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

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

May 19, 2023

Abstract

We model and analyze the Signal end-to-end secure messaging protocol within the Universal Composability (UC) framework. Specifically:

- We formulate an ideal functionality that captures end-to-end secure messaging in a setting with Public Key Infrastructure (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, obtaining their entire internal states. 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. These components 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 with minimal hardness assumptions.

Our modeling introduces additional innovations that enable arguing about the security of Signal, irrespective of the underlying communication medium, and facilitate the secure composition of dynamically generated modules that share state. These features, in conjunction 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.

*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 . . . 13
- 1.5 Related Work . . . 14

2 Universally Composable Security **15**
- 2.1 Universal Composability: A Primer . . . 15
- 2.2 New Capabilities: Global Functionalities, Adversarially Provided Code . . . 17

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

4 Overview of our Modular Decomposition **26**

5 A Signal-style Secure Messaging Protocol: Realizing $\mathcal{F}_{\mathsf{SM}}$ **32**
- 5.1 The Epoch Key Exchange Functionality $\mathcal{F}_{\mathsf{eKE}}$ . . . 33
- 5.2 The Forward Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ . . . 37
- 5.3 The Signal Protocol, Π_{SGNL} . . . 40
- 5.4 Security Analysis of Π_{SGNL} . . . 42

6 The Public Key Ratchet: Realizing $\mathcal{F}_{\mathsf{eKE}}$ **48**
- 6.1 Protocol Π_{eKE} . . . 48
- 6.2 Cascaded PRF-PRG (CPRFG) . . . 51
- 6.3 Security Analysis of Π_{eKE} . . . 58

7 Unidirectional Forward Secure Authenticated Channels: Realising $\mathcal{F}_{\mathsf{fs\_aead}}$ **72**
- 7.1 The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$ . . . 72
- 7.2 The Single-Message Authenticated Encryption Functionality $\mathcal{F}_{\mathsf{aead}}$ . . . 72
- 7.3 Protocol $\Pi_{\mathsf{fs\_aead}}$ . . . 77

8 The Symmetric Ratchet: Realising $\mathcal{F}_{\mathsf{mKE}}$ **81**

9 Authenticated Encryption for Single Messages: Realising $\mathcal{F}_{\mathsf{aead}}$ **84**

References **89**

List of Figures

1 Modeling and realizing secure messaging: The general subroutine structure 10
2 The code library functionality, $\mathcal{F}_{\mathsf{lib}}$ 18
3 The Public-Key Directory Functionality, $\mathcal{F}_{\mathsf{DIR}}$ 20
4 The Programamble Random Oracle Functionality, $\mathcal{F}_{\mathsf{pRO}}$ 20
5 The Long-Term Keys Module Functionality, $\mathcal{F}_{\mathsf{LTM}}$ 21
6 The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$ 22
7 The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$ 23
8 A pictorial rendition of Theorem 2 28
9 A pictorial rendition of Theorem 3 29
10 A pictorial rendition of Theorem 4 30
11 A pictorial rendition of Theorem 5 31
12 A pictorial rendition of Theorem 6 31
13 The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$ 38
14 The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$ (continued) 39
15 The Forward-Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ 41
16 The Signal Protocol, Π_{SGNL} 43
17 Secure Messaging Simulator, $\mathcal{S}_{\mathsf{SM}}$ 44
18 Secure Messaging Simulator $\mathcal{S}_{\mathsf{SM}}$ continued... 45
19 Internal Adversary $\mathcal{I}_{\mathsf{SM}}$ 46
20 The Epoch Key Exchange Protocol Π_{eKE} 50
21 The Epoch Key Exchange Protocol Π_{eKE} (continued) 52
22 Cascaded PRF-PRG Security Game 54
23 CPRFG Construction 56
24 Cascaded PRF-PRG Simulator 56
25 Internal adversarial code $\mathcal{I}_{\mathsf{eKE}}$ 59
26 Simulator $\mathcal{S}_{\mathsf{eKE}}$ for realizing $\mathcal{F}_{\mathsf{eKE}}$ 60
27 Simulator $\mathcal{S}_{\mathsf{eKE}}$ (continued) 61
28 The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$ 73
29 The Authenticated Encryption with Associated Data Functionality, $\mathcal{F}_{\mathsf{aead}}$ 74
30 The Forward-Secure Encryption $\Pi_{\mathsf{fs\_aead}}$ 75
31 Internal Adversary, $\mathcal{I}_{\mathsf{fsaead}}$ 76
32 Forward Secure Authenticated Encryption Simulator, $\mathcal{S}_{\mathsf{fsaead}}$ 78
33 $\mathcal{S}_{\mathsf{fsaead}}$ Continued... 79
34 The Message Key Exchange Protocol Π_{mKE} 82
35 Message Key Exchange Simulator, $\mathcal{S}_{\mathsf{mKE}}$ 83
36 The Authenticated Encryption with Associated Data Protocol, Π_{aead} 85
37 Internal adversarial code $\mathcal{I}_{\mathsf{aead}}$ 86
38 Authenticated Encryption with Associated Data Simulator, $\mathcal{S}_{\mathsf{aead}}$ 87

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 [56].

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, 10, 11, 14, 17, 20, 29–37, 39–41, 53–55, 59–62, 64]. 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, 47]). For example, the Signal protocol is combined with other cryptographic protocols in WhatsApp [63] to perform abuse reporting or Status [57] and Slyo [58] 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 [52], file sharing services like Keybase [42] (which performs less frequent ratcheting), and videoconferencing services like Zoom [45] (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 [48, 49], and concurrent work by Bienstock et al. [12] 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 [51].)

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 [21, 22]. We focus on modeling Signal at the level specified in their documentation [51] , taking care to adhere to the abstractions within the specification and not limiting our analysis to any single choice of cipher suite.

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 [51]. 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 of instantiating the key exchange module in the plain model, we propose a new abstraction for Signal's continuous key derivation module, which we call a Cascaded PRF-PRG (CPRFG), and we show that this primitive 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 primitive may be useful as a building block 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 to facilitate the use of Signal (or any protocol that realizes $\mathcal{F}_{\mathsf{SM}}$) as a component within other protocols in a 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. If the provided representative string at the time of decapsulation is does not match the original encapsulation exactly, then whether to decapsulate is also decided via adversarially provided code that doesn't get any information about the encapsulated message. This approach avoids restricting to only strong MACs (Message Authentication Codes).

Simple user interface. The above encapsulation and decapsulation requests are the only ways that a parent protocol interacts with $\mathcal{F}_{\mathsf{SM}}$. This eliminates the need for the parent protocol to maintain session-related state such as epoch-ids or sequence numbers. In addition to simplicity, this modeling provides the guarantee that a badly designed parent protocol cannot harm the security of a protocol realising $\mathcal{F}_{\mathsf{SM}}$.[1]

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. In traditional UC modeling, the communication medium is considered part of the service provided by the protocol, and the actual communication is abstracted away.

Moreover, by having $\mathcal{F}_{\mathsf{SM}}$ return to the parent protocol an actual string (that represents an idealized encapsulated message), our model allows the parent protocol to further process the string as needed, similar to what is done in existing systems. This abstraction grants more flexibility in how the secure communication is utilized and integrated within various applications and contexts.

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

[1] Furthermore, this modeling forces any protocol that realizes $\mathcal{F}_{\mathsf{SM}}$ to handle all aspects that are critical to security, such as, say, the timely deletion of sensitive information (keys, internal random choices, epoch identifiers).

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. Similarily to [19, 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 corruption and recovery. One of the main design goals of Signal is its resilience to recurring but transient break-ins. We facilitate the exposition of these properties by modelling corruption as an instantaneous event where the adversary learns the entire state of the corrupted party.[2]

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 practical aspects of the Signal protocol, where 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 [17, 35]. Consequently, $\mathcal{F}_{\mathsf{SM}}$ exhibits similar behavior.

[2] 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" of a party's device 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 the internal state of a party suffices for full impersonation of that party to its peer.

Second, as remarked in the Signal documentation [51], 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, but 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 [24].) 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. [12]. In our modeling, $\mathcal{F}_{\mathsf{SM}}$ forks when one of the parties is compromised, and at the same time, the other party 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 [51] 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 Π_{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 part of the message itself.* This interdependence between message content and the authentication key might initially suggest that a modular security analysis along the above partitioning to modules would be impossible.

One way to get around this issue is to include the AEAD module (or at least the authentication of the first message in each epoch) within the key exchange module, thereby allowing $\mathcal{F}_{\mathsf{eKE}}$ to authenticate new epoch identifiers. However, this would alter the partitioning of modules as given in the Signal protocol. We thus keep the AEAD module separate, and explicitly model the fact that the $\mathcal{F}_{\mathsf{eKE}}$ does not determine the authenticity of new epoch identifiers. In our work, $\mathcal{F}_{\mathsf{eKE}}$ assigns a fresh pseudorandom secret key with each new epoch identifier, regardless of whether it is 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 by the calling protocol, which receives feedback from an authenticated encryption functionality that uses the key k output by $\mathcal{F}_{\mathsf{eKE}}$ for this epoch identifier. This is a separation of tasks successfully breaks-up the circularity issue, allowing it to be adressed across multiple components in a modular fashion. We now 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 other 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, Π_{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 key derivation function (KDF) (of which Signal's typical instantiation using HKDF is a special case.) The desired properties of the KDF are discussed at a high level in Signal's documentation, we introduce the new primitive cascaded PRF-PRG to capture the necessary properties of this module.

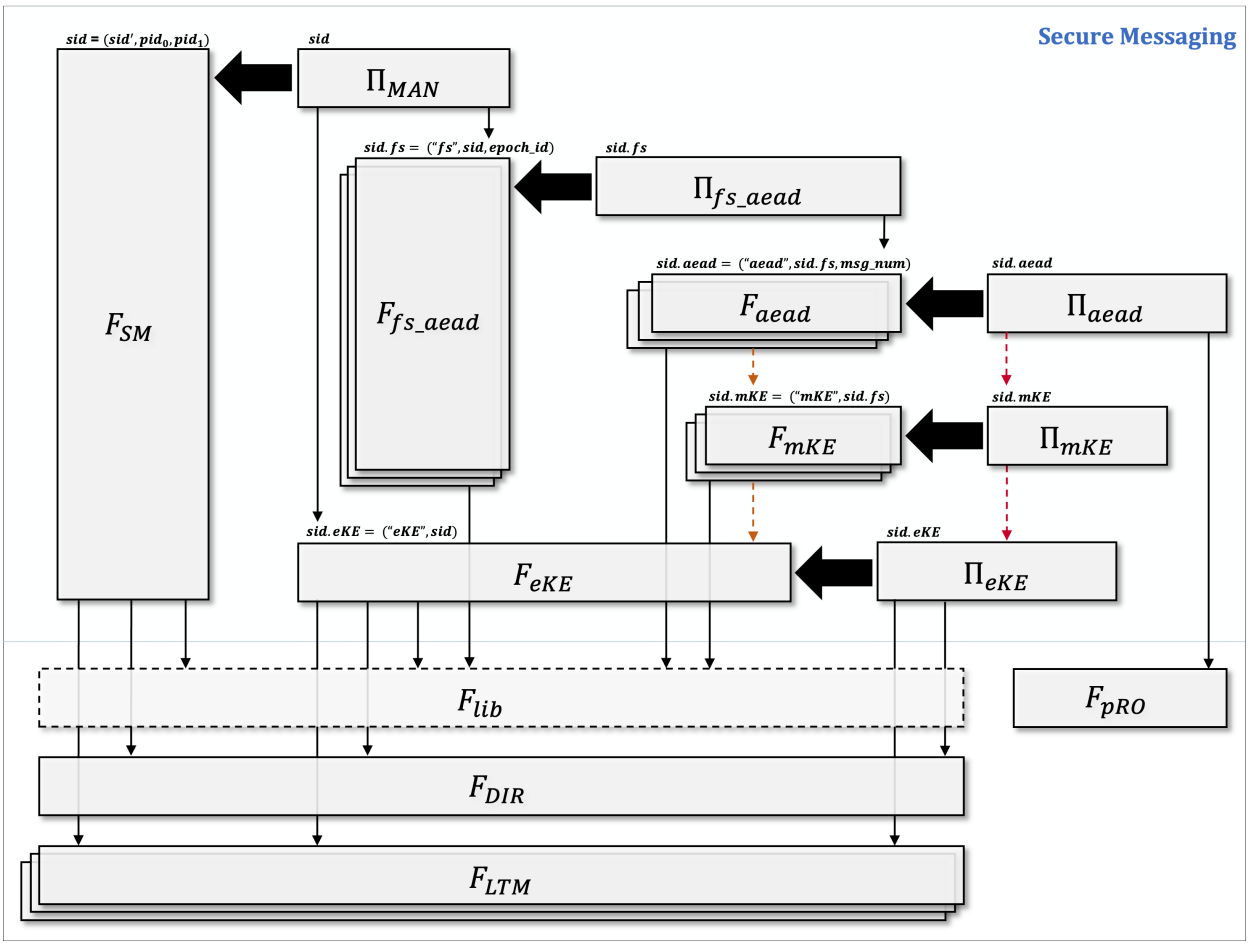

Figure 1: Modeling and realizing secure messaging: The general subroutine structure. Ideal functionalities are denoted by F and protocols by Π. Horizontal arrows denote realization, whereas vertical arrows denote subroutine calls; the red dashed arrows denote subroutine calls to F^{Π} (see Section 2.2 and 4.) Functionalities $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{pRO}}$ are global with respect to $\mathcal{F}_{\mathsf{SM}}$ and all its sub-functionalities, whereas $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{mKE}}$ are global only to a subset of functionalities.

Demonstrating that Π_{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 Π_{mKE}. Using the formalism of [5], we thus show that Π_{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 Π_{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 [50]. 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 Π_{aead}, we treat $\mathcal{F}_{\mathsf{mKE}}$ as a global functionality with respect to Π_{aead}. That is, we show that Π_{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.

Π_{SGNL}. At the highest level of abstraction, we have each of the two parties run protocol Π_{SGNL}. When initiating a session, or starting a new epoch within a session, (i.e., when encapsulating the first message in an epoch), Π_{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 Π_{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 Π_{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 emphasised that Π_{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 [50] applies.

We propose a middle-ground solution: we show how to instantiate the KDF via a plain-model primitive which we call a *cascaded PRF-PRG (CPRFG)* whose 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) and realises a marginally weaker functionality $\mathcal{F}^{*}_{\mathsf{eKE}}$ that differs in how it handles the problem of the adversary sending malformed packets to the receiver during a compromised epoch. Our name "cascaded PRF-PRG" is based on the conceptual similarity between our definition and the PRF-PRG introduced by Alwen et al. [1]; nevertheless, we stress that technically the primitives are quite different, as we elaborate below in Section 1.5.

Modularity with weaker adaptivity. Some earlier analyses of Signal (see Section 1.5) consider security against adversaries which are limited in their adaptivity. Our modular decomposition of Signal as presented here remains valid for such adversaries. That is, as long as the building blocks can withstand a certain level of adaptivity, the overall protocol remains secure.[3]

The benefits of modularity In conclusion, we emphasise the advantages of a fully composable and modular approach to analysing the Signal architecture for secure instant messaging. We mentioned earlier in this section that currently existing implementations of AEAD and the asymmetric ratchet could 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 Π_{aead} and Π_{eKE} to be easily plugged in.

Additional functionalities and methods. Our modeling and instantiation of Signal also relies on the existence of several functionalities at the network level and on individual devices. For instance, we rely on the existence of a public key infrastructure $\mathcal{F}_{\mathsf{lib}}$, which can be modeled as a global UC functionality as described in the next section. We also assume that each party's corresponding secret key is stored in a long-term module $\mathcal{F}_{\mathsf{LTM}}$ that is protected at the system level from exfiltration even if the device is corrupted. Moreover, we presume that local devices have the ability to perform secure deletion, in order to erase old keys as part of the key ratcheting process.

One of the contributions of this work is to specify concretely what functionalities Signal relies upon, and in which components of the system. When a protocol is not specified clearly, an instantiation that adheres to the specified protocol could mess up implied aspects of the protocol (such as deletion of old keys) and, in doing so, open up avenues for attacks. The protocol specifications in

[3] For more details on modeling adaptivity levels of corruptions in the UC framework, see [22, Section 7.1 on Page 69].

this work are somewhat longer than expected because they address such concerns, whereas Signal's specification [51] sometimes lacks such detailed guidance.

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]. Specifically:

- 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.
- Additionally, 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.

Furthermore, since our use of global state is not completely covered by the UCGS Theorem of [5], we extend that theorem so as to cover our use case. To see why our use case is not covered by the UCGS Theorem, recall that the UCGS Theorem states that if a protocol Π UC-realizes functionality $\mathcal{F}$ in the presence of a globally accessible functionality $\mathcal{G}$, then for any protocol ρ, the protocol $\rho^{\mathcal{F}\to\Pi}$ UC emulates ρ in the presence of $\mathcal{G}$.

While this theorem suffices for most of our uses, for our multi-level use of UCGS we would like to additionally show that $\rho^{\mathcal{F}\to\Pi}$ UC emulates ρ even in the presence of $\Pi_{\mathcal{G}}$, where $\Pi_{\mathcal{G}}$ is a protocol that UC-emulates $\mathbb{G}$. However, such implication is not true in general [6, 28].

We get around this problem by proving the following simple-but-useful lemma (Lemma 1 in Section 2): Assume that $\Pi_{\mathcal{G}}$ UC-realizes $\mathcal{G}$ via a simulator $\mathcal{S}$, then any protocol ρ 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 ρ 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., [8, 9, 26, 27]) and self-healing after corruption (e.g., [24, 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, 10, 36, 40, 64]). 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, 53], 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. [12] 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. 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 [12] and [41] differ from our $\mathcal{F}_{\mathsf{SM}}$ in several ways.

- *Differences between our work and both of [12, 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. [12]:* While the modeling of the Signal protocol in [12] 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, [12] accounts for adversarial choice of randomness, which our modeling does not account for. They also propose and analyze an enhancement of the double ratchet structure, which they call the Triple Ratchet protocol, that helps parties regain security faster following a compromise event.

- *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.

Comparison to the CRYPTO 2022 version of this work. In addition to providing full specifications of the ideal functionalities and protocols, as well as full proofs of security, the current version of $\mathcal{F}_{\mathsf{SM}}$ provides a marginally weaker guarantee than the version in [25], in terms of recovery from session compromise. Specifically, the version of $\mathcal{F}_{\mathsf{SM}}$ in [25] guarantees that a session regains its security at the third epoch since the last party corruption event, unless the environment chooses to prevent recovery by delivering a specifically crafted bogus message to the party that expects to receive the first message in a new epoch. If such a message is delivered, the session is forked indefinitely.

The current version of $\mathcal{F}_{\mathsf{SM}}$ allows the environment to deliver yet another form of bogus message to the party that expected the first message in a new epoch. However, in this case the recovery is only delayed so as to take three epochs from the delivery event. We view this weakening as having a secondary impact of the actual security of the protocol, yet it simplifies the analysis and prevents us from using stronger cryptographic hardness assumptions. See more details in Sections **??** The current version of $\mathcal{F}_{\mathsf{SM}}$ allows the environment to deliver yet another form of bogus message to the party that expected the first message in a new epoch. However, in this case the recovery is only delayed so as to take three epochs from the delivery event. We view this weakening as having a secondary impact of the actual security guarantees provided by our treatment, yet it simplifies the analysis and prevents us from using stronger cryptographic hardness assumptions. See more details in Sections 3.2 and 5.1.

2 Universally Composable Security

2.1 Universal Composability: A Primer

UC security [22] is an instantiation of the simulation-based security paradigm in which the real world execution of a protocol Π is compared with an idealized abstraction $\mathcal{F}$. The UC security

framework gives two special powers to the distinguisher (also known as the *environment* Env) in order to provide maximum flexibility to distinguish Π 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 Π 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 Π, 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 (sid) and party ID (pid). All machines that have the same sid and same program Π are called a *session* of Π. 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 sid; this combination uniquely identifies a single machine. Within sid, many (but not all) of the functionalities in this work will include the 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 Π UC-realizes $\mathcal{F}$, and there exists another "hybrid" protocol ρ 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 Π. The composition guarantees that the instantiation $\rho^{\mathcal{F}\to\Pi}$ is "just as secure" as the ρ itself, in the same sense defined above. In this case we say that $\rho^{\mathcal{F}\to\Pi}$ UC-emulates ρ.

UC with global subroutines. Crucially, the UC theorem requires that both $\mathcal{F}$ and Π 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 [23]. 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 Π 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 Π 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 Π or of $\mathcal{F}$, and at the same time take inputs directly from the environment and provide outputs directly to the environment[4].

Now, consider protocol ρ 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 $\rho^{\mathcal{F}\to\Pi}$, that is identical to ρ except that all instances of the ideal functionality $\mathcal{F}$ are replaced by instances of the real protocol Π, UC emulates ρ *in the presence of* G. Note that in both ρ and in $\rho^{\mathcal{F}\to\Pi}$, G may take input from and provide outputs to multiple instance of Π (or of $\mathcal{F}$), of ρ, 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 ρ 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 ρ 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 ρ UC-realizes $\mathcal{F}$ in the presence of $\Pi_{\mathcal{G}}$, then ρ 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 ρ UC-realizes $\mathcal{F}$ in the presence of Π, it suffices to demonstrate that ρ 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 ρ and ideal functionality $\mathcal{F}$ we have that ρ UC-realizes $\mathcal{F}$ in the presence of $\Pi_{\mathcal{G}}$ if and only if ρ UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}^{\mathcal{S}}$.*

[4]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" δ that simply runs Π alongside G as subroutines of δ.

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 be used by ideal functionalities. (We call the uploaded code $\mathcal{I}$ an internal 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 τ represents the code of the target machines to obtain code α. 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 a request for code from some party τ, 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 α 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 α' starting at step 2.
     (d) Inline the code for the calls to α'.
     (e) If this is the last dependency, record $(\tau, \alpha, \mathsf{linking} = \mathit{false})$.
3. Output α.

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 Section 1.2, we dive right into the detailed descriptions of the main components. Section 3.1 describes the functionalities ($\mathcal{F}_{\mathsf{DIR}}$, $\mathcal{F}_{\mathsf{LTM}}$, and $\mathcal{F}_{\mathsf{pRO}}$) that are global with respect to the top level secure messaging functionality $\mathcal{F}_{\mathsf{SM}}$ (other than $\mathcal{F}_{\mathsf{lib}}$ which was described on Page 18). Section 3.2 describes the the top-level secure messaging functionality $\mathcal{F}_{\mathsf{SM}}$.

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 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.

$\mathcal{F}_{\mathsf{pRO}}$. Next, we design a programmable random oracle module $\mathcal{F}_{\mathsf{pRO}}$ (Fig. 4) 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 [18]. Anyone can query the random oracle, but only the (real or ideal world) adversary has the power to program it.

$\mathcal{F}_{\mathsf{LTM}}$. Finally, we design a module $\mathcal{F}_{\mathsf{LTM}}$ (Fig. 5) 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 [46]. 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.

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

This section presents our *secure instant messaging functionality* $\mathcal{F}_{\mathsf{SM}}$ (the complete details of which can be found in Figure 6.) This functionality takes two types of inputs from the peers that are using it, `SendMessage`and `ReceiveMessage`; `SendMessage` inputs are used to encapsulate messages

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

$\mathcal{F}_{\mathsf{DIR}}$ has a fixed session ID, denoted $\mathcal{F}_{\mathsf{DIR}}$.
RecordKeys: On input $(\texttt{RecordKeys}, \mathsf{pid}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}})$ from $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$ do: //For simplicity $\mathcal{F}_{\mathsf{DIR}}$ records just one public key per pid. Multiple public keys per "user level party" can be handled by having multiple 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 pid, respectively, and
2. Output $(\texttt{RecordKeys}, \mathsf{pid}, \texttt{Success})$ to the caller.

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

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

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

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

1. If there is no entry for pid_j then output $(\texttt{GetInitKeys}, \texttt{Fail})$ 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 $(\texttt{GetInitKeys}, \mathsf{pid}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{ok}^{\mathsf{pk}}_{\mathsf{pid}})$ to the caller.

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

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

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

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

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

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

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

   Else choose $h' \xleftarrow{\$} \{0,1\}^{\ell}$ 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 4: The Programamble Random Oracle Functionality, $\mathcal{F}_{\mathsf{pRO}}$

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

$\mathcal{F}_{\mathsf{LTM}}$ is parameterized by a specific activator program Root, a key derivation function HKDF and key generation function keyGen(), and an algebraic group $\mathbb{G}$. All algebraic operations are done in $\mathbb{G}$. The local session ID is of the form $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$. Inputs from senders whose party ID is different than 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 $(\mathsf{ik}^{\mathsf{sk}}_{\mathsf{pid}}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}), (\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}) \xleftarrow{\$}$ 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 $(\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}}) \xleftarrow{\$}$ 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 $(\mathsf{ok}^{\mathsf{sk}}_1, \mathsf{ok}^{\mathsf{pk}}_1), \ldots, (\mathsf{ok}^{\mathsf{sk}}_j, \mathsf{ok}^{\mathsf{pk}}_j) \xleftarrow{\$}$ 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 $(\Pi_{\mathsf{SGNL}}, \mathsf{pid})$, do:

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

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 pid and code Π_{eKE}:

1. Choose an ephemeral key pair $(\mathsf{ek}^{\mathsf{sk}}, \mathsf{ek}^{\mathsf{pk}}) \xleftarrow{\$}$ 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 5: The Long-Term Keys Module Functionality, $\mathcal{F}_{\mathsf{LTM}}$

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

This functionality has a session id $\mathsf{sid} = (sid', \mathsf{pid}_0, \mathsf{pid}_1)$. Inputs arriving from machines whose identity is neither pid_0 nor pid_1 are ignored. //For notational simplicity we assume some fixed interpretation of pid_0 and pid_1 as complete identities of the two calling machines.

`SendMessage`: On input $(\texttt{SendMessage}, m)$ from pid: //pid is the extended identity of a party's machine.

1. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. **If** `initialized` not set do: //initialization
   - **If** $\mathsf{pid} \neq \mathsf{pid}_0$, end the activation. **Else**, send $(\texttt{ConfirmRegistration})$ to $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$.
   - Upon output $(\texttt{ConfirmRegistration}, t)$ from $\mathcal{F}_{\mathsf{LTM}}$: **If** $t = \texttt{Success}$, send $(\texttt{GetInitKeys})$ to $\mathcal{F}_{\mathsf{DIR}}$. **Else**, end the activation.
   - Upon receiving a response $(\texttt{GetInitKeys}, \mathsf{pid}_1, \mathsf{ik}_1^{\mathsf{pk}}, \mathsf{rk}_1^{\mathsf{pk}}, \mathsf{ok}_1^{\mathsf{pk}})$ from $\mathcal{F}_{\mathsf{DIR}}$: **If** $\mathsf{ok}_1^{\mathsf{pk}} \neq \perp$, continue. **Else**, end the activation.
   - Initialize boolean variables `initialized` $= \mathit{true}$, $\mathsf{diverge\_parties} = \mathit{false}$, integer variables (set to 0) $\mathsf{epoch\_num}_0, \mathsf{sent\_msgnum}_0, \mathsf{rcv\_msgnum}_0, N\_\mathsf{self}_0 = 0$ and empty dictionaries $\mathsf{advControl}, \mathsf{id\_dict}, \mathsf{N\_dict} = \{\}$.
     //advControl will record which parties are adversarially controlled in each epoch, id_dict maps epoch id's to epoch numbers, and N_dict will hold the number of messages sent in each epoch.
   - For all $e \geq 0$, set $\mathsf{N\_dict}[e] = \infty$. Set $\mathsf{advControl}[\mathsf{epoch\_num}_0] = \perp$.
   - Initialize $\mathsf{state}_{\mathcal{I}} = \perp$ and call $\mathcal{F}_{\mathsf{lib}}$ to obtain the internal code $\mathcal{I}$.
3. Increment $\mathsf{sent\_msgnum}_i$ by 1.
4. **If** $\texttt{leak} \in \mathsf{advControl}[\mathsf{epoch\_num}_i] \vee \mathsf{diverge\_parties} = \mathit{true}$, send backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{SendMessage}, \mathsf{pid}, m)$ to $\mathcal{A}$. **Else**, run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{SendMessage}, \mathsf{pid}, |m|)$.
5. Upon obtaining $(\mathsf{state}'_{\mathcal{I}}, \texttt{SendMessage}, \mathsf{pid}, \mathsf{epoch\_id}, c)$ from $\mathcal{A}$ or $\mathcal{I}$, continue.
6. Update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$.
7. **If** $\mathsf{sent\_msgnum}_i == 1$: //If this is the start of a new sending epoch store the returned epoch id.
   - **If** $\mathsf{epoch\_id} \notin \mathsf{keys}(\mathsf{id\_dict})$, record $\mathsf{id\_dict}[\mathsf{epoch\_id}] = \mathsf{epoch\_num}_i$. **Else**, end the activation.
8. Set $h = (\mathsf{epoch\_id}, \mathsf{sent\_msgnum}_i, N\_\mathsf{self}_i)$.
   //N_self_i is the number of messages sent by pid_i in its previous sending epoch.
9. **If** $\mathsf{diverge\_parties} = \mathit{false}$, record (pid, h, c, m). **Else**, continue.
   //If the parties' states have diverged, then encrypted messages are no longer recorded.
10. Output $(\texttt{SendMessage}, \mathsf{sid}, \mathsf{pid}, h, c)$ to pid.

`Corrupt`: On input $(\texttt{Corrupt}, \mathsf{pid})$ from Env:

1. **If** $\mathsf{pid} \notin \{\mathsf{pid}_0, \mathsf{pid}_1\}$, end the activation. **Else**, continue.
2. Initialize the list $\mathsf{corruptions}_i$ if it does not exist.
3. Append $(\mathsf{epoch\_num}_i, \mathsf{sent\_msg\_num}_i, \mathsf{received\_msg\_num}_i)$ to the list $\mathsf{corruptions}_i$.
4. For all epochs $e \leq \mathsf{epoch\_num}_i$, set $\mathsf{advControl}[e] = \{\texttt{leak}, \texttt{Inject}\}$ to allow the adversary to influence messages still in transit.
5. Set $\mathsf{advControl}[\mathsf{epoch\_num}_i{+}1], \mathsf{advControl}[\mathsf{epoch\_num}_i{+}2] = \{\texttt{leak}, \texttt{Inject}\}$. Set $\mathsf{advControl}[\mathsf{epoch\_num}_i{+}3] = \{\texttt{leak}\}$.
6. Initialize a list $\mathsf{pending\_msgs} = []$. For each record $(\mathsf{pid}_{1-i}, h, c, m)$ do the following:
   //Make a list of all the messages still in transit to party pid_1.
   (a) **If** there already was a successful `ReceiveMessage` for h (i.e there is a record $(\texttt{Authenticate}, h, c', 1)$ for some c'), continue to the next record. **Else**, append $(\mathsf{pid}_{1-i}, h, c, m)$ to $\mathsf{pending\_msgs}$.
7. Send a request $(\mathsf{state}_{\mathcal{I}}, \texttt{ReportState}, i, \mathsf{pending\_msgs})$ to $\mathcal{A}$.
8. On receiving $(\texttt{ReportState}, i, S)$ from $\mathcal{A}$, output $(\texttt{Corrupt}, S)$ to Env.

(The rest of this functionality is in Fig. 7 on Page 23)

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

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

(This functionality begins in Fig. 7 on Page 23)

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

1. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. **If** this is the first ReceiveMessage request: //initialize the responder
   - **If** $i = 1$, continue. **Else**, end the activation.
   - Send $(\texttt{ConfirmRegistration})$ to $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$.
   - Upon receiving the output $(\texttt{ConfirmRegistration}, t)$ from $\mathcal{F}_{\mathsf{LTM}}$: **If** $t = \texttt{Success}$, continue. **Else**, end activation.
   - Send $(\texttt{GetResponseKeys}, \mathsf{pid}_0, \mathsf{pid}_1)$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving output $(\texttt{GetResponseKeys}, \mathsf{pid}_0, \mathsf{ik}_0^{\mathsf{pk}})$ from $\mathcal{F}_{\mathsf{DIR}}$, continue.
   - Initialize the variables $\mathsf{sent\_msgnum}_1, \mathsf{rcv\_msgnum}_1 = 0$ and $\mathsf{epoch\_num}_1 = 1$.
3. **If** there already was a successful ReceiveMessage for h (i.e there is a record $(\texttt{Authenticate}, h, c', 1)$ for some c'), or this ciphertext previously failed to authenticate (i.e. a record $(\texttt{Authenticate}, h, c, 0)$ exits), output $(\texttt{ReceiveMessage}, h, c, \texttt{Fail})$ to pid.
4. **If** $\mathsf{epoch\_id} \in \mathsf{keys}(\mathsf{id\_dict})$, set temporary variable $\mathsf{epoch\_num} = \mathsf{id\_dict}[\mathsf{epoch\_id}]$. **Else**:
   (a) **If** $\mathsf{sent\_msgnum}_i = 0$, output $(\texttt{ReceiveMessage}, h, c, \texttt{Fail})$ to pid. **Else**, continue.
   //If pid is in a receiving state and hasn't sent any messages in its current sending epoch, it will not accept messages with a new epoch id.
   (b) Set temporary variable $\mathsf{epoch\_num} = \mathsf{epoch\_num}_i + 1$.
   (c) **If** $\texttt{leak} \in \mathsf{advControl}[\mathsf{epoch\_num}]$:[a]
   - For epochs $e \in \{\mathsf{epoch\_num}, \mathsf{epoch\_num} + 1\}$: Set $\mathsf{advControl}[e] = \{\texttt{leak}, \texttt{Inject}\}$.
   - Add leak to $\mathsf{advControl}[\mathsf{epoch\_num} + 2]$
5. **If** $\mathsf{msg\_num} > \mathsf{N\_dict}[\mathsf{epoch\_num}]$, output $(\texttt{ReceiveMessage}, h, c, \texttt{Fail})$ to pid
   //For epoch_num's that are not finished yet, the N_dict returns a default value of ∞, so this check passes automatically.
6. **If** $\mathsf{diverge\_parties} = \mathit{false} \wedge \texttt{Inject} \notin \mathsf{advControl}[\mathsf{epoch\_num}]$, run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, h, c)$. //honest case
7. **Else**, send backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, h, c)$ to $\mathcal{A}$.
8. On receiving $(\mathsf{state}'_{\mathcal{I}}, \texttt{Inject}, h, c, v)$ from $\mathcal{A}$ or $\mathcal{I}$:
   - Update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$.
   - **If** $v = \perp$, record $(\texttt{Authenticate}, \mathsf{pid}, h, c, 0)$ and output $(\texttt{ReceiveMessage}, h, c, \texttt{Fail})$. **Else**, continue.
   - **If** $\mathsf{diverge\_parties} = \mathit{false}$ and $\texttt{Inject} \notin \mathsf{advControl}[\mathsf{epoch\_num}]$: //honest case
     – **If** there is a record $(\mathsf{sender}, h, c^*, m)$ for header h, record $(\texttt{Authenticate}, h, c, 1)$ and set $m^* = m$. **Else**, output $(\texttt{ReceiveMessage}, h, c, \texttt{Fail})$. //allow authentication of a message with a different mac in the honest case.
   - **Else**: //compromised case
     – Record $(\texttt{Authenticate}, h, c, 1)$, and set $m^* = v$.
     – **If** $\mathsf{epoch\_id}$ does not appear as a key in $\mathsf{id\_dict}$ then set $\mathsf{diverge\_parties} = \mathit{true}$. //diverge parties is being set here.
9. **If** $\mathsf{epoch\_num}_i < \mathsf{epoch\_num}$: //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 $\mathsf{sent\_msgnum}_i = 0$.
10. Output $(\texttt{ReceiveMessage}, h, m^*)$ to pid.

[a] This provision (which was missing in the [25] version) allows the environment to delay recovery from corruption by delivering messages with bogus epoch ids, this occurs even if these messages do not contain a payload that passes authentication.

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

for sending to the peer, whereas `ReceiveMessage` inputs are used to decapsulate received packets. The functionality takes a few other 'modeling inputs' that are not part of the interface of protocols realising this functionality. The first of these, `Corrupt`, captures the meaning of party corruptions in our model. The others are a number of 'side channel' messages from the adversary, these are used to fine-tune the security guarantees within the `SendMessage` and `ReceiveMessage` interfaces.

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 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 pid_0 is the identity of the creating party (ie, the *initiator*), and 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. Upon the first $(\texttt{SendMessage}, m)$ activation (which is the first activation of the functionality overall) $\mathcal{F}_{\mathsf{SM}}$ runs its initialization by doing the following four things:

1. Verifying the existence of the instance of $\mathcal{F}_{\mathsf{LTM}}$ corresponding to the initiator pid_0.
2. Checking if the desired peer (pid_1) has an available one-time key $\mathsf{ok}^{\mathsf{pk}}$ registered with the directory $\mathcal{F}_{\mathsf{DIR}}$.
3. Initiating variables that will record subsequent epoch identifiers, message numerals within each epoch, compromised epochs, etc.
4. Initializing internal adversarial state $\mathsf{state}_{\mathcal{I}}$ and calling $\mathcal{F}_{\mathsf{lib}}$ to obtain internal adversarial code $\mathcal{I}$.[5]

If the local state of $\mathcal{F}_{\mathsf{SM}}$ indicates that this `SendMessage` activation is the first one in a new epoch, then $\mathcal{F}_{\mathsf{SM}}$ will have already allowed the ideal-model adversary (namely, the simulator) to choose a new epoch identifier for this epoch within the `ReceiveMessage` interface. When the epoch is uncompromised, $\mathcal{I}$ will have been run internally by $\mathcal{F}_{\mathsf{SM}}$ to allow the simulator to choose the new epoch identifier. Otherwise the epoch is compromised and $\mathcal{F}_{\mathsf{SM}}$ will have asked the simulator for a new epoch identifier and waited to receive a response. In both cases $\mathcal{F}_{\mathsf{SM}}$ will have verified that the newly chosen identifier is different than all previously used ones before continuing.

After initialization during the first activation and at the start of all subsequent activations, $\mathcal{F}_{\mathsf{SM}}$ lets the ideal-model adversary choose the ciphertext c that will correspond to m. There are two cases **(Case 1: Uncompromised sending epoch.)** The functionality allows the simulator to make the choice of ciphertext by running the code $\mathcal{I}$ internally with only the length of m as input. **(Case 2: Compromised sending epoch.)** The message m is leaked in full to the simulator by $\mathcal{F}_{\mathsf{SM}}$ who then waits for the simulator to send back its chosen ciphertext c.

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. The output (c, h) is sent to pid_0. 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.

[5] When no party is compromised, the functionality never hands over control to the simulator. Instead, it allows adversarial choices by internally running the adversarially provided code $\mathcal{I}$. This enables the functionality to achieve immediate decryption.

ReceiveMessage. This input allows the receiving party to perform an "idealized authenticated decryption" operation, even though the ciphertext was generated without knowledge of the message and $\mathcal{F}_{\mathsf{SM}}$ itself has no keying material.

Upon the first $(\texttt{ReceiveMessage}, c, h)$ input, $\mathcal{F}_{\mathsf{SM}}$ perform initialization by (1) Verifying that this request is coming from pid_1 (2) Verifying that the instance of $\mathcal{F}_{\mathsf{LTM}}$ that corresponds to pid_1 exists, and (3) Verifying that pid_0 is registered with $\mathcal{F}_{\mathsf{DIR}}$.

After initialization during the first activation and on all subsequent activations, decryption proceeds with respect to the following cases:

1. if there is an encryption record (m, c, h) for the header h with the exact ciphertext c, then $\mathcal{F}_{\mathsf{SM}}$ returns the corresponding message m to pid regardless of whether the epoch is compromised.

2. Otherwise, if there is some encryption record (m, c', h) for the header h with a different ciphertext c' (presumably, c is a "mauled ciphertext") then $\mathcal{F}_{\mathsf{SM}}$ gives the adversary the latitude to decide whether decryption should succeed. This behavior combines the standard EU-CMA guarantee for the underlying authentication scheme, combined with one-time decryption. There are two sub-cases **1) Uncompromised receiving epoch.** The functionality allows the simulator to decide if the ciphertext c successfully decrypts by running the code $\mathcal{I}$ internally and providing it h and c as inputs. If the output of $\mathcal{I}$ isn't $\perp$ then decryption succeeds and the message m is output. Hence immediate decryption is preserved. **2) Compromised receiving epoch.** The ideal-model adversary can cause the receiving party to accept any plaintext of its choosing. To this end, the functionality provides h and c as inputs to the adversary and waits for the adversary to return a plaintext m' to decrypt to. As long as $m' \neq \perp$, decryption succeeds.

3. Finally, if there is no record (m, c', h) for the header h, then $\mathcal{F}_{\mathsf{SM}}$ fails to decrypt unless the next receiving epoch of decrypting party is compromised. In the case that the next receiving epoch is compromised, the adversary may attempt a person-in-the-middle attack which succeeds if and only if the honest recipient accepts a new epoch created by the adversary rather than the honest sender. The functionality proceeds as in the previous case and gives the adversary the latitude to decide whether decryption should succeed and what the decrypted plaintext should be. If decryption succeeds then the party has now started receiving messages in a new epoch where its peer can never send a message and therefore the parties' states and keys have diverged from each other. 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, the functionality behaves as if both parties are compromised for the rest of the session. This divergence(i.e. fork) event represents a complete break of security of the session. If decryption doesn't succeed then the compromise is extended to two epochs past the next receiving epoch of the decrypting party.

It is stressed that $\mathcal{F}_{\mathsf{SM}}$ only supports decrypting a message for each header once; whenever a successful decryption is output by $\mathcal{F}_{\mathsf{SM}}$ for some header h, the functionality will note this and will refuse to participate in all subsequent calls with the same header h. This ensures forward secrecy. In addition, $\mathcal{F}_{\mathsf{SM}}$ stores the exact number of messages sent in previous epochs so that honest parties can detect if there is an attempt to inject a message after the sender's planned ending to an epoch.

Finally, if this happens to be the first successfully received message for party pid_i in the newest epoch, then $\mathcal{F}_{\mathsf{SM}}$ notes that the next $\texttt{SendMessage}$ activation with sender pid will start a new epoch and allows the ideal-model adversary (namely, the simulator) to choose a new epoch identifier for this sending epoch as described earlier.

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).

First, 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 $\mathcal{F}_{\mathsf{SM}}$. However, any messages in transit at the time of corruption is sent to the ideal-model adversary instantly, who sends back a simulated state that can be output to the environment. We've already described the behavior of $\mathcal{F}_{\mathsf{SM}}$ within the `SendMessage` and `ReceiveMessage` interfaces while epochs are compromised.

So all that's left is describing the process by which the functionality recovers from a compromise. The point by which a compromised session 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 newest sending epoch. For a compromised session to become uncompromised, the corrupted party must start at least two new sending epochs since the last corruption event. Specifically, if party pid_i is corrupted when it is in sending epoch e, the resulting compromise lasts at least until epoch number $e+3$. The compromise period may be extended further if a party receives a packet with a mauled header that corresponds to some compromised epoch e^*. In this case, one of two things can happen:

- **If decryption fails, the compromise extends until epoch e^*+2.** This outcome corresponds to a weakness in the standalone security of Π_{eKE}, as it does not include any authentication of keys. [6]
- **If decryption succeeds, the functionality behaves as if both parties are compromised for the rest of the session.** The success of decryption corresponds to the party starting to receive messages in a new epoch where its peer can never send a message, causing the parties' states and keys to diverge from each other. In this scenario, $\mathcal{F}_{\mathsf{SM}}$ notes that the session is *forked*—or in other words, that the parties' states have *diverged*, and the functionality then behaves as if both parties are compromised for the rest of the session. This divergence (i.e., fork) event represents a complete break of security for the session.

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

[6] This provision was missing in the [25] version of this work.

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 10, which graphically depicts the various components and how they fit together.

A Signal-style realization of $\mathcal{F}_{\mathsf{SM}}$. In our first instantiation depicted by the thick black arrow at the top left of Fig. 1, we decompose the Signal protocol into two ideal modules representing the interconnected components of the double ratchet architecture: (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 Π_{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 through the incorporation of randomness, and (2) long-term forward secrecy. Meanwhile, the short-lived forward secure encryption modules $\mathcal{F}_{\mathsf{fs\_aead}}$ each grant fine-grained forward secrecy within the epochs they are associated with. By relying on $\mathcal{F}_{\mathsf{eKE}}$, the $\mathcal{F}_{\mathsf{fs\_aead}}$ modules additionally enable the overall protocol to realise the long-term forward secrecy and healing properties endowed by $\mathcal{F}_{\mathsf{eKE}}$. At this level of abstraction, the reliance of $\mathcal{F}_{\mathsf{fs\_aead}}$ on $\mathcal{F}_{\mathsf{eKE}}$ is indirect (via the epoch identifiers.) However, at a lower level of abstraction, the security of this module will depend directly on $\mathcal{F}_{\mathsf{eKE}}$ since the chain keys provided by $\mathcal{F}_{\mathsf{eKE}}$ will be used eventually to derive the encapsulation and decapsulation keys for individual messages. 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 (Π_{SGNL}, Fig. 16)*: The protocol Π_{SGNL} (see Section 5.3) is the top-level protocol that interfaces only with the ideal functionalities $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{fs\_aead}}$ and realises the functionality $\mathcal{F}_{\mathsf{SM}}$. There are three primary takeaways from the design of Π_{SGNL}: 1) it has the same input-output API as our ideal functionality $\mathcal{F}_{\mathsf{SM}}$, 2) it displays an idealized version of the double ratchet with clearly distinct roles for the two subroutines, and finally 3) it moves one level of abstraction closer to the specification of the real Signal architecture. 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. 13)*: The epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$ persists throughout the entire secure messaging session between two parties. It represents the root ratchet of the Signal protocol which comprises the public key "backbone" of continuous key agreement in the secure messaging architecture of Signal. At the level of abstraction where Π_{SGNL} realises $\mathcal{F}_{\mathsf{SM}}$, The only role of the epoch key exchange functionality is to produce the epoch id's that identify the communicating parties' sending epochs. At this level the functionality interacts only with the protocol Π_{SGNL} and the global functionalities $\mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$. However, $\mathcal{F}_{\mathsf{eKE}}$ doesn't only produce epoch identifiers. It also maps identifier pairs $(\mathsf{epoch\_id}_0, \mathsf{epoch\_id}_1)$ to sending and receiving chain keys that can be leveraged by the symmetric ratchet to harness $\mathcal{F}_{\mathsf{eKE}}$'s properties of *healing from compromise* and *coarse-grained forward secrecy*. Looking ahead, this property of $\mathcal{F}_{\mathsf{eKE}}$ is used two levels of abstraction below this one, where the module $\mathcal{F}_{\mathsf{mKE}}$ is instantiated by a protocol Π_{mKE}. The message

key exchange protocol retrieves the chain keys produced by $\mathcal{F}_{\mathsf{eKE}}$ and uses them to produce encapsulation and decapsulation keys for individual messages.

- *Forward Secure Authenticated Encryption ($\mathcal{F}_{\mathsf{fs\_aead}}$, Fig. 15)*: 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 sender's (resp. receiver's) Π_{SGNL} for the epoch may send a `StopEncrypting` (resp. `StopDecrypting`) request to the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance, after which the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance will no longer allow any more messages to be encrypted (resp. decrypted) 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.

In Section 5 we provide a rigorous specification for each of the modules $\Pi_{\mathsf{SGNL}}, \mathcal{F}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{fs\_aead}}$ discussed above. We then prove Theorem 2 (Figure 8): we provide a concrete simulator $\mathcal{S}_{\mathsf{SM}}$ and show that (together with $\mathcal{F}_{\mathsf{SM}}$) it is perfectly indistinguishable from the Π_{SGNL} hybrid world.

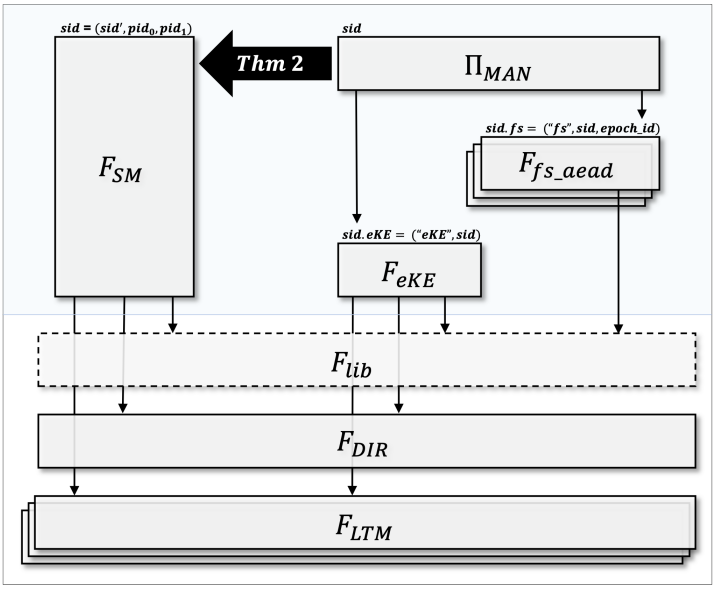

Figure 8: A pictorial rendition of Theorem 2

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

Instantiating the public ratchet (realizing $\mathcal{F}_{\mathsf{eKE}}$). The instantiation of $\mathcal{F}_{\mathsf{eKE}}$ in a modular fashion is one of the most delicate parts of this work (alongside the teasing apart of a modular functionality $\mathcal{F}_{\mathsf{eKE}}$ to represents the security provided by Signal's public key ratchet). Signal's public key ratchet is specified by Π_{eKE} in Section 6.1 of our work and the instantiation of $\mathcal{F}_{\mathsf{eKE}}$ is depicted in Figure 1 by the lowermost thick black arrow in the diagram. One main challenge, as observed by Alwen et al. [1] and others, is that the key derivation function (KDF) within the public ratchet (Π_{eKE}), must maintain security if either of the previous root key or the newly generated

ephemeral keys are uncompromised. As described 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.[7]

Alwen et al. formalized the guarantee they realised KDF must satisfy by way of constructing a new primitive: a PRF-PRG. In Section 6.2 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 a plain model instantiation of this primitive based only on punctured PRFs (Thm. 12 in Section 6.2.3). Using this CPRFG construction, we are able to UC-realise the functionality $\mathcal{F}_{\mathsf{eKE}}$ (described in Section 6.3) The proof of Theorem 3 (Figure 9) is quite intricate; See Section 6.3 for the details.

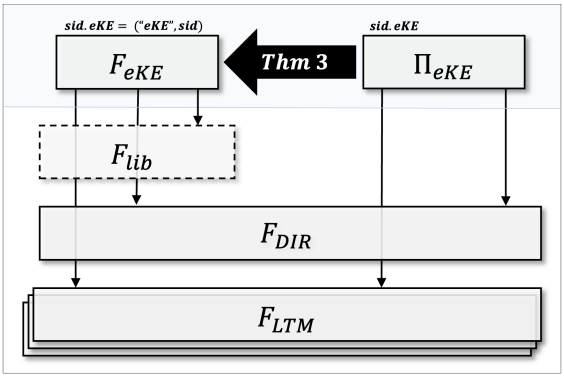

Figure 9: A pictorial rendition of Theorem 3

Theorem 3 *Assume that KDF is a CPRFG with security parameter λ, that the DDH assumption holds in the group $\mathbb{G}$ where $|\mathbb{G}| \geq 2^{\lambda}$. Then protocol Π_{eKE} UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ in the presence of global functionalities $\mathcal{F}_{\mathsf{lib}}$, $\mathcal{F}_{\mathsf{DIR}}$ 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. 28)*: Each message key exchange functionality instance handles the key derivation for the symmetric ratchet for a particular epoch. Specifically, it provides key_seed's to Π_{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 key_seed's up to N and later disallowing the generation of any further key seeds for its epoch. The protocol (Π_{mKE}, Fig. 34) realizes $\mathcal{F}_{\mathsf{mKE}}$ by iteratively applying a length-doubling pseudorandom function to the chain_key provided by Π_{eKE} to generate key_seed's. If it needs to skip message key seeds (for example, if the messages arrive out of order), Π_{eKE} applies the PRG several times until reaching the correct key_seed, meanwhile storing intermediate 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 chain_key. The functionality

[7]Note that it is relatively straightforward to achieve both these guarantees by modeling the KDF as a programmable random oracle.

(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 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. 29)*: 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 key_seed from Π_{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 (Π_{aead}, Fig. 36) realizes $\mathcal{F}_{\mathsf{aead}}$ by querying the random oracle $\mathcal{F}_{\mathsf{pRO}}$ on the key_seed to get the full 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 sid (which contains information about the pid's, epoch_id, msg_num, and such). To decrypt, Π_{aead} similarly gets the key_seed from $\mathcal{F}_{\mathsf{mKE}}$ and queries $\mathcal{F}_{\mathsf{pRO}}$ to expand it. Then, Π_{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).

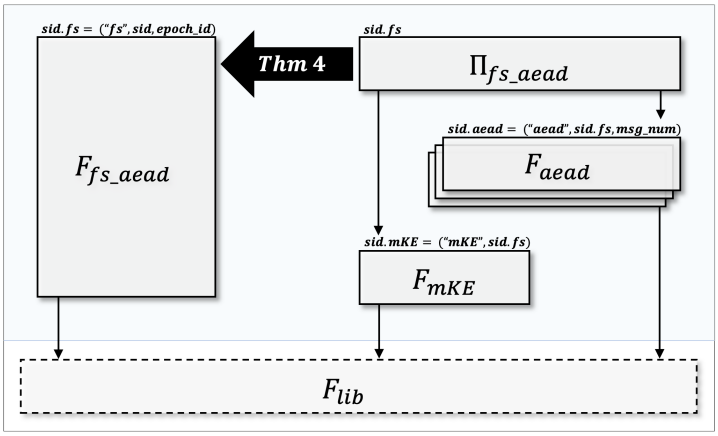

Figure 10: A pictorial rendition of Theorem 4

Theorem 4 *Protocol $\Pi_{\mathsf{fs\_aead}}$ (perfectly) UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ in the presence of $\mathcal{F}_{\mathsf{lib}}$.*

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.

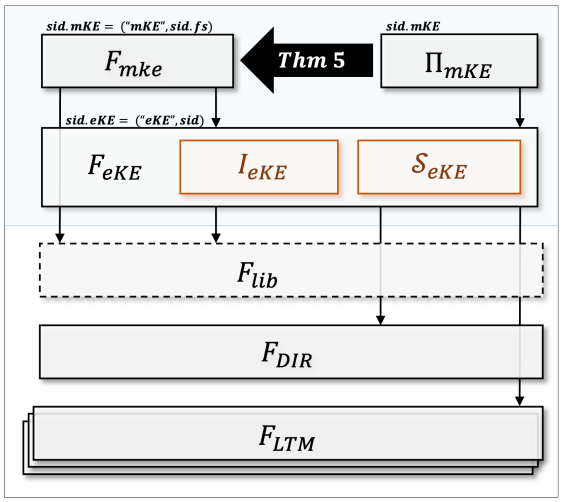

Figure 11: A pictorial rendition of Theorem 5

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

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 [21] that introduced the UC security framework. We show the following theorem in Section 9.

