)$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$ do:

1. If this is the first activation, initialize a dictionary $\mathsf{records}$.
   //Simulating a $\texttt{RetrieveKey}$ request to $\mathcal{F}_{\mathsf{mKE}}$
2. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
3. If this is the first activation for $\mathsf{pid}_i$:
   - Set $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} = 0$ and provide input $(\texttt{GetSendingKey})$ to $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid}))$ on behalf of $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE\text{”}, \mathsf{sid}.fs), \mathsf{pid})$.
   - Upon receiving $(\texttt{GetSendingKey}, \mathsf{sending\_chain\_key})$ from $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ set $\mathcal{V}iew_i.\mathsf{curr\_chain\_key} = \mathsf{sending\_chain\_key}$.
4. If $\mathsf{msg\_num} \neq \mathcal{V}iew_i.\mathsf{curr\_msg\_num} + 1$, then end the activation.
5. Otherwise increment $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$ and store it in $\mathsf{state}_{\mathcal{I}}$.
6. $(\mathcal{V}iew_i.\mathsf{curr\_chain\_key}, -) = PRG(\mathcal{V}iew_i.\mathsf{curr\_chain\_key})$.
7. Store $\mathcal{V}iew_i.\mathsf{curr\_chain\_key}$ in $\mathsf{state}_{\mathcal{I}}$
   //End simulation of $\texttt{RetrieveKey}$ from $\mathcal{F}_{\mathsf{mKE}}$
8. Parse $\mathsf{state}.aead.\mathsf{msg\_num}$ from $\mathsf{state}_{\mathcal{I}}$
9. Send a backdoor message $(\mathsf{state}.aead.\mathsf{msg\_num}, \texttt{Encrypt}, \mathsf{pid}, m, N)$ to $\mathsf{Env}$ on behalf of $(\mathcal{F}_{\mathsf{aead}}, \text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num})$.
10. Upon receiving a response $(\mathsf{state}, \texttt{Encrypt}, \mathsf{pid}, c)$ update $\mathsf{state}.aead.\mathsf{msg\_num} \leftarrow \mathsf{state}$ and add the tuple $(c, N, 1)$, the message $m$, sender $i$, and $\mathsf{ready2decrypt} = true$ to $\mathsf{records}[\mathsf{msg\_num}]$.
11. Update (or store for the first time) $\mathsf{records}[\mathsf{msg\_num}]$ in $\mathsf{state}_{\mathcal{I}}$
12. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, c)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

**Inject:** On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \mathsf{pid}, c^*, \mathsf{msg\_num}, N)$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$ do:

1. If this is the first activation then end the activation.
2. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
3. If this is the first activation for $\mathsf{pid}_i$, initialize $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} = 0$, $\mathcal{V}iew_i.\mathsf{missed\_msgs} = []$.
   //Simulating a send $(\texttt{Decrypt}, c, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
4. Get the message $m$, sender $i$, $\mathsf{ready2decrypt}$, and tuples of the form $(c, N, b)$, from $\mathsf{records}[\mathsf{msg\_num}]$ in $\mathsf{state}_{\mathcal{I}}$. If there is no such record or if $\mathsf{ready2decrypt} \neq true$, then output $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
   //Simulating a $\texttt{RetrieveKey}$ request to $\mathcal{F}_{\mathsf{mKE}}$
5. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
6. If this is the first activation for $\mathsf{pid}_i$ set $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} = 0$ and provide input $(\texttt{GetReceivingKey})$ to $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid}))$ on behalf of $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE\text{”}, \mathsf{sid}.fs), \mathsf{pid})$.
7. Upon receiving $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key})$ from $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ set $\mathcal{V}iew_i.\mathsf{curr\_chain\_key} = \mathsf{sending\_chain\_key}$.
8. Store $\mathcal{V}iew_i.\mathsf{curr\_chain\_key}$ in $\mathsf{state}_{\mathcal{I}}$ //End simulation of $\texttt{RetrieveKey}$ from $\mathcal{F}_{\mathsf{mKE}}$
9. Parse $\mathsf{state}.aead.\mathsf{msg\_num}$ from $\mathsf{state}_{\mathcal{I}}$
10. Send a backdoor message $(\mathsf{state}.aead.\mathsf{msg\_num}, \texttt{inject}, \mathsf{pid}, c^*, N)$ to $\mathcal{A}$ on behalf of $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid} = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
11. Upon receiving response $(\mathsf{state}, \texttt{inject}, \mathsf{pid}, c^*, v^*)$ from $\mathcal{A}$, update $\mathsf{state}.aead.\mathsf{msg\_num} \leftarrow \mathsf{state}$ in $\mathsf{state}_{\mathcal{I}}$.
12. If the tuple $(c^*, N, 1)$ is in $\mathsf{records}[\mathsf{msg\_num}]$, set $\mathsf{ready2decrypt} = false$ in the entry $\mathsf{records}[\mathsf{msg\_num}]$ and set $v = m$.
13. If the tuple $(c^*, N, 0)$ is in $\mathsf{records}[\mathsf{msg\_num}]$, output $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
14. Else (there is no record $(c^*, N, b)$ in $\mathsf{records}[\mathsf{msg\_num}]$),
    - If $v^* = \perp$ then record $(c^*, N, 0)$, and output $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
    - Else, set $\mathsf{ready2decrypt} = false$ in the entry $\mathsf{records}[\mathsf{msg\_num}]$, and set $v = v^*$.

