Universally Composable End-to-End Secure Messaging

Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia

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

March 22, 2022

Abstract. We provide a full-fledged security analysis of the Signal end-to-end messaging protocol within the UC framework. In particular:

- We formulate an ideal functionality that captures end-to-end secure messaging, in a setting with PKI and an untrusted server, against an adversary that has full control over the network and can adaptively and momentarily compromise parties at any time and obtain their entire internal states. In particular our analysis captures the forward and backwards secrecy properties of Signal and the conditions under which they break.
- We model the various components of Signal (PKI and long-term keys, backbone "asymmetric ratchet," epoch-level symmetric ratchets, authenticated encryption) as individual ideal functionalities that are analyzed separately and then composed using the UC and Global-State UC theorems.
- We use the Random Oracle Model to model non-committing encryption for arbitrary-length messages, but the rest of the analysis is in the plain model based on standard primitives. In particular, we show how to realize Signal's key derivation functions in the standard model, from generic components, and under minimalistic cryptographic assumptions.

Our analysis improves on previous ones in the guarantees it provides, in its relaxed security assumptions, and in its modularity. We also uncover some weaknesses of Signal that were not previously discussed.

Our modeling differs from previous UC models of secure communication in that the protocol is modeled as a set of *local* algorithms, keeping the communication network completely out of scope. We also make extensive, layered use of global-state composition within the plain UC framework. These innovations may be of separate interest.

[*] 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.

Table of Contents

List of Figures

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 to 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 large periods of time, in 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 [57] (built on top of predecessors like Off-The-Record [17]) has been designed to give a response to these specific challenges of secure messaging. In doing so it has also revolutionized the concept of secure communication over the Internet in many ways. Indeed, the end-to-end encrypted (e2ee) Signal protocol is used to transmit hundreds of billions of messages per day [61].

However, modeling the requirements of secure messaging in general, and analyzing the security properties of Signal in particular, have proved to be challenging. One of the main sticking points has been 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 backwards security (i.e., preventing an attacker from learning past messages), as well as forwards security (i.e., 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 authentication and 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.

Many works have risen to this challenge and provide security analyses of different aspects of the protocol [1, 3, 4, 7, 11, 12, 15, 20, 22, 31–37, 39–43, 58–60, 62–64, 66, 68]. Some of these works formally analyze the Signal protocol as-is, whereas other works propose modifications to Signal that provide additional security guarantees. However, the intricacy and apparent circularity of the internals of the procotol have forced analytical works to either make relatively strong and non-standard security assumptions on the modules in use, or else punt on analyzing some components and instead model them as abstract random functions.

Furthermore, many of these works provide game-based definitions that capture specific aspects of the protocol but do not attempt to provide a complete account of the security of the protocol. In contrast, the Signal messaging protocol is rarely used on its own. For instance, it forms the basis of modern online chat services like WhatsApp [67], file sharing services like Keybase [44], and videoconferencing services like Zoom [48]. As a result, standalone security analyses of the Signal protocol are not easy to work with when attempting to capture the security of an entire messaging ecosystem where the Signal protocol is but one component. (Consider for instance the typical situation in which people are participating concurrently in several conversations spanning several services, jointly processing information coming from the different services, etc.) Indeed, the subtle distinctions between a chat service and the underlying e2ee messaging protocol have been shown to lead to security concerns based on, e.g., the use of read receipts [53] or the lack of security awareness when healing from a compromise [37, 39].

This state of affairs seems to call for 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 and Maurer [43], which defines, within the Constructive Cryptography framework [54,55], an abstract "ratcheting" service that is inspired by Signal's main "continuous key exchange plus authenticated encryption" component. However, it is not clear how to use that modeling to either argue about the security of applications that use Signal, nor how to realize Signal (or their service) from simpler components. Very recently, Bienstock et al [13] formulate, within the UC framework [23, 24], an ideal functionality that aims at capturing the security guarantees provided by actual Signal and highlights a number of important points of weakness of existing game-based analysis of its underlying double ratchet mechanism. They also show that the general structure of Signal realizes their functionality, and analyze the security of some significant improvements over the current design. However they too do not take advantage of the framework to modularize the analysis itself, and instead "break" the circularity in Signal's design by modeling Signal's continuous key derivation function as a random oracle.

1.1 Contributions of This Work

We provide fully modular modeling of Signal-style end-to-end pairwise secure messaging within the UC security framework. On the one hand, our modeling can be used as a basis for analyzing the security of applications that use Signal or other secure messaging protocols. On the other hand, we decompose Signal to smaller modules in a way that breaks the circularity of the security dependence, and show how to realize the various components based on minimal cryptographic assumptions, and with only minimal and "localized" use of the random oracle abstraction. More specifically:

- We provide an ideal functionality $\mathcal{F}_{\mathsf{SM}}$ that captures end-to-end secure messaging (with some Signal-specific caveats, see below). Our formulation provides a very simple interface with the user, namely encapsulating a message for sending and decapsulating received messages, along with idealized security guarantees. Our intention is for $\mathcal{F}_{\mathsf{SM}}$ to be readily usable as a component within other protocols.
- We decompose the Signal protocol into more basic modules, formalized as ideal functionalities within the UC framework. Our decomposition is inspired by (but not identical to) the abstractions within the Signal protocol specification [57].
- We use the above decomposition, along with the UC and UC with joint state theorems [5, 24], to show that the Signal messaging protocol (with a few small changes) realizes our ideal functionality $\mathcal{F}_{\mathsf{SM}}$ against adaptive and transient attackers. In particular, we make crucial use of modularity to demonstrate how the apparent circular security dependence with the components of Signal can be resolved while still using standard primitives.
- We pinpoint the conditions under which the security guarantees of Signal break and an adversary can successfully "fork" a Signal session into two sessions where in each session one of the parties interacts with the adversary, whereas both parties still think that they are interacting with each other.
- In the process, we provide a new modeling for Signal's continuous key derivation protocol. We call the new primitive Cascaded PRF-PRG (CPRFG), and show that it suffices for Signal to achieve adaptive security. We also show how to construct CPRFGs from PRGs and puncturable PRFs. This building block may be of independent interest.

Our modeling and use of the UC framework involves some additional innovations that might be of independent interest:

- We model messaging applications (and, in particular, the Signal protocol) as local algorithms that encapsulate outgoing messages and decapsulate incoming encapsulated messages, henceforth leaving out of scope the mechanism of transmitting encapsulated messages from one party to the other. This provides substantial simplification relative to the traditional UC modeling of secure communication, where the communication medium is modeled as part of the service provided by the protocol.
- We make extensive use of universal composition with global subroutines. This allows for finer-grain de-composition of the Signal protocol to smaller components that are well aligned with the original design. Specifically, we provide single-instance analysis of the module for encrypting and decrypting a single message ($\mathcal{F}_{\mathsf{aead}}$), while letting all modules obtain their keys from a single idealized message key

generation module (Π_{mKE}). At the same time we provide single-instance analysis of Π_{mKE} while having multiple instances of Π_{mKE} obtain epoch keys from a single epoch key exchange module ($\mathcal{F}_{\mathsf{eKE}}$), etc.

- We make only minimal use of the random oracle abstraction (ROM). Specifically, we use the ROM only within protocol Π_{aead} that provides simple symmetric authenticated encryption and decryption of an individual message. Furthermore, we use the ROM only for the inevitable task [56] of making that protocol non-committing so as to withstand adaptive corruptions in the case where the messages are longer than the keys. In particular, in a setting where the number and lengths of the plaintext messages per epoch is fixed and known, our analysis can be set in the plain model.

In Sections 1.2 and 1.3 we provide additional details on our modeling and realization of $\mathcal{F}_{\mathsf{SM}}$.

1.2 On the ideal secure messaging functionality, $\mathcal{F}_{\mathsf{SM}}$

We highlight some additional properties of our modeling of ideal secure messaging.

Modeling of PKI and long term keys. We directly model Signal's specific design for public keys and associated long and medium term secret keys that are used to identify parties across multiple sessions. Specifically, we formulate a "PKI" functionality that stores the long-term, ephemeral, and one-time public keys associated with party identities, as well as a "long term private key module" for each party identity, that stores the private keys associated with the public keys of that 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.

Immediate decryption with efficient message loss resilience [1]. To promote asynchronous communication, the recipient of a message can process it immediately (i.e., without interacting with an external entity such as the sender or some server). To ensure this property, $\mathcal{F}_{\mathsf{SM}}$ responds to decapsulation requests immediately, without interacting with other entities (such as the adversary). Furthermore, this holds even if only a subset of the messages arrive, and arrival is out of order.

Forward and post-compromise security [36]. To facilitate the analysis of recovery from temporary compromise, we model party corruptions as momentary events where the adversary obtains access to all the messages that have been sent to that party and were not yet received. In addition, the party is marked as compromised. While compromised, all the messages sent and received by the party are disclosed to the adversary. Still, all other messages, including the messages sent and received by the party until the point of corruption, remain hidden even after corruption. Furthermore, the adversary obtains no other information on the history of the session such as its duration or the long term identity of the peer.

The point by which a compromised party becomes uncompromised is Signal-specific, and is determined by inspecting the sequence of encapsulation and decapsulation activations of that two parties. More specifically, Signal partitions the messages sent in a session into *sending epochs,* where each sending epoch is associated with one of the two parties, and consists of all the messages sent by that party from the end of its previous sending epoch until the first time this party successfully decapsulates an incoming message that belongs to the peer's latest sending epoch. A compromised party becomes uncompromised as soon as it has started *two* new sending epoch since the last corruption event.

We stress that securely realizing $\mathcal{F}_{\mathsf{SM}}$ guarantees full simulatability for the realizing protocol against an adversary that adaptively corrupts parties as the computation proceeds [63].

Authentication and security awareness [14, 39, 42]. As long as both parties are uncompromised, both are guaranteed that they only accept messages that were sent by the legitimate peers. However, the adversary is allowed to cause a party to accept arbitrary messages as coming from its peer, as long as these messages are associated with an epoch where *either the party or its peer* is compromised Furthermore, $\mathcal{F}_{\mathsf{SM}}$ does not give parties a way to detect whether their peers have received forged messages in their name while they were corrupted. This represents a known weakness of Signal [20, 37].