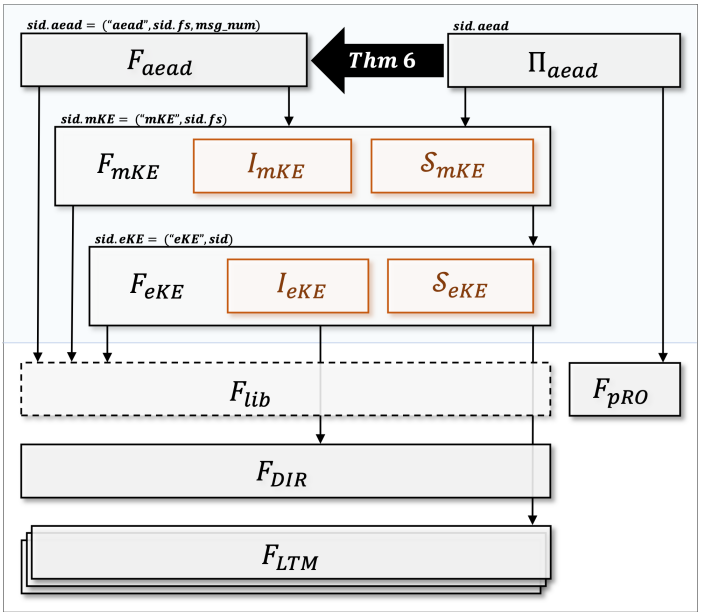

Figure 12: A pictorial rendition of Theorem 6

Theorem 6 *Assuming the unforgeability of* (MAC, Verify)*, protocol* Π_{aead} *UC-realizes the ideal*

functionality $\mathcal{F}_{\mathsf{aead}}$ *in the presence of global functionalities* $\mathcal{F}_{\mathsf{pRO}}$, $\mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$, *as well as* $\mathcal{F}_{\mathsf{mKE}}^{\Pi_{\mathsf{mKE}}}$, $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}}$, *where* $\mathcal{F}_{\mathsf{mKE}}^{\Pi_{\mathsf{mKE}}} = (\mathcal{S}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{mKE}})$, *and* $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{I}_{\mathsf{eKE}}, \mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.

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 Π_{mKE} UC-realizes $\mathcal{F}_{\mathsf{mKE}}$ in the presence of Π_{eKE}, as well as $\mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}$:

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

Next we use Theorem 6 together with Corollary 7 and Lemma 1 to deduce that Π_{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* Π_{aead} *is unforgeable, that the KDF module used by* Π_{eKE} *is a CPRFG, that the DDH assumption holds for the group* $\mathbb{G}$ *used in* $\mathcal{F}_{\mathsf{LTM}}$ *and* Π_{eKE}*, and that the* PRG *used in* Π_{mKE} *is a secure length-doubling pseudorandom generator. Then protocol* Π_{aead} *UC-realizes* $\mathcal{F}_{\mathsf{aead}}$ *in the presence of* $\Pi_{\mathsf{mKE}}, \Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$.

Now, recall that $\Pi_{\mathsf{fs\_aead}}^{\mathcal{F}_{\mathsf{mKE}} \to \Pi_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{aead}} \to \Pi_{\mathsf{aead}}}$ is the protocol that's identical to $\Pi_{\mathsf{fs\_aead}}$, except that calls to $\mathcal{F}_{\mathsf{mKE}}$ are replaced with calls to Π_{mKE} and calls to $\mathcal{F}_{\mathsf{aead}}$ are replaced with calls to Π_{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* Π_{aead} *is unforgeable, that the KDF module used by* Π_{eKE} *is a CPRFG, that the DDH assumption holds for the group* $\mathbb{G}$ *used in* $\mathcal{F}_{\mathsf{LTM}}$ *and* Π_{eKE}*, and that the* PRG *used in* Π_{mKE} *is a secure length-doubling pseudorandom generator. Then protocol* $\Pi_{\mathsf{fs\_aead}}^{\mathcal{F}_{\mathsf{mKE}} \to \Pi_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{aead}} \to \Pi_{\mathsf{aead}}}$ *UC-realizes* $\mathcal{F}_{\mathsf{fs\_aead}}$ *in the presence of* $\Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$.

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

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

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 (Π_{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 Π_{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 Π_{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 13 on page 38) 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 pid_i receives a ciphertext from pid_{1-i} (through $\mathcal{F}_{\mathsf{fs\_aead}}$) marked with an $\mathsf{epoch\_id}$ that it has not seen before, pid_i knows that pid_{1-i} has started a new epoch. The party 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 pid_i. If the verification succeeds, 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 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. pid_0 is the sender for all even numbered epochs and 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. [8] 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.

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

[8] 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.

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 Π_{mKE} respectively. At this level, when a Π_{mKE} protocol instance belonging to 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 Π_{SGNL}, $\mathcal{F}_{\mathsf{eKE}}$ receives such a request when the party 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.)[9] 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. 13.) The epoch key exchange functionality takes inputs from and provide output to the corresponding two machines of Π_{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.

Confirm Receiving Epoch After initialization, the epoch key exchange functionality expects an input of the form $(\mathtt{ConfirmRcvEpoch}, \mathsf{epoch\_id}^*)$ 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

[9] Note that even at the level where $\mathcal{F}_{\mathsf{fs\_aead}}$ is instantiated by $\Pi_{\mathsf{fs\_aead}}+\mathcal{F}_{\mathsf{mKE}}+\mathcal{F}_{\mathsf{aead}}$, no keys are used and therefore this request is never sent to $\mathcal{F}_{\mathsf{eKE}}$.

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 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 Π_{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 `GetSendingKey` requests from the parties' Π_{mKE} instances. For each epoch, the functionality $\mathcal{F}_{\mathsf{eKE}}$ expects one `GetSendingKey` request from a Π_{mKE} instance that belongs to the sender 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 `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 `GetReceivingKey` requests from the parties' Π_{mKE} instances. For each epoch, the functionality $\mathcal{F}_{\mathsf{eKE}}$ expects a non-zero number of `GetReceivingKey` requests from Π_{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 Π_{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 Π_{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 Π_{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 Π_{SGNL} instance that made the corruption request. Before responding to the calling instance of Π_{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 may be achieved. This is because the recovery for a single compromise goes through the following phases:

1. **full state compromise (the party's entire state is known during epochs e, $e+1$.)** If a party is corrupted during sending epoch e then they will be a receiver in epoch $e+1$. Since the party does not add any secret randomness to its state in epoch $e+1$, the adversary still knows the party's entire state at this point.

2. **sender's randomness is updated (epoch $e+2$)** In epoch $e+2$, the corrupted party will update its randomness and send a value $\mathsf{epoch\_id}_{e+2}$ to their peer. Once their peer confirms the new epoch id, the adversary will be unable to tamper with the communication in a significant way; however, full deniability has not been restored.

3. **both parties' randomness is updated (epoch $e+3$)** In epoch $e+3$ the peer will update its randomness and send a value $\mathsf{epoch\_id}_{e+3}$ to the corrupted party. Once the corrupted party confirms the new epoch id, full deniability will be restored. However, since such a confirmation is evidenced by the start of epoch $e+4$, this is the first uncompromised epoch after the corruption.

The compromise period may be extended further if a party receives a packet with a bogus epoch id that corresponds to some compromised epoch e^*. In this case, one of two things can happen:

- **If the party does not confirm the bogus epoch id, the compromise extends until epoch e^*+2.** This outcome corresponds to a weakness in the standalone security of Π_{eKE}, as it does not include any authentication of keys.[10]

- **If the party *does* confirm the bogus epoch id, the functionality behaves as if both parties are compromised for the rest of the session.** The confirmation of a bogus epoch id corresponds to the divergence of the parties' states and keys. In this scenario, $\mathcal{F}_{\mathsf{eKE}}$ notes that the parties have *diverged*, and the functionality then behaves as if both parties are compromised for the rest of the session.

[10] This provision was missing in the [25] version of this work.

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 $\Pi_{\mathsf{SGNL}} + \Pi_{\mathsf{fs\_aead}} + \Pi'_{\mathsf{eKE}}$ 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 Π_{eKE} in fact allows healing in 2 rounds instead of 3. This modification is discussed in the remark on Page 51

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

The forward secure authenticated encryption functionality (see Figure 15 on page 41) processes encryptions and decryptions for a single epoch (specified in $\mathsf{sid}.fs$) for the protocol Π_{SGNL}. 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 protocol Π_{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 Π_{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 Π_{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 Π_{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 Π_{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.

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

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 pid_0 nor pid_1 are ignored. //For notational simplicity we assume some fixed interpertation of pid_0 and 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, initialize conversation using $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$ and get internal code from $\mathcal{F}_{\mathsf{lib}}$:
   (a) Parse sid to retrieve two party ids $(\mathsf{pid}_0, \mathsf{pid}_1)$ for the initiator and responder parties and store them. If $\mathsf{pid}_0 \neq \mathsf{pid}_i$ end the activation.
   (b) Provide input $(\texttt{GetInitKeys}, \mathsf{pid}_1, \mathsf{pid}_0)$ to $(\mathcal{F}_{\mathsf{DIR}})$.
   (c) Upon receiving output $(\texttt{GetInitKeys}, \mathsf{ik}_1^{\mathsf{pk}}, \mathsf{rk}_1^{\mathsf{pk}}, \mathsf{ok}_1^{\mathsf{pk}})$ from $(\mathcal{F}_{\mathsf{DIR}})$:
      i. **If** $\mathsf{ok}_{\mathsf{pid}_1}^{\mathsf{pk}} = \perp$ then output $(\texttt{ConfirmReceivingEpoch}, \texttt{Fail})$.
      //Don't start the conversation if the one time keys belonging to the other party have run out.
      ii. Initialize empty lists $\mathsf{corruptions}_0$, $\mathsf{corruptions}_1$, $\mathsf{compromised\_epochs}$.
      iii. Set $\mathsf{epoch\_id\_partner}_0 = \mathsf{epoch\_id\_self}_1 = \mathsf{ok}_{\mathsf{pid}_1}^{\mathsf{pk}}$, $\mathsf{epoch\_num}_0 = -2$, and $\mathsf{epoch\_num}_1 = -1$.
      iv. Send $(\texttt{ComputeSendingRootKey}, \mathsf{ik}_1^{\mathsf{pk}}, \mathsf{rk}_1^{\mathsf{pk}}, \mathsf{ok}_1^{\mathsf{pk}})$ to $\mathcal{F}_{\mathsf{LTM}}$.
   (d) On receiving $(\texttt{ComputeSendingRootKey}, k, \mathsf{ek}^{\mathsf{pk}})$ from $\mathcal{F}_{\mathsf{LTM}}$, continue.
   (e) Initialize $\mathsf{state}_{\mathcal{I}} = \perp$, call $\mathcal{F}_{\mathsf{lib}}$ to obtain internal adversarial code $\mathcal{I}_{\mathsf{eKE}}$.
2. **Else** (this is not first activation):
   (a) Set $\mathsf{epoch\_id\_partner}_i = \mathsf{epoch\_id}^*$.
   (b) **If** $\mathsf{epoch\_id}^* \neq \mathsf{epoch\_id\_self}_{1-i}$: Set $\mathsf{diverge\_parties} = \mathit{true}$, run step 3 of $\texttt{Corrupt}$ to set $\mathsf{recv\_chain\_key}^*$ and $\mathsf{leakage}$, and send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}^*, \mathsf{leakage})$ to the adversary. On receiving a response, continue. //$\mathsf{diverge\_parties}$ is set here
   //If this may be the first divergence, make sure the simulator has control.
3. **If** $\mathsf{epoch\_num}_i + 2 \in \mathsf{compromised\_epochs}$ or $\mathsf{diverge\_parties}$: Send backdoor message $(\texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$ to the adversary. **Else** run $\mathcal{I}_($$\mathsf{state}_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch_id}^*)$.
   //If the parties are diverged or compromised send a backdoor message to the adversary, otherwise run the internal adversarial code.
4. Upon receiving $(\texttt{GenEpochId}, \mathsf{state}'_{\mathcal{I}}, i, \mathsf{epoch\_id})$ from $\mathcal{A}$ or from $\mathcal{I}$, update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$ and do the following:
   (a) **If** $\mathsf{epoch\_id}$ is the same as any previous invocation of $\texttt{GenEpochId}$, end the activation.
   (b) Set $\mathsf{epoch\_id\_self}_i = \mathsf{epoch\_id}$, $\mathsf{epoch\_num\_dict}[\mathsf{epoch\_id}] = \mathsf{epoch\_num}_i$, $\mathsf{got\_sending\_key}_i = \mathit{false}$, and $\mathsf{epoch\_num}_i \mathrel{+}= 2$. //save the next sending epoch id.
   (c) 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. Parse $\mathsf{sid}.mKE = (\text{“}mKE\text{”}, \text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$. **If** $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_i$ end the activation; **else** set i such that $\mathsf{pid} = \mathsf{pid}_i$.
2. **If** $\mathsf{got\_sending\_key}_i = \mathit{true}$ or $\texttt{ConfirmReceivingEpoch}$ has never been run successfully then end the activation.
//the functionality isn't initialized or the sending key for the current epoch has already been retrieved
3. Sample $\mathsf{sending\_chain\_key}_i \xleftarrow{\$} \mathcal{K}_{ep}$ 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.
4. **If** $\mathsf{diverge\_parties} = \mathit{true}$, or $\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$ send backdoor message $(\texttt{GetSendingKey}, i)$ to the adversary; on receiving backdoor message $(\texttt{GetSendingKey}, i, K_{\mathtt{send}})$ from $\mathcal{A}$ set $\mathsf{sending\_chain\_key}_i = K_{\mathtt{send}}$.
//If the parties are diverged or compromised let the adversary choose $\mathsf{sending\_chain\_key}_i$.
5. 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. 14 on Page 39)

Figure 13: The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$

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

(This functionality begins in Fig. 13 on Page 38)

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. **Else**, set i such that $\mathsf{pid} = \mathsf{pid}_i$.
2. **If** $\mathsf{sending\_chain\_key}_{1-i}$ has been deleted or `ConfirmReceivingEpoch` has never been run successfully end the activation.
3. Parse $\mathsf{epoch\_id} = (\mathsf{epoch\_id}', \mathsf{ek}_j^{\mathsf{pk}}, \mathsf{ok}_{i\leftarrow j}^{\mathsf{pk}})$ 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 `GetReceivingKey`:
   (a) **If** $\mathsf{pid}_1 \neq \mathsf{pid}_i$, then end the activation.
   (b) Send $(\texttt{GetResponseKeys}, \mathsf{pid}_{1-i})$ to $\mathcal{F}_{\mathsf{DIR}}$.
   (c) Upon receiving $(\texttt{GetResponseKeys}, \mathsf{ik}_j^{\mathsf{pk}})$, send input $(\texttt{ComputeReceivingRootKey}, \mathsf{ik}_j^{\mathsf{pk}}, \mathsf{ek}_j^{\mathsf{pk}}, \mathsf{ok}_{i\leftarrow j}^{\mathsf{pk}})$ to $\mathcal{F}_{\mathsf{LTM}}$.
5. **If** $\mathsf{diverge\_parties} = \mathit{true}$ or $\mathsf{epoch\_num}_i + 1 \in \mathsf{compromised\_epochs}$: //Let $\mathcal{A}$ choose key
   (a) **If** $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_{1-i}$ add $\mathsf{epoch\_num}_i + 2, \mathsf{epoch\_num}_i + 3$ to $\mathsf{compromised\_epochs}$.[a]
   (b) Send $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$ to the adversary.
   (c) Upon receiving $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*)$ from $\mathcal{A}$, output $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key}^*)$.
6. **Else**, **if** $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_{1-i}$: //No corruptions or divergence but $\mathsf{epoch\_id}$ doesn't match $\mathsf{epoch\_id}_{1-i}$.
   (a) Sample $\mathsf{recv\_chain\_key}_i \xleftarrow{\$} \mathcal{K}_{ep}$.
   (b) Add $(\mathsf{epoch\_id}, \mathsf{recv\_chain\_key})$ to $\mathsf{receive\_attempts}[\mathsf{epoch\_num}]$.
   (c) Output $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key})$.
7. **Else**, 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:

1. Add $\mathsf{epoch\_id\_self}_i$ to the list $\mathsf{corruptions}_i$.
2. Add $\mathsf{epoch\_num}_i, \mathsf{epoch\_num}_i + 1, \ldots, \mathsf{epoch\_num}_i + 3$ to the list $\mathsf{compromised\_epochs}$.
   //Compromise goes through the following stages: fully compromised for 2-3 epochs, sender randomness updated, both parties' randomness updated.
   //$\mathsf{epoch\_num}_i + 5$ is protected by forward secrecy even in the case of re-corruption.
3. Let j be such that $\mathsf{epoch\_num}_j > \mathsf{epoch\_num}_{1-j}$ and:
   (a) **If** $\mathsf{got\_sending\_key}_j = \mathit{false}$ set $\mathsf{recv\_chain\_key}^* = \perp$. **Else** set $\mathsf{recv\_chain\_key}^* = \mathsf{sending\_chain\_key}_j$.
   (b) **If** $\mathsf{epoch\_num}_j \in \mathsf{receive\_attempts}.keys$ then set $\mathsf{leakage} = \mathsf{receive\_attempts}[\mathsf{epoch\_num}_j]$. **Else** set $\mathsf{leakage} = []$.
4. Send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}^*, \mathsf{leakage})$ to the adversary.
5. Upon receiving $(\texttt{ReportState}, i, S)$ from $\mathcal{A}$, output $(\texttt{Corrupt}, S)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

[a] This provision (which was missing in the [25] version) allows the environment to delay recovery from corruption by delivering messages with bogus epoch ids, this occurs even if these messages do not contain a payload that passes authentication.

Figure 14: The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$ (continued)

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, Π_{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 Π_{SGNL}. Since this functionality corresponds to a single epoch, there is no provision for recovery of security properties after a corruption.

5.3 The Signal Protocol, Π_{SGNL}

Protocol Π_{SGNL} (see Figure 16 on page 43) 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 Π_{SGNL} (which have identical API's as $\mathcal{F}_{\mathsf{SM}}$).

Send Message When receiving a `SendMessage` request from party pid, Π_{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 protocol Π_{SGNL} 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 pid.

Receive Message On receiving a `ReceiveMessage` command with ciphertext c and header $h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N)$ from a party with id pid, the protocol Π_{SGNL} first initializes the

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

This functionality processes encryptions and decryptions for a *single* epoch and has session id $\mathsf{sid}.fs$, where $\mathsf{sid}.fs = (\textit{"fs\_aead"}, \mathsf{sid} = (\mathsf{sid}', (\mathsf{pid}_0, \mathsf{pid}_1)), \mathsf{epoch\_id})$. Inputs arriving from machines other than pid_0, pid_1 are ignored. //For notational simplicity we assume a fixed interpertation of pid_0 and pid_1 as complete identities of the two calling machines.

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

1. If this is the first activation then:
   - Let i be such that $\mathsf{pid} = \mathsf{pid}_i$. Initialize $\mathsf{msg\_num} = 0$ and sender $\mathsf{sender} = i$.
   - Initialize $\mathsf{state}_{\mathcal{I}} = \perp$, call $\mathcal{F}_{\mathsf{lib}}$ to obtain the internal adversarial code $\mathcal{I}$.
2. Verify that 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 $\texttt{IsCorrupt?} = \textit{false}$: Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, N, |m|)$.
6. If $\texttt{IsCorrupt?} = \textit{true}$: Send a backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, m)$ to $\mathcal{A}$.
7. On obtaining the output $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ from $\mathcal{I}$ or $\mathcal{A}$, update $\mathsf{state}_{\mathcal{I}}$ and record $(m, c, \mathsf{msg\_num}, N)$.
8. Output $(\texttt{Encrypt}, c)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

1. Verify that 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 $(\texttt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$, then output $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N, \texttt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
3. If $\texttt{IsCorrupt?} = \textit{false}$:
   - Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ and obtain updated state $\mathsf{state}_{\mathcal{I}}$ and output $(\texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, v)$.
   - If $v = \perp$, then record $(\texttt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$ and output $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N, \texttt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
   - Otherwise, mark $\mathsf{msg\_num}$ as inaccessible and output $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N, m)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
4. Else ($\texttt{IsCorrupt?} = \textit{true}$):
   - Send $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ to $\mathcal{A}$.
   - On receiving $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, c, \mathsf{msg\_num}, N, v)$ from $\mathcal{A}$, update $\mathsf{state}_{\mathcal{I}}$ and do:
     - If $v = \perp$, record $(\texttt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$ and output $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N, \texttt{Fail})$.
     - Else, then mark $\mathsf{msg\_num}$ as inaccessible and output $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N, v)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

1. If 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 pid_i has deleted the ability to encrypt future messages. Output $(\texttt{StopEncrypting}, \texttt{Success})$.

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

1. If 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 pid_i: end the activation.
2. Mark all $\mathsf{msg\_num} > \mathsf{msg\_num}^*$ as inaccessible, and output $(\texttt{StopDecrypting}, \texttt{Success})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

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

Figure 15: The Forward-Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$

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 value epoch_id in header h. On receiving a response from $\mathcal{F}_{\mathsf{fs\_aead}}$, if decryption failed, Π_{SGNL} outputs a failure message. Otherwise, it updates the list of msg_num's that were skipped in the epoch corresponding to the value epoch_id. If the value epoch_id is new (i.e. no messages have been received by this party under this value before) then Π_{SGNL} closes its last sending and receiving epochs (by sending a `StopDecrypting` request to the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance corresponding to $\mathsf{epoch\_id}_{\mathsf{partner}}$, and a `StopEncrypting` request to the $\mathcal{F}_{\mathsf{fs\_aead}}$ instance corresponding to $\mathsf{epoch\_id}_{\mathsf{self}}$.) The protocol Π_{SGNL} then updates $\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}$ and sends $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id})$ to $\mathcal{F}_{\mathsf{eKE}}$ to ratchet forward and receive a new $\mathsf{epoch\_id}^*$ for its next sending epoch. Finally, Π_{SGNL} deletes the decrypted message v returned by $\mathcal{F}_{\mathsf{fs\_aead}}$ and outputs the ciphertext c, message v, and header h to pid.

Corruption The protocol Π_{SGNL} has one additional interface, a `Corrupt` interface that is accessible only to Env. This interface is not part of the real protocol, but is included only for UC-modelling purposes. On a corruption from the environment, Π_{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 Π_{SGNL} who forwards the union of their states up to Env.

5.4 Security Analysis of Π_{SGNL}

In this section we prove Theorem 2 which says that Π_{SGNL}, $\mathcal{F}_{\mathsf{eKE}}$, and $\mathcal{F}_{\mathsf{fs\_aead}}$ together UC-realize $\mathcal{F}_{\mathsf{SM}}$ in the presence of global functionalities $\mathcal{F}_{\mathsf{lib}}$, $\mathcal{F}_{\mathsf{LTM}}$, and $\mathcal{F}_{\mathsf{DIR}}$.

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. We refer readers to Section 2 for a more detailed primer on the universally composable security framework.

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

Proof: To prove Theorem 2, we first construct the simulator $\mathcal{S}_{\mathsf{SM}}$ and the internal code $\mathcal{I}_{\mathsf{SM}}$. Then, we argue that the environment Env has an identically distributed view in its interaction with $\Pi_{\mathsf{SGNL}} + \mathcal{F}_{\mathsf{fs\_aead}} + \mathcal{F}_{\mathsf{eKE}}$ as it does in its interaction with $\mathcal{F}_{\mathsf{SM}} + \mathcal{S}_{\mathsf{SM}} + \mathcal{I}_{\mathsf{SM}}$.

In an interaction between Env and $\mathcal{F}_{\mathsf{SM}}$, the simulator $\mathcal{S}_{\mathsf{SM}}$ and internal adversarial code $\mathcal{I}_{\mathsf{SM}}$ are provided with only the information that $\mathcal{F}_{\mathsf{SM}}$ gives to its ideal process adversary and internal adversarial code. When the session is not compromised, the functionality never calls the simulator $\mathcal{S}_{\mathsf{SM}}$. Instead, the only adversarial choices are made when the functionality runs the internal code $\mathcal{I}_{\mathsf{SM}}$. The objective of $\mathcal{S}_{\mathsf{SM}}$ and $\mathcal{I}_{\mathsf{SM}}$ is to respond in such a way that simulates the artifacts that would be generated if Env were interacting with Π_{SGNL}. The detailed versions of $\mathcal{I}_{\mathsf{SM}}$ and $\mathcal{S}_{\mathsf{SM}}$ can be found in Figure 19 (Page 46) and Figure 17 (Page 44) respectively.

Before arguing that the adversary's view is identical in the two scenarios, we describe where $\mathcal{S}_{\mathsf{SM}}$ and $\mathcal{I}_{\mathsf{SM}}$ are called by $\mathcal{F}_{\mathsf{SM}}$ within each of its methods:

- Within `SendMessage`, as long as initialization has been properly performed then $\mathcal{F}_{\mathsf{SM}}$ will generate tuple $(\texttt{SendMessage}, \mathsf{pid}, \ell)$ and either run the code $\mathcal{I}_{\mathsf{SM}}$ on it or send it to $\mathcal{S}_{\mathsf{SM}}$ based on whether the parties are compromised, diverged, or neither. When the parties are neither compromised nor diverged, $\ell = |m|$ and the code $\mathcal{I}_{\mathsf{SM}}$ is run on $(\texttt{SendMessage}, \mathsf{pid}, \ell)$. Otherwise, $\ell = m$ and the tuple is sent to $\mathcal{S}_{\mathsf{SM}}$.

Π_{SGNL}

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

1. If this is the first activation do: //initialization for the initiator of the session
   - Parse the local session id sid to retrieve the party identifiers $(\mathsf{pid}_0, \mathsf{pid}_1)$ for the initiator and responder. If pid_0 is different from either the local party identifier pid, or the party identifier of 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 $\mathsf{sent\_msg\_num}$ += 1, output $(\texttt{SendMessage}, \mathsf{sid}, h, c)$ to 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 pid:

1. If this is the first activation then do: //initialization for the responder of the session
   - Parse the local session identifier sid to retrieve the party identifiers $(\mathsf{pid}_0, \mathsf{pid}_1)$ for the initiator and responder. If pid_1 is different from either the local party identifier, or the party identifier for 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 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 pid while deleting the decrypted message v.

Corruption: Upon receiving $(\texttt{Corrupt}, \mathsf{pid})$ from 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 $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE = "eKE", \mathsf{sid})$.
2. On receiving $(\texttt{Corrupt}, S_{eKE})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE = "eKE", \mathsf{sid})$, 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 $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs = ("fs\_aead", \mathsf{sid}, \mathsf{epoch\_id}))$.
   - On receiving $S_{\mathsf{epoch\_id}}$, add it to S.
4. Output $(\texttt{Corrupt}, \mathsf{pid}_i, S)$ to Env.

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

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 `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 Env (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$).
   - Upon receiving $(\mathsf{state}, \texttt{GenEpochId}, i, \mathsf{epoch\_id})$ from 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 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 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 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 Env (in the name of $(\mathcal{F}_{\mathsf{fs\_aead}}$, sid.fs) where $\mathsf{sid}.fs = (\textit{"fs\_aead"}, \mathsf{sid}, \mathsf{epoch\_id})$).
5. On receiving $(\mathsf{state}, \texttt{Inject}, v)$ from 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 `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 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 $\mathsf{epoch\_id} = \mathsf{sent\_ids}_i.top$ 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 $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$.