    //End simulation of $\texttt{Decrypt}$ from $\mathcal{F}_{\mathsf{aead}}$
15. While $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} \leq \mathsf{msg\_num}$ do: //we only get to this step if decryption succeeds.
    - Increment $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$

$\mathcal{I}_{\mathsf{fs}}$

**Encrypt:** On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, |m|)$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$ do:

1. If this is the first activation, initialize a dictionary $\mathsf{records}$.
   //Simulating a `RetrieveKey` request to $\mathcal{F}_{\mathsf{mKE}}$
2. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
3. If this is the first activation for $\mathsf{pid}_i$:
   - Set $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} = 0$ and provide input $(\texttt{GetSendingKey})$ to $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE = (\text{"}eKE\text{"}, \mathsf{sid}))$ on behalf of $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{"}mKE\text{"}, \mathsf{sid}.fs), \mathsf{pid})$.
   - Upon receiving $(\texttt{GetSendingKey}, \mathsf{sending\_chain\_key})$ from $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ set $\mathcal{V}iew_i.\mathsf{curr\_chain\_key} = \mathsf{sending\_chain\_key}$.
4. If $\mathsf{msg\_num} \neq \mathcal{V}iew_i.\mathsf{curr\_msg\_num} + 1$ then end the activation.
5. Otherwise increment $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$ and store it in $\mathsf{state}_{\mathcal{I}}$.
6. $(\mathcal{V}iew_i.\mathsf{curr\_chain\_key}, -) = PRG(\mathcal{V}iew_i.\mathsf{curr\_chain\_key})$.
7. Store $\mathcal{V}iew_i.\mathsf{curr\_chain\_key}$ in $\mathsf{state}_{\mathcal{I}}$
   //End simulation of `RetrieveKey` from $\mathcal{F}_{\mathsf{mKE}}$ .
8. Parse $\mathsf{state}.aead.\mathsf{msg\_num}$ from $\mathsf{state}_{\mathcal{I}}$
9. Run $\mathcal{I}_{\mathsf{aead}}(\mathsf{state}.aead.\mathsf{msg\_num}, \texttt{Encrypt}, \mathsf{pid}, |m|, N)$ specifically for $(\mathcal{F}_{\mathsf{aead}}, \text{"}aead\text{"}, \mathsf{sid}.fs, \mathsf{msg\_num})$.
10. Upon receiving output $(\mathsf{state}, \texttt{Encrypt}, \mathsf{pid}, c)$, update $\mathsf{state}.aead.\mathsf{msg\_num} \leftarrow \mathsf{state}$ in $\mathsf{state}_{\mathcal{I}}$, add the tuple $(c, N, 1)$, the message $m$, sender $i$, and $\mathsf{ready2decrypt} = \mathit{true}$ to $\mathsf{records}[\mathsf{msg\_num}]$.
11. Update or store for the first time $\mathsf{records}[\mathsf{msg\_num}]$ in $\mathsf{state}_{\mathcal{I}}$
12. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, c)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

**Inject:** On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$ do:

1. If this is the first activation then end the activation.
2. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
3. If this is the first activation for $\mathsf{pid}_i$, initialize $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} = 0$, $\mathcal{V}iew_i.\mathsf{missed\_msgs} = []$.
   //Simulating a send $(\texttt{Decrypt}, c, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{"}aead\text{"}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
4. Get the message $m$, sender $i$, $\mathsf{ready2decrypt}$, and tuples of the form $(c, N, b)$, from $\mathsf{records}[\mathsf{msg\_num}]$ in $\mathsf{state}_{\mathcal{I}}$. If there is no such record or if $\mathsf{ready2decrypt} \neq \mathit{true}$, then output $(\texttt{inject}, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
   //Simulating a `RetrieveKey` request to $\mathcal{F}_{\mathsf{mKE}}$
5. Let $i$ be such that $\mathsf{pid} = \mathsf{pid}_i$.
6. If this is the first activation for $\mathsf{pid}_i$ set $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} = 0$ and provide input $(\texttt{GetReceivingKey})$ to $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE = (\text{"}eKE\text{"}, \mathsf{sid}))$ on behalf of $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{"}mKE\text{"}, \mathsf{sid}.fs), \mathsf{pid})$.
7. Upon receiving $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key})$ from $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ set $\mathcal{V}iew_i.\mathsf{curr\_chain\_key} = \mathsf{sending\_chain\_key}$.
8. Store $\mathcal{V}iew_i.\mathsf{curr\_chain\_key}$ in $\mathsf{state}_{\mathcal{I}}$ //End simulation of `RetrieveKey` from $\mathcal{F}_{\mathsf{mKE}}$
9. Parse $\mathsf{state}.aead.\mathsf{msg\_num}$ from $\mathsf{state}_{\mathcal{I}}$
10. Run $\mathcal{I}_{\mathsf{aead}}(\mathsf{state}.aead.\mathsf{msg\_num}, \texttt{Authenticate}, \mathsf{pid}, c, N)$ specifically for $(\mathcal{F}_{\mathsf{aead}}, \text{"}aead\text{"}, \mathsf{sid}.fs, \mathsf{msg\_num})$.
11. Upon receiving output $(\mathsf{state}, \texttt{Authenticate}, \mathsf{pid}, v)$, update $\mathsf{state}.aead.\mathsf{msg\_num} \leftarrow \mathsf{state}$ in $\mathsf{state}_{\mathcal{I}}$.
12. If the tuple $(c, N, 1)$ is in $\mathsf{records}[\mathsf{msg\_num}]$, set $\mathsf{ready2decrypt} = \mathit{false}$ in the entry $\mathsf{records}[\mathsf{msg\_num}]$ and set $v = m$.
13. If the tuple $(c, N, 0)$ is in $\mathsf{records}[\mathsf{msg\_num}]$, output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
14. Else (there is no record $(c, N, b)$ in $\mathsf{records}[\mathsf{msg\_num}]$),
    - If $v = \perp$ then record $(c, N, 0)$, and output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
    - Otherwise $(v \neq \perp)$, set $\mathsf{ready2decrypt} = \mathit{false}$ in the entry $\mathsf{records}[\mathsf{msg\_num}]$, and set $v = m$. //$m$ is the message in the entry $\mathsf{records}[\mathsf{msg\_num}]$.

    //End simulation of `Decrypt` from $\mathcal{F}_{\mathsf{aead}}$

15. While $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} \leq \mathsf{msg\_num}$ do: //we only get to this step if decryption succeeds.
    - Increment $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$.

**$\Pi_{\mathsf{mKE}}$**

The protocol has a party id $\mathsf{pid}$ and a session id $\mathsf{sid}.mKE$ that takes the following format: $\mathsf{sid}.mKE = (\text{“}mKE\text{”}, sid.fs)$. Where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$, and $\mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1)$.

The protocol is parametrized by a length doubling pseudorandom generator $PRG$.