Session forking. As remarked in [57], when one of the parties is compromised, Signal allows an adversary to "fork" a messaging session. That is, the adversary can create a situation where both parties believe they are talking with each other in a joint session, and yet they are 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 [26].) Essentially, $\mathcal{F}_{\mathsf{SM}}$ 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's different than the one used by the sender. In that case, $\mathcal{F}_{\mathsf{SM}}$ remains forked indefinitely, without any additional corruptions.)

1.3 Realizing $\mathcal{F}_{\mathsf{SM}}$, Modularly

We overview our decomposition of Signal to individual components, highlighting the main insights and the differences from the Signal specification [57].

Modular design of secure messaging. As described by Alwen et al [1], the Signal protocol has two main components: symmetric authenticated encryption with associated data (AEAD), which is applied to individual messages, and a key evolution or *ratcheting* mechanism. The protocol is designed so that each key is used to protect at most one message. To help keep the parties in synch 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.

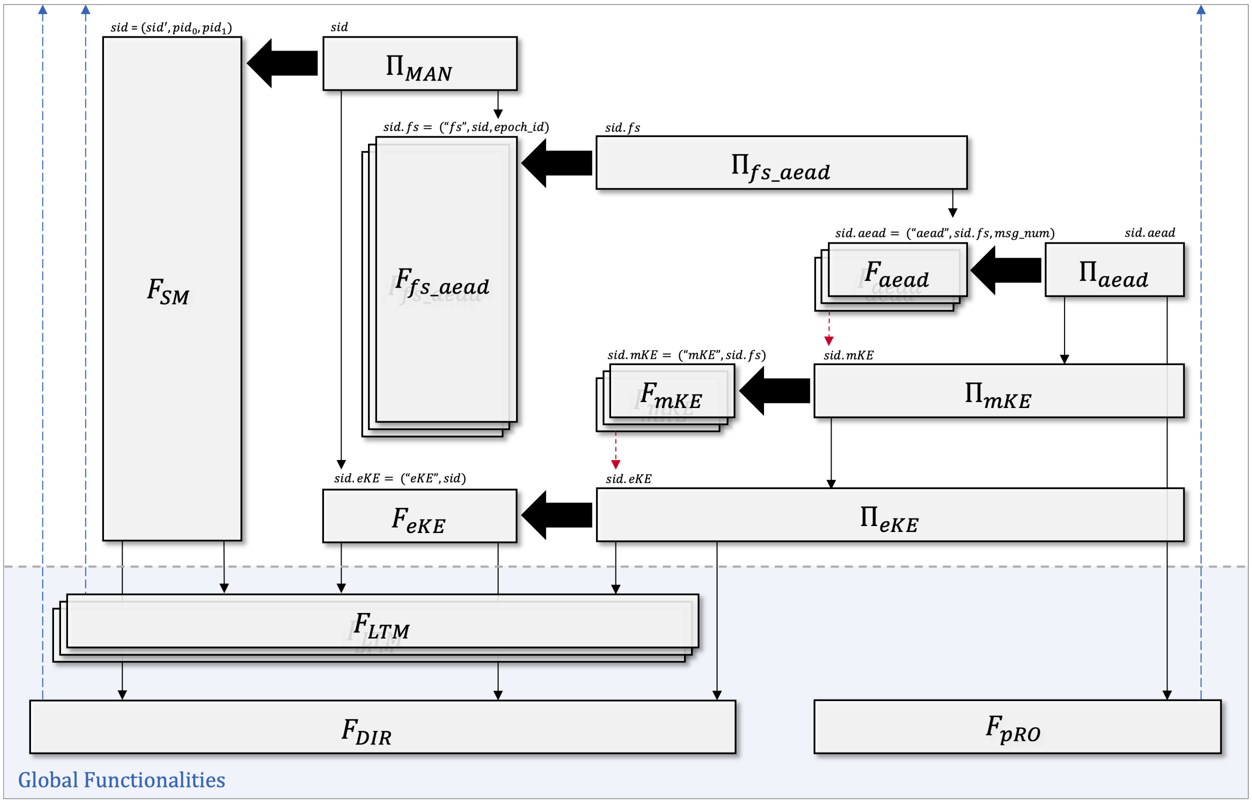
Now, within each sending epoch, the keys are generated one after the other, along a simple hash (or, pseudorandom generation) chain, without any forward-security mechanisms. The initial chaining key for each epoch is generated from a "trunk" chain that ratchets forward every time a new sending epoch starts. Each ratcheting of the trunk chain involves an additional Diffie-Hellman key exchange that's piggibacked on the message and authenticated using the same AEAD used for the data; the resulting Diffe-Hellman secret is then used as input to the ratcheting (along with an existing chaining value). Furthermore, the public values of each such Diffie-Hellman exchange are used as unique identifiers of the sending epoch that a message is part of. This mechanism allows the parties to keep in synch without storing any long-term information about the history of the session.

This "continuous Diffie Hellman" is the main tool that guarantees forward and backwards secrecy, as well as recovery from past key compromise. However, it also creates a circular security dependence: the first message in each new sending epoch needs to be authenticated (by the AEAD in use) using a key *k that's derived from the message itself.* Thus modular security analysis along the above partitioning to modules may not appear possible. Still, we are able to do so, as follows (see 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 "trunk 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 the epoch identifier, and is then given the associated secret key.

To break the above circularity and formulate $\mathcal{F}_{\mathsf{eKE}}$ as a stand alone functionality that's realizable without the use of the authenticated encryption, we allow the receiving party of a new epoch to present multiple potential epoch identifiers, and obtain a secret epoch key associated with each one of these identifiers. Furthermore, while only one of these keys is the one used by the sender for this epoch, all the keys provided by $\mathcal{F}_{\mathsf{eKE}}$ are guaranteed to appear random and independent to the adversary. In other words, $\mathcal{F}_{\mathsf{eKE}}$ leaves it to the receiver to determine which of the candidate identifiers for the new epoch is the correct one. (If $\mathcal{F}_{\mathsf{eKE}}$ recognizes, from observing the corruption activity and the generated epoch IDs, that the session has forked, then it exposes the secret keys to the adversary.)

Creating multiple alternative keys for each epoch, while preserving simulatability and adaptive corruptions is indeed more challenging. Still, the lack of circularity allows us to realize $\mathcal{F}_{\mathsf{eKE}}$ via a protocol, Π_{eKE},



$sid = (sid', pid_0, pid_1)$
sid
Π_{MAN}
$sid.fs = (\text{"}fs\text{"}, sid, epoch\_id)$
$sid.fs$
Π_{fs_aead}
F_{SM}
F_{fs_aead}
$sid.aead = (\text{"}aead\text{"}, sid.fs, msg\_num)$
$sid.aead$
F_{aead}
Π_{aead}
$sid.mKE = (\text{"}mKE\text{"}, sid.fs)$
$sid.mKE$
F_{mKE}
Π_{mKE}
$sid.eKE = (\text{"}eKE\text{"}, sid)$
$sid.eKE$
F_{eKE}
Π_{eKE}
F_{LTM}
F_{DIR}
F_{PRO}
Global Functionalities

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

that uses only basic cryptographic primitivies. We do this by way of a new primitive, Cascaded PRF-PRG. (Formally, Π_{eKE} UC-realizes $\mathcal{F}_{\mathsf{eKE}}$ in the presence of global functionalities $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$ discussed above.)

$\mathcal{F}_{\mathsf{mKE}}$. The per-epoch key chain is captured by an ideal functionality $\mathcal{F}_{\mathsf{mKE}}$ that generates a sequence of random keys, one at a time. 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 (each key is retrievable at most once by each party and becomes irrecoverable upon retrieval, even for a corrupted party), but no backwards secrecy (once corrupted, all the future keys in the sequence are exposed to the adversary.)

$\mathcal{F}_{\mathsf{mKE}}$ is realized by Π_{mKE}, which employs a straighforward key chaining using a length-doubling PRG, starting from the initial chain key obtained from $\mathcal{F}_{\mathsf{eKE}}$. Demonstrating that Π_{mKE} realizes $\mathcal{F}_{\mathsf{mKE}}$ requires some care, given that $\mathcal{F}_{\mathsf{eKE}}$ is used by multiple instance of Π_{mKE}. Furthermore, as we'll see, we may have multiple instances of Π_{mKE} ask $\mathcal{F}_{\mathsf{eKE}}$ for keys associated with different identifiers, all associated with the same epoch. We address such situations by demonstrating that Π_{mKE} UC-realizes $\mathcal{F}_{\mathsf{mKE}}$ in the presence of $\mathcal{F}_{\mathsf{eKE}}$, where $\mathcal{F}_{\mathsf{eKE}}$ serves as a global functionality. This guarantees that each instance of Π_{mKE} generates keys that are effectively independent from the keys generated by all other instances of Π_{mKE}.

$\mathcal{F}_{\mathsf{aead}}$. 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 $\mathcal{F}_{\mathsf{mKE}}$. Had we only needed to assert security against non-adaptive corruptions, any standard AEAD scheme would do. However, we need to provide simulation-based security in the presence of adaptive corruptions, which is provably impossible in the plain model whenever the key is shorter than the plaintext [56]. We get around this issue by providing a simple AEAD scheme which UC-realizes $\mathcal{F}_{\mathsf{aead}}$ in the programmable random oracle model. 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 $\mathcal{F}_{\mathsf{mKE}}$.

$\mathcal{F}_{\mathsf{fs\_aead}}$. Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ is an abstraction of the module that handles the encapsulation and decapsulation of 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. (Indeed, it is this backwards security property that justifies treating $\mathcal{F}_{\mathsf{fs\_aead}}$ as a separate module.)

$\mathcal{F}_{\mathsf{fs\_aead}}$ is realized 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 incapsulated 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.)

Modeling corruptions. Resilience to recurring but transient break-ins is one of the main design goals of Signal. We facilitate the exposition of these properties as follows. First, we model corruption as an instantaneous event where the adversary learns the entire state of the corrupted party.[1] Next, we make explicit how the behavior of $\mathcal{F}_{\mathsf{SM}}$ and each one of the building blocks depends, at each point in time, on when was the last corruption event at each party.

We also make sure that our modular analysis is consistent with party-wise corruptions. That is, we model corruptions in such a way that allows analyzing each module separately, with respect to corruption of this module alone, while at the same time guaranteeing that the composite protocol exhibits the expected behavior with respect to corruptions of all modules at the same time. Specifically, our modeling of corruption make sure that all the components of a single party are corrupted together. (An alternative modeling, where different modules can be corrupted while others remain uncorrupted, might be of value - say, when different components run on different physical computers. However we leave this extension out of scope.)