(The rest of this simulator is in Fig. 18 on Page 45)

Figure 17: Secure Messaging Simulator, $\mathcal{S}_{\mathsf{SM}}$

$\mathcal{S}_{\mathsf{SM}}$ continued...

(This simulator begins in Fig. 17 on Page 44)

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 $\mathsf{recv\_chain\_key} \xleftarrow{\$} \mathcal{K}_{ep}$ from the key distribution.
4. Send $(\mathsf{state}.eKE, \texttt{ReportState}, i, \mathsf{recv\_chain\_key})$ to Env (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$).
5. Upon receiving $(\texttt{ReportState}, i, S^*)$ from Env, add S^* to S_i.
6. For all $\mathsf{epoch\_id}^*$ such that there exists a header $h \in \mathsf{pending\_msgs}$ containing $\mathsf{epoch\_id}^*$:
   - Send $(\mathsf{state}.fs.\mathsf{epoch\_id}^*, \texttt{Corrupt}, \mathsf{pid}, \mathsf{leakage})$ to Env (in the name of $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$ where $\mathsf{sid}.fs = (\textit{“fs\_aead”}, \mathsf{sid}, \mathsf{epoch\_id}^*)$.
   - Upon receiving a response $(\texttt{Corrupt}, \mathsf{pid}, S^*)$ from Env, add S^* to the set S_i.
7. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{ReportState}, \mathsf{pid}_i, S_i)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$.

Figure 18: Secure Messaging Simulator $\mathcal{S}_{\mathsf{SM}}$ continued...

- 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 generate a tuple $(\mathsf{state}.fs.\mathsf{epoch\_id}, \texttt{Inject}, \mathsf{pid}, \mathsf{msg\_}$ When the parties are not compromised or diverged, $\mathcal{F}_{\mathsf{SM}}$ runs the code $\mathcal{I}_{\mathsf{SM}}$ on the tuple to decide whether the message is authentic. Otherwise, it sends the tuple to $\mathcal{S}_{\mathsf{SM}}$ to allow it to decide whether the message is authentic or even run a rushing attack to change the message contents. If this is the first successfully received message of a new epoch then $\mathcal{I}_{\mathsf{SM}}/\mathcal{S}_{\mathsf{SM}}$ additionally generates a new $\mathsf{epoch\_id}$ for the recipient party to use when it sends its next message.
- If the corruption of a party is requested by environment, then $\mathcal{F}_{\mathsf{SM}}$ calls the simulator's `ReportState` method to generate a simulated state for the party which it sends to the environment in response.

In the remainder of this proof, we describe why the actions of $\mathcal{S}_{\mathsf{SM}}$ and $\mathcal{I}_{\mathsf{SM}}$ ensure that Env's view in its interaction with $\mathcal{F}_{\mathsf{SM}}$ is identically distributed (when treated as a random variable) to its view when interacting with the real protocol Π_{SGNL}.

This argument is divided into two cases based on whether the session is compromised or not. When the session is not compromised, the proof proceeds via induction over the steps of the internal code $\mathcal{I}_{\mathsf{SM}}$ where we argue that each individual action taken within the internal code maintains the indistinguishability property between the environment's view in the real and ideal worlds. Likewise, when the session is compromised, the proof proceeds by induction over the steps of the simulator. Observing that corruption, uncorruption, and divergence occur at the same times in $\mathcal{F}_{\mathsf{SM}}$ and Π_{SGNL}, on can see that checking the above cases suffices to complete the argument that the environment's view is identical in the real and ideal cases.

Note 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 Π_{SGNL} terminate before ever invoking the ideal-world

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

This internal adversary is only called when no parties are compromised.
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 `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 $\mathcal{I}_{\mathsf{eKE}}(\mathsf{state}.eKE, \texttt{GenEpochId}, i, \perp)$
   - Upon receiving $(\mathsf{state}', \texttt{GenEpochId}, i, \mathsf{epoch\_id})$ from 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 $\mathcal{I}_{\mathsf{fsaead}}(\mathsf{state}.fs.\mathsf{epoch\_id}, \texttt{Encrypt}, \mathsf{pid}, \mathsf{sent\_msg\_num}_i, |m|)$ for specifically $(\mathcal{I}_{\mathsf{fsaead}}, \mathsf{sid}.fs)$, 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 $\mathcal{I}_{\mathsf{fsaead}}$:
   - 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 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 $\mathcal{I}_{\mathsf{fsaead}}(\mathsf{state}.fs.\mathsf{epoch\_id}, \texttt{Inject}, \mathsf{pid}, \mathsf{msg\_num}, c)$ specifically for $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.\text{fs})$ where $\mathsf{sid}.fs = (\textit{"fs\_aead"}, \mathsf{sid}, \mathsf{epoch\_id}))$.
4. On receiving $(\mathsf{state}, \texttt{Inject}, v)$ 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 $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, h, c, \perp)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$. //in this case $\mathcal{F}_{\mathsf{SM}}$ should output `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 $\mathcal{I}_{\mathsf{eKE}}(\mathsf{state}.eKE, \texttt{GenEpochId}, i, \mathsf{epoch\_id})$
   - On receiving $(\mathsf{state}', \texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$, 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 $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, h, c, v)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$.

Figure 19: Internal Adversary $\mathcal{I}_{\mathsf{SM}}$

adversary $\mathcal{S}_{\mathsf{SM}}$ or the real-world adversary $\mathcal{A}$, respectively.

Without loss of generality, we restrict our attention to a deterministic environment Env and a dummy adversary $\mathcal{A}$. Consequently, there are only two sources of randomness in the entire execution: first, the choice of ik and rk within $\mathcal{F}_{\mathsf{DIR}}$, which influences the epoch key generated in $\mathcal{F}_{\mathsf{eKE}}$, and second, the choice of epoch key $\mathsf{recv\_chain\_key}$ within the view of a corrupted receiver.

Case 1: neither party is compromised. The functionality $\mathcal{F}_{\mathsf{SM}}$ notifies the internal code when any message is sent or received, as long as it passes the input validation checks described above. During `ReceiveMessage`, the internal code 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 Π_{SGNL} invokes the specific instance of $\mathcal{F}_{\mathsf{fs\_aead}}$ corresponding to $\mathsf{epoch\_id}$, which then sends a notification to the internal code $\mathcal{I}_{\mathsf{fsaead}}$ asking for the desired message to inject. In the ideal world, the internal code $\mathcal{I}_{\mathsf{SM}}$ sends this exact backdoor message to the internal code $\mathcal{I}_{\mathsf{fsaead}}$ and retrieves a value v. It is straightforward to confirm by inspection that the logic in $\mathcal{I}_{\mathsf{SM}}$ 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 Π_{SGNL} invokes $\mathcal{F}_{\mathsf{eKE}}$ to perform a public ratchet; the internal code $\mathcal{I}_{\mathsf{SM}}$ emulates the corresponding backdoor message that $\mathcal{F}_{\mathsf{eKE}}$ sends to its internal adversarial code $\mathcal{I}_{\mathsf{eKE}}$ to receive the $\mathsf{epoch\_id}$ for the new epoch. Hence, the real and ideal worlds move to a new epoch in lockstep. The internal code 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 Π_{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 internal adversarial code is given the identity of the sending party pid and the length of the desired message $\ell = |m|$. In the ideal world, by the time that $\mathcal{F}_{\mathsf{SM}}$ has invoked $\mathcal{I}_{\mathsf{SM}}$, it has properly initialized `SendMessage` and is using $\mathcal{I}_{\mathsf{SM}}$ to generate a ciphertext so that it can complete the message transmission. Note that $\mathcal{I}_{\mathsf{SM}}$ generates the ciphertext by invoking the code $\mathcal{I}_{\mathsf{fsaead}}$ with state $\mathsf{state}.fs.\mathsf{epoch\_id}$ that corresponds to the newest epochid stored in the state $\mathsf{state}_{\mathcal{I}}$ given to $\mathcal{I}_{\mathsf{SM}}$. For the corresponding call to `SendMessage` in the real world, Π_{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 uses the internal code $\mathcal{I}_{\mathsf{fsaead}}$ directly to generate the ciphertext. Because $\mathcal{I}_{\mathsf{SM}}$ has ℓ and the newest $\mathsf{epoch\_id}$ (from previous calls to `ReceiveMessage`), the views of the environment in both scenarios are the same.

Finally, it is 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 analogously to the way $\mathcal{I}_{\mathsf{SM}}$ does in the case described above with some small differences. Specifically, the simulator $\mathcal{S}_{\mathsf{SM}}$ sends backdoor messages to the environment (addressed to $\mathcal{S}_{\mathsf{fsaead}}$) in the situations where the internal code $\mathcal{I}_{\mathsf{SM}}$ would have run the code $\mathcal{I}_{\mathsf{fsaead}}$, there is a small difference in how the returned value is processed in the case of `Inject`.

The difference between $\mathcal{I}_{\mathsf{SM}}$ and $\mathcal{S}_{\mathsf{SM}}$ in `SendMessage` is the following: While $\mathcal{I}_{\mathsf{SM}}$ invokes $\mathcal{I}_{\mathsf{fsaead}}$ with the length $|m|$, the simulator $\mathcal{S}_{\mathsf{SM}}$ sends a nearly identical backdoor message with the length $|m|$ replaced by message m, this backdoor message is marked for $\mathcal{S}_{\mathsf{fsaead}}$ corresponding to the latest epoch id stored in $\mathsf{state}_{\mathcal{I}}$.

During `Inject`, the difference between $\mathcal{S}_{\mathsf{SM}}$ and $\mathcal{S}_{\mathsf{SM}}$ is in how the returned value from the the environment (resp. $\mathcal{I}_{\mathsf{fsaead}}$) is processed. The simulator $\mathcal{S}_{\mathsf{SM}}$ sends a backdoor message to the environment that's identical to the input that $\mathcal{I}_{\mathsf{SM}}$ runs $\mathcal{I}_{\mathsf{fsaead}}$ with. On receiving a response, $\mathcal{S}_{\mathsf{SM}}$ must compute an input validation predicate that's different than the one in $\mathcal{I}_{\mathsf{SM}}$, because in this case whenever the ciphertext c isn't exactly the one generated in the corresponding `Encrypt` call, both $\mathcal{F}_{\mathsf{SM}}$ and Π_{SGNL} allow the adversary to inject any message of its choice.

To complete the proof, it only remains to check 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 Π_{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 Π_{SGNL} does, with only one exception. Because Π_{SGNL} does not expose to the environment the commands `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 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 Π_{SGNL} uncorrupt a party and mark the parties as diverged at the same times.

□

6 The Public Key Ratchet: Realizing $\mathcal{F}_{\mathsf{eKE}}$

This section describes a protocol Π_{eKE} (Fig. 20) that UC-realizes $\mathcal{F}_{\mathsf{eKE}}$ (Fig. 13). This protocol mirrors the public key ratchet from the Signal protocol – in particular it uses values from a continuous Diffie-Hellman ratchet as inputs to a key derivation function (KDF) chain. The properties of a KDF chain are captured by a new primitive, Cascaded PRF-PRG (CPRFG) (Section 6.2) that is realisable in the plain model and may be of independent interest. In Section 6.3 it is shown that Π_{eKE} (instantiated with a CPRFG) UC-realizes $\mathcal{F}_{\mathsf{eKE}}$ in the plain model (without a random oracle) in the presence of the global functionalities $\mathcal{F}_{\mathsf{DIR}}$, $\mathcal{F}_{\mathsf{lib}}$, and $\mathcal{F}_{\mathsf{LTM}}$.

6.1 Protocol Π_{eKE}

This section describes a protocol Π_{eKE} (Fig. 20) that UC-realizes the epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$ (Fig. 13). The protocol Π_{eKE}, which persists for the entire duration of the secure messaging session between two parties, mirrors the public key ratchet from the Signal protocol. It uses values from a continuous Diffie-Hellman ratchet as inputs to a key derivation function (KDF) chain. The epoch key exchange protocol Π_{eKE} (Section 6.1) has a security tradeoff: (1) When the KDF used in Π_{eKE} is modeled as a random oracle, the protocol realises $\mathcal{F}_{\mathsf{eKE}}$ as is, this allows the overall protocol Π_{SGNL} to realise $\mathcal{F}_{\mathsf{eKE}}$ as is. Allowing an adversary to send malformed packets during corruption means that the adversary can get extra information about the parties secret Diffie-Hellman exponents:

Say that Alice and Bob have secret exponents a and b respectively. Alice tried to share her new epoch identifier g^a with Bob, but Bob has not yet received this information. The adversary can

now send as many malformed epoch identifiers $g^{a'}$ to the Bob as it chooses and get back the value $KDF(g^{a'b})$ for each. Additionally $g^{a'}$ can depend on the honest epoch identifier g^a of Alice.[11]

This is the reason that the adversary to $\mathcal{F}_{\mathsf{eKE}}$ can extend the healing time after corruption by sending malformed packets to parties during the first epoch after recovery. Without step 5a $\mathcal{F}_{\mathsf{eKE}}$ could not be realises by Π_{eKE} in the plain model without specific strong properties being enforced on KDF.[12]

Recall that each of the two parties will run their own copy of epoch key-exchange protocol Π_{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 Π_{eKE} is also initialized on the first call to `ConfirmReceivingEpoch` by the initiator. However, the responder's instance of Π_{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 Π_{eKE} below. The full details can be found in Figure 20 on Page 50.

Each of the parties involved in the protocol execution run an instance of Π_{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 $(\texttt{GetReceivingKey}, \mathsf{epoch\_id})$ 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 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 KDF) which is stored temporarily until the

[11] We note that in the context of the full secure messaging protocol the adversary does not actually see these chain keys. It instead only learns if decryption succeeds or not.

[12] Another option for weakening the functionality is to assume that the adversary is completely inactive during the recovery epoch, or even during the entire healing period as is done in the work by Alwen et al. [1], we prefer instead to characterize the exact additional power gained by the adversary.

Π_{eKE}

This protocol has a party id 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 KDF as subroutines: (1) keyGen chooses a random Diffie-Hellman exponent $\mathsf{epoch\_key} \xleftarrow{\$} |\mathbb{G}|$ for a known group $\mathbb{G}$ and sets $\mathsf{epoch\_id} = g^{\mathsf{epoch\_key}}$. It then outputs $(\mathsf{epoch\_key}, \mathsf{epoch\_id})$. (2) The protocol 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 $(\mathsf{root\_key}_a, \mathsf{root\_key}_c), \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Send $(\texttt{GetInitKeys}, \mathsf{pid}_{1-i}, \mathsf{pid}_i)$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving $(\texttt{GetInitKeys}, \mathsf{ik}_j^{\mathsf{pk}}, \mathsf{rk}_j^{\mathsf{pk}}, \mathsf{ok}_{j\leftarrow i}^{\mathsf{pk}})$, if $\mathsf{ok}_{j\leftarrow i}^{\mathsf{pk}} = \perp$ then output $(\texttt{ConfirmReceivingEpoch}, \texttt{Fail})$. Otherwise, send input $(\texttt{ComputeSendingRootKey}, \mathsf{ik}_j^{\mathsf{pk}}, \mathsf{rk}_j^{\mathsf{pk}}, \mathsf{ok}_{j\leftarrow i}^{\mathsf{pk}})$ to $\mathcal{F}_{\mathsf{LTM}}$.
   - Upon receiving $(\texttt{ComputeSendingRootKey}, s = (s_a, s_c), \mathsf{ek}_i^{\mathsf{pk}})$, set $(\mathsf{root\_key}_a, \mathsf{root\_key}_c) = (s_a, s_c)$.
   - Run the subroutine **Compute Sending Chain Key**.
   - Erase $\mathsf{ek}_i^{\mathsf{pk}}$ and output $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}_{\mathsf{self}} || \mathsf{ek}_i^{\mathsf{pk}} || \mathsf{ok}_{j\leftarrow i}^{\mathsf{pk}})$ 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 $(\mathsf{root\_key}_a, \mathsf{root\_key}_c), \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Parse $\mathsf{epoch\_id} = (\mathsf{epoch\_id}', \mathsf{ek}_j^{\mathsf{pk}}, \mathsf{ok}_{i\leftarrow j}^{\mathsf{pk}})$ and set $\mathsf{temp\_epoch\_id\_partner} = \mathsf{epoch\_id}'$
   - Send $(\texttt{GetResponseKeys}, \mathsf{pid}_{1-i})$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving $(\texttt{GetResponseKeys}, \mathsf{ik}_j^{\mathsf{pk}})$, send input $(\texttt{ComputeReceivingRootKey}, \mathsf{ik}_j^{\mathsf{pk}}, \mathsf{ek}_j^{\mathsf{pk}}, \mathsf{ok}_{i\leftarrow j}^{\mathsf{pk}})$ to $\mathcal{F}_{\mathsf{LTM}}$.
   - Upon receiving $(\texttt{ComputeReceivingRootKey}, s = (s_a, s_c))$, set $(\mathsf{root\_key}_a, \mathsf{root\_key}_c) = (s_a, s_c)$.
4. Run the subroutine **Compute Receiving Chain Key.**
5. Output $(\texttt{GetReceivingKey}, \mathsf{temp\_recv\_chain\_key})$ and erase $\mathsf{temp\_recv\_chain\_key}$.

(The rest of this protocol is in Fig. 21 on Page 52)

Figure 20: The Epoch Key Exchange Protocol Π_{eKE}

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 Π_{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 Π_{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 20 defines the model of corruption we are considering. On corruption, Π_{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.

Remark

We remark that a simple modification to Π_{eKE}, also mentioned in [1], 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 Π_{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 ≥ 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.
Bienstock et al. [13] provide a different modification, which they call the 'triple ratchet', that achieves similarly fast healing from corruption without the additional communication overhead.

6.2 Cascaded PRF-PRG (CPRFG)

A crucial component of the epoch key exchange protocol Π_{eKE} (fully specified later in Fig. 20 on Page 50) is a stateful key derivation function (KDF). The KDF consists of a secret state $s = (s_a, s_c)$, and two deterministic functions, $KDF.\mathbf{Advance}$ and $KDF.\mathbf{Compute}$, which take an values s_a, s_c respectively along with externally provided input r, and do the following:

Π_{eKE} continued...

(This protocol begins in Fig. 20 on Page 50)

Corrupt: On receiving (`Corrupt`) from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$: return $(\mathsf{epoch\_key}, \mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{partner}}, (\mathsf{root\_key}_a, \mathsf{root\_key}_c))$ 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 KDF is used.

Compute Sending Chain Key:

1. If this is the first activation start at step 5.
2. Compute a root input $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathsf{epoch\_key}_{\mathsf{self}})$.
3. Compute the new root key $KDF.\mathbf{Advance}(\mathsf{root\_key}_a, \mathsf{root\_input}) = (\mathsf{root\_key}'_a, \mathsf{root\_key}'_c)$ and update the value $(\mathsf{root\_key}_a, \mathsf{root\_key}_c) = (\mathsf{root\_key}'_a, \mathsf{root\_key}'_c)$.
4. Generate a key pair $(\mathsf{epoch\_key}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{self}}) \leftarrow \mathrm{keyGen}()$.
5. Compute a new root input $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathsf{epoch\_key}_{\mathsf{self}})$.
6. Compute the sending chain key $KDF.\mathbf{Compute}(\mathsf{root\_key}_c, \mathsf{root\_input}) = (\mathsf{root\_key}'_c, \mathsf{sending\_chain\_key})$ and update $\mathsf{root\_key}_c = \mathsf{root\_key}'_c$.
7. 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 $KDF.\mathbf{Compute}(\mathsf{root\_key}_c, \mathsf{root\_input}) = (\mathsf{root\_key}'_c, \mathsf{temp\_recv\_chain\_key})$ and update the value $\mathsf{root\_key}_c = \mathsf{root\_key}'_c$.
3. Erase $\mathsf{root\_input}$. //The old root key is overwritten and therefore already erased.

Figure 21: The Epoch Key Exchange Protocol Π_{eKE} (continued)

- **KDF.Advance($\mathbf{s_a}, \mathbf{r}$):** Update the state $s = (s_a, s'_c)$ according to the input s_a, r to generate an updated state $s^* = (s^*_a, s^*_c)$. (Within the Signal protocol, an application of KDF.**Advance** corresponds to advancing an epoch.)

- **KDF.Compute($\mathbf{s_c}, \mathbf{r}$):** Generate a key k for a potential next epoch corresponding to a given the input s_c, r, along with an updated state s'_c. (Within the Signal protocol, an application of KDF.**Compute** corresponds to computing a receiving chain key k *without advancing the epoch.* Note that we allow an update to the state even in this case; this is a crucial ingredient in providing security against adaptive corruptions.)

In this subsection we define a new primitive, **Cascaded PRF-PRG (CPRFG)**, that captures the security requirements from this KDF. Additionally, we present a candidate construction of a **CPRFG** in the standard model. (The HKDF function [44] used in the Signal application also meets this definition of a **CPRFG**, albeit in the random oracle model.)

6.2.1 Defining the CPRFG

The definition of a **Cascaded PRF-PRG (CPRFG)** extends the PRF-PRNG model from Alwen et al. [1] to capture the requirements from a KDF-chain under adaptive corruptions. The benefit of the CPRFG definition is that it directly requires the exact adaptive properties needed from a KDF-chain as specified in the Signal architecture [51] while the PRF-PRNG definition from [1] has to be stringent enough to provide the right adaptive guarantees when used in a chain. This definition follows the real/ideal paradigm. Specifically, it specifies an ideal game and a real game, and requires existence of a simulator $\mathcal{S}_{\mathsf{KDF}}$ such that no adversary can distinguish whether it is interacting in a real game that represents an execution of an actual KDF, or in an ideal game with simulator $\mathcal{S}_{\mathsf{KDF}}$, where the ideal game represents the expected behavior of the system that exhibits the properties described above.

In both games, the adversary can repeatedly make one of four queries: $(\mathsf{Compute}, r)$, $(\mathsf{Advance}, r)$, $(\mathsf{Compromise}, r)$, or $(\mathsf{Recover})$ which are answered as defined in Figure 22 below.

We note that, while the definition does follow the real/ideal paradigm, we found it more convenient not to frame it within the UC framework. In particular, this allows us to directly require that KDF.**Compute** and KDF.**Advance** be deterministic functions, and to make the queries in the game different than the normal API of KDF.**Compute** and KDF.**Advance**.

A formal definition follows in Theorem 11 with the full game details shown in Fig. 22.

Definition 11 (Cascaded PRF-PRG (CPRFG)) *Let m be a polynomial, $\mathcal{O}_{\mathsf{CPRFG}}$ be an oracle that runs either the real or ideal version of the CPRFG security game (Figure 22, pg Page 54), $\lambda \in \mathbb{N}$ be the security parameter, and $|R_\lambda| \geq 2^\lambda$ be an input space. Let the module KDF have a secret state $(s_a, s_c) \in \{0,1\}^\lambda$ and deterministic functions $(KDF.\textbf{\textit{Compute}}, KDF.\textbf{\textit{Advance}})$ such that: $KDF.\textbf{\textit{Advance}}$ takes inputs $s_a, r \in \{0,1\}^\lambda \times R_\lambda$ and produces an output $(s^*_a, s^*_c) \in \{0,1\}^\lambda \times \{0,1\}^\lambda)$. $KDF.\textbf{\textit{Compute}}$ takes inputs $s_c, r \in \{0,1\}^\lambda \times R_\lambda$ and produces an output $s'_c, k \in \{0,1\}^\lambda \times \{0,1\}^{m(\lambda)}$.*

*KDF is a **cascaded PRF-PRG (CPRFG)** if $\exists$ probabilistic polynomial time (PPT) algorithm $\mathcal{S}_{\mathsf{KDF}}$ such that any PPT machine $\mathcal{A}$ interacting with oracle $\mathcal{O}_{\mathsf{CPRFG}}$ can distinguish whether the oracle is running the real game or the ideal game with advantage at most negligible in λ.*

6.2.2 Properties of the CPRFG

To provide intuition for this new primitive, in this section we provide an informal discussion of some of the properties that it guarantees.

Cascaded PRF-PRG Security Game

Let m be a polynomial, $\lambda \in \mathbb{N}$ a security parameter. This figure describes the Cascaded PRF-PRG security game for a KDF with secret state $(s_a, s_c) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ and deterministic functions $(KDF.\textbf{Compute}, KDF.\textbf{Advance})$ such that: $KDF.\textbf{Advance}$ takes inputs $s_a, r \in \{0,1\}^\lambda \times R_\lambda$ and produces an output $(s_a^*, s_c^*) \in \{0,1\}^\lambda \times \{0,1\}^\lambda)$. $KDF.\textbf{Compute}$ takes inputs $s_c, r \in \{0,1\}^\lambda \times R_\lambda$ and produces an output $s_c', k \in \{0,1\}^\lambda \times \{0,1\}^{m(\lambda)}$.

Real game:

- On $\mathsf{Initialization}$: Oracle $\mathcal{O}_{\mathsf{CPRFG}}$ is initialized with uniformly sampled state $s = (s_a, s_c) \xleftarrow{\$} \{0,1\}^\lambda$.
- On input $(\mathsf{Compute}, r)$: $\mathcal{O}_{\mathsf{CPRFG}}$ runs $KDF.\textbf{Compute}(s_c, r) = (s_c', k)$, outputs k, changes state $s = s'$.
- On input $(\mathsf{Advance}, r)$: $\mathcal{O}_{\mathsf{CPRFG}}$ runs $KDF.\textbf{Advance}(s_a, r) = (s_a^*, s_c^*)$, and changes state $(s_a, s_c) = (s_a^*, s_c^*)$.
- On input $(\mathsf{Compromise}, r^*)$: $\mathcal{O}_{\mathsf{CPRFG}}$ outputs the old state s. Mark the state as compromised.
- On input $(\mathsf{Recover})$: $\mathcal{O}_{\mathsf{CPRFG}}$ samples $r' \xleftarrow{\$} R_\lambda$ uniformly at random, computes $KDF.\textbf{Compute}(s_c, r') = s_c', k$, and $KDF.\textbf{Advance}(s_a, r') = (s_a^*, s_c^*)$, and changes state $s = (s_a^*, s_c^*)$. Mark the state as recovered and output k.

Ideal game:

- On $\mathsf{Initialization}$: Oracle $\mathcal{O}_{\mathsf{CPRFG}}$ is initialized with a uniformly sampled state $F \xleftarrow{\$} \{f : R_\lambda \to \{0,1\}^{m(\lambda)}\}$.
- On input $(\mathsf{Compute}, r)$:
  1. **If** the state is compromised it outputs $\mathcal{S}(\mathsf{Compute}, r) = k$ and adds the tuple (r, k) to the list $\mathsf{computed\_values}$.
  2. **Else** (the state is not compromised), it outputs $F(r) = k$ and adds the tuple (r, k) to the list $\mathsf{computed\_values}$.
- On input $(\mathsf{Advance}, r)$:
  1. The oracle $\mathcal{O}_{\mathsf{CPRFG}}$ resets $\mathsf{computed\_values} = \{\}$.
  2. **If** the state isn't compromised, $\mathcal{O}_{\mathsf{CPRFG}}$ updates its state to a new function $F \xleftarrow{\$} \{f : R_\lambda \to \{0,1\}^{m(\lambda)}\}$ sampled uniformly at random.
  3. **Else** (the state is compromised), $\mathcal{O}_{\mathsf{CPRFG}}$ runs $\mathcal{S}(\mathsf{Advance}, r)$.
- On input $(\mathsf{Compromise}, r^*)$: **If** the state is not compromised then $\mathcal{O}_{\mathsf{CPRFG}}$ marks the state as compromised and outputs $\mathcal{S}(\mathsf{computed\_values}, (r^*, F(r^*))$ (or $\mathcal{S}(\mathsf{computed\_values}, \perp)$ if $r^* = \perp$), where $\mathsf{computed\_values}$ correspond to all the $\mathsf{Compute}$ queries made by $\mathcal{A}$ since the last $\mathsf{Advance}$ or $\mathsf{Recover}$ query and F is the current state. //If the state is compromised, do nothing.
- On input $(\mathsf{Recover})$: $\mathcal{O}_{\mathsf{CPRFG}}$ marks the state as recovered. It samples $k \xleftarrow{\$} \{0,1\}^\lambda$ uniformly at random and updates its state to a new function $F \xleftarrow{\$} \{f : R_\lambda \to \{0,1\}^{m(\lambda)}\}$ sampled uniformly at random. It runs $\mathcal{S}(\mathsf{Recover})$. Reset $\mathsf{computed\_values} \leftarrow []$.