**RetrieveKey:** On receiving $(\texttt{RetrieveKey}, \mathsf{pid})$ from $(\Pi_{\mathsf{aead}}, \mathsf{sid}.aead, \mathsf{pid})$, where $\mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num})$,

1. if $\mathsf{pid} \notin \{\mathsf{pid}_0, \mathsf{pid}_1\}$ then end the activation.
2. If this is the first activation, set $\mathsf{curr\_msg\_num} = 0$ and provide input $(\texttt{GetSendingKey})$ to $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid}))$.
3. Upon receiving $(\texttt{GetSendingKey}, \mathsf{sending\_chain\_key})$ from $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ set $\mathsf{curr\_chain\_key} = \mathsf{sending\_chain\_key}$.
4. While $\mathsf{curr\_msg\_num} \leq \mathsf{msg\_num}$ do:
   - Increment $\mathsf{curr\_msg\_num}$.
   - $(\mathsf{curr\_chain\_key}, k) = PRG(\mathsf{curr\_chain\_key})$.
   - Store $\mathsf{missed\_msgs}[\mathsf{curr\_msg\_num}] = k$.
5. Output $(\texttt{RetrieveKey}, \mathsf{pid}, \mathsf{missed\_msgs}[\mathsf{msg\_num}])$ to $(\Pi_{\mathsf{aead}}, \mathsf{sid}.aead, \mathsf{pid})$ while erasing the entry $\mathsf{missed\_msgs}[\mathsf{msg\_num}]$.

**StopKeys:** On receiving $(\texttt{StopKeys}, \mathsf{msg\_num}^*)$ from $(\Pi_{fs\_aead}, \mathsf{sid}.fs, \mathsf{pid})$ do:

1. If $\texttt{StopKeys}$ has already been called, return $(\texttt{StopKeys}, \texttt{Success})$.
2. While $\mathsf{curr\_msg\_num} \leq \mathsf{msg\_num}^*$ do:
   - Increment $\mathsf{curr\_msg\_num}$.
   - $(\mathsf{curr\_chain\_key}, k) = PRG(\mathsf{curr\_chain\_key})$.
   - Store $\mathsf{missed\_msgs}[\mathsf{curr\_msg\_num}] = k$.
3. Set $\mathsf{curr\_chain\_key} = \perp$.
4. Return $(\texttt{StopKeys}, \texttt{Success})$.

**Corruption:** On receiving $(\texttt{Corrupt})$ from $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$:
//Note that the `Corrupt` interface is not part of the “real” protocol; it is only included for UC-modelling purposes.

1. Let $S = (\mathsf{curr\_chain\_key}, \mathsf{curr\_msg\_num}, \mathsf{missed\_msgs})$.
2. Output $(\texttt{Corrupt}, S)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.

Figure 25: The Message Key Exchange Protocol $\Pi_{\mathsf{mKE}}$

$\mathcal{S}_{\mathsf{mKE}}$

The global session id $\mathsf{sid}$ encodes a ciphersuite, including the $\mathsf{PRG}$ (used by $\Pi_{\mathsf{mKE}}$), and the length $\lambda$ of key seeds.

**ReportState:** On receiving ($\texttt{ReportState}, \mathsf{pid}_i, \mathsf{keys\_in\_transit}, \mathsf{msg\_num}_i, \mathsf{chain\_key}$) from ($\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE$) do:

```
1. Initialize a dictionary $\mathsf{seeds} = \{\}$.
2. For $(\mathsf{msg\_num}, k) \in \mathsf{keys\_in\_transit}$ such that $\mathsf{msg\_num} < \mathsf{msg\_num}_i$:
    • Store $\mathsf{seeds}[\mathsf{msg\_num}] = k$
3. If $\mathsf{chain\_key} = \perp$, output $(\texttt{ReportState}, S = \perp)$ to $\mathcal{F}_{\mathsf{mKE}}$. //This happens when the party has already run StopKeys for this epoch.
4. Let $\mathsf{curr\_msg\_num} = \mathsf{msg\_num}_i$, and let the $\mathsf{curr\_chain\_key} = \mathsf{chain\_key}$.
5. While there is some $(\mathsf{msg\_num}, k) \in \mathsf{keys\_in\_transit}$ such that $\mathsf{msg\_num} > \mathsf{msg\_num}_i$:
    • $\mathsf{curr\_msg\_num}+ = 1$
    • Let $(\mathsf{curr\_chain\_key}, \mathsf{key\_seed}) = PRG(\mathsf{curr\_chain\_key})$.
    • Store $\mathsf{seeds}[\mathsf{msg\_num}] = \mathsf{key\_seed}$.
    • Set $\mathsf{latest\_seed\_num} = \mathsf{curr\_msg\_num}$.
    • Delete $(\mathsf{msg\_num}, k)$ from $\mathsf{keys\_in\_transit}$.
6. Delete local variable $\mathsf{curr\_msg\_num}$.
7. Output $(\texttt{ReportState}, S = \{\mathsf{chain\_key}\})$.
```

**RetrieveKey:** On receiving ($\texttt{RetrieveKey}, \mathsf{pid}, \mathsf{msg\_num}$) from ($\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE$) do:
//This happens only if `IsCorrupt?` = *true*. In particular, $\mathcal{S}_{\mathsf{mKE}}$ has already gotten a `ReportState` directive.

```
1. If $\mathsf{msg\_num} \leq \mathsf{latest\_seed\_num}$ then output $(\texttt{RetrieveKey}, \mathsf{pid}, k = \mathsf{seeds}[\mathsf{msg\_num}])$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE)$. //The only keys not in this dictionary are keys that were already retrieved by both parties at the time of corruption.
2. If $\mathsf{msg\_num} > \mathsf{latest\_seed\_num}$, initialize $j = \mathsf{latest\_seed\_num} + 1$.
3. While $j < \mathsf{msg\_num}$:
    • $(\mathsf{curr\_chain\_key}, \mathsf{key\_seed}) \leftarrow PRG(\mathsf{curr\_chain\_key})$.
    • Store $\mathsf{seeds}[j] = \mathsf{key\_seed}$, $j+ = 1$.
4. Output $(\texttt{RetrieveKey}, \mathsf{pid}, k = \mathsf{seeds}[\mathsf{msg\_num}])$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE)$.
```

Figure 26: Message Key Exchange Simulator, $\mathcal{S}_{\mathsf{mKE}}$

at random. Therefore, the state produced at compromise is indistinguishable from the real world. □

## 9 Authenticated Encryption for Single Messages: Realising $\mathcal{F}_{\mathsf{aead}}$

As outlined in Section 4, the ideal authenticated encryption functionality $\mathcal{F}_{\mathsf{aead}}$ is realized by way of a specific symmetric authenticated encryption scheme, which obtains its secret key from $\mathcal{F}_{\mathsf{mKE}}$, and provides security against adaptive curruptions in the programmable random oracle model (which is captured by way of $\mathcal{F}_{\mathsf{pRO}}$).