[1] We don't directly model "Byzantine" corruptions, where the adversary is allowed to destroy or modify the state or program of the corrupted party. Indeed, in our setting where the adversary has complete control over the communication, Byzantine corruption of one party does not help the adversary in compromising the security of the peer. On the other hand, analyzing the ability to recover from break-ins becomes significantly more complex in such situations; in particular, Signal does not appear to have been designed to enable recovery in such extreme cases.

Composition with global state. As discussed above, our analysis universal composition with global state [5], which allows demonstrating that protocol π UC-realizes functionality $\mathcal{F}$ in the presence of other functionality $\mathcal{G}$, where $\mathcal{G}$ is 'global', in the sense that it takes inputs from π, $\mathcal{F}$, and also potentially directly from the environment. Furthermore, we use *multiple levels* of UC with global subroutines. In particular, after showing that π UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}$, we wish to argue that π UC-realizes $\mathcal{F}$ in the presence of protocol γ, where γ is some protocol that UC-realizes $\mathcal{G}$. While such implication is not true in general [6, 30], we demonstrate that does indeed hold for the protocols considered in this work.

Altogether, we show that $\mathit{\Pi}_{\mathsf{SGNL}}$, along with $\mathcal{F}_{\mathsf{eKE}}$ and multiple insances of $\mathcal{F}_{\mathsf{fs\_aead}}$, UC realizes $\mathcal{F}_{\mathsf{SM}}$ in the presence of $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$, and $\mathcal{F}_{\mathsf{pRO}}$.

1.4 Related Work

As mentioned above, Bienstock et. al [13] provide security analysis of Signal within the UC framework. They formulate an "ideal double ratchet" functionality, with a number of variations that capture different levels of security. They also demonstrate several shortcomings of previous formulations, such as overlooking the effect of choosing keys too early or keeping them around for too long. They also propose and analyze an enhancement of the double ratchet structure that helps parties regain security faster following a compromise event.

The ideal functionality of [13] differs from our $\mathcal{F}_{\mathsf{SM}}$ in a number of ways: First, their modeling does not account for the initial setup of sessions; it also includes the communication medium as part of the protocol (thus making it harder to argue about immediate encapsulation and decapsulation). On the other hand, it accounts for adversarial choice of randomness, which our modeling does not account for. We defer further comparison to later versions of this work.

Apart from the works of [13, 43], there is a long history of designing key exchange protocols that form the basis of Signal's double ratchet. Building upon the work of Bellare and Rogaway [9, 10], Canetti and Krawczyk [28,29] introduce a composable analysis of authenticated key exchange (AKE) that combines long-term and ephemeral keys to establish a secure channel. Subsequently, Kudla and Paterson [49], HMQV [47], KEA+ [51], NAXOS [50], and CMQV [65] extend this framework to improve forward security and resilience against key compromise impersonation. Cohn-Gordon et al. [36] provide the first formal treatment of post-compromise security or self-healing, and Cremers et al. [37] strengthen this property against active attackers. Unger and Goldberg [63, 64] introduce deniable AKE for secure messaging, which ensures that an adversary cannot use the network transcript to verify a coerced party's claim about the plaintext messages, and Vatandas et al. [66] analyze the deniability of the Signal protocol.

Recently, several works have analyzed Signal's actual secure channel and key ratcheting protocols. Cohn-Gordon et al. [33, 34] provide the first formal treatment of Signal. Bellare et al. [11] introduce generic game-based definitions of Signal's ratcheting, but only show security only against a corrupted sender. Alwen et al. [1] contribute new game-based definitions that add the immediate decryption property, and constructions from generic crypto primitives. By contrast, our work is the first to consider simulation-based security against an adaptive, arbitrary environment.

Another line of research explores modifications to the Signal protocol. Jaeger-Stepanovs [41] and Poettering and Rösler [58] achieve optimal post-compromise security against both parties, but use public key cryptography that Balli et al. [7] show is necessary. Jost et al. [42], Durak and Vaudenay [39], and Yan and Vaudenay [68] design leaner Signal variants that sacrifice some or all post-compromise security for better efficiency. Blazy et al. [15] splits Signal's directory service into a trusted identity-based signing authority and an untrusted key distribution center. Brendel et al. [19], Drucker and Gueron [38], and Hashimoto et al. [40] propose post-quantum alternatives to Signal's X3DH public ratchet. Chase et al. [31] add anonymity to a server-assisted group messaging system. By contrast, our work attempts to stay as close to the original Signal messaging protocol as possible, with minimal changes as necessary to achieve adaptive, concurrent security. Put another way: while our modular construction could be used to analyze alternatives (e.g., using post-quantum crypto rather than Diffie-Hellman for epoch key exchange), in this work we only instantiate a Signal-like messaging protocol.

Finally, some recent work explores extensions to the basic two-device communication model. Campion et al. [22] construct and analyze a slight variant of Signal's Sesame protocol that supports multiple devices per participant [57]. Several works examine secure group messaging with communication sub-linear in the group size [3,4,32,35,59,60], but Bienstock et al. [12] show that this approach is incompatible with this paper's goal of concurrent security. Finally, Caforio et al. [20] provide a generic construction to add security awareness to any Signal-style protocol. In this work, we incorporate all of these extensions in a modular fashion on top of our point-to-point secure messaging functionality.

2 Universally Composable Security: Some Background

UC security [24] 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 $\mathrm{exec}_{\mathcal{E},\Pi} \approx \mathrm{exec}_{\mathcal{E},\mathcal{F},\mathcal{S}}$ 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 [25]. 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[2].

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 $\rho^{\mathcal{F}\rightarrow\Pi}$, 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 $\rho^{\mathcal{F}\rightarrow\Pi}$, 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.

We make the following additional observation that, while simple, ends up playing a key role in our analysis, and may be of independent interest. Consider protocol Π that UC-realizes an ideal functionality $\mathcal{F}$, and assume that we want to assert that protocol ρ UC-realizes Γ in the presence of Π. Can we use the fact that Π UC-realizes $\mathcal{F}$, to simplify this analysis? In general, the answer is negative: the fact that ρ UC-realizes Γ in the presence of $\mathcal{F}$ does *not* in general imply that ρ UC-realizes Γ in the presence of Π (see [6, 30]). Still, we observe that since Π UC-realizes $\mathcal{F}$ there must exist a simulator $\mathcal{S}$ such that no environment can distinguish between an interaction with Π and an interaction with $\mathcal{F}$ and $\mathcal{S}$. Now consider the machine $\mathcal{F}_{\mathcal{S}}$ that represents the combination of $\mathcal{F}$ and $\mathcal{S}$ (where the communication between $\mathcal{F}$ and $\mathcal{S}$ now becomes internal to the combined machine $\mathcal{F}_{\mathcal{S}}$. We observer that Π and $\mathcal{F}_{\mathcal{S}}$ UC-emulate each other. This means that instead of demonstrating that ρ UC-realizes Γ in the presence of Π, it suffices to demonstrate that ρ UC-realizes Γ in the presence of $\mathcal{F}_{\mathcal{S}}$ [6, 30]. That is:

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

3 Modelling Secure Messaging

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

3.1 Global Functionalities

Before describing $\mathcal{F}_{\mathsf{SM}}$ itself, we briefly summarize three global sub-routines that all of the functionalities and protocols in this work rely upon. Our global subroutines 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.

[2] 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.

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

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

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

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

UpdateRotatingKey: On input $(\texttt{UpdateRotatingKey})$ from $(\mathsf{pid}, \mathrm{Root})$, do:

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

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

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

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

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

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

1. If the list $\mathsf{onetime\_keys}_{\mathsf{pid}}$ is empty then output an error message to $(\Pi_{\mathsf{eKE}}, \mathsf{pid})$. Else, delete the one-time key pair $(\mathsf{ok}^{\mathsf{sk}}, \mathsf{ok}^{\mathsf{pk}})$ from the list and compute:
   - $DH_1 = (\mathsf{ik}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}}$
   - $DH_2 = (\mathsf{ek}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ik}^{\mathsf{sk}}_{\mathsf{pid}}}$
   - $DH_3 = (\mathsf{ek}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{rk}^{\mathsf{sk}}_{\mathsf{pid}}}$
   - $DH_4 = (\mathsf{ek}^{\mathsf{pk}}_{\mathsf{partner}})^{\mathsf{ok}^{\mathsf{sk}}}$
2. Output $(\texttt{ComputeReceivingRootKey}, HKDF(DH_1||DH_2||DH_3||DH_4))$.

Fig. 2. The Long-Term Keys Module Functionality, $\mathcal{F}_{\mathsf{LTM}}$

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

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

RecordKeys: On input $(\mathtt{RecordKeys}, \mathsf{pid}, \mathsf{ik}^{\mathsf{pk}}_{\mathsf{pid}}, \mathsf{rk}^{\mathsf{pk}}_{\mathsf{pid}})$ from $(\mathcal{F}_{\mathsf{LTM}}, \mathsf{pid})$ do: //For simplicity $\mathcal{F}_{\mathsf{DIR}}$ records just one public key per 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 $(\mathtt{RecordKeys}, \mathsf{pid}, \mathtt{Success})$ to the caller.

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

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

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

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

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

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

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

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

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

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

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

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 $h_{end} \xleftarrow{\$} \{0,1\}^{\ell(n)-|h|}$, let $h' = h||h_{end}$, and replace the record (m, h) with (m, h').

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

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

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

Fig. 4. The Programamble Random Oracle Functionality, $\mathcal{F}_{\mathsf{pRO}}$

$\mathcal{F}_{\mathsf{LTM}}$ Next, we design a module $\mathcal{F}_{\mathsf{LTM}}$ (fig. 2) 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 [52]. Concretely, each party can generate short-term rotating and one-time keys that limit the period of vulnerability if a long-term key is compromised.

$\mathcal{F}_{\mathsf{pRO}}$ Finally, we design a programmable random oracle module $\mathcal{F}_{\mathsf{pRO}}$ (fig. 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 [21]. Anyone can query the random oracle, but only the (real or ideal world) adversary has the power to program it.

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

This section presents our *secure message functionality* $\mathcal{F}_{\mathsf{SM}}$. The complete details of the functionality can be found in Figure 5. We start with high level description of the functionality and some motivation for specific choices.