Figure 22: Cascaded PRF-PRG Security Game

Property 1: As long as the initial state $s = (s_a, s_c)$ is random and secret, any sequence of calls to KDF.**Compute** that does not contain a call to KDF.**Advance** should behave like a sequence of calls to a random function. That is, for each new, adversarially chosen value of r, the key k generated by running KDF.**Compute**(s_c, r) should look like a fresh random key. Furthermore, two calls with the same input r should result with the same k. That is, regardless of the sequence of KDF.**Compute** queries run before the call with input r, the output key k should be the same.

(This property is intended to ensure that even though an adversary for Π_{eKE} may cause a receiver to compute epoch keys using r's that don't match the one used by the sender, the adversary cannot actually guess what any of the resulting epoch keys will be. Furthermore, the consistency of the outputs to KDF.**Compute** queries ensures that two parties that have the same initial state (s_a, s_c) and each make a different sequence of KDF.**Compute** queries, will get the same output key k for any query r contained in both sequences.)

This property holds for the following reasons: (1) Compute queries in the ideal game (from Figure 22), are answered via a function F chosen uniformly at random as long as the state is not compromised. (2) The outputs of the randomly chosen function F on inputs r are fixed regardless of the order of Compute queries (till an Advance occurs and a new random function is chosen). (3) Finally, when a call to (Compute, r) is answered with a key k, the tuple (r, k) is stored till the next Advance query to make sure all calls to (Compute, r) till then are answered with the same key k.

Property 2: As long as either s_a or r is both random and secret, running KDF.**Advance**(s_a, r) updates the state to a new pseudorandom state (s_a^*, s_c^*). This in turn causes KDF.**Compute** to behave like a fresh random function. Furthermore, a party that starts with some initial state $s = (s_a, s_c)$ and runs some sequence of KDF.**Compute** queries that lead to a state $s' = (s_a, s_c')$, will always get the same resulting state s^* when it queries KDF.**Advance** for some input r, regardless of the sequence of KDF.**Compute** queries that it made before advancing to lead to the state s'.

(This property provides forward secrecy of any keys computed before the last time KDF.**Advance** was run. It also allows parties who can agree on a fresh random and secret r to recover from the corruption. Additionally, the consistency property ensures that even though an adversary for Π_{eKE} can cause parties to compute keys for several incorrect randomizer values, the adversary still can't make the parties go out of sync without corrupting them.)

In the case where the state is not compromised, the ideal game samples a fresh random function on receiving (Advance, root_input). On the other hand, the case where the state is compromised running KDF.**Advance**(s_a, r') with a secret random value r' is indistinguishable from updating the state to a newly chosen random function because the actions for (Recover) in the real game look indistinguishable from those for (Recover) in the ideal game. Finally, consistency of the resulting state s^* with respect to the input r holds because KDF.**Advance** is deterministic with respect to its inputs (s_a, r) and queries to KDF.**Compute** don't change s_a.

Property 3: To demonstrate the security properties of a CPRFG against break-ins we would like to show that: (1) Exposing the current state of the KDF does not expose any keys generated before the last Advance. (2) At any point during the execution, the current local state of the system is simulatable given only the inputs and outputs of KDF.**Compute** since the last KDF.**Advance**.

(This guarantees that an adversary does not obtain any computational advantage over what is allowed in an ideal execution where the generated keys are truly random.)

This property is modeled in the ideal (Compromise, root_input*) operation in which the simulator is given only the inputs and outputs since the last advance and produces a state $s = (s_a, s_c)$ that is indistinguishable from that seen during compromise in the real world.

6.2.3 Constructing the CPRFG

In this section, we provide a construction for the CPRFG in the standard model. The construction requires a puncturable pseudorandom function [15] along with standard crypto primitives.

CPRFG Construction

The construction is parameterized by a puncturable PRF $\{f\}$, PRF g, and length-tripling PRG h.
$KDF.\textbf{Compute}(s_c, r)$:

1. Parse $s_c = (s_1, s_2)$, and further parse $s_1 = (\widehat{s}_1, (a_1, b_1), ..., (a_k, b_k))$
2. If $r = a_i$ for some i: set $\widehat{f}_{s_1}(\mathsf{root\_input}) = b_i$ and $s'_c = s_c$.
3. Else:
   (a) $\widehat{s}'_1 \leftarrow Puncture(\widehat{s}_1, r)$
   (b) Update $s'_1 = (\widehat{s}'_1, (a_1, b_1), ..., (a_k, b_k), (r, f_{\widehat{s}_1}(r))$
   (c) Compute $\widehat{f}_{s_1}(r) \leftarrow f_{\widehat{s}_1}(r)$
4. Output $(s'_c, \widehat{f}_{s_1}(r) \oplus s_2)$

$KDF.\textbf{Advance}(s_a, r)$:

1. Output $(s_a, s_c) \leftarrow h(g_{s_a}(r))$

Figure 23: CPRFG Construction

Theorem 12 *Assume that $\{f\}$ is a puncturable PRF, $\{g\}$ is a PRP, and h is a PRG. Then the KDF construction in Figure 23 is a cascaded PRF-PRG.*

Cascaded PRF-PRG Simulator

On input $(\mathsf{Compute}, r)$ run the honest $KDF.\textbf{Compute}(s_c, r)$ from Figure 23 and update s_c accordingly.
On input $(\mathsf{Advance}, r)$ run the honest $KDF.\textbf{Advance}(s_a, r)$ from Figure 23 and update $s = (s_a, s_c)$ accordingly.
On input $(\mathsf{computed\_values}, (r, k))$ or $(\mathsf{computed\_values}, \perp)$:

- Sample $s_1, s_a \xleftarrow{\$} \{0,1\}^\lambda$ independently and uniformly at random.
- Parse $\mathsf{computed\_values} = ((a_1, b_1), \ldots, (a_k, b_k))$.
- Let $\widehat{s_1}$ be the result of puncturing s_1 at points $a_1, \ldots, a_k$.
- Set $s'_1 = (\widehat{s}_1, (a_1, b_1), \ldots, (a_k, b_k))$.
- If $(r, k) \neq \perp$, set $s_2 = k \oplus \hat{f}_{s'_1}(r)$.
- Finally, set $s_c = (s'_1, s_2)$ and output $s = (s_a, s_c)$.

On input $(\mathsf{Recover})$: delete state s.

Figure 24: Cascaded PRF-PRG Simulator

Proof: Consider the simulator $\mathcal{S}_{\mathsf{KDF}}$ in Figure 24 (on Page 56). 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 (as in [15]), or the fact that $\{g\}$ is a PRF, or the PRG property of h.

The proof uses the following hybrid argument. Let the hybrid experiment H_i^1 consist of the ideal CPRFG game up to and including the i-th instance of $(\mathsf{Advance}, \cdot)$ or $(\mathsf{Recover})$ in the CPRFG

game; all following calls to the game are real. Next, let H_i^2 consist of the ideal CPRFG game up to the i-th instance of $(\mathsf{Advance}, \cdot)$ or $(\mathsf{Recover})$ *and* the subsequent calls to the CPRFG game before the next advance; only then do all subsequent calls act as in the real game. Specifically, if epoch i was not corrupted, the $(i+1)$-th $(\mathsf{Advance}, \cdot)$ will use a random $\mathsf{root\_key}_3$ to key g.

We will begin with the first case. Suppose there is some $\mathcal{A}$ that can distinguish H_i^1 from H_i^2 (which differ only in whether the operations computed during the i-th epoch are real or ideal). Then, clearly epoch i is not compromised because the simulator ensures that the two hybrids are identical (the simulation is perfect). Then, construct the following adversary $\mathcal{A}_{PPRF}$ against the puncturable PRF property of f:

1. Emulate the ideal CPRFG game to $\mathcal{A}$ up to and including the i-th $\mathsf{Advance}$.
2. Until the next $\mathsf{Advance}$ or $\mathsf{Compromise}$:
   (a) Initialize a list of computed values $\mathsf{computed\_values}$
   (b) Let $\mathcal{K}$ be the set of possible values for r.
   (c) For inputs of the form $(\mathsf{Compute}, r)$, for some r from $\mathcal{A}$:
   Query
   $$k \leftarrow PPRF.Challenge(r)$$
   and return k; Also, set $\mathcal{K} = \mathcal{K} \setminus \{r\}$ to puncture this input.
3. If we exited Step 2 with a $(\mathsf{Compromise}, r^*)$ for some r^*, query $s_1 \leftarrow PPRF.Constrain(\mathcal{K} \cup \{r^*\})$, sample s_2' and s_a uniformly and set $s_2 = PPRF.Challenge(r^*) \oplus s_2'$. Output $s = ((s_1, s_2), s_a)$.
4. If we exited Step 2 with a $(\mathsf{Advance}, r)$ for some r, sample $(s_a, s_c) \xleftarrow{\$} \{0,1\}^\lambda$ uniformly.
5. Emulate the real CPRFG game with the state initialized to s for the remaining calls from $\mathcal{A}$.
6. If $\mathcal{A}$ outputs "H_i^1", then output "pseudorandom function". Otherwise ($\mathcal{A}$ outputs H_i^2), output "random function".

Notice that if the function in the PPRF game is pseudorandom, then hybrid H_i^1 is exactly emulated to $\mathcal{A}$, because the PPRF key is truly random; otherwise, H_i^2 is perfectly emulated to $\mathcal{A}$. Thus, $\mathcal{A}$'s distinguishing probability is identical to $\mathcal{A}_{PRF\_PRG}$'s.

Now, consider the case that $\mathcal{A}$ can distinguish between hybrids H_{i-1}^2 and H_i^1 (which differ only on whether the i-th $\mathsf{Advance}$ is real or ideal). Clearly then epoch i is not corrupt because the simulator ensures that the two hybrids are identical (the simulation is perfect) Then, we will construct the following PRF-PRG adversary $\mathcal{A}_{PRF\_PRG}$:

1. Emulate the ideal CPRFG game to $\mathcal{A}$ up to the i-th $\mathsf{Advance}$.
2. On the i-th $(\mathsf{Advance}, \cdot)$ or $\mathsf{Recover}$:
   (a) If $(\mathsf{Advance}, r)$ for some r: $(s_a, s_c) \leftarrow h(PRF.Challenge(r))$.
   (b) Else $(\mathsf{Recover})$: the PRF key k is known to $\mathcal{A}$, but the r is uniformly random. Set $k \leftarrow PRG.Challenge()$.
3. For the remaining calls from $\mathcal{A}$, emulate the real CPRFG game with state initialized to $s = (s_a, s_c)$.

4. If $\mathcal{A}$ outputs "H^2_{i-1}", then output "pseudorandom". Otherwise ($\mathcal{A}$ outputs "H^1_i"), output "truly random".

Notice that if the functions in the PRF or PRG games are pseudorandom, then we have perfectly emulated hybrid H^2_{i-1} to $\mathcal{A}$; otherwise, if the functions in the PRF or PRF games are random, we have perfectly emulated hybrid H^1_i. Thus, the distinguishing advantage of $\mathcal{A}$ is the same as that of $\mathcal{A}_{PRF\_PRG}$.

Lastly, since the CPRFG adversary $\mathcal{A}$ runs in polynomial time in the security parameter, there can be at most a polynomial number of Advance and Recover 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 KDF.**Compute** between two consecutive applications of KDF.**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 [50]. 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.

6.3 Security Analysis of Π_{eKE}

Proof: The proof of Theorem 3 proceeds in two parts. The first part is the construction of the simulator $\mathcal{S}_{\mathsf{eKE}}$ and the internal code $\mathcal{I}_{\mathsf{eKE}}$. The second part of the proof is a hybrid argument used to show the following indistinguishability (for the previously constructed $\mathcal{S}_{\mathsf{eKE}}, \mathcal{I}_{\mathsf{eKE}}$):

$$(\mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \boldsymbol{\mathcal{F}}^*_{\mathbf{eKE}}, \boldsymbol{\mathcal{S}}_{\mathbf{eKE}}, \boldsymbol{\mathcal{I}}_{\mathbf{eKE}}) \approx (\mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \boldsymbol{\Pi}_{\mathbf{eKE}}).$$

When neither party has ever been corrupted, the functionality never calls the simulator. Instead, the only adversarial choices are made when the functionality runs the internal code $\mathcal{I}_{\mathsf{eKE}}$ to receive new epoch ids. The internal adversarial code for this proof simply samples (epoch_key, epoch_id) pairs uniformly at random for each new epoch. The detailed version of $\mathcal{I}_{\mathsf{eKE}}$ can be found in Fig. 25 on Page 59.

Once a party has been corrupted, the functionality will begin to call the simulator $\mathcal{S}_{\mathsf{eKE}}$. (The detailed simulator $\mathcal{S}_{\mathsf{eKE}}$ can be found in Figs. 26 to 27 on Page 60.) The simulator is first called by the functionality at the time of corruption. At this time, the simulator must return a simulated state state for the corrupted party which will be passed to the environment by the functionality. The receipt of party state state simulates a passive cloning attack. Then, during the healing period when the parties are still compromised from the state corruption, the simulator must perform two actions: (1) provide sending_chain_keys and recv_chain_keys for the parties to $\mathcal{F}_{\mathsf{eKE}}$ until compromise ends (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 Π_{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

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

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

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

1. If this is the first activation:
   - Set $\mathsf{epoch\_num}_0 = 0$, $\mathsf{epoch\_num}_1 = -1$ in $\mathsf{state}_{\mathcal{I}}$.
2. Else (this is not the first activation):
   - Parse $\mathsf{state}_{\mathcal{I}}$ to recover the pair $(\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i)$.
   - Store $\mathsf{old\_epoch\_id}_i = \mathsf{epoch\_id}_i$ and $\mathsf{old\_epoch\_key}_i = \mathsf{epoch\_key}_i$ in $\mathsf{state}_{\mathcal{I}}$.
3. Sample a new pair $(\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i) \xleftarrow{\$}$ keyGen() and record this pair in $\mathsf{state}_{\mathcal{I}}$.
4. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch\_id}_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

Figure 25: Internal adversarial code $\mathcal{I}_{\mathsf{eKE}}$

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 Π_{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 $(\texttt{ReportState}, i, \mathsf{recv\_chain\_key})$ to the simulator. The simulator must return 'the state of 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 Π_{eKE} are $\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{1-i}, s = s_a||s_c$.

Note that $\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i$, and $\mathsf{epoch\_id}_{1-i}$ were chosen by the adversarial code $\mathcal{I}_{\mathsf{eKE}}$ just like they would be in the real protocol and can be provided as-is. So, the only remaining question is what value $s = s_a||s_c$) should the simulator 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 $s_a||s_c$ produced here only needs to take account past chain keys. Fortunately, because KDF is a cascaded PRF-PRG, the current $s_a||s_c$ in the state of a party will be unrelated to all previous keys provided by $\mathcal{F}_{\mathsf{eKE}}$ in most cases and can simply be chosen at random. The only case where s is related to a previous output of $\mathcal{F}_{\mathsf{eKE}}$ is when the instance of protocol Π_{SGNL} for 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 party's instance $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$. In this case, the provided $(s_a||s_c)$ must satisfy $KDF.\mathbf{Compute}(s_c, \mathsf{epoch\_id}_{1-i}^{\mathsf{epoch\_key}_i}) = -, \mathsf{recv\_chain\_key}$. In these cases only, the functionality will provide $\mathsf{recv\_chain\_key}$ to the simulator at the time of making the request. On receiving a request containing $\mathsf{recv\_chain\_key}$, the simulator will invoke the KDF simulator $\mathcal{S}_{\mathsf{KDF}}$ that proves the security of KDF; since the KDF protocol is a secure CPRFG, such a simulator $\mathcal{S}_{\mathsf{KDF}}$ must exist.

After pid_i is corrupted, the simulator will be able to provide all the keys for the parties by running the instructions for Π_{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 will end the execution of the

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

$\mathcal{S}_{\mathsf{eKE}}$ runs only if the parties are diverged, or if the current epoch is compromised (i.e. we're within the quarantine period for a corruption.)

The keyGen(·) and KDF components are the same ones from Π_{eKE}: (1) keyGen chooses a random Diffie-Hellman pair $(\mathsf{epoch\_key}, \mathsf{epoch\_id})$. (2) The protocol KDF is a CPRFG (Cascaded PRF-PRG).

At first activation: Send $(\mathcal{F}_{\mathsf{eKE}}, \mathcal{I}_{\mathsf{eKE}})$ 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:
   - Output $(\texttt{GetSendingKey}, i, \mathcal{V}iew_i.\mathsf{sending\_chain\_key})$ and delete $\mathcal{V}iew_i.\mathsf{sending\_chain\_key}$.
2. 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. If $\mathsf{diverge\_parties} = \mathit{false}$:
   - If $\mathsf{epoch\_id} \neq \mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{self}}$: Add $\mathsf{epoch\_num}_{1-i}+1$, $\mathsf{epoch\_num}_{1-i}+2$ to $\mathsf{compromised\_epochs}$.
   - Run the KDF simulator $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compute}, \mathcal{V}iew_i.\mathsf{root\_input})$ and set the output as $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key}$.
4. Else ($\mathsf{diverge\_parties} = \mathit{true}$):
   - Set $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key} = KDF.\mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
5. Erase $\mathcal{V}iew_i.\mathsf{root\_input}$ and $\mathcal{V}iew_i.\mathsf{temp\_epoch\_id}_{\mathsf{partner}}$.
6. 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 $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, \mathsf{pid}_i, \mathsf{recv\_chain\_key}^*, \mathsf{leakage})$ 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.
//$\mathsf{recv\_chain\_key}^* = \perp$ if and only if $\mathsf{epoch\_num}_{1-i} < \mathsf{epoch\_num}_i$.

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. Create two dictionaries $\mathcal{V}iew_0, \mathcal{V}iew_1$.
   //This and following steps will run only if the parties weren't compromised when `ReportState` was called.
4. From $\mathsf{state}_{\mathcal{I}}$ get the variables $\mathsf{epoch\_id}_0, \mathsf{epoch\_id}_1, \mathsf{epoch\_key}_0, \mathsf{epoch\_key}_1, \mathsf{old\_epoch\_id}_0, \mathsf{old\_epoch\_id}_1$.
5. Set the following values in the view objects $\mathcal{V}iew_0, \mathcal{V}iew_1$:
   (a) Set $\mathcal{V}iew_0.\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}_0$, $\mathcal{V}iew_0.\mathsf{epoch\_key}_{\mathsf{self}} = \mathsf{epoch\_key}_0$.
   (b) Set $\mathcal{V}iew_1.\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}_1$, $\mathcal{V}iew_1.\mathsf{epoch\_key}_{\mathsf{self}} = \mathsf{epoch\_key}_1$.
   (c) Let $j \in \{0,1\}$ be such that pid_j started the latest sending epoch. (i.e $\mathsf{epoch\_num}_j > \mathsf{epoch\_num}_{1-j}$.)
      i. Set $\mathcal{V}iew_j.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}_{1-j}$.
      ii. Set $\mathcal{V}iew_{1-j}.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{old\_epoch\_id}_j$.
      iii. Let $\mathsf{root\_input}^* = \mathsf{Exp}(\mathsf{epoch\_id}_j, \mathsf{epoch\_key}_{1-j})$.
      iv. Run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compromise}, \mathsf{leakage}, (\mathsf{root\_input}^*, \mathsf{recv\_chain\_key}^*))$ to get an output $\mathsf{root\_key}$.
      v. Set $\mathcal{V}iew_{1-j}.\mathsf{root\_key} = \mathsf{root\_key}$.
      vi. If $\mathsf{recv\_chain\_key}^* \neq \perp$, set $\mathcal{V}iew_j.\mathsf{sending\_chain\_key} = \mathsf{recv\_chain\_key}^*$.
      vii. Set $\mathcal{V}iew_j.\mathsf{root\_key} = KDF.\mathbf{Advance}(\mathsf{root\_key}, \mathsf{root\_input}^*)$. //Only the receiver tells $\mathcal{S}_{\mathsf{KDF}}$ to advance. Compute sender's root key locally.
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 $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

(The rest of this simulator is in Fig. 27 on Page 61)

Figure 26: Simulator $\mathcal{S}_{\mathsf{eKE}}$ for realizing $\mathcal{F}_{\mathsf{eKE}}$

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

(This simulator begins in Fig. 26 on Page 60)

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

1. Set $\mathsf{epoch\_num}_i$ += 2.
2. Set $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}^*$.
3. Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}^*, \mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}})$.
4. If $\mathsf{diverge\_parties} = \mathit{false}$:
   (a) Run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Advance}, \mathsf{root\_input})$. //The receiver will tell $\mathcal{S}_{\mathsf{KDF}}$ to advance its epoch.
   //Next, we check if the states have diverged.
   (b) Compute $s = KDF.\mathbf{Advance}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathsf{root\_input})$ and set $\mathcal{V}iew_i.\mathsf{root\_key} = s$.
   (c) If $\mathcal{V}iew_i.\mathsf{root\_key} \neq \mathcal{V}iew_{1-i}.\mathsf{root\_key}$ then set $\mathsf{diverge\_parties} = \mathit{true}$. //divergence occurs here
5. Else ($\mathsf{diverge\_parties} = \mathit{true}$):
   - Set $\mathcal{V}iew_i.\mathsf{root\_key} = KDF.\mathbf{Advance}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathsf{root\_input})$.
6. If $\mathsf{epoch\_num}_i \notin \mathsf{compromised\_epochs}$ and $\mathsf{diverge\_parties} \neq \mathit{true}$):
   //If the views are in sync and $\mathsf{epoch\_num}_i + 1$ is the first uncompromised epoch after corruption.
   //Delete the view objects and generate a new epoch id.
   (a) Sample $(\mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}}, \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}}) \xleftarrow{\$} \mathrm{keyGen}()$.
   (b) Let $j \in \{0,1\}$ be such that pid_j started the latest sending epoch. (i.e $\mathsf{epoch\_num}_j > \mathsf{epoch\_num}_{1-j}$.)
   (c) Set the following values in $\mathsf{state}_{\mathcal{I}}$:
      i. $\mathsf{epoch\_id}_j = \mathcal{V}iew_j.\mathsf{epoch\_id}_{\mathsf{self}}$ and $\mathsf{epoch\_key}_j = \mathcal{V}iew_j.\mathsf{epoch\_key}_{\mathsf{self}}$.
      ii. $\mathsf{epoch\_id}_{1-j} = \mathcal{V}iew_{1-j}.\mathsf{epoch\_id}_{\mathsf{self}}$ and $\mathsf{epoch\_key}_{1-j} = \mathcal{V}iew_{1-j}.\mathsf{epoch\_key}_{\mathsf{self}}$.
      iii. $\mathsf{old\_epoch\_id}_j = \mathcal{V}iew_{1-j}.\mathsf{epoch\_id}_{\mathsf{partner}}$ and $\mathsf{old\_epoch\_id}_{1-j} = \perp$.
   (d) Delete $\mathcal{V}iew_0, \mathcal{V}iew_1$ and output $(\texttt{GenEpochId}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{epoch\_id}_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
7. Otherwise ($\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$ or $\mathsf{diverge\_parties} = \mathit{true}$):
   //If the parties are still compromised or the views have diverged.
   //Continue to update $\mathcal{V}iew_i$ according to Π_{eKE}.
   (a) Sample $(\mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}}, \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}}) \xleftarrow{\$} \mathrm{keyGen}()$.
   (b) Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}^*, \mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}})$.
   (c) If $\mathsf{diverge\_parties} = \mathit{false}$, run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compute}, \mathsf{root\_input})$ and set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key}$ to be its output.
   (d) Else ($\mathsf{diverge\_parties} = \mathit{true}$), set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key} = KDF.\mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathsf{root\_input})$.
   (e) Set $\mathcal{V}iew_i.\mathsf{root\_key} = KDF.\mathbf{Advance}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathsf{root\_input})$.
   (f) Erase $\mathsf{root\_input}$.
   (g) Output $(\texttt{GenEpochId}, i, \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

Figure 27: Simulator $\mathcal{S}_{\mathsf{eKE}}$ (continued)

dummy parties. However, in the case of divergence, it must continue to run the dummy parties to provide chain keys.

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

To demonstrate the validity of $\mathcal{S}_{\mathsf{eKE}}, \mathcal{I}_{\mathsf{eKE}}$, One must prove that that the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ with access to the simulator $\mathcal{S}_{\mathsf{eKE}}$ and internal code $\mathcal{I}_{\mathsf{eKE}}$ is indistinguishable from the real world. This indistinguishability is proven via the following hybrid executions $H_0, \ldots, H_5$ that bridge the gap between a real-world execution H_0 (namely an execution with $\Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$) and an ideal execution H_5 (namely an execution with $\mathcal{F}_{\mathsf{eKE}}, \mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$). The full hybrid argument follows this brief description of each hybrid:

- **Hybrid H_0 (Ideal World):** This is the ideal world. Here you can think of $\mathcal{S}_{\mathsf{eKE}}+\mathcal{I}_{\mathsf{eKE}}+\mathcal{F}_{\mathsf{eKE}}$ all together as one combined unit with combined state.

- **Hybrid H_1:** In this hybrid the main change is to the method of choosing chain keys *when both parties are uncompromised*. In hybrid H_0 these keys are chosen uniformly at random from $\mathcal{K}_{ep}$ while in this hybrid they're chosen via a function $F : \mathbb{G} \to \mathcal{K}_{ep}$ chosen uniformly at random for the corresponding epoch. the inputs to the random function are the values $\mathsf{root\_input}$ chosen uniformly at random from the group $\mathbb{G}$(as in the CPRFG game). The adversary's view in this hybrid is identically distributed as in the previous hybrid.

- **Hybrid H_2:** This hybrid again changes the method of choosing chain keys *when both parties are uncompromised*. In this case, H_2 transitions from using a version of the ideal CPRFG game to using the protocol KDF. In particular, the KDF.**Compute** and KDF.**Advance** methods of the KDF are now used to compute the chain keys and update the value $s = s_a||s_c$ respectively instead of choosing a new random function for each epoch. The root inputs and initial root key are chosen identically to the previous hybrid. This hybrid is indistinguishable to H_1 since KDF is a secure CascadedPRF-PRNG.

- **Hybrid H_3:** This hybrid again changes the method of choosing chain keys *when both parties are uncompromised*. In particular, we replace the random choices of $\mathsf{root\_inputs}$ with the Diffie-Hellman keys computed as specified in the protocol Π_{eKE} over a series of hybrids. Let $M = \mathsf{max\_epochnum}$, then the hybrids are denoted by $H_3^1, \ldots, H_3^M$ where H_3^k is the hybrid where the root inputs for the first k epochs are all using the specification of Π_{eKE}. The hybrid H_3^M is computationally indistinguishable from H_2 based on the DDH assumption on the group $\mathbb{G}$.