If corruptions were not adaptive, any authenticated encryption scheme would suffice here; in particular, there would have been no no need to resort to the random oracle model. In fact, this would have remained true even if the overall corruption structure was adaptive, as long as the adversary does not learn the keys that correspond to messages that were sent to the corrupted party but not yet received.

However, assert full-fledged security in our model requires coming up with a simulation process that first generates a ciphertext $c$, and is then given an arbitrary message $m$ and asked to generate a key $k$ such that $Dec(k, c) = m$. While such schemes exist, they require having a key that is longer than the total length of the messages encrypted with that key. Furthermore, impossibility holds even when authentication is not required: There do not exist adaptively secure encryption schemes in the plain model where the key is shorter t han the message [51].

We circumvent this impossibility by resorting to the random oracle model (again, using ideas from [51]). Specifically, we employ a simple Encrypt-then-MAC scheme [44] where the encryption is simply a one-time-pad, and the random oracle is used to expand the key to the length needed for the MAC algorithm, plus the length of the message.

We note that when the message is shorter than the overall keylength minus the length of the MAC key, the above scheme is adaptively secure even in the plain model. Consequently, in situations where there is a known bound on the total length of messages sent in each epoch, our solution is fully secure in the plain model.

Protocol $\Pi_{\mathsf{aead}}$ is presented in Figure 27 on page 69. We show:

**Theorem 6** *Assuming the unforgeability of* $(\mathsf{MAC}, \mathsf{Verify})$*, protocol* $\Pi_{\mathsf{aead}}$ *UC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{aead}}$ *in the presence of* $\mathcal{F}_{\mathsf{pRO}}$*, as well as* $\mathcal{F}_{\mathsf{mKE}}^{\Pi_{\mathsf{mKE}}}$.
$\left(\mathcal{F}_{\mathsf{mKE}}^{\Pi_{\mathsf{mKE}}} = (\mathcal{S}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{mKE}}), \mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}}), \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{lib}}\right)$.

**Proof:**

The ideal-process adversary $\mathcal{S}_{\mathsf{aead}}$ is presented in Figure 29. To demonstrate the validity of $\mathcal{S}_{\mathsf{aead}}$, we show that no environment that has global access to $\mathcal{F}_{\mathsf{pRO}}, \mathcal{F}_{\mathsf{mKE}}^{\Pi_{\mathsf{mKE}}} = (\mathcal{S}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{mKE}}), \mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}}), \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{lib}}$, can tell whether it is interacting with $\Pi_{\mathsf{aead}}$, or else with $\mathcal{F}_{\mathsf{aead}}$ and $\mathcal{S}_{\mathsf{aead}}$.

This is done as follows: We first argue that, conditioned on two bad events not happening, the simulation is perfect. Then, we show that the bad events happen with negligible probability. The first, $\mathsf{Forge}$, is the event that the environment produces a verifying message tag pair $(m', t')$ for a fresh message $m'$, when no party has been corrupted. The second, $\mathsf{Collision}$, is the bad event that a message seed key provided by $\mathcal{F}_{\mathsf{mKE}}^{\Pi_{\mathsf{mKE}}}$ collides with an input to the random oracle that was previously programmed by the environment or simulator.

**$\Pi_{\mathsf{aead}}$**

This protocol has a session id $\mathsf{sid}.aead = (\text{“}aead\text{''}, \mathsf{sid}.fs, \mathsf{msg\_num})$ where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1), \mathsf{epoch\_id})$ and party id $\mathsf{pid}$.

It uses a message authentication code $(\mathsf{MAC}, \mathsf{Verify})$ with key length $\lambda$.

**Encrypt:** On receiving $(\texttt{Encrypt}, m, N)$ from $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$:

1. If this is not the first activation or $\mathsf{pid} \notin (\mathsf{pid}_0, \mathsf{pid}_1)$, end the activation.
2. Provide input $(\texttt{RetrieveKey}, \mathsf{pid})$ to $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$.
3. Upon receiving output $(\texttt{RetrieveKey}, k)$ from $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$:
   - Let $\ell = |m| + \lambda$.
   - Send $(\texttt{HashQuery}, k, \ell)$ to $\mathcal{F}_{\mathsf{pRO}}$.
   - Upon receiving the output $(\texttt{HashQuery}, \mathsf{msg\_key})$, parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$, where the $\mathsf{k_{otp}}$ has length $|m|$, and $\mathsf{k_{mac}}$ has length $\lambda$.
   - Compute ciphertext $c' = \mathsf{k_{otp}} \oplus m$
   - Compute tag $t = \mathsf{MAC}(\mathsf{k_{mac}}, (c', \mathsf{sid}.aead, N))$.
   - Finally, set $c = (c', t)$.
   - Delete $\mathsf{msg\_key}, k, m$, and $c$ and output $(\texttt{Encrypt}, c)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
4. If the response from $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$ is $(\texttt{RetrieveKey}, \texttt{Fail})$, then output $(\texttt{Encrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.

**Decryption:** On receiving $(\texttt{Decrypt}, c = (c', t), N)$ from $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$:

1. If this is not the first activation or $\mathsf{pid} \notin (\mathsf{pid}_0, \mathsf{pid}_1)$, end the activation.
2. Provide input $(\texttt{RetrieveKey}, \mathsf{pid})$ to $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$.
3. Upon receiving output $(\texttt{RetrieveKey}, k)$ from $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$:
   - Let $\ell = |m| + \lambda$.
   - Send $(\texttt{HashQuery}, k, \ell)$ to $\mathcal{F}_{\mathsf{pRO}}$.
   - Upon receiving the output $(\texttt{HashQuery}, \mathsf{msg\_key})$, parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$, where the $\mathsf{k_{otp}}$ has length $|m|$, and $\mathsf{k_{mac}}$ has length $\lambda$.
   - If $\mathsf{Verify}(\mathsf{k_{mac}}, t, (c', \mathsf{sid}.aead, N)) \neq 1$, then output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
   - Else (the tag is valid), compute message $m = \mathsf{k_{otp}} \oplus c'$.
   - Delete $\mathsf{msg\_key}, k, m$, and $c$ and output $(\texttt{Decrypt}, m)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
4. If the response from $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$ is $(\texttt{RetrieveKey}, \texttt{Fail})$, then output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.