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

An instance of $\mathcal{F}_{\mathsf{SM}}$ is created by some party (namely an ITM, or collquially 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 idendity, 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. At the first $(\texttt{SendMessage}, m)$ activation (which is the first activation overall), $\mathcal{F}_{\mathsf{SM}}$ first verifies that the instance of $\mathcal{F}_{\mathsf{LTM}}$ that corresponds to the initiator pid_0 exists, and that the desired peer (pid_1) is registered with $\mathcal{F}_{\mathsf{DIR}}$. It also initiates variables that will record subsequent epoch identifiers and the numeral of each message in its epoch, as well as which party is compromised at each epoch.

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

Furthermore, if the local state of $\mathcal{F}_{\mathsf{SM}}$ indicates that this `SendMessage` activation is the first one in a new epoch, then $\mathcal{F}_{\mathsf{SM}}$ asks the adversary for a new epoch identifier for this epoch, and verifies that the received identifier is different than all previously used ones[3].

[3] In the UC framework, letting the adversary determine values (such as a ciphertext, epoch-ID, or decrypted plaintext) can be done in one of two ways. One way is to directly ask the adversary to provide the value. This requires handing the control of the single-threaded execution over to the adversary, which means leaking the fact that an activity (such as the encapsulation of a message, as here) took place; it also means that the activity (here, the encapsulation process) may be delayed indefinitely by the adversary. Alternatively, the functionality could instead ask the adversary to provide, at first activation, a "proxy" program P. Now, instead of asking the adversary to choose the ciphertext, the functionality will run P with the same information that it would have given the adversary, and take the response of P as the response of the adversary. The same program is used also for decryption requests. Additional aspects of this modeling choice are discussed in [27]. (It is stressed that the interaction with the adversary at corruption time is still performed with the actual adversary, not with the program P).

Currently $\mathcal{F}_{\mathsf{SM}}$, as well as the rest of the functionalities, are formulated as using the first mechanism that has weaker security guarantees. However, the Signal protocol (as well as our rendering of it) can actually be shown

Finally, $\mathcal{F}_{\mathsf{SM}}$ records (m, c, h) where h is the"header information" that includes the epoch identifier $\mathsf{epoch\_id}$ of the message and the message number $\mathsf{msg\_num}$ in the epoch, and outputs (c, h) to pid_0.

It is stressed that, as long as the epoch IDs are unique, no two records of encrypted messages have the same header information. Indeed, uniqueness is the only property that the epoch IDs need to satisfy.

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

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

- If this is the first ReceiveMessage activation, then $\mathcal{F}_{\mathsf{SM}}$ verifies that this request is coming from pid_1, that the instance of $\mathcal{F}_{\mathsf{LTM}}$ that corresponds to pid_1 exists, and that pid_0 is registered with $\mathcal{F}_{\mathsf{DIR}}$.
- If a record (m, c, h) exists then $\mathcal{F}_{\mathsf{SM}}$ returns the corresponding message m to pid *and deletes the record* (m, c, h). (That is, a message is decryptable exactly once.)
- If there exists no record (m, c', h') with $h' = h$ then return $\perp$.
- If there exists a record (m, c', h') where $c' \neq c$ and $h' = h$ (presumably, c' is a "mauled ciphertext") then $\mathcal{F}_{\mathsf{SM}}$ gives the adversary the latitude to decide whether decryption should succeed; still, in case of successful decryption the returned value is m, and the record is deleted. This behavior combines the standard EU-CMA guarantee for the underlying authentication scheme, combined with one-time decryption.
- Finally, if this happens to be the first successfully received message for party pid_i in the newest epoch, then $\mathcal{F}_{\mathsf{SM}}$ notes that the next SendMessage activation with sender pid will start a new epoch.

It is stressed that $\mathcal{F}_{\mathsf{SM}}$ only supports decrypting each header once; it will refuse to participate in subsequent calls with the same h to ensure forward secrecy. In addition, $\mathcal{F}_{\mathsf{SM}}$ stores the number of messages sent in previous epochs so that honest parties can detect if an adversary has injected a message after the planned ending to a particular epoch.

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

Before describing the actual behavior of $\mathcal{F}_{\mathsf{SM}}$ we note two ways in which our modeling of corruptions differs from the traditional. First, corruption is captured as an instantaneous exposure event, as opposed to having separate*corrupt* and *leave* events. (This yields a simpler model while not restricting adversarial capabilities.) Second, corruptions are modeled as inputs (coming from the environment, as opposed to the traditional UC modeling corruption as message coming from the adversary. (This again simplifies the mechanics without restricting the adversarial power.)

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

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

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

to realize the alternative mechanism that hides the fact that processing took place and guarantees immediate completion. We intend to make it an explicit part of the formalism in future versions of this work.

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

Third, if pid_{1-i} calls $\mathcal{F}_{\mathsf{SM}}$ to receive an incoming ciphertext (h, c), where h specifies an epoch ID $\mathsf{epoch\_id}$ that corresponds to a new epoch, whereas party pid_i has already sent messages for that epoch but with a different epoch ID $\mathsf{epoch\_id}'$, *and in addition* pid_i *is compromised (but not recovering),* then the addversary can instruct $\mathcal{F}_{\mathsf{SM}}$ to successfully decrypt (h, c). In that case, $\mathcal{F}_{\mathsf{SM}}$ notes that the session is *forked* (or, in other words, the parties have diverged). In this case, both parties become compromised for the rest of the session. This event represent a complete break of security of the session.

4 Modular Decomposition of $\mathcal{F}_{\mathsf{SM}}$

We provide a brief overview of our modular decomposition in Section 1.3. In this section, we provide more details about our modular, iterative process for decomposing the ideal secure messaging functionality $\mathcal{F}_{\mathsf{SM}}$ into a collection of real protocols that each address one specific purpose. Rigorous UC security analyses are deferred to the Supplementary Material.

4.1 The Double Ratchet

In our first layer, we decompose $\mathcal{F}_{\mathsf{SM}}$ into two components that model the interconnected pieces of the double ratchet: a public key exchange component $\mathcal{F}_{\mathsf{eKE}}$ and a symmetric key authenticated encryption component $\mathcal{F}_{\mathsf{fs\_aead}}$. These components are 'glued' together with a manager protocol Π_{SGNL}. We describe the protocols and functionalities here, and provide a rigorous reduction in Supplementary Material 5.

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

Epoch Key Exchange ($\mathcal{F}_{\mathsf{eKE}}$, fig. 6). The epoch key exchange functionality comprises the public key "backbone" of the secure messaging continuous key agreement. The functionality is persistent during the entire session, mapping $(\mathsf{epoch\_id}_0, \mathsf{epoch\_id}_1)$ pairs to to sending and receiving chain keys for the symmetric ratchet. The protocol (Π_{eKE}, fig. 10) realizes $\mathcal{F}_{\mathsf{eKE}}$ by using continuous Diffie-Hellman key exchange together with a key derivation function (KDF) to produce $\mathsf{chain\_key}$'s for the symmetric ratchet. Critically, the epoch key exchange provides recovery from a state compromise (post-compromise security), as each party generates a fresh Diffie-Hellman exponent to begin a new sending epoch.

Forward Secure Authenticated Encryption ($\mathcal{F}_{\mathsf{fs\_aead}}$, fig. 7). The forward secure authenticated encryption functionality models the symmetric key ratchet for secure messaging. Each $\mathcal{F}_{\mathsf{fs\_aead}}$ instance handles the encryption and decryption of messages for a single epoch. The protocol ($\Pi_{\mathsf{fs\_aead}}$, fig. 13) realizes $\mathcal{F}_{\mathsf{fs\_aead}}$ by outsourcing authenticated encryption and decryption of each message to separate $\mathcal{F}_{\mathsf{aead}}$ instances. Furthermore, $\Pi_{\mathsf{fs\_aead}}$ uses the subfunctionality $\mathcal{F}_{\mathsf{mKE}}$ to handle the key exchange for the symmetric ratchet. As the name suggests, the forward secure encryption functionality (and protocol) provides forward security by deleting a message ciphertext pair after the recipient has successfully decrypted it. Additionally, $\Pi_{\mathsf{fs\_aead}}$ instructs $\mathcal{F}_{\mathsf{mKE}}$ to "close" an old epoch so as to only allow for the decryption of messages still in transit once the parties have moved on to newer epochs.

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

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

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 interpertation of pid_0 and pid_1 as complete identities of the two calling machines.

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

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

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

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

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

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

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

Fig. 5. The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$

4.2 The Symmetric Ratchet: Realizing $\mathcal{F}_{\mathsf{fs\_aead}}$

Next, 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.

Message Key Exchange ($\mathcal{F}_{\mathsf{mKE}}$, fig. 11). Each message key exchange functionality instance handles the key derivation for the symmetric ratchet for a particular epoch. Specifically, it provides $\mathsf{key\_seed}$'s to 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 $\mathsf{key\_seed}$'s up to N and later disallowing the generation of any further key seeds for its epoch. The protocol (Π_{mKE}, fig. 15) realizes $\mathcal{F}_{\mathsf{mKE}}$ by iteratively applying a length-doubling pseudorandom function to the $\mathsf{chain\_key}$ provided by Π_{eKE} to generate $\mathsf{key\_seed}$'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 $\mathsf{key\_seed}$, meanwhile storing intermediate $\mathsf{key\_seed}$'s. To close an epoch at a particular message number N, it generates all message key seeds up to N and then deletes the $\mathsf{chain\_key}$. The functionality (and protocol) enforces the forward security guarantee by: deleting key seeds for messages that have been retrieved, ensuring that message keys look independent and random from each other (using the PRG), and deleting the $\mathsf{chain\_key}$ when the parties have moved on to future epochs (to prevent message injection in old epochs).

Authenticated Encryption with Associated Data ($\mathcal{F}_{\mathsf{aead}}$, fig. 12). Each authenticated encryption functionality instance handles the encryption, decryption, and authentication of a particular message for a particular epoch and hands the ciphertext or message back to $\Pi_{\mathsf{fs\_aead}}$. It gets a $\mathsf{key\_seed}$ from 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. 17) realizes $\mathcal{F}_{\mathsf{aead}}$ by querying the random oracle $\mathcal{F}_{\mathsf{pRO}}$ on the $\mathsf{key\_seed}$ to get the full $\mathsf{msg\_key}$. It then computes the ciphertext c using a One Time Pad and then a secure message authentication code to authenticate c as well as its sid (which contains information about the pid's, $\mathsf{epoch\_id}$, $\mathsf{msg\_num}$, and such). To decrypt, Π_{aead} similarly gets the $\mathsf{key\_seed}$ 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 Supplementary Material 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 $\mathcal{F}_{\mathsf{LTM}}$, $\mathcal{F}_{\mathsf{pRO}}$, and $\mathcal{F}_{\mathsf{DIR}}$).