- **Hybrid H_4:** In this hybrid the random choice of initial root key is replaced by the value computed using $\mathcal{F}_{\mathsf{LTM}}$. The indistinguishability of this hybrid is also by reduction to the DDH assumption on group $\mathbb{G}$ since $\mathcal{F}_{\mathsf{LTM}}$ computes the value $s = s_a||s_c$ using the the X3DH protocol.

- **Hybrid H_5: (Real World)** This hybrid is the real world. The only real difference between this hybrid and the previous one is the fact that the parties run separate instances of KDF instead of a joint instance. This hybrid is indistinguishable from the previous one because: (1) KDF consists of deterministic functions,(2) the two parties run KDF on identical inputs to the ones they use in H_4.

Note

In hybrids H_0, H_1, and H_2 we display the lines of the simulator and functionality that are relevant to the current hybrid. This note explains the colors used in the display boxes; **black text existed in the previous hybrid and does not change in the current hybrid; brown text was added in the current hybrid;** ~~struck out grey text was erased in the current hybrid.~~

6.3.1 Hybrid H_0 (Ideal World)

This is the combination of the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$, the internal code $\mathcal{I}_{\mathsf{eKE}}$, and the simulator $\mathcal{S}_{\mathsf{eKE}}$ all in the presence of $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$. Here you can consider $\mathcal{S}_{\mathsf{eKE}} + \mathcal{I}_{\mathsf{eKE}} + \mathcal{F}_{\mathsf{eKE}}$ all together as one combined unit (with combined state) since this combined unit must be proven indistinguishable from the real protocol Π_{eKE}. In other words, this hybrid is exactly the ideal world, we are simply considering the code of $\mathcal{S}_{\mathsf{eKE}}$ and $\mathcal{I}_{\mathsf{eKE}}$ to be directly inlined into the functionality $\mathcal{F}_{\mathsf{eKE}}$ as shown below:

$\mathcal{S}_{\mathsf{eKE}}$ (Hybrid 0)

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

$\mathcal{F}_{\mathsf{eKE}}$ (Hybrid 0)

ConfirmReceivingEpoch:

1. (e) ~~Initialize $\mathsf{state}_{\mathcal{I}} = \perp$, call $\mathcal{F}_{\mathsf{lib}}$ with input "eKE" to obtain internal adversarial code $\mathcal{I}_{\mathsf{eKE}}$.~~
2. (b) **If** $\mathsf{epoch\_id}^* \neq \mathsf{epoch\_id\_self}_{1-i}$: Set $\mathsf{diverge\_parties} = \mathit{true}$, run step 3 of `Corrupt` to set $\mathsf{recv\_chain\_key}^*$ and leakage. Run $\mathcal{S}_{\mathsf{eKE}}(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}^*, \mathsf{leakage})$ and discard the output. ~~, and send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}^*, \mathsf{leakage})$ to the adversary. On receiving a response, continue.~~
3. **If** $\mathsf{epoch\_num}_i + 2 \in \mathsf{compromised\_epochs}$ or $\mathsf{diverge\_parties}$: Run $\mathcal{S}_{\mathsf{eKE}}(\texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$. ~~Send backdoor message $(\texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$ to the adversary.~~ **Else** run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$.

GetSendingKey:

4. **If** $\mathsf{diverge\_parties} = \mathit{true}$, or $\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$ run $\mathcal{S}_{\mathsf{eKE}}(\texttt{GetSendingKey}, i)$, on producing output $(\texttt{GetSendingKey}, i, K_{\mathtt{send}})$, set $\mathsf{sending\_chain\_key}_i = K_{\mathtt{send}}$.
~~send backdoor message $(\texttt{GetSendingKey}, i)$ to the adversary; on receiving backdoor message $(\texttt{GetSendingKey}, i, K_{\mathtt{send}})$ from $\mathcal{A}$ set $\mathsf{sending\_chain\_key}_i = K_{\mathtt{send}}$.~~

GetReceivingKey:

5. (a) **If** $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_{1-i}$, add $\mathsf{epoch\_num}_i + 2$ to $\mathsf{compromised\_epochs}$.
   (b) Run $\mathcal{S}_{\mathsf{eKE}}(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$. On producing output $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*)$, output $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key}^*)$.
   ~~Send $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$ to the adversary.~~
   (c) ~~Upon receiving $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*)$ from $\mathcal{A}$, output $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key}^*)$.~~

Corrupt:

4. Run $\mathcal{S}_{\mathsf{eKE}}(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}^*, \mathsf{leakage})$. On producing the output $(\texttt{ReportState}, i, \mathcal{V}iew_i)$ output $(\texttt{Corrupt}, \mathcal{V}iew_i)$.
   ~~Send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}^*, \mathsf{leakage})$ to the adversary.~~
5. ~~Upon receiving $(\texttt{ReportState}, i, \mathcal{V}iew_i)$ from $\mathcal{A}$, output $(\texttt{Corrupt}, \mathcal{V}iew_i)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.~~

6.3.2 Hybrid H_1

The changes made in this step are very important for setting up the next few hybrids. The two main changes between this hybrid and the previous one are:

1. This hybrid chooses the initial root key completely at random in Step 1d of $\mathcal{F}_{\mathsf{eKE}}$, ignoring the output of $\mathcal{F}_{\mathsf{LTM}}$.
2. When parties are uncompromised, this hybrid chooses root inputs randomly from G and runs those though a function $F : \mathbb{G} \to \mathcal{K}_{ep}$ (chosen uniformly at random for the corresponding epoch.)

> Note the exception in the choice of $\mathsf{root\_input}$ that happens for the very first sending chain key chosen after the end of a compromise. The random choice of $\mathsf{root\_input}$ in this case corresponds to the `Recover` interface in the 'real execution' of the CPRFG security game. The properties provided to the $\mathsf{root\_input}$ via the DDH assumption on group $\mathbb{G}$ are only required for secure recovery. In a later hybrid, this random choice of $\mathsf{root\_input}$ in the case of recovery is replaced with the computation of a Diffie-Hellman key as in Π_{eKE} via a reduction to the DDH assumption on group $\mathbb{G}$.

Note that the view of the environment in this hybrid is identically distributed to its view in the previous hybrid: The first change doesn't impact the view of the adversary at all in this hybrid since the initial root key isn't used anywhere in the rest of this hybrid. Moreover, from the security of the X3DH protocol, the initial root key computed by $\mathcal{F}_{\mathsf{LTM}}$ is indistinguishable from a randomly chosen initial root key. This will be important in hybrid H_3 where a CPRFG protocol KDF is used to choose the chainkeys.

The edits to the code of $\mathcal{S}_{\mathsf{eKE}} + \mathcal{I}_{\mathsf{eKE}} + \mathcal{F}_{\mathsf{eKE}}$ are presented in detail below. In particular, the only edits in this hybrid are to the code of $\mathcal{F}_{\mathsf{eKE}}$. Recall that this functionality can now access the internal states of $\mathcal{I}_{\mathsf{eKE}}$ and $\mathcal{S}_{\mathsf{eKE}}$; the values $\mathsf{epoch\_id}_i, \mathsf{epoch\_key}_i$ referred to below belong to $\mathcal{I}_{\mathsf{eKE}}$'s state. These values are used to compute $\mathsf{root\_input}$ for every epoch except when recovering from compromise.

$\mathcal{F}_{\mathsf{eKE}}$ (Hybrid 1)

ConfirmReceivingEpoch:

1. (d) On receiving $(\texttt{ComputeSendingRootKey}, k, \mathsf{ek}^{\mathsf{pk}})$ from $\mathcal{F}_{\mathsf{LTM}}$, sample a value $s_a||s_c \xleftarrow{\$} \{0,1\}^n$. ~~continue.~~

4. (c) Choose a new random function $F : \mathbb{G} \to \mathcal{K}_{ep}$.

   (d) Output $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id\_self}_i)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

GetSendingKey:

3. **If** there is a tuple $(\mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{i-1}, \mathsf{chain\_key})$ in the list $\mathsf{computed\_values}$, set $\mathsf{sending\_chain\_key}_i = \mathsf{chain\_key}$. **Else** :

   ~~Sample $\mathsf{sending\_chain\_key}_i \xleftarrow{\$} \mathcal{K}_{ep}$ from the key distribution.~~

   (a) **If** $\mathsf{epoch\_num}_i - 1 \notin \mathsf{compromised\_epochs}$ compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$. **Else** choose a random value $\mathsf{root\_input} \xleftarrow{\$} \mathbb{G}$ and store the tuple $(\mathsf{epoch\_num}_i, \mathsf{root\_input})$.

   (b) Compute $\mathsf{sending\_chain\_key}_i = F(\mathsf{root\_input})$ and store the tuple $(\mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{i-1}, \mathsf{sending\_chain\_key}_i)$ in the list $\mathsf{computed\_values}$.

GetReceivingKey:

6. (a) **If** there is a tuple $(\mathsf{epoch\_id}, \mathsf{epoch\_id}_i, \mathsf{chain\_key})$ in the list $\mathsf{computed\_values}$, set $\mathsf{recv\_chain\_key}_i = \mathsf{chain\_key}$.
~~Sample $\mathsf{recv\_chain\_key}_i \xleftarrow{\$} \mathcal{K}_{ep}$.~~

(b) **Else**:

i. Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}, \mathsf{epoch\_key}_i)$.

ii. Compute $\mathsf{recv\_chain\_key}_i = F(\mathsf{root\_input})$ and store the tuple $(\mathsf{epoch\_id}, \mathsf{epoch\_id}_i, \mathsf{recv\_chain\_key}_i)$ in the list $\mathsf{computed\_values}$.

Note that the changes here closely map to the ideal interface in the CPRFG security game when $b = 1$. (In the next hybrid, a secure CPRFG KDF will be used instead.)

Lemma 13 *The view of the environment in H_0 is identically distributed to that in hybrid H_1.*

Proof: As mentioned earlier, the first change doesn't impact the view of the adversary at all in this hybrid since the initial root key isn't used anywhere in the rest of this hybrid. Additionally, the outputs of a function $F : \mathbb{G} \to \mathcal{K}_{ep}$ chosen uniformly at random are identically distribution to $\mathsf{chain\_key} \xleftarrow{\$} \mathcal{K}_{ep}$ chosen uniformly at random, this remains true both when $\mathsf{root\_input}$ is sampled randomly and when $\mathsf{root\_input}$ is known. □

6.3.3 Hybrid H_2 (This hybrid uses a CPRFG protocol KDF to choose the output keys.)

This hybrid transitions from using a version of the ideal CPRFG game + $\mathcal{S}_{\mathsf{KDF}}$ to using the actual protocol. In particular, the KDF.**Compute** and KDF.**Advance** methods of the KDF are now used to compute the chain keys and update the value $s = s_a||s_c$ for each party instead functions chosen uniformly at random for each epoch.

The detailed edits to the version of $\mathcal{S}_{\mathsf{eKE}} + \mathcal{I}_{\mathsf{eKE}} + \mathcal{F}_{\mathsf{eKE}}$ from the previous hybrid are presented below. The edits in the first box pertain to the same lines of code in $\mathcal{F}_{\mathsf{eKE}}$ as were edited in the previous hybrid. Following that are the edits to $\mathcal{S}_{\mathsf{eKE}}$ which replace the use of $\mathcal{S}_{\mathsf{KDF}}$ with the use of the protocol KDF. As before, the values $\mathsf{epoch\_id}_i, \mathsf{epoch\_key}_i$ for $i \in \{0, 1\}$ belong to $\mathcal{I}_{\mathsf{eKE}}$'s state. These values are used to compute $\mathsf{root\_input}$ for every epoch except for during recovery from compromise.

$\mathcal{F}_{\mathsf{eKE}}$ (Hybrid 2)

ConfirmReceivingEpoch:

1. (d) On receiving $(\texttt{ComputeSendingRootKey}, k, \mathsf{ek}^{\mathsf{pk}})$ from $\mathcal{F}_{\mathsf{LTM}}$, sample a value $s_a||s_c \xleftarrow{\$} \{0,1\}^n$.

4. (c) **If** $\mathsf{epoch\_num}_i - 2 \notin \mathsf{compromised\_epochs}$, compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$. **Else**, get $\mathsf{root\_input}$ from the tuple $(\mathsf{epoch\_num}_i, \mathsf{root\_input})$.
~~Choose a new random function $F : \mathbb{G} \to \mathcal{K}_{ep}$.~~

(d) Run KDF.**Advance**$(s_a, \mathsf{root\_input}) = s'$, and change state $s = s'$.

GetSendingKey:

3. **If** there is a tuple $(\mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{i-1}, \mathsf{chain\_key})$ in the list $\mathsf{computed\_values}$, set $\mathsf{sending\_chain\_key}_i = \mathsf{chain\_key}$. **Else**:

(a) **If** $\mathsf{epoch\_num}_i - 1 \notin \mathsf{compromised\_epochs}$ compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$. **Else** choose a random value $\mathsf{root\_input} \xleftarrow{\$} \mathbb{G}$ and store the tuple $(\mathsf{epoch\_num}_i, \mathsf{root\_input})$.

(b) ~~Compute $\mathsf{sending\_chain\_key}_i = F(\mathsf{root\_input})$ and s~~
Run KDF.**Compute**$(s_c, \mathsf{root\_input}) = (s'_c, \mathsf{sending\_chain\_key}_i)$ and change state $s_c = s'_c$. Store the tuple $(\mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{i-1}, \mathsf{sending\_chain\_key}_i)$ in the list $\mathsf{computed\_values}$.

GetReceivingKey:

6. (a) **If** there is a tuple $(\mathsf{epoch\_id}, \mathsf{epoch\_id}_i, \mathsf{chain\_key})$ in the list $\mathsf{computed\_values}$, set $\mathsf{recv\_chain\_key}_i = \mathsf{chain\_key}$.

   (b) **Else**:

   i. Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}, \mathsf{epoch\_key}_i)$.

   ii. ~~Compute $\mathsf{recv\_chain\_key}_i = F(\mathsf{root\_input})$ and s~~
   Run $KDF.\mathbf{Compute}(s_c, \mathsf{root\_input}) = (s'_c, \mathsf{recv\_chain\_key}_i)$ and update the value $s_c = s'_c$.
   Store the tuple $(\mathsf{epoch\_id}, \mathsf{epoch\_id}_i, \mathsf{recv\_chain\_key}_i)$ in the list $\mathsf{computed\_values}$.

$\mathcal{S}_{\mathsf{eKE}}$ (Hybrid 2)

GetReceivingKey:

3. If $\mathsf{diverge\_parties} = \mathit{false}$:
   - Run the KDF simulator $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compute}, \mathcal{V}iew_i.\mathsf{root\_input})$. Set the output as $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key}$.
4. Else ($\mathsf{diverge\_parties} = \mathit{true}$):
   - Set $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key} = KDF.\mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.

ReportState:

5. (c) iv. ~~Run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compromise}, \mathsf{leakage}, (\mathsf{root\_input}^*, \mathsf{recv\_chain\_key}^*))$ to get an output root_key.~~

   v. Set $\mathcal{V}iew_{1-j}.\mathsf{root\_key} = s_a||s_c$~~root_key~~.

GenEpochId:

4. (a) ~~Run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Advance}, \mathsf{root\_input})$.~~

6. (c) Set $s_a||s_c = \mathcal{V}iew_{1-j}.\mathsf{root\_key}$.

   (d) Delete the objects $\mathcal{V}iew_0, \mathcal{V}iew_1$ and output $(\mathtt{GenEpochId}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{epoch\_id}_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

7. (c) Let $s_a||s_c = \mathcal{V}iew_i.\mathsf{root\_key}$ and compute $KDF.\mathbf{Compute}(s_c, \mathcal{V}iew_i.\mathsf{root\_input}) = s'_c, k$.
   ~~If $\mathsf{diverge\_parties} = \mathit{false}$, run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compute}, \mathsf{root\_input})$ and set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key}$ to be its output.~~

   (d) Update $\mathcal{V}iew_i.\mathsf{sending\_chain\_key} = k$ and $\mathcal{V}iew_i.\mathsf{root\_key} = s_a||s'_c$.
   ~~Else ($\mathsf{diverge\_parties} = \mathit{true}$), set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key} = KDF.\mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathsf{root\_input})$.~~

This hybrid is indistinguishable from the previous one because our KDF is a secure CPRFG. This is proved via a reduction to the security of the KDF protocol. An environment that can distinguish this hybrid from the previous one can be used to build an adversary that wins the CPRFG security game from Figure 22.

Lemma 14 *Assume that KDF is a CPRFG. Then the view of the environment in hybrid H_1 is computationally indistinguishable from that in hybrid H_2.*

$\mathcal{A}$: $\mathcal{F}_{\mathsf{eKE}}$ (proof of Lemma 14)

ConfirmReceivingEpoch:

1. (d) On receiving $(\mathtt{ComputeSendingRootKey}, k, \mathsf{ek}^{\mathsf{pk}})$ from $\mathcal{F}_{\mathsf{LTM}}$, initialise $\mathcal{O}_{\mathsf{CPRFG}}$.
   ~~sample a value $s_a||s_c \xleftarrow{\$} \{0,1\}^n$.~~

4. (c) **If** $\mathsf{epoch\_num}_i - 2 \notin \mathsf{compromised\_epochs}$, compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$. **Else**, get $\mathsf{root\_input}$ from the tuple $(\mathsf{epoch\_num}_i, \mathsf{root\_input})$.

   (d) Send $(\mathsf{Advance}, \mathsf{root\_input})$ to $\mathcal{O}_{\mathsf{CPRFG}}$.

~~Run $KDF.\mathbf{Advance}(s_a, \mathsf{root\_input}) = s'$, and change state $s = s'$.~~
//If $\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$, recovery just occurred so the epoch was advanced by the other party calling the `GetSendingKey` method.

GetSendingKey:

3. **If** there is a tuple $(\mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{i-1}, \mathsf{chain\_key})$ in the list $\mathsf{computed\_values}$, set $\mathsf{sending\_chain\_key}_i = \mathsf{chain\_key}$. **Else**:
   (a) **If** $\mathsf{epoch\_num}_i - 1 \notin \mathsf{compromised\_epochs}$ compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$. and send $(\mathsf{Compute}, \mathsf{root\_input})$ to the oracle $\mathcal{O}_{\mathsf{CPRFG}}$. **Else** send $(\mathtt{Recover})$ to the oracle $\mathcal{O}_{\mathsf{CPRFG}}$.
   ~~choose a random value $\mathsf{root\_input} \xleftarrow{\$} \mathbb{G}$ and store the tuple $(\mathsf{epoch\_num}_i, \mathsf{root\_input})$.~~
   (b) ~~Run $KDF.\mathbf{Compute}(s_c, \mathsf{root\_input}) = (s'_c, \mathsf{sending\_chain\_key}_i)$ and change state $s_c = s'_c$.~~ S
   On getting a response $\mathsf{sending\_chain\_key}_i$, store the tuple $(\mathsf{epoch\_id}_i, \mathsf{epoch\_id}_{i-1}, \mathsf{sending\_chain\_key}_i)$ in the list $\mathsf{computed\_values}$.

GetReceivingKey:

6. (a) **If** there is a tuple $(\mathsf{epoch\_id}, \mathsf{epoch\_id}_i, \mathsf{chain\_key})$ in the list $\mathsf{computed\_values}$, set $\mathsf{recv\_chain\_key}_i = \mathsf{chain\_key}$.
   (b) **Else**:
      i. Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}, \mathsf{epoch\_key}_i)$.
      ii. If $\mathsf{epoch\_id}_i \notin \mathsf{compromised\_epochs}$, then send $(\mathsf{Compute}, \mathsf{root\_input})$ to the oracle $\mathcal{O}_{\mathsf{CPRFG}}$ for the CPRFG security game and get an output $\mathsf{chain\_key}$. **Else**, run $KDF.\mathbf{Compute}(s_c, \mathsf{root\_input}) = (s'_c, \mathsf{recv\_chain\_key}_i)$ and update the value $s_c = s'_c$. Store the tuple $(\mathsf{epoch\_id}, \mathsf{epoch\_id}_i, \mathsf{recv\_chain\_key}_i)$ in the list $\mathsf{computed\_values}$.

$\mathcal{A}$: $\mathcal{S}_{\mathsf{eKE}}$ (proof of Lemma 14)

GetReceivingKey:

3. If $\mathsf{diverge\_parties} = \mathit{false}$:
   - Send $(\mathsf{Compute}, \mathcal{V}iew_i.\mathsf{root\_input})$ to $\mathcal{O}_{\mathsf{CPRFG}}$ ~~Run the KDF simulator $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compute}, \mathcal{V}iew_i.\mathsf{root\_input})$~~ and set the output as $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key}$.
4. Else ($\mathsf{diverge\_parties} = \mathit{true}$):
   - set $s_a||s_c = \mathcal{V}iew_i.\mathsf{root\_key}$ and compute $KDF.\mathbf{Compute}(s_c, \mathcal{V}iew_i.\mathsf{root\_input}) = s'_c, k$.
     ~~Set $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key} = KDF.\mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.~~
   - Update $\mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key} = k$ and $\mathcal{V}iew_i.\mathsf{root\_key} = s_a||s'_c$.

ReportState:

5. (c) iv. Send $(\mathsf{Compromise}, \mathsf{root\_input}^*)$ to $\mathcal{O}_{\mathsf{CPRFG}}$ and get back a state $s = s_a||s_c$.
   ~~Run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compromise}, \mathsf{leakage}, (\mathsf{root\_input}^*, \mathsf{recv\_chain\_key}^*))$ to get an output root_key.~~
   v. Set $\mathcal{V}iew_{1-j}.\mathsf{root\_key} = s_a||s_c$~~root_key~~.

GenEpochId:

4. (a) Send $(\mathsf{Advance}, \mathsf{root\_input})$ to $\mathcal{O}_{\mathsf{CPRFG}}$.
   ~~Run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Advance}, \mathsf{root\_input})$.~~
6. (c) Set $s_a||s_c = \mathcal{V}iew_{1-j}.\mathsf{root\_key}$.
   (d) Delete the objects $\mathcal{V}iew_0, \mathcal{V}iew_1$ and output $(\mathtt{GenEpochId}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{epoch\_id}_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
7. (c) If $\mathsf{diverge\_parties} = \mathit{false}$, send $(\mathsf{Compute}, \mathsf{root\_input})$ to $\mathcal{O}_{\mathsf{CPRFG}}$
   ~~run $\mathcal{S}_{\mathsf{KDF}}(\mathsf{Compute}, \mathsf{root\_input})$~~ and set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key}$ to be its output.
   (d) Else, set $s_a||s_c = \mathcal{V}iew_i.\mathsf{root\_key}$ and compute $KDF.\mathbf{Compute}(s_c, \mathcal{V}iew_i.\mathsf{root\_input}) = s'_c, k$.
   ~~set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key} = KDF.\mathbf{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathsf{root\_input})$.~~
   (e) Update $\mathcal{V}iew_i.\mathsf{sending\_chain\_key} = k$ and $\mathcal{V}iew_i.\mathsf{root\_key} = s_a||s'_c$.

Proof: Let $\mathcal{D}$ be an environment that distinguishes between hybrids H_1 and H_2 with advantage greater than negligible in λ, let the oracle $\mathcal{O}_{\mathsf{CPRFG}}$ be as in Definition 11 (an oracle that runs either the real or the ideal version of the CPRFG game from Figure 22 on Page 54). The oracle $\mathcal{D}$ will be used to construct an adversary $\mathcal{A}$ (described below) that can distinguish whether the oracle $\mathcal{O}_{\mathsf{CPRFG}}$ is running the real game or the ideal game with advantage greater than negligible in λ. The adversary $\mathcal{A}$ maps messages from the distinguishing environment $\mathcal{D}$ into messages for the CPRFG game and vice versa. The environment $\mathcal{D}$'s view in its interaction with the adversary $\mathcal{A}$ exactly matches its view in either hybrid H_1 or hybrid H_2 depending on whether $\mathcal{O}_{\mathsf{CPRFG}}$ represents the 'ideal game' or the 'real game' respectively. Therefore, to acquire the exact same distinguishing probability as $\mathcal{D}$, $\mathcal{A}$ mimics the distinguishing decisions of $\mathcal{D}$ exactly. That is, if at any point $\mathcal{D}$ stops and produces an output, $\mathcal{A}$ will produce the same output.

The adversary $\mathcal{A}$ responds to requests from the distinguishing environment $\mathcal{D}$ by running the instructions for $\mathcal{F}_{\mathsf{eKE}} + \mathcal{S}_{\mathsf{eKE}} + \mathcal{I}_{\mathsf{eKE}}$ from H_2 with some minor changes where it queries $\mathcal{O}_{\mathsf{CPRFG}}$ to choose the sending and receiving chain keys that it outputs. The tricky part of replacing the choice of chain keys from H_2 with calls to $\mathcal{O}_{\mathsf{CPRFG}}$ is the decision of when to send an $\mathsf{Advance}$ query to $\mathcal{O}_{\mathsf{CPRFG}}$.

Most of the time, the $\mathsf{Advance}$ queries (in $\mathcal{A}$'s code) ensure that the epoch of $\mathcal{O}_{\mathsf{CPRFG}}$ is consistent with the epoch of the party that is behind, i.e the party that is not the latest sender. This allows $\mathcal{A}$ to respond to `GetReceivingKey` requests for this party by making $\mathsf{Compute}$ queries to $\mathcal{O}_{\mathsf{CPRFG}}$. However, when recovery from compromise occurs, $\mathcal{A}$ must send a $\mathsf{Recover}$ query to $\mathcal{O}_{\mathsf{CPRFG}}$ to get the parties' new $\mathsf{sending\_chain\_key}$, this necessarily advances the epoch of $\mathcal{O}_{\mathsf{CPRFG}}$ to be consistent with the latest sender. However due to the preceeding corruption, $\mathcal{A}$ has enough information in these situations to respond directly to `GetReceivingKey` requests for the party that is behind.

When the oracle $\mathcal{O}_{\mathsf{CPRFG}}$ corresponds to the 'real game,' $\mathcal{A}$ is simply executing hybrid H_2. Correspondingly, when the oracle $\mathcal{O}_{\mathsf{CPRFG}}$ corresponds to the 'ideal game,' $\mathcal{A}$ is simply executing hybrid H_1. Since KDF is a CPRFG, there is a simulator $\mathcal{S}_{\mathsf{KDF}}$ for which the 'real execution' and 'ideal execution' of the CPRFG game are computationally indistinguishable. For this same simulator then, hybrids H_1 and H_2 are therefore computationally indistinguishable. □

6.3.4 Hybrid H_3

This hybrid again changes the method of choosing chain keys *when both parties are uncompromised.* In particular, we replace the random choices of $\mathsf{root_input}$s with the Diffie-Hellman keys computed as specified in the protocol Π_{eKE} over a series of hybrids. Let $\mathbb{G}, q, g$ be fixed, let $M = \mathsf{max\_epochnum}$, then the hybrids are denoted by $H_3^1, \ldots, H_3^M$ where H_3^k is the hybrid where the root inputs for the first k epochs are DDH values computed using the specification of Π_{eKE}. All remaining root inputs are randomly sampled as in hybrid H_2. The hybrid H_3^M is computationally indistinguishable from H_2 based on the DDH assumption on the group $\mathbb{G}$.