**Corruption:** On receiving $(\texttt{Corrupt})$ from $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$:
//Note that the `Corrupt` interface is not part of the “real” protocol; it is only included for UC-modelling purposes.

1. Output $(\texttt{Corrupt}, S = \perp)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$. //$\Pi_{\mathsf{aead}}$ has no persistent state.

Figure 27: The Authenticated Encryption with Associated Data Protocol, $\Pi_{aead}$

$\mathcal{I}_{aead}$

The internal code has a ciphersuite, including the MAC protocol $(\mathsf{MAC}, \mathsf{Verify})$, the encryption protocol $OTP$, and the length of the MAC keys $\lambda$.

**To compute** $\mathcal{I}(\mathsf{state}_\mathcal{I}, \mathtt{Encrypt}, \mathsf{pid}, |m|, N)$**:**

```
1. Set $\ell = |m|$ and $m = 0^\ell$.
2. Choose a random message key $\mathsf{msg\_key} \xleftarrow{\$} \{0,1\}^{\ell+\lambda}$.
3. Parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = \ell$ and $|\mathsf{k_{mac}}| = \lambda$.
4. Compute ciphertext $c' = \mathsf{k_{otp}} \oplus m$ and tag $t = \mathsf{MAC}(\mathsf{k_{mac}}, (c', \mathsf{sid}.aead, N))$.
5. Finally, set $c = (c', t)$, and record $(\mathsf{msg\_key}, m, c, N)$ in $\mathsf{state}_\mathcal{I}$.
6. Return $(\mathsf{state}_\mathcal{I}, \mathtt{Encrypt}, \mathsf{pid}, c)$.
```

**To compute** $\mathcal{I}(\mathsf{state}_\mathcal{I}, \mathtt{Authenticate}, \mathsf{pid}, c, N)$**:**

```
1. Parse $c^* = (c', t')$
2. Let $\ell = |c'|$.
3. If there is no record $(\mathsf{msg\_key}, m, c = (c', t), N)$ in $\mathsf{state}_\mathcal{I}$ then end the activation.
4. Else, parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = |m|$ and $|\mathsf{k_{mac}}| = \lambda$.
5. If $\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 0$: set $v = \perp$.
6. Else $(\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 1)$: set $v = m$.
7. Return $(\mathsf{state}_\mathcal{I}, \mathtt{Authenticate}, \mathsf{pid}, c, v)$.
```

Figure 28: Internal adversarial code $\mathcal{I}_{aead}$

$\mathcal{S}_{\mathsf{aead}}$

**At first activation:** Send $(\mathcal{F}_{\mathsf{aead}}, \mathcal{I})$ to $\mathcal{F}_{\mathsf{lib}}$.

**On receiving** $(\mathsf{state}_{\mathcal{I}}, \texttt{ReportState}, \mathsf{pid}, k, m^*)$ **from** $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$**:**

1. Initialize state $S = \perp$.
2. If there is a record $(\mathsf{msg\_key}, m, c = (c', t))$ in $\mathsf{state}_{\mathcal{I}}$:
    - Parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = |m^*|$.
    - Let $\mathsf{k_{otp}}^* = c' \oplus m^*$ and $\mathsf{msg\_key}^* = \mathsf{k_{otp}}^*||\mathsf{k_{mac}}$.
    - Send a backdoor message $(\texttt{Program}, k, \mathsf{msg\_key}^*)$ to $\mathcal{F}_{\mathsf{pRO}}$.
    - On receiving $(\texttt{Program})$, continue.
3. Store the variable $k$ in $\mathsf{state}_{\mathcal{I}}$.
4. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{ReportState}, \mathsf{pid}, S)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$.

**On receiving** $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, m, N)$**:** //This is called in the case that either party has been corrupted and no ciphertext was released yet. In this case $\mathcal{S}_{\mathsf{aead}}$ simply runs the instructions for $\Pi_{\mathsf{aead}}$.

1. Set $\ell = |m|$.
2. Send $(\texttt{HashQuery}, k, \ell + \lambda)$ to $\mathcal{F}_{\mathsf{pRO}}$. //Note that $k$ is sent to $\mathcal{S}_{\mathsf{aead}}$ during `ReportState`
3. Upon receiving the output $(\texttt{HashQuery}, \mathsf{msg\_key})$, set $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = \ell$ and $|\mathsf{k_{mac}}| = \lambda$.
4. Compute ciphertext $c' = \mathsf{k_{otp}} \oplus m$ and tag $t = \mathsf{MAC}(\mathsf{k_{mac}}, (c', \mathsf{sid}.aead, N))$.
5. Finally, set $c = (c', t)$ and record $(\mathsf{msg\_key}, m, c, N)$ in $\mathsf{state}_{\mathcal{I}}$.
6. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, c)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$.

**On receiving** $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \mathsf{pid}, c^*, N)$ **from** $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$**:**
//This is called in the case that either party has been corrupted and no output was generated yet. $\mathcal{S}_{\mathsf{aead}}$ simply runs the instructions for $\Pi_{\mathsf{aead}}$.

1. Parse $c^* = (c', t')$
2. Let $\ell = |c'|$.
3. Send $(\texttt{HashQuery}, k, \ell + \lambda)$ to $\mathcal{F}_{\mathsf{pRO}}$. //Note that $k$ is sent to $\mathcal{S}_{\mathsf{aead}}$ during `ReportState`
4. Upon receiving the output $(\texttt{HashQuery}, \mathsf{msg\_key}')$, continue.
5. Parse $\mathsf{msg\_key}' = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = \ell$, $|\mathsf{k_{mac}}| = \lambda$.
6. If $\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 0$: set $v = \perp$.
7. Else $(\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 1)$: compute $v = \mathsf{k_{otp}} \oplus c'$.
8. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{inject}, \mathsf{pid}, c, v)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$.