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

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. This process is straightforward for some of the functionalities; for instance, $\mathcal{F}_{\mathsf{aead}}$ is a slight variant of the *secure message transmission* functionality $\mathcal{F}_{\mathsf{SMT}}$ that has been analyzed in the original work of Canetti [23] that introduced the UC security framework.

Instantiating $\mathcal{F}_{\mathsf{eKE}}$, on the other hand, is more complex and deserves more attention. The main challenge, as observed by Alwen et al. [1] and others, is that the key derivation module within the public ratchet must maintain security if either of the previous root key or the newly generated ephemeral keys are uncompromised. Alwen et al. formalized this guarantee by way of constructing a new primitive: a PRF-PRG. In this work, we make two important improvements upon this construction. First, we contribute a new construction called a *Cascaded PRF-PRG* that allows for equivocation, in order to maintain security against adaptive adversaries. Second, we provide an instantiation in the plain model (i.e., without random oracles) based on punctured PRFs. While we defer to Supplementary Material 7 many details of our UC analysis of the public ratchet, we discuss the Cascaded PRF-PRG construction in detail in the next section.

5 Top-Level Realization of Ideal Secure Messaging

In this section, we describe the top-level realization of $\mathcal{F}_{\mathsf{SM}}$ by the protocol Π_{SGNL} and its sub-functionalities $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{fs\_aead}}$. We begin by describing the epoch key exchange functionality in Section 5.1, with a formal description on fig. 6. Next, in Section 5.2, we present a description of the forward secure encryption functionality together with a formal description in the figure on fig. 7. In Section 5.3 we present the protocol Π_{SGNL} that UC-realizes $\mathcal{F}_{\mathsf{SM}}$ (together with its sub-functionalities), with a formal description of Π_{SGNL} on fig. 8. Lastly, we prove the UC-realization stated in Theorem 1 by constructing a simulator $\mathcal{S}_{\mathsf{SM}}$, defined on fig. 9.

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

The fundamental security objective of the epoch key exchange functionality (see Figure 6 on page 22) 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.

Each epoch has a sending party pid_i and a receiving party pid_{1-i}. Furthermore, we identify each epoch with the sending party's $\mathsf{epoch\_id}$, a unique adversarially chosen value (in Signal, the $\mathsf{epoch\_id}$ is the sender's public Diffie-Hellman key). An important usability feature of the epoch key exchange functionality 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. The functionality 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 out of band of the $\mathcal{F}_{\mathsf{eKE}}$ protocol. In our top-level realization of $\mathcal{F}_{\mathsf{SM}}$, this verification is done by using $\mathcal{F}_{\mathsf{fs\_aead}}$ to check that the ciphertext and its associated information decrypts (and authenticates) successfully under the new key. If the verification succeeds, pid_i *confirms* the new epoch by replacing the $\mathsf{epoch\_id}$ in which it is currently receiving messages with the new one, and by updating its own randomness via the choice of a new $\mathsf{epoch\_id}$ with which to send any future messages. If, on the other hand, the verification fails, then the party deletes the temporary variables it computed and remains in the same sending and receiving 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. These functionalities are used by the epoch key exchange protocol to provide the identity binding that is. automatically provided by the functionality. The epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$ models the post-compromise behavior of the public ratchet in the Signal protocol, but *without providing any epoch keys* at this level.

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 Π_{mKE} (not present at the top-level) and the adversary $\mathcal{A}$. Inputs coming from other sources are ignored. 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 $(\texttt{ConfirmRcvEpoch}, \mathsf{epoch\_id}^*)$ from $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$ only after a message has been successfully received using $\mathsf{epoch\_id}^*$ in a new receiving epoch. This input triggers the functionality to provide a new adversarially chosen $\mathsf{epoch\_id}$ to the party. On the first activation $\mathcal{F}_{\mathsf{eKE}}$ gets the first receiving epoch id from the global directory $\mathcal{F}_{\mathsf{DIR}}$. For all other epochs, $\mathsf{epoch\_id}^*$ is stored as the current receiving epoch id. Lastly, the calling party's epoch number is incremented, the $\mathsf{epoch\_num}$ is stored in a dictionary keyed by $\mathsf{epoch\_id}$, and the new $\mathsf{epoch\_id}$ is returned to Π_{SGNL} for the calling party.

In the case that the input $\mathsf{epoch\_id}^*$ is *not* the identifier for the previous epoch (for example, if the adversary provided an epoch identifier that resulted in a ciphertext that authenticates), this indicates that the parties' views of the conversation have diverged irreversibly. In the protocol, if the parties' epoch identifiers diverge in this way, their symmetric keys diverge as well and can not be restored; after just a single state compromise, the adversary can do a person-in-the-middle attack against both parties simultaneously forever. Thus, to model this outcome in $\mathcal{F}_{\mathsf{eKE}}$, if $\mathsf{epoch\_id}^*$ is *not* the $\mathsf{epoch\_id}$ for the previous epoch, the variable $\mathsf{diverge\_parties}$ is set to true, which will allow $\mathcal{A}$ to choose the sending and receiving chain keys for both parties for all time. We highlight that this man-in-the-middle attack is always possible in signal's protocol after the adversary sees the state of one of the parties, this is our modelling specifically includes this possibility instead of trying to get around it. We note that even though our proofs assume passive corruptions, the security provided by our modelling to the party who was not corrupted is equivalent even in the case of malicious corruptions.

Get Sending Key As was mentioned, the keys generated by $\mathcal{F}_{\mathsf{eKE}}$ (and Π_{eKE}) are used to derive message keys for the epoch. On receiving a `GetSendingKey` request from Π_{mKE}, $\mathcal{F}_{\mathsf{eKE}}$ samples a uniform $\mathsf{sending\_chain\_key}$, stores it by overwriting the old key value, and returns it to Π_{mKE}. In the case that the parties are diverged or the epoch has been compromised, $\mathcal{F}_{\mathsf{eKE}}$ allows the adversary to choose the $\mathsf{sending\_chain\_key}$.

Note that although our state compromises do not allow the adversary to directly modify a party's state, in the protocol $\mathcal{A}$ could test several $\mathsf{epoch\_id}$'s to find one that results in $\mathsf{chain\_key}$'s restricted to some subset of the keyspace and verify these outputs in compromised epochs. Thus, we model this power by allowing it to simply choose the chain keys of compromised epochs.

Get Receiving Key This method is the receiving party's interface for getting its $\mathsf{recv\_chain\_key}$ for decrypting messages in the epoch. In an honest execution (i.e., when the parties have not diverged and the epoch is not compromised), $\mathcal{F}_{\mathsf{eKE}}$ returns to Π_{mKE} the same $\mathsf{sending\_chain\_key}$ that the sender got–thus preserving the symmetry of the message key exchange keys. On the other hand, if the parties have diverged or they are *about to* diverge (their views of the epoch id's do not match and the next epoch is compromised), $\mathcal{F}_{\mathsf{eKE}}$ allows the adversary to choose the $\mathsf{recv\_chain\_key}$ to give to Π_{mKE}. In the case that the parties have not diverged, and the following epoch is not compromised, but the epoch id's do not match (meaning the adversary naively tampered with the ciphertext), $\mathcal{F}_{\mathsf{eKE}}$ samples a random $\mathsf{recv\_chain\_key}$, records the tampering attempt with an entry $(\mathsf{epoch\_id}, \mathsf{recv\_chain\_key})$ in $\mathsf{receive\_attempts}$, and outputs $\mathsf{recv\_chain\_key}$ to Π_{mKE}. 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); however, in the Signal protocol, if $\mathcal{A}$ corrupts the receiver after naive tampering attempts, the adversary can compute the resulting $\mathsf{recv\_chain\_key}$'s after the fact. We prevent this by using a puncturable KDM and removing the ability to compute chain keys using epochids that failed in the past.

Corrupt On receiving a `Corrupt` notification for a party from Π_{SGNL} above, $\mathcal{F}_{\mathsf{eKE}}$ notes that the current epoch is corrupted, and adds the next 3 epochs to a list of compromised epochs until full post-compromise recovery is achieved. The recovery for a single compromise goes through the following phases: fully compromised (both party's entire $\mathcal{F}_{\mathsf{eKE}}$ states are known), the sender's randomness is updated (upon starting a new epoch), both parties' randomness is updated (upon starting a new epoch). . Lastly, the adversary gets the $\mathsf{chain\_key}$ for the party, and if the receiver is the corrupted party, $\mathcal{A}$ also gets the dictionary of $\mathsf{receive\_attempts}$ containing entries of the form $(\mathsf{epoch\_id}, \mathsf{recv\_chain\_key})$ from times when the adversary attempted a naive tampering for the epoch. Lastly, $\mathcal{A}$ returns some state S and $\mathcal{F}_{\mathsf{eKE}}$ forwards this state up to the calling Π_{SGNL}.

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

The forward secure authenticated encryption functionality (see Figure 7 on page 24) processes encryptions and decryptions for a single epoch (specified in $\mathsf{sid}.fs$) for the manager 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).

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

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.

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

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

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

1. Set i such that $\mathsf{pid} = \mathsf{pid}_i$.
2. If $\texttt{ConfirmReceivingEpoch}$ has never been run successfully (i.e $\mathsf{epoch\_id\_self}_0$ hasn’t been initialized) or $\mathsf{got\_sending\_key}_i = true$, 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 $\mathsf{diverge\_parties} = true$, or $\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$, then:
   - Send backdoor message $(\texttt{GetSendingKey}, i)$ to $\mathcal{A}$
   - On receiving backdoor message $(\texttt{GetSendingKey}, i, K_{\mathsf{send}})$ from $\mathcal{A}$, set $\mathsf{sending\_chain\_key}_i = K_{\mathsf{send}}$.
5. Set $\mathsf{got\_sending\_key}_i = true$ and output $(\texttt{GetSendingKey}, \mathsf{sending\_chain\_key}_i)$.