Lemma 15 *Assume that the DDH assumption holds for group $\mathbb{G}$. Then the view of the environment in hybrid H_2 is computationally indistinguishable from that of the last hybrid in series H_3.*

Lemma 15 will be proven via reduction to the following distinguishing game: The adversary $\mathcal{A}$ is given a challenge triple $\mathbb{G}, q, g, (g^x, g^y, g^z)$. If $b = 0$ then $z \xleftarrow{\$} |\mathbb{G}|$, if $b = 1$ then $z = x \cdot y$. $\mathcal{A}$ must distinguish whether $b = 0$ or $b = 1$. Because the DDH assumption holds for group $\mathbb{G}$, no PPT adversary $\mathcal{A}$ can distinguish whether $b = 0$ or $b = 1$ with advantage greater than negligible in the security parameter λ.

Say that there exists a distinguishing environment $\mathcal{D}$ that can distinguish between H_2 and H_3 with non-negligible probability. Then $\mathcal{D}$ can be used by an adversary $\mathcal{A}$ to win the DDH distinguishing game with non-negligible advantage. The adversary $\mathcal{A}$ is constructed differently on a case by case basis but in both cases the adversary $\mathcal{A}$ will respond to requests from $\mathcal{D}$ by running the instructions for $\mathcal{F}_{\mathsf{eKE}} + \mathcal{S}_{\mathsf{eKE}} + \mathcal{I}_{\mathsf{eKE}}$ from H_3 up to epoch $k-1$ and from H_2 starting epoch $k+1$ for some value k. Also in both cases $\mathcal{A}$ will choose the root input for epoch k to be g^z from the challenge triple (g^x, g^y, g^z), and the epoch ids for epochs $k-1$ and k to be g^x and g^y respectively. (Since $\mathcal{A}$ not doesn't have the secret exponent for the epoch g^x, it chooses the root input for epoch $k-1$ to be $(g^x)a$ where a is the known secret exponent for epoch $k-2$.)

Instructions for $\mathcal{A}$ in both cases:

1. If there is some environment $\mathcal{D}$ that successfully distinguishes between hybrids H_2 and H_3 with non-negligible probability in the parameter λ, then it can also distinguish neighboring hybrids H_3^{k-1} and H_3^k for at least one k. First, find such a k.

2. When the parties are compromised, simply follow the code of hybrid H_2 (which is the same as hybrid H_3^M.)

3. When the parties are not compromised follow the instructions below to choose root inputs for all epochs and to choose epoch ids for epochs $k-1$ and k. All the rest of the code of hybrids H_2 and H_3^M is the same, so follow it exactly.

   - Up to epoch k, choose root inputs to be real Diffie-Hellman keys.
   - During epoch k, choose root input g^z from the challenge triple (g^x, g^y, g^z). Correspondingly, choose epoch ids g^x, g^y for epochs $k-1$ and k respectively. (choose the root input for epoch $k-1$ to be $(g^x)a$ where a is the known secret exponent for epoch $k-2$)
   - For each epoch starting $k+1$, randomly sample a fresh root input $h \xleftarrow{\$} \mathbb{G}$.

Case 1: The probability that epoch k is compromised by $\mathcal{D}$ is higher in one of the two hybrids H_3^{k-1} and H_3^k by an amount that is non-negligible in λ.

If epoch k is compromised by $\mathcal{D}$ significantly more often in one of the two hybrids, then the adversary can win the DDH distinguishing game by choosing its outputs as below:

3 If $\mathcal{D}$ takes any actions that cause epoch k to be compromised, stop and output 1, otherwise output a randomly chosen bit at the end of the execution. (The actions that could cause epoch k to be compromised include: 1) corrupting the party in epochs $\mathsf{epoch\_num}^* + 1, \mathsf{epoch\_num}^*, \mathsf{epoch\_num}^* - 1, \mathsf{epoch\_num}^* - 2, \mathsf{epoch\_num}^* - 3$, 2) diverging the parties 3) sending bogus $\mathsf{epoch_id}$s to the receiver of epoch $k-2$ or $k-1$ while the epoch is compromised.)

$$\begin{aligned}&\Pr[\mathcal{A}(g^x, g^y, g^{xy}) = 1] - \Pr[\mathcal{A}(g^x, g^y, g^z) = 1] \\ &= \Pr[\text{epoch } k \text{ compromised in } H_3^{k-1}] - \Pr[\text{epoch } k \text{ compromised in } H_3^k]\end{aligned}$$

Case 2: Now, if the probability that epoch k is compromised $\mathcal{D}$ in hybrid H_3^{k-1} is within a negligible distance of the probability that epoch k is compromised in hybrid H_3^k, then this can be used by $\mathcal{A}$ to win the DDH distinguishing game by choosing its outputs as follows:

3 If $\mathcal{D}$ takes any actions that cause epoch k to be compromised, then stop and output a randomly chosen bit, otherwise wait till $\mathcal{D}$ stops, and then produce the same output that $\mathcal{D}$ produces.

Let the event that epoch k is compromised be cprm and let the event that epoch k is not compromised be $\neg\,\mathsf{cprm}$. Note that when epoch k is compromised then the hybrids H_3^{k-1} and H_3^k are identical. So, an environment $\mathcal{D}$ that corrupts epoch k roughly the same amount in hybrids H_3^k and H_3^{k+1} must have that $\Pr[\mathcal{D}(H_3^{k+1}) = 1 \mid \mathsf{cprm}] - \Pr\left[\mathcal{D}(H_3^k) = 1 \mid \mathsf{cprm}\right] = \mathsf{negl}(\lambda)$ for some negligible function $\mathsf{negl}(\cdot)$. Therefore,

$$\begin{aligned}
&\Pr[\mathcal{A}(g^x, g^y, g^{xy}) = 1] - \Pr[\mathcal{A}(g^x, g^y, g^z) = 1] \\
&= \Pr[\mathcal{D}(H_3^{k+1}) = 1 \mid \neg\,\mathsf{cprm}] - \Pr[\mathcal{D}(H_3^k) = 1 \mid \neg\,\mathsf{cprm}] \\
&= \Pr[\mathcal{D}(H_3^{k+1}) = 1] - \Pr[\mathcal{D}(H_3^k) = 1] - \mathsf{negl}(\lambda)
\end{aligned}$$

Given that epoch k is not compromised, the environment's view is exactly as in H_3^{k-1} when $b = 0$ and H_3^k when $b = 1$. So if $\mathcal{D}$ distinguishes between h_3^{k-1} and H_3^k with non-negligible probability then so does $\mathcal{A}$.

6.3.5 Hybrid H_4

In this hybrid the randomly chosen initial root keys are replaced by root keys computed using $\mathcal{F}_{\mathsf{DIR}}$, $\mathcal{F}_{\mathsf{LTM}}$ as in Π_{eKE}. 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 Π_{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 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 ik, rk.
- $\mathcal{F}_{\mathsf{DIR}}$ reliably provides pairs of parties with the same public diffie hellman half ok.
- The provided ok is unique to a pair of parties.

Lemma 16 *Assume that the DDH assumption holds for group $\mathbb{G}$. Then the view of the environment in hybrid H_4 is computationally indistinguishable from that in H_3.*

6.3.6 Hybrid H_5 (Real World)

Finally, we can now separate the epoch key exchange instances that the two parties interact with by splitting the functionality into two instances. In the hybrids before this, if either party is corrupted or confirms an incorrect epoch id then control is handed over to the simulator who runs separate instances of the protocol for each party anyway. So the interesting case is when the parties' views of the $\mathsf{epoch\_id}$'s so far agree. In this case, agreement for all previous hybrids is by fiat. Here, agreement holds because of the consistency of KDF.**Compute** and because KDF.**Advance** is a deterministic function of s_a, r.

Lemma 17 *The view of the environment in hybrid H_5 is identically distributed to that in H_4.*

□

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

In this section, we prove that protocol $\Pi_{\mathsf{fs\_aead}}$ (which uses ideal functionalities $\mathcal{F}_{\mathsf{mKE}}$ and $\mathcal{F}_{\mathsf{aead}}$ as subroutines) UC-realizes the functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ (which is described in section Section 5.2).

The protocol $\Pi_{\mathsf{fs\_aead}}$ and the proof that it realises $\mathcal{F}_{\mathsf{fs\_aead}}$ are presented in Section 7.3. Prior to that, the component functionalities $\mathcal{F}_{\mathsf{mKE}}$ and $\mathcal{F}_{\mathsf{aead}}$ are defined in Sections 7.1 and 7.2 respectively.

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

In this section, we describe the ideal message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$ (as described previously in Section 4). This functionality, which is shown in full detail in Figure 28 on page 73, 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. In the context of $\Pi_{\mathsf{fs\_aead}}$, $\mathcal{F}_{\mathsf{mKE}}$ provides $\mathsf{key\_seed}$'s to $\mathcal{F}_{\mathsf{aead}}$ instances for each message within the epoch that $\mathcal{F}_{\mathsf{mKE}}$ represents. For each $\mathsf{msg\_num}$, the functionality allows at most one $(\texttt{RetrieveKey}, \mathsf{pid})$ request for each party pid. If a second request is made then it simply ends the activation. That is the functionality will provide the $\mathsf{key\_seed}$ at most once for the encryption and for the decryption of each mesage in the epoch. The functionality also closes epochs by allowing the $\Pi_{\mathsf{fs\_aead}}$ instance for each party to make a request $(\texttt{StopKeys}, N)$. Once such a request is made by a party, the functionality will disallow any `RetrieveKey` requests made on behalf of that party for $\mathsf{msg\_num} > N$.

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

In this section, we define an single-message authenticated encryption functionality $\mathcal{F}_{\mathsf{aead}}$, with the full details shown in Figure 29 on page 74). Each $\mathcal{F}_{\mathsf{aead}}$ instance handles the encryption, decryption, and authentication of a particular message for a particular epoch and hands the ciphertext or message back to $\Pi_{\mathsf{fs\_aead}}$.

Encryption: On receiving an encryption request, while not corrupted, $\mathcal{F}_{\mathsf{aead}}$ first consults the message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$ to decide whether encryption of this message is possible for the party pid. It then runs adversarial code $\mathcal{I}_{\mathsf{aead}}$ with the message length $|m|$ to choose the corresponding ciphertext c. It then simply stores the message and adversarially-generated ciphertext in a table. While corrupted, the functionality instead provides the message m to the adversary and allows it to choose a ciphertext directly with this knowledge.

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

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 pid_0 nor pid_1 are ignored. //For notational simplicity we assume some fixed interpretation of pid_0 and pid_1 as complete identities of the two calling machines.

This functionality is parametrized by a seed length λ

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

1. If this is the first activation,
   - Parse sid to recover the two party ids $(\mathsf{pid}_0, \mathsf{pid}_1)$.
   - Initialize dictionary $\mathsf{key\_dict}$ and variables $\texttt{IsCorrupt?} = \mathit{false}$, $\mathsf{msg\_num}_0, \mathsf{msg\_num}_1 = 0$.
     //If this epoch is corrupted then this isn't the first activation because there was already a (Corrupt) request.
     //$\mathsf{msg\_num}_i$ is the largest message number whose key has abeen successfully retrieved by pid_i.
2. **If** there is record $(\texttt{Retrieved}, i, \mathsf{msg\_num})$ or a record $(\texttt{StopKeys}, i, N)$ for $N < \mathsf{msg\_num}$, end the activation.
3. 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 $k \xleftarrow{\$} \{0,1\}^\lambda$.
4. Else ($\texttt{IsCorrupt?} = \mathit{true}$):
   - Send $(\texttt{RetrieveKey}, \mathsf{pid}, \mathsf{msg\_num})$ to the the adversary.
   - Upon receiving $(\texttt{RetrieveKey}, \mathsf{pid}, k)$ from the adversary, continue.
5. Store $\mathsf{key\_dict}[\mathsf{msg\_num}] = k$.
6. If $\mathsf{msg\_num} > \mathsf{msg\_num}_i$, set $\mathsf{msg\_num}_i = \mathsf{msg\_num}$. //$\mathsf{msg\_num}_i$ is the largest message number whose key has abeen successfully retrieved by pid_i.
7. 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 2-6 of 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 $\texttt{IsCorrupt?} = \mathit{true}$, create empty lists $\mathsf{keys\_in\_transit}, \mathsf{pending\_msgs}$.
3. 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 $\mathsf{pending\_msgs}$.
4. If there is a record $(\texttt{StopKeys}, i, N)$ set $\mathsf{stop\_request} = (\texttt{StopKeys}, i, N)$, **Else** set $\mathsf{stop\_request} = \perp$.
5. Send $(\texttt{ReportState}, i, \mathsf{keys\_in\_transit}, \mathsf{msg\_num}_0, \mathsf{msg\_num}_1, \mathsf{stop\_request})$ to $\mathcal{A}$.
6. On receiving a response $(\texttt{ReportState}, i, S)$ from $\mathcal{A}$, output $(\texttt{Corrupt}, S)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid}_i)$.

Figure 28: The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$

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

This functionality has a session id $\mathsf{sid}.aead = (\text{“}aead\text{''}, \mathsf{sid}.fs, \mathsf{msg\_num})$ where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1), \mathsf{epoch\_id})$. Inputs arriving from machines whose identity is neither pid_0 nor pid_1 are ignored. //For notational simplicity we assume some fixed interpretation of pid_0 and pid_1 as complete identities of the two calling machines.

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

1. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. If this is not the first encryption request end the activation.
3. **If** $\texttt{IsCorrupt?} = \mathit{true}$, Send a backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, m, N)$ to $\mathcal{A}$.
4. **Else** $(\texttt{IsCorrupt?} = \mathit{false})$:
   - Provide input $(\texttt{RetrieveKey}, \mathsf{pid})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$.
   - 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.
   - Initialize $\mathsf{state}_{\mathcal{I}} = \perp$, call $\mathcal{F}_{\mathsf{lib}}$ to obtain the internal code $\mathcal{I}$, and run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, |m|, N)$.
5. Upon obtaining $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, c)$ from $\mathcal{A}$ or $\mathcal{I}$, record the tuple $(c, m, N, 1)$, record the sender $\mathsf{S} = i$, and set $\mathsf{ready2decrypt} = \mathit{true}$.
6. 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** $\texttt{IsCorrupt?} = \mathit{false}$:
   - If there isn't a stored key k, provide input $(\texttt{RetrieveKey}, \mathsf{pid})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$. Upon obtaining a response $(\texttt{RetrieveKey}, \mathsf{pid}, k)$ from $\mathcal{F}_{\mathsf{mKE}}$: Store k.
   - If there hasn't been a successful encryption request or $\mathsf{pid} \neq \mathsf{pid}_{\mathsf{S}-i}$, output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
2. If $k = \perp$, if $\mathsf{ready2decrypt} = \mathit{false}$, or if there is a record $(c, N, 0)$, output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$. //Failure of decryption can occur for an honest receiver so we need an explicit failure notification.
3. If there is a record $(c, m, N, 1)$, note $\mathsf{ready2decrypt} = \mathit{false}$ and output $(\texttt{Decrypt}, m)$.
4. **If** $\texttt{IsCorrupt?} = \mathit{false}$:
   - Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, N)$ to obtain the output $(\mathsf{state}_{\mathcal{I}}, v)$.
   - **If** $v = \perp$ then record $(c, N, 0)$, and output $(\texttt{Decrypt}, \texttt{Fail})$. **Else**, note $\mathsf{ready2decrypt} = \mathit{false}$, and output $(\texttt{Decrypt}, m)$.
5. **Else** $(\texttt{IsCorrupt?} = \mathit{true})$:
   - Send backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, c, N)$ to $\mathcal{A}$.
   - Upon receiving response $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, c, N, v)$ from $\mathcal{A}$: **If** $v = \perp$ then record $(c, N, 0)$, and output $(\texttt{Decrypt}, \texttt{Fail})$. **Else** output $(\texttt{Decrypt}, v)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.

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

1. Provide input $(\texttt{RetrieveKey}, \mathsf{pid})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$.
2. Upon obtaining a response $(\texttt{RetrieveKey}, \mathsf{pid}, k)$ from $\mathcal{F}_{\mathsf{mKE}}$, store k.
3. If a message m was successfully decrypted then set $m^* = m$, otherwise, set $m^* = \perp$.
4. Set $\texttt{IsCorrupt?}$ to true, and send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, \mathsf{pid}, k, m^*)$ to $\mathcal{A}$.
5. 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 29: 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 $(\texttt{Encrypt}, m, N)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ do:

1. Check that sid matches the one in the local session ID, and that 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 $(\texttt{Encrypt}, m, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$ and delete m.
5. Upon receiving $(\texttt{Encrypt}, c)$, output $(\texttt{Encrypt}, c)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

1. Check that sid matches the one in the local session ID, and that 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 $(\texttt{Decrypt}, c, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
4. Upon receiving $(\texttt{Decrypt}, v)$, if $v = \texttt{Fail}$ then output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
5. Otherwise ($v \neq \texttt{Fail}$):
   - 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 $(\texttt{Decrypt}, v)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

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

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

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

Corruption: On receiving $(\texttt{Corrupt})$ 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 sid matches the one in the local session ID, and that 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 $(\texttt{Corrupt})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE\text{”}, \mathsf{sid}.fs))$.
4. On receiving $(\texttt{Corrupt}, S_{mKE})$, add it to S and do the following.
5. For each record $\mathsf{msg\_num} \in \mathsf{pending\_msgs}$:
   - Send $(\texttt{Corrupt}, \mathsf{pid}_i)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$
   - On receiving a response $(\texttt{Corrupt}, \mathsf{pid}_i, S_{\mathsf{msg\_num}})$, add $S_{\mathsf{msg\_num}}$ to S.
6. Output $(\texttt{Corrupt}, S)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

Figure 30: The Forward-Secure Encryption $\Pi_{\mathsf{fs\_aead}}$

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

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

1. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. **If** $\mathsf{state}_{\mathcal{I}} = \perp$:
   (a) Initialize objects $\mathcal{V}iew_0$, $\mathcal{V}iew_1$, and dictionary $\mathsf{records}$.
   (b) Set $\mathcal{V}iew_0.\mathsf{curr\_msg\_num} = \mathcal{V}iew_1.\mathsf{curr\_msg\_num} = 0$.
   (c) Set $\mathcal{V}iew_0.\mathsf{pending\_msgs} = \mathcal{V}iew_1.\mathsf{pending\_msgs} = []$
3. **Else**, parse the objects $\mathcal{V}iew_0$, $\mathcal{V}iew_1$, and the dictionary $\mathsf{records}$ from $\mathsf{state}_{\mathcal{I}}$.
4. Set $\mathsf{msg\_num} = \mathcal{V}iew_i.\mathsf{curr\_msg\_num} + 1$ and update $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} += 1$.
5. If $\mathsf{msg\_num\_state}_{\mathsf{aead}}$ exists in $\mathsf{state}_{\mathcal{I}}$ then parse it otherwise initialize $\mathsf{msg\_num\_state}_{\mathsf{aead}} = \perp$.
6. Run $\mathcal{I}_{\mathsf{aead}}(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \texttt{Encrypt}, \mathsf{pid}, |m|, N)$.
7. Upon receiving output $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \texttt{Encrypt}, \mathsf{pid}, c)$, store $(\ell = |m|, \mathsf{pid}, \mathsf{ready2decrypt} = \mathit{true}), (c, N, 1)$ in $\mathsf{records}[\mathsf{msg\_num}]$.
8. Update or store for the first time the values $\mathcal{V}iew_0$, $\mathcal{V}iew_1$, $\mathsf{records}$, $\mathsf{msg\_num\_state}_{\mathsf{aead}}$, in $\mathsf{state}_{\mathcal{I}}$
9. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, c)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

`Authenticate`**:** 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. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Parse the dictionary $\mathsf{records}$ from $\mathsf{state}_{\mathcal{I}}$:
   (a) **If** $\mathsf{records}[\mathsf{msg\_num}] = \perp$ then output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
   (b) **Else** get $(\ell, \mathsf{pid}, \mathsf{ready2decrypt})$ and any record of the form (c, N, v) from $\mathsf{records}[\mathsf{msg\_num}]$.
3. **If** $\mathsf{ready2decrypt} = \mathit{false}$ in the retrieved record, output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
4. **If** there is no record (c, n, v):
   - Run $\mathcal{I}_{\mathsf{aead}}(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \texttt{Authenticate}, \mathsf{pid}, c, N)$. On obtaining the output $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \texttt{Authenticate}, \mathsf{pid}, c, N, v)$, append (c, N, v) to $\mathsf{records}[\mathsf{msg\_num}]$ and update $\mathsf{records}$ in $\mathsf{state}_{\mathcal{I}}$.
5. **If** $v = 0$ then output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$. **Else** continue.
6. While $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} \leq \mathsf{msg\_num}$ do: //we only get to this step if decryption succeeds.
   - Increment $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} += 1$.
   - Append $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$ to $\mathcal{V}iew_i.\mathsf{pending\_msgs}$.
7. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, 1)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

Figure 31: Internal Adversary, $\mathcal{I}_{\mathsf{fsaead}}$

Decryption: Once a successful decryption occurs, an instance of the functionality $\mathcal{F}_{\mathsf{aead}}$ will never try to decrypt a ciphertext again. While not corrupted, the encryption functionality $\mathcal{F}_{\mathsf{aead}}$ will only ever output the originally encrypted message m or a failure notification. On receiving a decryption request, if the instance is not corrupted, then the functionality first consults with the message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$ to decide whether decryption of this message is possible for the party pid at all, if not then it outputs a failure notification. (If the instance has been corrupted already then this key has already been retrieved and sent to $\mathcal{A}$ by the functionality.) Otherwise, if the ciphertext provided matches the ciphertext c chosen at the time of corruption then

$\mathcal{F}_{\mathsf{aead}}$ always outputs m in both the honest and corrupted case. In general $\mathcal{F}_{\mathsf{aead}}$ keeps consistency with its prior outputs, always outputting a failure notification for ciphertexts it previously refused to decrypt. If the ciphertext does not match the one produced at the time of encryption, and if decryption of this ciphertext has never been attempted before, then based on whether corruption has occurred, the functionality will either run the adversarial code to decide if the ciphertext authenticates or it will allow the adversary to decrypt the ciphertext to any value of its choosing (up to consistency of outputs.).

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 30 on page 75. In this subsection we prove Theorem 4.

Theorem 4 *Protocol $\Pi_{\mathsf{fs\_aead}}$ (perfectly) UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ in the presence of $\mathcal{F}_{\mathsf{lib}}$.*

The proof of Theorem 4 takes up the rest of this section. For legibility, we first provide a high-level overview of the proof, and then we provide the details in the two cases where the parties are and aren't corrupted.

Proof: To prove Theorem 4, we first construct the simulator $\mathcal{S}_{\mathsf{fsaead}}$ and the internal code $\mathcal{I}_{\mathsf{fsaead}}$. Then, we argue that the environment Env has an identically distributed view in its interaction with $\Pi_{\mathsf{fs\_aead}} + \mathcal{F}_{\mathsf{mKE}} + \mathcal{F}_{\mathsf{aead}}$ as it does in its interaction with $\mathcal{F}_{\mathsf{fs\_aead}} + \mathcal{S}_{\mathsf{fsaead}} + \mathcal{I}_{\mathsf{fsaead}}$.

The objective of $\mathcal{I}_{\mathsf{fsaead}}$ is to use $\mathcal{I}_{\mathsf{aead}}$ to provide ciphertexts that match the distribution of ciphertexts in the real world. This is necessary because we do not restrict the ciphertexts provided by $\mathcal{F}_{\mathsf{fs\_aead}}$ to any particular distribution or format. (The detailed specification of this internal code can be found in Figure 31 on page 76.)

The objective of the ideal process adversary $\mathcal{S}_{\mathsf{fsaead}}$ is to simulate the interactions that would take place between the environment and the protocol $\Pi_{\mathsf{fs\_aead}}$ once a party has been corrupted. Since $\Pi_{\mathsf{fs\_aead}}$'s security properties derive entirely from its component functionalities $\mathcal{F}_{\mathsf{mKE}}$ and $\mathcal{F}_{\mathsf{aead}}$, the view of an environment Env will be identically distributed in the real and ideal scenarios. When the simulator $\mathcal{S}_{\mathsf{fsaead}}$ receives messages from $\mathcal{F}_{\mathsf{fs\_aead}}$ in the ideal world, it takes the actions specified in the pseudocode in Figure 32 on page 78.

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 compare the actions taken by $\Pi_{\mathsf{fs\_aead}}$ in the real world with the actions taken by $\mathcal{F}_{\mathsf{fs\_aead}}$ together with $\mathcal{I}_{\mathsf{fsaead}}$ and $\mathcal{S}_{\mathsf{fsaead}}$ in the ideal world. As in the proof of Theorem 2, this proof holds by induction over the actions taken by the internal adversarial code $\mathcal{I}_{\mathsf{fsaead}}$ and the simulator $\mathcal{S}_{\mathsf{fsaead}}$. We show that each action leaves the distribution in the real and ideal worlds the same: that is, each action of $\mathcal{S}_{\mathsf{fsaead}}$ and $\mathcal{I}_{\mathsf{fsaead}}$ maintains the property that the view of the environment 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 Env. As a consequence, the entire execution is deterministic, since the message keys themselves are never used for encryption in $\mathcal{F}_{\mathsf{aead}}$.