Figure 29: Authenticated Encryption with Associated Data Simulator, $\mathcal{S}_{\mathsf{aead}}$

**MAC Forgery Game**

We define MAC forgery in the standard chosen-plaintext setting:

1. A key $\mathsf{k_{mac}} \xleftarrow{\$} \mathcal{K}_{mac}$ is sampled uniformly.
2. The adversary $\mathcal{A}$ is given access to the MAC oracle $\mathsf{MAC}(\mathsf{k_{mac}}, \cdot)$ which on message $m$ computes and outputs the tag $t$ of that message under the MAC. Let $Q$ denote the set of all queries that $\mathcal{A}$ makes to $\mathsf{MAC}(\mathsf{k_{mac}}, \cdot)$.
3. The adversary $\mathcal{A}$ then outputs $(m', t')$
4. The event $\mathsf{Forge}_1$ occurs if and only if $\mathsf{Verify}(\mathsf{k_{mac}}, m', t') = 1$ and $m' \notin Q$.

We will call a successful forgery event of this kind $\mathsf{Forge}$.

**Second Forgery Game**

Now we define a different type of forgery:

1. Two keys $\mathsf{k_{mac}}, \mathsf{k_{mac}}' \xleftarrow{\$} \mathcal{K}_{mac}$ are sampled uniformly and independently.
2. The adversary $\mathcal{A}$ is given access to the MAC oracle $\mathsf{MAC}(\mathsf{k_{mac}}, \cdot)$ which on message $m$ computes and outputs the tag $t$ of that message under the MAC.
3. The adversary $\mathcal{A}$ is given access to the $\mathsf{Verify}$ oracle $\mathsf{Verify}(\mathsf{k_{mac}}', \cdot, \cdot)$ which on input $(m', t')$ outputs whether the message tag pair verifies.
4. The adversary $\mathcal{A}$ then outputs $(m', t')$
5. The event $\mathsf{Forge}_2$ occurs if and only if $\mathsf{Verify}(\mathsf{k_{mac}}', m', t') = 1$.

We will call a successful forgery event of this kind $\mathsf{Forge}_2$.

This experiment models the setting where the two parties have diverged and thus have independent MAC keys. Lastly, we define $\mathsf{Collision}$ to be the event that $\mathsf{Env}$ already programmed $\mathcal{F}_{\mathsf{pRO}}$ on input $k$.

**Lemma 13** *If neither* $\mathsf{Forge}$ *nor* $\mathsf{Collision}$ *events happen during the executions, then the the simulation by* $\mathcal{S}_{\mathsf{aead}}$ *is perfect.*

**Proof:**Assume that the bad events $\mathsf{Forge}$ and $\mathsf{Collision}$ do not occur during the executions. We will prove that, for all environments $\mathsf{Env}$ and adversaries $\mathcal{A}$, the simulator $\mathcal{S}_{\mathsf{aead}}$ together with $\mathcal{F}_{\mathsf{aead}}$ *perfectly* simulate the real-world views of $\mathsf{Env}$ and $\mathcal{A}$ when they interact with $\Pi_{\mathsf{aead}}$.

**Encryption** Observe that encryption in the ideal world occurs exactly as in the real world, except that the $\mathcal{I}_{\mathsf{aead}}$ module in $\mathcal{F}_{\mathsf{aead}}$ chooses a random key under which to encrypt the all 0's message of the correct length using the one time pad to produce a ciphertext $c$. It then authenticates the produced ciphertext using the randomly chosen $\mathsf{k_{mac}}$. Later, when a corruption occurs, the simulator can easily use the leaked message $m$ to compute a key $\mathsf{k_{otp}} = c \oplus m$ that will decrypt the ciphertext to the correct message. It then uses the leaked key-seed and programs $\mathcal{F}_{\mathsf{pRO}}$ to output the key $\mathsf{k_{otp}} \parallel \mathsf{k_{mac}}$ on this seed.

**Decryption** In the real world, $\Pi_{\mathsf{aead}}$ retrieves the message key seed from $\Pi_{\mathsf{mKE}}$, and if the key is available, $\Pi_{\mathsf{aead}}$ queries $\mathcal{F}_{\mathsf{pRO}}$ to get the expanded $\mathsf{msg\_key}$. If the tag $t$ verifies, then it decrypts the ciphertext using $\mathsf{msg\_key}$; otherwise, it returns a failure message. If the adversary has compromised the state of $\Pi_{\mathsf{aead}}$, then $\Pi_{\mathsf{mKE}}$ for the same epoch is compromised as well, and $\mathcal{A}$ will get the $\mathsf{key\_seed}$ (which it can expand to inject ciphertexts that will authenticate).

In the ideal world, $\mathcal{F}_{\mathsf{aead}}$ retrieves the message key from $\Pi_{\mathsf{mKE}}$ and if the key is available, $\mathcal{F}_{\mathsf{aead}}$ returns the message $m$ that it encrypted to $c = (c', t)$ in the case that the other party asks to decrypt $c$. This case is identical to the real world, by definition. If, on the other hand, $\mathcal{F}_{\mathsf{aead}}$ gets a different ciphertext $c^* \neq c$, then $\mathcal{F}_{\mathsf{aead}}$ runs the internal $\mathcal{I}_{\mathsf{aead}}$ module on $(\mathtt{inject}, c^*)$ to see if it wants to inject a potentially different message and waits for a response value $v$. $\mathcal{I}_{\mathsf{aead}}$ uses the $\mathsf{k}_{\mathsf{mac}}$ it used during encryption to verify the tag in $c^*$; if the tag fails to verify for $c^*$, then it returns $v = \perp$. In the case that the epoch is not compromised, $\mathcal{F}_{\mathsf{aead}}$ will check that $\mathcal{I}_{\mathsf{aead}}$'s returned value $v \neq \perp$. If $v \neq \perp$, then $\mathcal{F}_{\mathsf{aead}}$ will output the original message $m$. Assuming that $\mathsf{Forge}_1$ and $\mathsf{Forge}_2$ do not occur, the original $m$ will be returned if and only if $c$ is the input ciphertext.

In the case that the epoch is compromised, $\mathcal{S}_{\mathsf{aead}}$ uses the message key seed $k$ that it received from $\mathcal{F}_{\mathsf{aead}}$ to query the random oracle $\mathcal{F}_{\mathsf{pRO}}$ to expand the key. It then verifies the tag and outputs the decryption of the ciphertext (which may be different from $m$). In this case, $\mathcal{F}_{\mathsf{aead}}$ outputs the injected message from $\mathcal{S}_{\mathsf{aead}}$. This is identical to the powers of the real-world adversary after a state compromise.