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

1. If $\mathsf{pid} \notin \{\mathsf{pid}_0, \mathsf{pid}_1\}$ then end this activation. Otherwise, set i such that $\mathsf{pid} = \mathsf{pid}_i$.
2. If $\texttt{ConfirmReceivingEpoch}$ has never been run successfully (i.e $\mathsf{epoch\_id\_self}_0$ hasn’t been initialized) or $\mathsf{sending\_chain\_key}_{1-i}$ has been deleted then end the activation.
3. If this is the first activation:
   - Initialize state variables $\mathsf{root\_key}, \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Parse $\mathsf{epoch\_id} = (\mathsf{epoch\_id}', \mathsf{ek}_j^{\mathsf{pk}}, \mathsf{ok}_{i \leftarrow j}^{\mathsf{pk}})$ 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 $(\texttt{ComputeReceivingRootKey}, k)$, continue.
4. If $\mathsf{diverge\_parties} = true$ or $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_{1-i}$: //Let $\mathcal{A}$ choose key
   - Send $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$ to $\mathcal{A}$
   - Upon receiving $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*)$ from $\mathcal{A}$, set $\mathsf{recv\_chain\_key}_i = \mathsf{recv\_chain\_key}^*$.
   - If $\mathsf{diverge\_parties} = false$ and $\mathsf{epoch\_num}_i + 1 \notin \mathsf{compromised\_epochs}$, add $\mathsf{epoch\_id}$ to $\mathsf{receive\_attempts}[\mathsf{epoch\_num}]$.
5. Else ($\mathsf{diverge\_parties} = false$ and $\mathsf{epoch\_id} = \mathsf{epoch\_id\_self}_{1-i}$), set $\mathsf{recv\_chain\_key}_i = \mathsf{sending\_chain\_key}_{1-i}$ //Expected case
6. Output $(\texttt{GetReceivingKey}, \mathsf{recv\_chain\_key}_i)$ and erase $\mathsf{recv\_chain\_key}_i$.

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

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

Fig. 6. The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$

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

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 messages are leaked to the adversary, since in the Signal protocol the sender does not store any message keys. 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}.

5.3 The Signal Protocol, Π_{SGNL}

Protocol Π_{SGNL} (see Figure 8 on page 26) 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 $\mathcal{F}_{\mathsf{SM}}$).

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

This functionality processes encryptions and decryptions for a *single* epoch and has session id $\mathsf{sid}.fs$ that takes the following format: $\mathsf{sid}.fs = (\textit{"fs\_aead"}, \mathsf{sid} = (\mathsf{sid}', (\mathsf{pid}_0, \mathsf{pid}_1)), \mathsf{epoch\_id})$. Inputs arriving from machines whose identity is neither pid_0 nor pid_1 are ignored. //For notational simplicity we assume some fixed interpertation of pid_0 and pid_1 as complete identities of the two calling machines.

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

1. If this is the first activation then:
   - Let i be such that $\mathsf{pid} = \mathsf{pid}_i$ and initialize $\mathsf{msg\_num} = 0$, sender $b = i$.
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 $\mathtt{IsCorrupt?} = \textit{true}$ then set $\ell = m$, else $\ell = |m|$.
6. Send a backdoor message $(\mathtt{Encrypt}, \mathsf{pid}, \mathsf{msg\_num}, N, \ell)$ to $\mathcal{A}$
7. Upon receiving a response $(\mathtt{Encrypt}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ from $\mathcal{A}$, record $(m, c, \mathsf{msg\_num}, N, \ell)$ and output $(\mathtt{Encrypt}, c)$ to $(\Pi_{MAN}, \mathsf{sid}, \mathsf{pid})$.

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

1. Verify that sid matches the one in the local state and $\mathsf{pid} = \mathsf{pid}_{1-b}$, otherwise end the activation. //end the activation if the decrypt request is not from the receiving party
2. If $\mathsf{msg\_num}$ is set as inaccessible, or there is a record $(\mathtt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$, then output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, \mathtt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
3. Send $(\mathtt{inject}, \mathsf{pid}, c, \mathsf{msg\_num}, N)$ to $\mathcal{A}$. //this notifies the adversary that a decryption is being attempted and allows it to decide on injection/authentication only if it should have that power.
4. On receiving $(\mathtt{inject}, v)$ from $\mathcal{A}$, do:
   - If there is a record $(m, c, \mathsf{msg\_num}, N, \ell)$ then mark $\mathsf{msg\_num}$ as inaccessible and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, m)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
   - If $v = \perp$ record $(\mathtt{Authenticate}, c, \mathsf{msg\_num}, N, 0)$ and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, \mathtt{Fail})$.
   - Else if $\mathtt{IsCorrupt?} == \textit{true}$, mark $\mathsf{msg\_num}$ as inaccessible and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, v)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
   - Else if there is a record $(m, c^*, \mathsf{msg\_num}, N, \ell)$, with a different ciphertext, mark $\mathsf{msg\_num}$ as inaccessible and output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, m)$.
   - Otherwise (there is no such record), output $(\mathtt{Decrypt}, c, \mathsf{msg\_num}, N, \mathtt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

1. If sid doesn't match the one in the local state, if $\mathsf{pid} \neq \mathsf{pid}_b$, 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 $(\mathtt{StopEncrypting}, \mathtt{Success})$.

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

1. If sid doesn't match the one in the local state, $\mathsf{pid} \neq \mathsf{pid}_{1-b}$, 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 $(\mathtt{StopDecrypting}, \mathtt{Success})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

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

Fig. 7. The Forward-Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$

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

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

Corruption The manager has one additional interface, a $\texttt{Corrupt}$ 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, $\mathit{\Pi}_{\mathsf{SGNL}}$ sends $\texttt{Corrupt}$ 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 $\mathit{\Pi}_{\mathsf{SGNL}}$ who forwards the union of their states up to Env.

5.4 Security Analysis

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

Theorem 1. *Protocol $\mathit{\Pi}_{\mathsf{SGNL}}$ (perfectly) UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{SM}}$ in the presence of $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$.*

Proof. We construct an ideal-process adversary $\mathcal{S}_{\mathsf{SM}}$ in the figure on fig. 9 that interacts with functionality $\mathcal{F}_{\mathsf{SM}}$. The objective of $\mathcal{S}_{\mathsf{SM}}$ is to simulate the interactions that would take place between the environment and the manager protocol $\mathit{\Pi}_{\mathsf{SGNL}}$ (together with its subroutine functionalities $\mathcal{F}_{\mathsf{fs\_aead}}$ and $\mathcal{F}_{\mathsf{eKE}}$), so that the views of the environment Env are perfectly identical in the real and ideal scenarios.

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

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

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 $\mathsf{epoch\_id}_{\mathsf{self}} = \bot$, $\mathsf{epoch\_id}_{\mathsf{partner}} = \bot$, $\mathsf{sent\_msg\_num} = 0$, $N_{\mathsf{last}} = 0$.
   - Provide input $(\texttt{ConfirmReceivingEpoch}, \bot)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
   - On receiving $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$, set $\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}$.
   - Initialize a list $\mathsf{receiving\_epochs} = []$.
2. Provide input $(\texttt{Encrypt}, m, N_{\mathsf{last}})$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$, where $\mathsf{sid}.fs = (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{self}})$. //$\mathcal{F}_{\mathsf{fs\_aead}}$ already knows $\mathsf{epoch\_id}$ and $\mathsf{msg\_num}$
3. On receiving $(\texttt{Encrypt}, c, N_{\mathsf{last}})$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$, delete m, increment $\mathsf{sent\_msg\_num} \mathrel{+}= 1$, output $(\texttt{SendMessage}, \mathsf{sid}, h, c)$ to 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 $\mathsf{epoch\_id}_{\mathsf{self}} = \bot$, $\mathsf{epoch\_id}_{\mathsf{partner}} = \bot$, $\mathsf{sent\_msg\_num} = 0$ 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 $\mathsf{epoch\_id} \neq \mathsf{epoch\_id}_{\mathsf{partner}}$: //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 $\texttt{Corrupt}$ interface is not part of the "real" protocol; it is only included for modelling purposes.

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

Fig. 8. The Signal Protocol, Π_{SGNL}

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

SendMessage: On receiving $(\mathtt{SendMessage}, \mathsf{pid}, \ell)$ from $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$ do:

1. Set i such that $\mathsf{pid} = \mathsf{pid}_i$. //this is the identity of the sender
2. If this is the first invocation of $\mathtt{SendMessage}$ do:
   - Initialize $\mathsf{diverge\_parties} = \mathit{false}$, $\mathsf{injectable} = \mathit{false}$, $\mathsf{corrupted\_party} = \perp$, $\mathsf{sent\_msg\_num}_0 = 0$, 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 $\mathsf{injectable\_ids}_1 = \emptyset$.
   - Send $(\mathtt{GenEpochId}, i, \perp)$ to Env (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$).
   - Upon receiving $(\mathtt{GenEpochId}, i, \mathsf{epoch\_id})$ from Env, 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. Send $(\mathtt{Encrypt}, \mathsf{pid}, \mathsf{sent\_msg\_num}_i, \ell)$ to Env (in the name of $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$, where $\mathsf{sid}.fs = (\text{“}\mathit{fs\_aead}\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$). //recall that if a party is corrupted then $\ell = m$, else $\ell = |m|$
5. On receiving $(\mathtt{Encrypt}, \mathsf{pid}, \mathsf{sent\_msg\_num}, c)$ from Env:
   - Add $(h, c) \in \mathsf{sentheaders}$.
   - Send $(\mathtt{SendMessage}, \mathsf{pid}, \mathsf{epoch\_id}, c)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$

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

1. Set i such that $\mathsf{pid} = \mathsf{pid}_i$. //this is the identity of the attempted sender, and pid_{1-i} is the receiver
2. Parse $h = (\mathsf{epoch\_id}, \mathsf{msg\_num}, N)$.
3. Send $(\mathtt{inject}, \mathsf{pid}, \mathsf{msg\_num}, c)$ to Env (in the name of $(\mathcal{F}_{\mathsf{fs\_aead}}$, $\mathsf{sid}.\mathsf{fs})$ where $\mathsf{sid}.fs = (\text{“}\mathit{fs\_aead}\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$).
4. On receiving $(\mathtt{inject}, v)$ from Env, if $(h, c) \notin \mathsf{sentheaders} \wedge (v = \perp \vee (\nexists c^* \text{ s.t. } (h, c^*) \in \mathsf{sentheaders} \wedge \mathsf{diverge\_parties} = \mathit{false} \wedge \mathsf{injectable} = \mathit{false} \wedge \mathsf{epoch\_id} \notin \mathsf{injectable\_ids}))$: output $(\mathtt{inject}, h, c, \perp)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$. //in this case $\mathcal{F}_{\mathsf{SM}}$ should output $\mathtt{Fail}$, otherwise decryption succeeds
5. 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
6. If this is the first successfully received message with this $\mathsf{epoch\_id}$ do: //received first message from the sender's newest epoch
   - Send a message $(\mathtt{GenEpochId}, i, \mathsf{epoch\_id})$ to Env (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$)
   - On receiving $(\mathtt{GenEpochId}, i, \mathsf{epoch\_id}^*)$, add $\mathsf{epoch\_id}^*$ to the stack $\mathsf{sent\_ids}_{1-i}$. //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 $\mathsf{corrupted\_party} = \perp$. //if party i succeeds in establishing a new sending epoch, the adversary can no longer inject
7. Output $(\mathtt{inject}, h, c, v)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$.