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

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

ReportState: On receiving $(\mathsf{state}_{\mathcal{I}}, \mathtt{ReportState}, \mathsf{pid}, \mathsf{leakage})$ for $\mathsf{pid}_i \in \{\mathsf{pid}_0, \mathsf{pid}_1\}$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$:

1. Initialize $\mathsf{keys\_in\_transit} = [\,]$ and a simulated state object $\mathsf{state}_{\mathsf{pid}}$.
2. **If** $\mathtt{IsCorrupt?} = \mathit{true}$, skip to step 4. **Else**, set $\mathtt{IsCorrupt?} = \mathit{true}$ and continue.
3. **If** $\mathsf{state}_{\mathcal{I}} = \perp$:
   (a) Initialize objects $\mathcal{V}iew_0$, $\mathcal{V}iew_1$, and dictionary $\mathsf{records}$.
   (b) Set $\mathcal{V}iew_0.\mathsf{curr\_msg\_num} = \mathcal{V}iew_1.\mathsf{curr\_msg\_num} = 0$.
   (c) Set $\mathcal{V}iew_0.\mathsf{pending\_msgs} = \mathcal{V}iew_1.\mathsf{pending\_msgs} = [\,]$
4. **Else**, parse the objects $\mathcal{V}iew_0$, $\mathcal{V}iew_1$, and the dictionary $\mathsf{records}$ from $\mathsf{state}_{\mathcal{I}}$ and store them. Also parse and store any records $\mathsf{msg\_num\_state}_{\mathsf{aead}}$.
5. Add $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$ and $\mathcal{V}iew_i.\mathsf{pending\_msgs}$ to $\mathsf{state}_{\mathsf{pid}}$.
6. For each $h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N), c, m \in \mathsf{leakage}$:
   (a) Pick a new key $k_{\mathsf{msg\_num}} \xleftarrow{\$} \{0,1\}^{\lambda}$ (where λ is the key length that parametrizes $\mathcal{F}_{\mathsf{mKE}}$).
   (b) Append the tuple $(\mathsf{msg\_num}, k_{\mathsf{msg\_num}})$ to $\mathsf{keys\_in\_transit}$.
   (c) If $\mathsf{msg\_num\_state}_{\mathsf{aead}}$ doesn't exist, initialize $\mathsf{msg\_num\_state}_{\mathsf{aead}} = \perp$.
   (d) Send $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \mathtt{ReportState}, \mathsf{pid}, k_{\mathsf{msg\_num}}, m)$ to $\mathcal{A}$ on behalf of $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
   (e) On receiving a response $(\mathtt{ReportState}, \mathsf{pid}, S)$, add S to $\mathsf{state}_{\mathsf{pid}}$.
7. **If** there is a record $(\mathtt{StopKeys}, i, N)$ set $\mathsf{stop\_request} = (\mathtt{StopKeys}, i, N)$, **Else** set $\mathsf{stop\_request} = \perp$.
8. Send $(\mathtt{ReportState}, i, \mathsf{keys\_in\_transit}, \mathcal{V}iew_0.\mathsf{curr\_msg\_num}, \mathcal{V}iew_1.\mathsf{curr\_msg\_num}, \mathsf{stop\_request})$. to $\mathcal{A}$ on behalf of $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE\text{”}, \mathsf{sid}.fs))$
9. On receiving a response $(\mathtt{ReportState}, i, \mathsf{state}_{mKE})$ from $\mathcal{A}$, add it to the state $\mathsf{state}_{\mathsf{pid}}$.
10. Output $(\mathsf{state}_{\mathcal{I}}, \mathtt{ReportState}, \mathsf{state}_{\mathsf{pid}})$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

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

1. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathsf{msg\_num} = \mathcal{V}iew_i.\mathsf{curr\_msg\_num} + 1$ and update $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} \mathrel{+}= 1$.
3. If $\mathsf{msg\_num\_state}_{\mathsf{aead}}$ exists in $\mathsf{state}_{\mathcal{I}}$ then parse it otherwise initialize $\mathsf{msg\_num\_state}_{\mathsf{aead}} = \perp$.
4. Send a backdoor message $(\mathtt{RetrieveKey}, \mathsf{pid}, \mathsf{msg\_num})$ to $\mathcal{A}$ on behalf of $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE\text{”}, \mathsf{sid}.fs))$ and await a response $(\mathtt{RetrieveKey}, \mathsf{pid}, k)$. On receiving the response, continue.
5. Send a backdoor message $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \mathtt{ReportState}, \mathsf{pid}, k)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{aead}}$ with $\mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num})$ and await a response.
6. Send a backdoor message $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \mathtt{Encrypt}, \mathsf{pid}, m, N)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{aead}}$ with $\mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num})$.
7. On receiving a response $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \mathtt{Encrypt}, \mathsf{pid}, c)$ from $\mathcal{A}$, store $(\ell = |m|, \mathsf{pid}, \mathsf{ready2decrypt} = \mathit{true}), (c, N, 1)$ in $\mathsf{records}[\mathsf{msg\_num}]$.
8. Update or store for the first time the values $\mathcal{V}iew_0$, $\mathcal{V}iew_1$, $\mathsf{records}$, $\mathsf{msg\_num\_state}_{\mathsf{aead}}$, in $\mathsf{state}_{\mathcal{I}}$
9. Output $(\mathsf{state}_{\mathcal{I}}, \mathtt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, c)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

(The rest of this simulator is in Fig. 33 on Page 79.)

Figure 32: Forward Secure Authenticated Encryption Simulator, $\mathcal{S}_{\mathsf{fsaead}}$

$\mathcal{S}_{\mathsf{fsaead}}$ Continued...

(This simulator begins in Fig. 33 on Page 79.)

`Authenticate`: 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. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Provide input ()
3. Parse the dictionary records from $\mathsf{state}_{\mathcal{I}}$:
   (a) **If** $\mathsf{records}[\mathsf{msg\_num}] = \perp$ then output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
   (b) **Else** get $(\ell, \mathsf{pid}, \mathsf{ready2decrypt})$ and any record of the form (c, N, v) from $\mathsf{records}[\mathsf{msg\_num}]$.
4. **If** ready2decrypt $=$ *false* in the retrieved record, output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
5. **If** there is no record (c, n, v):
   - Send a backdoor message $(\texttt{RetrieveKey}, \mathsf{pid}, \mathsf{msg\_num})$ to $\mathcal{A}$ on behalf of $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE\text{”}, \mathsf{sid}.fs))$ and await a response $(\texttt{RetrieveKey}, \mathsf{pid}, k)$.
   - **If** $k = \perp$ then output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
   - **Else** a backdoor message $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \texttt{ReportState}, \mathsf{pid}, k)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{aead}}$ with $\mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num})$ and await a response.
   - Senda backdoor message $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \texttt{Inject}, \mathsf{pid}, c, N)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{aead}}$ with $\mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num})$.
   - On obtaining the output $(\mathsf{msg\_num\_state}_{\mathsf{aead}}, \texttt{Inject}, \mathsf{pid}, c, N, v)$, append (c, N, v) to $\mathsf{records}[\mathsf{msg\_num}]$ and update records in $\mathsf{state}_{\mathcal{I}}$.
6. **If** $v = 0$ then output $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, c, \mathsf{msg\_num}, N, \perp)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$. **Else** output $(\mathsf{state}_{\mathcal{I}}, \texttt{Authenticate}, \mathsf{pid}, c, \mathsf{msg\_num}, N, v)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

Figure 33: $\mathcal{S}_{\mathsf{fsaead}}$ Continued...

Case 1: No Corruptions Have Occurred

We begin by examining the action of the simulator and internal code when all parties are honest and uncorrupted. We show via induction that the real and ideal worlds remain indistinguishable after each action.

Encrypt. In the real world, $\Pi_{\mathsf{fs\_aead}}$ outsources the encryption of each message to a separate $\mathcal{F}_{\mathsf{aead}}$ instance $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“aead”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$ that is parametrized by the its msg_num. The functionality $\mathcal{F}_{\mathsf{aead}}$ then sends a $(\texttt{RetrieveKey}, \mathsf{pid})$ request to Π_{mKE} to check whether the message key seed for the message is available to party pid. If the message key is available, then the functionality $\mathcal{F}_{\mathsf{aead}}$ runs the internal code $\mathcal{I}_{\mathsf{aead}}$ with message length $|m|$ to produce a ciphertext. Meanwhile, the functionality $\mathcal{F}_{\mathsf{mKE}}$ samples a key at random for each msg_num and provides each key exactly once for each party (up to any `StopKeys` requests made by the parties).

In the ideal world, $\mathcal{F}_{\mathsf{fs\_aead}}$ makes the same checks as $\mathcal{F}_{\mathsf{mKE}}$ to make sure that the key for the message is available to party pid. It then runs $\mathcal{I}_{\mathsf{aead}}$ with message length $|m|$ to choose the ciphertext, just like $\mathcal{F}_{\mathsf{aead}}$ does in the real world.

Decrypt. In the real world, $\Pi_{\mathsf{fs\_aead}}$ again uses the appropriate $\mathcal{F}_{\mathsf{aead}}$ instance for decrypting ciphertexts. In this honest case $\mathcal{F}_{\mathsf{aead}}$ will only ever output the originally encrypted message m

or a failure request. Specifically, it will never output a new message m' that is different in any way to the originally encrypted message m for that $\mathsf{msg\_num}$. The authenticate encryption $\mathcal{F}_{\mathsf{aead}}$ sends a $(\texttt{RetrieveKey}, \mathsf{pid})$ request to Π_{mKE} to check whether the message key seed is available to the requesting party. If Π_{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 $\Pi_{\mathsf{fs\_aead}}$. Otherwise, if the ciphertext is different, then $\mathcal{F}_{\mathsf{aead}}$ runs internal code $\mathcal{I}_{\mathsf{aead}}$ to decide whether the message authenticates. Depending on the output v of the internal code, $\Pi_{\mathsf{fs\_aead}}$ also outputs either the message or a failure notification.

In the ideal world, $\mathcal{F}_{\mathsf{fs\_aead}}$ again makes the same checks as $\mathcal{F}_{\mathsf{mKE}}$ to ensure that the decryption key would have been available to the requester. If the key is available, then to decide whether to successfully decrypt, $\mathcal{F}_{\mathsf{fs\_aead}}$ sends an `Authenticate` request to $\mathcal{I}_{\mathsf{fsaead}}$ just like $\mathcal{F}_{\mathsf{aead}}$ does in the real world.

Case 2: One or Both Parties Have Been Corrupted

Next, we show that the real and ideal worlds remain indistinguishable at the moment of corruption, and also after subsequent encryption and decryption operations are executed.

Corruption. The protocol $\Pi_{\mathsf{fs\_aead}}$, on receiving a `Corrupt` request from Π_{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 receiving 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 Π_{SGNL}.

The ideal functionality, on receiving `Corrupt`, sends a `ReportState` request to $\mathcal{S}_{\mathsf{fsaead}}$ and includes a list of the ciphertexts and messages that are in transit. Since no message key seeds are used in the ideal world, the ideal functionality simply chooses random keys for any messages still in transit to the corrupted party (as $\mathcal{F}_{\mathsf{mKE}}$ would have done) and uses these keys to send the necessary `ReportState` requests to $\mathcal{S}_{\mathsf{mKE}}$ and $\mathcal{S}_{\mathsf{aead}}$. The messages that need to be leaked are simply stored in $\mathcal{F}_{\mathsf{fs\_aead}}$.

Encrypt. In the real world, $\Pi_{\mathsf{fs\_aead}}$ outsources the encryption of each message to a separate $\mathcal{F}_{\mathsf{aead}}$ instance $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“aead”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$ that is parametrized by the its $\mathsf{msg\_num}$. The functionality $\mathcal{F}_{\mathsf{aead}}$ then sends a $(\texttt{RetrieveKey}, \mathsf{pid})$ request to Π_{mKE} to check whether the message key seed for the message is available to party pid. If the message key is available, then the functionality $\mathcal{F}_{\mathsf{aead}}$ send the message m to the simulator $\mathcal{S}_{\mathsf{aead}}$ and allows it to choose the ciphertext c. Meanwhile, the functionality $\mathcal{F}_{\mathsf{mKE}}$ allows the simulator $\mathcal{S}_{\mathsf{mKE}}$ to choose the required key as long as a key for this $\mathsf{msg\_num}$ hasn't already been retrieved and there isn't a stopkeys request for a number smaller than $\mathsf{msg\_num}$.

In the ideal world, $\mathcal{F}_{\mathsf{fs\_aead}}$ also allows $\mathcal{S}_{\mathsf{mKE}}$ to choose the required key under the same conditions. It then sends the message m to the simulator $\mathcal{S}_{\mathsf{aead}}$ to allow it to choose the ciphertext. It is straightforward to validate that these actions look just like in the real world.

Decrypt. In the real world, $\Pi_{\mathsf{fs\_aead}}$ again uses the appropriate $\mathcal{F}_{\mathsf{aead}}$ instance for decrypting ciphertexts. The authenticate encryption $\mathcal{F}_{\mathsf{aead}}$ sends a $(\texttt{RetrieveKey}, \mathsf{pid})$ request to Π_{mKE} to check whether the message key seed is available to the requesting party. If Π_{mKE} says that the key isn't available which happens if the key has already been retrieved or if $(\texttt{StopKeys}, N)$ was run run

for $N <$ msg_num then a failure notification is output. Otherwise, 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 $\Pi_{\mathsf{fs\_aead}}$. Otherwise, if the ciphertext is different, then $\mathcal{F}_{\mathsf{aead}}$ allows the simulator $\mathcal{S}_{\mathsf{aead}}$ decide what the ciphertext decrypts to. Successful decryption is allowed only once even in the case of corruption.

In the ideal world, $\mathcal{F}_{\mathsf{fs\_aead}}$ again makes the same checks as $\mathcal{F}_{\mathsf{mKE}}$ to ensure that the decryption key would have been available to the requester. If the key is unavailable then $\mathcal{F}_{\mathsf{fs\_aead}}$ outputs a failure notification. If the key is available and the ciphertext matches the one at encryption, then again $\mathcal{F}_{\mathsf{fs\_aead}}$ takes an executive decision and outputs the message m that was originally encrypted. Otherwise, to decide how to decrypt, $\mathcal{F}_{\mathsf{fs\_aead}}$ sends an `Inject` request to $\mathcal{S}_{\mathsf{fsaead}}$ just like $\mathcal{F}_{\mathsf{aead}}$ does in the real world.

Both Cases 1 and 2

Finally, we must consider the remaining two methods within the API for forward-secure AEAD: the stop encrypting and stop decrypting methods. Our analysis of these two methods is the same whether any parties have been previously corrupted or not.

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 $(\texttt{StopDecrypting}, N)$ request from above, it sends $(\texttt{StopKeys}, N)$ to the $\mathcal{F}_{\mathsf{mKE}}$ instance for the epoch. Then, $\mathcal{F}_{\mathsf{mKE}}$ makes a note that the epoch is closed for the receiver at message number N. This prevents the receiver (or adversary) from decrypting any messages past N for the epoch.

The ideal world functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ simply mimiks this behavior by directly marking all messages beyond N as inaccessible and returns control to Π_{SGNL}.

In conclusion, $\mathcal{S}_{\mathsf{fsaead}}$ 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. This completes the proof of Theorem 4. □

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

In this section, we prove that the message key exchange protocol Π_{mKE} associated with a a particular epoch, realises the functionality $\mathcal{F}_{\mathsf{mKE}}$ for that epoch in the presence of the long-lived global epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$. As outlined in Section 4, protocol Π_{mKE} (presented in Figure 28 on page 73) 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 symmetric 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.

Before reading the following proof that Π_{mKE} (Figure 34) realises $\mathcal{F}_{\mathsf{mKE}}$ in the presence of $\mathcal{F}_{\mathsf{eKE}}$, one must understand that such a proof may not continue to hold in general when $\mathcal{F}_{\mathsf{eKE}}$ is realised by

some protocol Π_{eKE}. However, as explained in Section 2.2, one can instead prove that Π_{mKE} realises $\mathcal{F}_{\mathsf{mKE}}$ in the presence of $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}}$ for some protocol Π_{eKE} that realises $\mathcal{F}_{\mathsf{eKE}}$. This is the approach that we follow in this section.

Π_{mKE}

The protocol has a party id 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 $(\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}}, \mathsf{sid}.eKE)$ //$\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{I}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.
3. Upon receiving $(\texttt{GetSendingKey}, \mathsf{sending\_chain\_key})$ from $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}}$ 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 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 34: The Message Key Exchange Protocol Π_{mKE}

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

Proof: We construct an ideal-process adversary $\mathcal{S}_{\mathsf{mKE}}$ in the figure on Fig. 35 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 Π_{mKE} (in the presence of $\mathcal{F}_{\mathsf{eKE}} + \mathcal{S}_{\mathsf{eKE}} + \mathcal{I}_{\mathsf{eKE}}$), so that the views of the environment Env are computationally indistinguishable in the real and ideal scenarios.

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

The global session id sid encodes a ciphersuite, including the PRG (used by Π_{mKE}), and the length λ of key seeds.

ReportState: On receiving $(\texttt{ReportState}, i, \mathsf{keys\_in\_transit}, \mathsf{msg\_num}_i, \mathsf{stop\_request})$ from $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE)$ do:

1. If $\mathsf{msg\_num}_i \neq 0$ or $\mathsf{stop\_request} \neq \perp$ output $(\texttt{ReportState}, S = \perp)$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE)$.
   //Note that $\mathcal{F}_{\mathsf{mKE}}$ only gets corrupted if the sender has already initialized it. This means that the sender's $\mathsf{msg\_num}$ will never be 0.
2. **Else If** $\mathsf{msg\_num}_i \neq 0$ (i.e. $\mathsf{stop\_request} = \perp$), initialize $\mathsf{chain\_key} \xleftarrow{\$} \{0,1\}^\lambda$.
   //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.
3. **Else** ($\mathsf{msg\_num}_i = 0$ and $\mathsf{stop\_request} \neq \perp$), send $(\texttt{GetReceivingKey}, \mathsf{epoch\_id})$ to $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}}$. On receiving a response $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key})$ set $\mathsf{chain\_key} = \mathsf{recv\_chain\_key}$.
   //$\left(\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{I}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}}), \mathcal{F}_{\mathsf{lib}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}\right)$
   //If the receiver has not retrieved any keys, we get the chain key from $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}}$ so that it matches the key the sender used in the real world.
4. Initialize a dictionary $\mathsf{seeds} = \{\}$.
5. For $(\mathsf{msg\_num}, k) \in \mathsf{keys\_in\_transit}$ such that $\mathsf{msg\_num} < \mathsf{msg\_num}_i$:
   - Store $\mathsf{seeds}[\mathsf{msg\_num}] = k$
6. Let $\mathsf{curr\_msg\_num} = \mathsf{msg\_num}_i$, and let the $\mathsf{curr\_chain\_key} = \mathsf{chain\_key}$.
7. 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 $\mathsf{seeds}[\mathsf{curr\_msg\_num}] = \mathsf{key\_seed}$.
   - Set $\mathsf{latest\_seed\_num} = \mathsf{curr\_msg\_num}$.
   - Delete $(\mathsf{msg\_num}, k)$ from $\mathsf{keys\_in\_transit}$.
8. Delete local variable $\mathsf{curr\_msg\_num}$.
9. Output $(\texttt{ReportState}, S = \{\mathsf{chain\_key}, \mathsf{seeds}\})$.

RetrieveKey: On receiving $(\texttt{RetrieveKey}, \mathsf{pid}, \mathsf{msg\_num})$ from $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE)$ do:
//This happens only if $\texttt{IsCorrupt?} = \textit{true}$. In particular, $\mathcal{S}_{\mathsf{mKE}}$ has already gotten a $\texttt{ReportState}$ 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 35: Message Key Exchange Simulator, $\mathcal{S}_{\mathsf{mKE}}$

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. 35.

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 $\mathcal{F}_{\mathsf{eKE}} + \mathcal{S}_{\mathsf{eKE}} + \mathcal{I}_{\mathsf{eKE}}$ 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 $\mathsf{chain\_key}$ provided to it during compromise. Since $\mathcal{S}_{\mathsf{mKE}}$ generates future keys just like Π_{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 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 [50].

We circumvent this impossibility by resorting to the random oracle model (again, using ideas from [50]). Specifically, we employ a simple Encrypt-then-MAC scheme [43] 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 Π_{aead} is presented in Figure 36 on page 85.

Π_{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 pid.

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

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 $\mathsf{msg\_key} = \mathsf{k}_{\mathsf{otp}} || \mathsf{k}_{\mathsf{mac}}$, where the $\mathsf{k}_{\mathsf{otp}}$ has length $|m|$, and $\mathsf{k}_{\mathsf{mac}}$ has length λ.
   - Compute ciphertext $c' = \mathsf{k}_{\mathsf{otp}} \oplus m$
   - Compute tag $t = \mathsf{MAC}(\mathsf{k}_{\mathsf{mac}}, (c', \mathsf{sid}.aead, N))$.
   - 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 $\mathsf{msg\_key} = \mathsf{k}_{\mathsf{otp}} || \mathsf{k}_{\mathsf{mac}}$, where the $\mathsf{k}_{\mathsf{otp}}$ has length $|m|$, and $\mathsf{k}_{\mathsf{mac}}$ has length λ.
   - If $\mathsf{Verify}(\mathsf{k}_{\mathsf{mac}}, t, (c', \mathsf{sid}.aead, N)) \neq 1$, then output $(\texttt{Decrypt}, \texttt{Fail})$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.
   - Else (the tag is valid), compute message $m = \mathsf{k}_{\mathsf{otp}} \oplus c'$.
   - 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 $(\texttt{Corrupt}, S = \bot)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$. //$\Pi_{\mathsf{aead}}$ has no persistent state.

Figure 36: The Authenticated Encryption with Associated Data Protocol, Π_{aead}

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

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

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 λ.

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 $\mathsf{msg\_key} = \mathsf{k}_{\mathsf{otp}} || \mathsf{k}_{\mathsf{mac}}$ where $|\mathsf{k}_{\mathsf{otp}}| = \ell$ and $|\mathsf{k}_{\mathsf{mac}}| = \lambda$.
4. Compute ciphertext $c' = \mathsf{k}_{\mathsf{otp}} \oplus m$ and tag $t = \mathsf{MAC}(\mathsf{k}_{\mathsf{mac}}, (c', \mathsf{sid}.aead, N))$.
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 $\mathsf{msg\_key} = \mathsf{k}_{\mathsf{otp}} || \mathsf{k}_{\mathsf{mac}}$ where $|\mathsf{k}_{\mathsf{otp}}| = |m|$ and $|\mathsf{k}_{\mathsf{mac}}| = \lambda$.
5. If $\mathsf{Verify}(\mathsf{k}_{\mathsf{mac}}, t', (c', \mathsf{sid}.aead, N)) = 0$: set $v = \perp$.
6. Else $(\mathsf{Verify}(\mathsf{k}_{\mathsf{mac}}, t', (c', \mathsf{sid}.aead, N)) = 1)$: set $v = m$.
7. Return $(\mathsf{state}_{\mathcal{I}}, v)$.

Figure 37: Internal adversarial code $\mathcal{I}_{\mathsf{aead}}$

Proof:

The ideal-process adversary $\mathcal{S}_{\mathsf{aead}}$ is presented in Figure 38 and the internal adversarial code $\mathcal{I}_{\mathsf{aead}}$ is presented in Figure 37. To demonstrate the validity of $\mathcal{S}_{\mathsf{aead}}$, we show that no environment that has global access to $\mathcal{F}_{\mathsf{pRO}}, \mathcal{F}_{\mathsf{mKE}}^{\Pi_{\mathsf{mKE}}} = (\mathcal{S}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{mKE}}), \mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{I}_{\mathsf{eKE}}, \mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}}), \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{lib}}$, can tell whether it is interacting with Π_{aead}, or else with $\mathcal{F}_{\mathsf{aead}}$, $\mathcal{I}_{\mathsf{aead}}$, 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, 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, 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.

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

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 simulated state $\mathsf{state}_{\mathsf{pid}} = \{k\}$ and store the key k.
2. If $k \neq \perp$ and there is a record $(c = (c', t), \mathsf{msg\_key}, m, N, b)$ in $\mathsf{state}_{\mathcal{I}}$:
   - Parse $\mathsf{msg\_key} = \mathsf{k}_{\mathsf{otp}} || \mathsf{k}_{\mathsf{mac}}$ where $|\mathsf{k}_{\mathsf{otp}}| = |m^*|$.
   - Let ${\mathsf{k}_{\mathsf{otp}}}^* = c' \oplus m^*$ and $\mathsf{msg\_key}^* = {\mathsf{k}_{\mathsf{otp}}}^* || \mathsf{k}_{\mathsf{mac}}$.
   - Send a backdoor message $(\texttt{Program}, k, \mathsf{msg\_key}^*)$ to $\mathcal{F}_{\mathsf{pRO}}$.
   - On receiving $(\texttt{Program})$, continue.
3. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{ReportState}, \mathsf{pid}, \mathsf{state}_{\mathsf{pid}})$ 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 Π_{aead}.

1. If the stored key $k = \perp$, then end the activation.
2. Set $\ell = |m|$.
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 $(\texttt{HashQuery}, \mathsf{msg\_key})$, set $\mathsf{msg\_key} = \mathsf{k}_{\mathsf{otp}} || \mathsf{k}_{\mathsf{mac}}$ where $|\mathsf{k}_{\mathsf{otp}}| = \ell$ and $|\mathsf{k}_{\mathsf{mac}}| = \lambda$.
5. Compute ciphertext $c' = \mathsf{k}_{\mathsf{otp}} \oplus m$ and tag $t = \mathsf{MAC}(\mathsf{k}_{\mathsf{mac}}, (c', \mathsf{sid}.aead, N))$.
6. Finally, set $c = (c', t)$ and record $(\mathsf{msg\_key}, m, c, N)$ in $\mathsf{state}_{\mathcal{I}}$.
7. 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 Π_{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 $\mathsf{msg\_key}' = \mathsf{k}_{\mathsf{otp}} || \mathsf{k}_{\mathsf{mac}}$ where $|\mathsf{k}_{\mathsf{otp}}| = \ell$, $|\mathsf{k}_{\mathsf{mac}}| = \lambda$.
6. If $\mathsf{Verify}(\mathsf{k}_{\mathsf{mac}}, t', (c', \mathsf{sid}.aead, N)) = 0$: set $v = \perp$.
7. Else $(\mathsf{Verify}(\mathsf{k}_{\mathsf{mac}}, t', (c', \mathsf{sid}.aead, N)) = 1)$: compute $v = \mathsf{k}_{\mathsf{otp}} \oplus c'$.
8. Output $(\mathsf{state}_{\mathcal{I}}, \texttt{Inject}, \mathsf{pid}, c, N, v)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$.

Figure 38: 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 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 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 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 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 Forge_2.

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

Lemma 18 *If neither* Forge *nor* Collision *events happen during the executions, then the the simulation by* $\mathcal{S}_{\mathsf{aead}}$ *and* $\mathcal{I}_{\mathsf{aead}}$ *is perfect.*

Proof: Assume that the bad events Forge and Collision do not occur during the executions. We will prove that, for all environments Env and adversaries $\mathcal{A}$, the functionality $\mathcal{F}_{\mathsf{aead}}$ together with simulator $\mathcal{S}_{\mathsf{aead}}$ and internal adversarial code $\mathcal{I}_{\mathsf{aead}}$ *perfectly* simulates the real-world views of Env and $\mathcal{A}$ when they interact with Π_{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, Π_{aead} retrieves the message key seed from Π_{mKE}, and if the key is available, Π_{aead} queries $\mathcal{F}_{\mathsf{pRO}}$ to get the expanded msg_key. If the tag t verifies, then it decrypts the ciphertext using msg_key; otherwise, it returns a failure message. If the adversary has compromised

the state of Π_{aead}, then Π_{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 Π_{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 $\mathcal{F}_{\mathsf{aead}}$ either runs the internal adversarial module $\mathcal{I}_{\mathsf{aead}}$ on $(\texttt{Authenticate}, c^*)$ or calls the adversary om $(\texttt{Inject}, c^*)$.

If it is not corrupted then $\mathcal{F}_{\mathsf{aead}}$ runs the internal adversarial code to see if it wants to authenticate this ciphertext c^*, and waits for a response value. ($\mathcal{I}_{\mathsf{aead}}$ 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 Forge_1 and 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, $\mathcal{F}_{\mathsf{aead}}$ calls $\mathcal{S}_{\mathsf{aead}}$ which 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 Π_{aead} has no persistent state besides its 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 Π_{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 Forge and $\mathsf{Collision}$ do not occur. Notice that before corruptions the message key seeds provided by F^{Π}_{mKE} are uniformly random, so event Forge corresponds to the standard security game for message authentication codes. In particular, assuming that we have an environment that successfully induces a Forge event with non-negligible probability, we can construct a forger against $(\mathsf{MAC}, \mathsf{Verify})$, breaking our assumption. Thus, Forge happens with negligible probability.

Next, we discuss the $\mathsf{Collision}$ event. Since the key seeds provided by F^{Π}_{mKE} are uniformly random and independent, and the space of inputs to $\mathcal{F}_{\mathsf{pRO}}$ is exponential in the security parameter δ, 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 δ, 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., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO'93. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (Aug 1994)

[9] 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)

[10] 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)

[11] 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)

[12] 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

[13] Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the Signal double ratchet algorithm. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part I. LNCS, vol. 13507, pp. 784–813. Springer, Heidelberg (Aug 2022)

[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] 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)

[18] 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

[19] 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)

[20] 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)

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

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

[23] 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)

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

[25] Canetti, R., Jain, P., Swanberg, M., Varia, M.: Universally composable end-to-end secure messaging. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part II. LNCS, vol. 13508, pp. 3–33. Springer, Heidelberg (Aug 2022)

[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] 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

[44] 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)

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

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

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

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

[49] 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)

[50] 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)

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

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

[53] 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)

[54] 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)

[55] 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)

[56] 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)

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

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

[59] 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)

[60] 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)

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

[62] 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)

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

[64] 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)