**Corruption** Notice that the protocol $\Pi_{\mathsf{aead}}$ has no persistent state besides its $\mathsf{sid}$ and whether it has been activated already. The message, ciphertext, tag, and keys are all deleted after its activation. Accordingly, $\mathcal{S}_{\mathsf{aead}}$ (and thus $\mathcal{F}_{\mathsf{aead}}$) returns no state upon corruption.

The main job of $\mathcal{S}_{\mathsf{aead}}$ on corruption is to equivocate on the ciphertext it provided during encryption by programming the random oracle. Importantly, encryption occurs at most *once* in $\mathcal{F}_{\mathsf{aead}}$ (and $\Pi_{\mathsf{aead}}$). Thus, the random oracle is programmed at most once, and since we are assuming that the bad event $\mathsf{Collide}$ (that $\mathcal{F}_{\mathsf{pRO}}$ was already programmed or queried on input $k$) does not occur, the message seed key will be programmable. $\mathcal{S}_{\mathsf{aead}}$ receives the correct message $m^*$ along with the key seed $k$ from $\mathcal{F}_{\mathsf{aead}}$, after which it computes the message key from the ciphertext $c$ it generated and $m^*$ (and re-uses the MAC key $\mathsf{k}_{\mathsf{otp}}$) and programs these in $\mathcal{F}_{\mathsf{pRO}}$. □

So, all that's left to argue is why $\mathsf{Forge}$ and $\mathsf{Collision}$ do not occur. Notice that before corruptions the message key seeds provided by $F^{\Pi}_{mKE}$ are uniformly random, so event $\mathsf{Forge}$ corresponds to the standard security game for message authentication codes. In particular, assuming that we have an environment that successfully induces a $\mathsf{Forge}$ event with non-negligible probability, we can construct a forger against $(\mathsf{MAC}, \mathsf{Verify})$, breaking our assumption. Thus, $\mathsf{Forge}$ happens with negligible probability.

Next, we discuss the $\mathsf{Collision}$ event. Since the key seeds provided by $F^{\Pi}_{mKE}$ are uniformly random and independent, and the space of inputs to $\mathcal{F}_{\mathsf{pRO}}$ is exponential in the security parameter $\delta$, the probability that any two of them collide is negligible in the security parameter. Furthermore, since the environment runs in polynomial time with respect to the security parameter $\delta$, the environment can only program a polynomial number of inputs. So, for a seed length that is $\theta(\delta)$, the probability that the adversary programs an input that causes a collision is roughly $2^{\theta(\delta)} - \mathsf{poly}(\delta)$. □

## References

[1] Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019)

[2] Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Heidelberg (Aug 2020)

[3] Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 1463–1483. ACM Press (Nov 2021)

[4] Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Heidelberg (Nov 2020)

[5] Badertscher, C., Canetti, R., Hesse, J., Tackmann, B., Zikas, V.: Universal composition with global subroutines: Capturing global setup within plain UC. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part III. LNCS, vol. 12552, pp. 1–30. Springer, Heidelberg (Nov 2020)

[6] Badertscher, C., Hesse, J., Zikas, V.: On the (ir)replaceability of global setups, or how (not) to use a global ledger. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part II. LNCS, vol. 13043, pp. 626–657. Springer, Heidelberg (Nov 2021)

[7] Balli, F., Rösler, P., Vaudenay, S.: Determining the core primitive for optimally secure ratcheting. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part III. LNCS, vol. 12493, pp. 621–650. Springer, Heidelberg (Dec 2020)

[8] Bellare, M., Canetti, R., Krawczyk, H.: Pseudorandom functions revisited: The cascade construction and its concrete security. In: Proceedings of 37th Conference on Foundations of Computer Science. pp. 514–523. IEEE (1996)

[9] Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO'93. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (Aug 1994)

[10] Bellare, M., Rogaway, P.: Provably secure session key distribution: The three party case. In: 27th ACM STOC. pp. 57–66. ACM Press (May / Jun 1995)

[11] Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 619–650. Springer, Heidelberg (Aug 2017)

[12] Bienstock, A., Dodis, Y., Rösler, P.: On the price of concurrency in group ratcheting protocols. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 198–228. Springer, Heidelberg (Nov 2020)

[13] Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the signal double ratchet algorithm. Cryptology ePrint Archive, Report 2022/355 (2022), https://eprint.iacr.org/2022/355

[14] Blazy, O., Bossuat, A., Bultel, X., Fouque, P., Onete, C., Pagnin, E.: SAID: reshaping signal into an identity-based asynchronous messaging protocol with authenticated ratcheting. In: EuroS&P. pp. 294–309. IEEE (2019)

[15] Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 280–300. Springer, Heidelberg (Dec 2013)

[16] Borisov, N., Goldberg, I., Brewer, E.A.: Off-the-record communication, or, why not to use PGP. In: WPES. pp. 77–84. ACM (2004)

[17] Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 501–519. Springer, Heidelberg (Mar 2014)

[18] Caforio, A., Durak, F.B., Vaudenay, S.: Beyond security and efficiency: On-demand ratcheting with security awareness. In: Garay, J. (ed.) PKC 2021, Part II. LNCS, vol. 12711, pp. 649–677. Springer, Heidelberg (May 2021)

[19] Camenisch, J., Drijvers, M., Gagliardoni, T., Lehmann, A., Neven, G.: The wonderful world of global random oracles. Cryptology ePrint Archive, Report 2018/165 (2018), https://eprint.iacr.org/2018/165

[20] Camenisch, J., Drijvers, M., Lehmann, A.: Universally composable direct anonymous attestation. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) PKC 2016, Part II. LNCS, vol. 9615, pp. 234–264. Springer, Heidelberg (Mar 2016)

[21] Campion, S., Devigne, J., Duguey, C., Fouque, P.A.: Multi-device for signal. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 20, Part II. LNCS, vol. 12147, pp. 167–187. Springer, Heidelberg (Oct 2020)

[22] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001)

[23] Canetti, R.: Universally composable security. J. ACM 67(5), 28:1–28:94 (2020)

[24] Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (Feb 2007)

[25] Canetti, R., Halevi, S., Herzberg, A.: Maintaining authenticated communication in the presence of break-ins. J. Cryptol. 13(1), 61–105 (2000)

[26] Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 453–474. Springer, Heidelberg (May 2001)