ReportState: On receiving $(\mathtt{ReportState}, \mathsf{pid}, \mathsf{pending\_msgs})$ 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 $(\mathtt{ReportState}, i, \mathsf{recv\_chain\_key})$ to Env (in the name of $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$).
5. Upon receiving $(\mathtt{ReportState}, i, S^*)$ 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 $(\mathtt{Corrupt}, \mathsf{pid}, \mathsf{leak})$ to Env (in the name of $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$ where $\mathsf{sid}.fs = (\text{“}\mathit{fs\_aead}\text{”}, \mathsf{sid}, \mathsf{epoch\_id}^*)$.
   - Upon receiving a response $(\mathtt{Corrupt}, \mathsf{pid}, S^*)$ from Env, add S^* to the set S_i.
7. Output $(\mathtt{ReportState}, \mathsf{pid}_i, S_i)$ to $(\mathcal{F}_{\mathsf{SM}}, \mathsf{sid})$.

Fig. 9. Secure Messaging Simulator, $\mathcal{S}_{\mathsf{SM}}$

- If a corruption is requested from the environment to $\mathcal{F}_{\mathsf{SM}}$, then the simulator uses the `ReportState` methods within $\mathcal{F}_{\mathsf{eKE}}$ and $\mathcal{F}_{\mathsf{fs\_aead}}$ to recover the corrupted party's chain key and pending messages in transit, respectively.

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

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

Case 1: neither party is corrupted. The functionality $\mathcal{F}_{\mathsf{SM}}$ notifies the simulator when any message is sent or received, as long as it passes the input validation checks described above. During `ReceiveMessage`, the simulator is given a message header $(\mathsf{epoch\_id}, \mathsf{msg\_num}, N)$ together with a ciphertext c, and it is permitted to attempt to inject a message. In the analogous `ReceiveMessage` routine in the real world, the protocol Π_{SGNL} invokes the specific instance of $\mathcal{F}_{\mathsf{fs\_aead}}$ corresponding to $\mathsf{epoch\_id}$, which then sends a backdoor message asking for the desired message to inject. In the ideal world, the simulator $\mathcal{S}_{\mathsf{SM}}$ sends this exact backdoor message and retrieves a value v. It is straightforward to confirm by inspection that $\mathcal{S}_{\mathsf{SM}}$ matches the input-validation logic used within the real world: v is ignored if c is a valid ciphertext created by a previous invocation to `SendMessage`, and otherwise the message is authenticated if and only if $v \neq \perp$. Finally, if this is the first successfully received message of a new epoch, then the real world protocol Π_{SGNL} invokes $\mathcal{F}_{\mathsf{eKE}}$ to perform a public ratchet; the simulator $\mathcal{F}_{\mathsf{SM}}$ emulates the backdoor message required in order to receive the new $\mathsf{epoch\_id}$ from the environment. Hence, the ideal and emulated worlds move to a new epoch in lockstep. The simulator also records all $\mathsf{epoch_id}$s for later use during `SendMessage`, as described below. It is straightforward to check that the code of $\mathcal{F}_{\mathsf{SM}}$ and 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 simulator receives as input 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 $\mathcal{S}_{\mathsf{SM}}$, it has properly initialized `SendMessage` and is awaiting a ciphertext from $\mathcal{S}_{\mathsf{SM}}$ so it can complete the message transmission. For the corresponding call to `SendMessage` in the real world, 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 sends a backdoor message requesting a ciphertext. Because it knows ℓ and the newest $\mathsf{epoch\_id}$ (from previous calls to `ReceiveMessage`), our simulator $\mathcal{S}_{\mathsf{SM}}$ can also send this backdoor message on behalf of the appropriate instance of $\mathcal{F}_{\mathsf{fs\_aead}}$, and it receives back the desired ciphertext. The views of the environment in both scenarios are the same: this one backdoor message communication, together with a sent message with its desired ciphertext. It is also simple to observe by inspection that internal state variables like $\mathsf{msg\_num}$ and N remain in sync as well.

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

To complete the proof, it only remains to show that corruption and uncorruption happen at the same times in the real and ideal worlds and that the environment receives the same state in response to a `ReportState` command. Corruption is initiated by the environment itself; both 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 tcommands `GetSendingKey` and `GetReceivingKey` within its $\mathcal{F}_{\mathsf{eKE}}$ subroutine, it suffices for the simulator to sample $\mathsf{recv\_chain\_key}$ independently (from the same distribution as $\mathcal{F}_{\mathsf{eKE}}$ does) because 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 at the same time (when the corrupted party successfully begins a new sending epoch and has its message received by the honest recipient), and that $\mathcal{S}_{\mathsf{SM}}$ appropriately tracks when this event occurs using the $\mathsf{injectable}$ flag. This flag governs whether the simulator responds according to Case 1 or Case 2. Finally, divergence is even easier to analyze: in both $\mathcal{F}_{\mathsf{SM}}$ and Π_{SGNL} it occurs when a party receives a message with an $\mathsf{epoch\_id}$ that its partner never sent, the code of $\mathcal{S}_{\mathsf{SM}}$ properly tracks this event using the $\mathsf{diverge\_parties}$ flag and moves to Case 2, and the parties never recover from a divergence in either $\mathcal{F}_{\mathsf{SM}}$ or Π_{SGNL}.

6 The Public Key Ratchet: Realizing Epoch Key Exchange

This section describes protocol Π_{eKE} (Figure 10 on page 33) and shows that it UC-realizes $\mathcal{F}_{\mathsf{eKE}}$ (Figure 6 on page 22). This is done in the presence of the global functionalities $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$, and without using the random oracle abstraction.

Section 6.1 defines and realizes the Cascaded PRF-PRG (CPRFG) primitive, which is used to capture the key derivation module of Π_{eKE}. We then present Π_{eKE} in Section 6.2 and analyze it in Section 6.3.

6.1 Cascaded PRF-PRG (CPRFG)

The protocol Π_{eKE} (fully specified in fig. 10) uses a stateful key derivation module (KDM) that is started off at epoch 0 a random initial state $\mathsf{root\_key}_0$, and provides the following functionality: given a randomizer $\mathsf{root\_input}$, the KDM can either: (a) generate a chaining key $\mathsf{chain\_key}$ (to be used as a chaining key for the current epoch), or (b) advance to the next epoch. In this subsection we define the security requirements from this module and present a candidate construction.

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

1. As long as the state is random and secret and the epoch is not advanced, the KDM should behave like a random function with $\mathsf{root\_input}$ as input: there should be a fresh random $\mathsf{chain\_key}$ associated with each value of $\mathsf{root\_input}$. This is because an adversary that sends multiple bogus ciphertexts that purport to be a first message in a new epoch can cause the receiver to generate multiple new chaining keys for the same epoch.
2. Once the epoch is advanced, the KDM should behave like a fresh random function, independently from all previous ones. This should hold as long as either (a) the current state of the KDM is random and secret, or (b) the randomizer $\mathsf{root\_input}$ is random and secret. Indeed, condition (a) guarantees that an adversary that breaks into a party cannot re-compute keys generated by the party in previous epocs, and condition (b) allows parties to recover from break-ins as soon as they agree on a new secret randomizer $\mathsf{root\_input}$.
3. To demonstrate that adaptive break-ins do not compromise the overall security of the protocol, we would like to demonstrate that the process whereby the adversary first obtains the externally-visible outputs of the KDM when executed within Π_{eKE}, and then obtains the current internal state of the KDM, does not give the adversary any computational advantage whatsoever. To do that, we would like there to exist

a simulator that takes as input a sequence of pairs $(\mathsf{root\_input}_1, \mathsf{chain\_key}_1), ..., (\mathsf{root\_input}_k, \mathsf{chain\_key}_k)$, representing the inputs and resulting chaining keys that were generated by the KDM since the last change of root key, and generates a simulated local state that is consistent with the given sequence.

More concretely, a KDM consists of two algorithms, `Compute` and `Advance`. Algorithm

$$\mathtt{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) = (\mathsf{chain\_key}, \mathsf{root\_key}')$$

is given a state $\mathsf{root\_key}$ and randomizer $\mathsf{root\_input}$, and computes a chaining key $\mathsf{chain\_key}$ and an updated state $\mathsf{root\_key}'$. (It is stressed that this operation does not advance the epoch; still, the local state is updated and the old one is discarded.) Algorithm $\mathtt{Advance}(\mathsf{root\_key}, \mathsf{root\_input}) = \mathsf{root\_key}'$ is given a state $\mathsf{root\_key}$ and randomizer $\mathsf{root\_input}$, and updates the state for a new epoch.

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

In each one of the two executions, the adversary can repeatedly make one of three queries: either $(\mathtt{Compute}, \mathsf{root\_input})$, or $(\mathtt{Advance}, \mathsf{root\_input})$, or $(\mathtt{Compute\text{-}Expose\text{-}Advance}, \mathsf{root\_input})$.

In the real execution, the first two types of queries simply invoke the corresponding algorithm of the KDM, and the third query causes the oracle to return the current state s along with the resulting chain key $\mathsf{chain\_key} = \mathtt{Compute}(s, \mathsf{root\_input})$, followed by an advance of the state.