[27] Canetti, R., Krawczyk, H.: Universally composable notions of key exchange and secure channels. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 337–351. Springer, Heidelberg (Apr / May 2002)

[28] Canetti, R., Shahaf, D., Vald, M.: Universally composable authentication and key-exchange with global PKI. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) PKC 2016, Part II. LNCS, vol. 9615, pp. 265–296. Springer, Heidelberg (Mar 2016)

[29] Chase, M., Perrin, T., Zaverucha, G.: The signal private group system and anonymous credentials supporting efficient verifiable encryption. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1445–1459. ACM Press (Nov 2020)

[30] Chen, K., Chen, J.: Anonymous end to end encryption group messaging protocol based on asynchronous ratchet tree. In: Meng, W., Gollmann, D., Jensen, C.D., Zhou, J. (eds.) ICICS 20. LNCS, vol. 11999, pp. 588–605. Springer, Heidelberg (Aug 2020)

[31] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: EuroS&P. pp. 451–466. IEEE (2017)

[32] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. Journal of Cryptology 33(4), 1914–1983 (Oct 2020)

[33] Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 1802–1819. ACM Press (Oct 2018)

[34] Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: Hicks, M., Köpf, B. (eds.) CSF 2016 Computer Security Foundations Symposium. pp. 164–178. IEEE Computer Society Press (2016)

[35] Cremers, C., Fairoze, J., Kiesl, B., Naska, A.: Clone detection in secure messaging: Improving post-compromise security in practice. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1481–1495. ACM Press (Nov 2020)

[36] Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. In: Attrapadung, N., Yagi, T. (eds.) IWSEC 19. LNCS, vol. 11689, pp. 343–362. Springer, Heidelberg (Aug 2019)

[37] Hashimoto, K., Katsumata, S., Kwiatkowski, K., Prest, T.: An efficient and generic construction for signal's handshake (X3DH): Post-quantum, state leakage secure, and deniable. In: Garay, J. (ed.) PKC 2021, Part II. LNCS, vol. 12711, pp. 410–440. Springer, Heidelberg (May 2021)

[38] Hofheinz, D., Rao, V., Wichs, D.: Standard security does not imply indistinguishability under selective opening. In: Hirt, M., Smith, A.D. (eds.) TCC 2016-B, Part II. LNCS, vol. 9986, pp. 121–145. Springer, Heidelberg (Oct / Nov 2016)

[39] Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 33–62. Springer, Heidelberg (Aug 2018)

[40] Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 159–188. Springer, Heidelberg (May 2019)

[41] Jost, D., Maurer, U., Mularczyk, M.: A unified and composable take on ratcheting. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019, Part II. LNCS, vol. 11892, pp. 180–210. Springer, Heidelberg (Dec 2019)

[42] Keybase blog: New cryptographic tools on keybase. https://keybase.io/blog/crypto (2020)

[43] Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 669–684. ACM Press (Nov 2013)

[44] Krawczyk, H.: The order of encryption and authentication for protecting communications (or: How secure is ssl?). In: Kilian, J. (ed.) Advances in Cryptology - CRYPTO 2001, 21st Annual

International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2139, pp. 310–331. Springer (2001), https://doi.org/10.1007/3-540-44647-8_19

[45] Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (Aug 2010)

[46] Krohn, M.: Zoom rolling out end-to-end encryption offering. https://blog.zoom.us/zoom-rolling-out-end-to-end-encryption-offering/ (2020)

[47] Marlinspike, M., Perrin, T.: The X3DH key agreement protocol. https://signal.org/docs/specifications/x3dh/ (2016)

[48] Martiny, I., Kaptchuk, G., Aviv, A.J., Roche, D.S., Wustrow, E.: Improving signal's sealed sender. In: NDSS. The Internet Society (2021)

[49] Maurer, U.: Constructive cryptography - a primer (invited paper). In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, p. 1. Springer, Heidelberg (Jan 2010)

[50] Maurer, U.: Constructive cryptography - A new paradigm for security definitions and proofs. In: TOSCA. Lecture Notes in Computer Science, vol. 6993, pp. 33–56. Springer (2011)

[51] Nielsen, J.B.: Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 111–126. Springer, Heidelberg (Aug 2002)

[52] Open Whisper Systems: Technical information: Specifications and software libraries for developers. https://signal.org/docs/ (2016)

[53] Perrin, T.: The noise protocol framework. https://noiseprotocol.org/noise.html (2018)

[54] Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 3–32. Springer, Heidelberg (Aug 2018)

[55] Rösler, P., Mainka, C., Schwenk, J.: More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In: EuroS&P. pp. 415–429. IEEE (2018)

[56] Rotem, L., Segev, G.: Out-of-band authentication in group messaging: Computational, statistical, optimal. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 63–89. Springer, Heidelberg (Aug 2018)

[57] Singh, M.: Whatsapp is now delivering roughly 100 billion messages a day. https://techcrunch.com/2020/10/29/whatsapp-is-now-delivering-roughly-100-billion-messages-a-day/ (2020)

[58] Status: Private, secure communication. https://status.im (2022)

[59] Sylo: Comms for the metaverse. https://sylo.io (2022)

[60] Unger, N., Dechand, S., Bonneau, J., Fahl, S., Perl, H., Goldberg, I., Smith, M.: SoK: Secure messaging. In: 2015 IEEE Symposium on Security and Privacy. pp. 232–249. IEEE Computer Society Press (May 2015)

[61] Unger, N., Goldberg, I.: Deniable key exchanges for secure messaging. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 1211–1223. ACM Press (Oct 2015)

[62] Unger, N., Goldberg, I.: Improved strongly deniable authenticated key exchanges for secure messaging. PoPETs 2018(1), 21–66 (Jan 2018)

[63] Vatandas, N., Gennaro, R., Ithurburn, B., Krawczyk, H.: On the cryptographic deniability of the Signal protocol. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 20, Part II. LNCS, vol. 12147, pp. 188–209. Springer, Heidelberg (Oct 2020)

[64] WhatsApp LLC: About end-to-end encryption. https://faq.whatsapp.com/general/security-and-privacy/end-to-end-encryption/ (2021)

[65] Yan, H., Vaudenay, S.: Symmetric asynchronous ratcheted communication with associated data. In: Aoki, K., Kanaoka, A. (eds.) IWSEC 20. LNCS, vol. 12231, pp. 184–204. Springer, Heidelberg (Sep 2020)