In the ideal execution, `Compute` applies a random function to $\mathsf{root\_input}$, `Advance` switches to a new random function, and `Compute-Expose-Advance` first returns a random value for $\mathsf{chain\_key}$, and as simulated current state s'. The value s' is generated by the simulator, given $\mathsf{chain\_key}$ and the list of queries made by the adversary since the last `Advance`. Here the simulator needs to be able to generate the simulated state so that it is consistent with all the $\mathsf{root\_input}$'s generated by the adversary and the generated $\mathsf{chain\_key}$. A more formal presentation follows.

Cascaded PRF-PRG Security Game

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

Real game:

- Oracle O is initialized with random state $s \leftarrow \{0,1\}^n$.
- On input $(\mathtt{Compute}, \mathsf{root\_input})$: O runs $\mathtt{Compute}(s, \mathsf{root\_input}) = (\mathsf{chain\_key}, \mathsf{root\_key}')$, outputs $\mathsf{chain\_key}$ and changes state $s = \mathsf{root\_key}'$.
- On input $(\mathtt{Advance}, \mathsf{root\_input})$: O runs $\mathtt{Advance}(s, \mathsf{root\_input}) = \mathsf{root\_key}'$ and changes state $s = \mathsf{root\_key}'$.
- On input $(\mathtt{Compute\text{-}Expose\text{-}Advance}, \mathsf{root\_input})$: O runs $\mathtt{Compute}(s, \mathsf{root\_input}) = (\mathsf{chain\_key}, \mathsf{root\_key}')$, outputs $\mathsf{chain\_key}$ and the old state s, chooses $\mathsf{root\_input}' \leftarrow R_n$ at random, computes $\mathtt{Advance}(s, \mathsf{root\_input}) = (\mathsf{root\_key}')$, and changes state $s = \mathsf{root\_key}'$.

Ideal game:

- Oracle O is initialized with a state consisting of a random function $F : R_n \to \{0,1\}^m$.
- On input $(\mathtt{Compute}, \mathsf{root\_input})$: O outputs $F(\mathsf{root\_input}) = \mathsf{chain\_key}$.
- On input $(\mathtt{Advance}, \mathsf{root\_input})$: O updates its state to a new random function $F : R_n \to \{0,1\}^m$.
- On input $(\mathtt{Compute\text{-}Expose\text{-}Advance}, \mathsf{root\_input})$: O outputs $\mathsf{chain\_key} = F(\mathsf{root\_input})$ and $\mathcal{S}(\mathsf{root\_input}_1, ..., \mathsf{root\_input}_k, \mathsf{chain\_key})$, where $\mathsf{root\_input}_1, ..., \mathsf{root\_input}_k$ are all the queries made by $\mathcal{A}$ since the last `Advance` query. Finally O updates its state to a new random function $F : R_n \to \{0,1\}^m$.

Definition 1 (Cascaded PRF-PRG (CPRFG)). *A KDM $(\mathtt{Compute}, \mathtt{Advance})$ is a* cascaded PRF-PRG (CPRFG) *if there exist polytime algorithm $\mathcal{S}$ such that any polytime oracle machine $\mathcal{A}$ can distinguish between the real and ideal interactions described above only with advantage that's negligible in n.*

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

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

where H_c, H_s are two random functions. That is, $\mathtt{Compute}$ matches a random net chaining key with each $(\mathsf{root\_key}, \mathsf{root\_input})$ pair and does not change the local state. $\mathtt{Advance}$ simply provides a new random state for each $(\mathsf{root\_key}, \mathsf{root\_input})$ pair. The simulator will just provide a random value for $\mathsf{root\_key}$ and program H_c, H_s in the right locations to match.

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

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

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

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

As a second approximation, we allow $\mathtt{Compute\text{-}Expose\text{-}Advance}$ queries, but only immediately after an $\mathtt{Advance}$ query. Furthermore we assume the oracle only exposes the old state s, without exposing the value $\mathsf{chain\_key}$. In this case, plain PRFs may not suffice since the new root key may no longer be pseudorandom when the PRF key is exposed. However, as in [2], this can be easily fixed by making sure that the pseudorandom function family $\{g\}$ is a PRF-PRG, namely $g_a(\cdot)$ is a pseudorandom number generator for a random and public a.

As a third approximation, assume that the oracle does expose the value $\mathsf{chain\_key}$ as part of the response to a $\mathtt{Compute\text{-}Expose\text{-}Advance}$ query. Now the simulator appears to be in a bind: in the real game, the adversary provides $\mathsf{root\_input}$ and obtains $\mathsf{root\_key}, \mathsf{chain\_key}$ such that $\mathtt{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) = \mathsf{chain\_key}$. In contrast, in the ideal game the simulator is given $\mathsf{root\_input}$ and a random $\mathsf{chain\_key}$ and is expected to come up with $\mathsf{root\_key}$ such that $\mathtt{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) = \mathsf{chain\_key}$. We enable this "programmability" property by expanding the state with one-universal hash function applied to the output of the pseudorandom function f. That is, given a function family $\{f\}$ where both the key and output are n bits, construct $\{f'\}$ so that a key for f' is of the form $a = a_1, a_2$ where $|a_1| = |a_2|$ and $f'_{a_1,a_2}(x) = f_{a_1}(x) \oplus a_2$. It is easy to see that f' is programmable as needed, and in addition f' is a PRF as long as f is.

Finally, we consider the full-fledged game, where a $\mathtt{Compute\text{-}Expose\text{-}Advance}$ query can be made after multiple $\mathtt{Compute}$ queries. The construction so far appears inadequate: Indeed, in the real game the adversary first obtains the transcript $(\mathsf{root\_input}_1, f_s(\mathsf{root\_input}_1)), ..., (\mathsf{root\_input}_k, f_s(\mathsf{root\_input}_k))$, and then at a later query obtains the key s. In contrast, in the ideal game the adversary first obtains $(\mathsf{root\_input}_1, r_1)), ..., (\mathsf{root\_input}_k, r_k)$ where $r_1, ..., r_k$ are truly random values, and is then supposed to provide a matching key s — whereas such a key may not even exist.

We get around this difficulty by having $\mathtt{Compute}$, at each generation of an output value $f_s(\mathsf{root\_input})$, update the local state so that the new state consists of two separate values: the value $f_s(\mathsf{root\_input})$, plus a residual value s' that allows evaluating f_s at all points except $\mathsf{root\_input}$, and is computationally independent from the value $f_s(\mathsf{root\_input})$. Specifically, we let f be a *puncturable* PRF (as in [16, 18, 45]), and have $\mathtt{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) = (\mathsf{chain\_key}, (\mathsf{root\_key}', (\mathsf{root\_input}, \mathsf{chain\_key}))$ where $\mathsf{chain\_key} = f_{\mathsf{root\_key}}(\mathsf{root\_input})$ and $\mathsf{root\_key}'$ is the result of puncturing $\mathsf{root\_key}$ at point $\mathsf{root\_input}$.

So, in sum, our final construction has state $\mathsf{root\_key} = (\mathsf{root\_key}_1, \mathsf{root\_key}_2, \mathsf{root\_key}_3)$, and:

$$\begin{aligned}\mathtt{Compute}(\mathsf{root\_key}, \mathsf{root\_input}) &= (\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input}) \oplus \mathsf{root\_key}_2, \mathsf{root\_key}')\\ \mathtt{Advance}(\mathsf{root\_key}, \mathsf{root\_input}) &= g_{\mathsf{root\_key}_3}(\mathsf{root\_input})\end{aligned}$$

where $\{g\}$ is a PRF-PRG, and $\mathsf{root\_key}' = (\mathsf{root\_key}'_1, \mathsf{root\_key}_2, \mathsf{root\_key}_3))$. The values $\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input})$ and $\mathsf{root\_key}'_1$ are computed as follows: Parse $\mathsf{root\_key}_1 = (\widehat{\mathsf{root\_key}}_1, (a_1, b_1), ..., (a_k, b_k))$. Then, if $\mathsf{root\_input} =$

a_i for some i then let $\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input}) = b_i$ and $\mathsf{root\_key}'_1 = \mathsf{root\_key}_1$. Else, let $\widehat{f}_{\mathsf{root\_key}_1}(\mathsf{root\_input}) = f_{\widehat{\mathsf{root\_key}_1}}(\mathsf{root\_input})$, and let $\mathsf{root\_key}'_1 = (\widehat{\mathsf{root\_key}_1}', (a_1, b_1), ..., (a_k, b_k), (\mathsf{root\_input}, f_{\widehat{\mathsf{root\_key}_1}}(\mathsf{root\_input})))$, and $\widehat{\mathsf{root\_key}_1}'$ is the result of puncturing $\widehat{\mathsf{root\_key}_1}$ at point $\mathsf{root\_input}$. Furthermore, once a key is advanced, the old key is erased.

Theorem 2. *Assume that $\{f\}$ is a puncturable PRF and $\{g\}$ is a PRF-PRG. Then the above KDM is a cascaded PRF-PRG.*

Proof. (Sketch.) Consider the following simulator $\mathcal{S}$: Given $(\mathsf{root\_input}_1, r_1)$, ..., $(\mathsf{root\_input}_k, r_k)$, $\mathcal{S}$ chooses random n-bit keys s_0 and s_3, computes $s_2 = r_k \oplus f_{s_0}(\mathsf{root\_input}_k)$, sets $\widehat{s}_1 = s_0$, and for $i = 2..k$ sets $\widehat{s}_i$ as the result of puncturing $\widehat{s}_{i-1}$ at point $\mathsf{root\_input}_{i-1}$. Finally, $\mathcal{S}$ outputs the simulated state $s = (s_1, s_2, s_3)$ where $s_1 = (\widehat{s}_k, (\mathsf{root\_input}_1, r_1), ..., (\mathsf{root\_input}_k, r_k))$.

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

Observe that the size of the state of above CPRFG grows roughly linearly with the number of applications of `Compute` between two consecutive applications of `Advance`, which may in principle lend to a denial of service attack on the protocol. However, we argue that a linear 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 [56]. Furthermore, it may be reasonable to mitigate such denial of service attacks by imposing an a priori bound on the number of failed attempts to advance an epoch before the protocol raises an alarm to the user.

6.2 Protocol Π_{eKE}

The epoch key exchange protocol (See Figure 10 on page 33) computes the public key ratchet in the Signal protocol. It persists for the entire duration of the secure messaging session between two parties, and it has four methods (the same as $\mathcal{F}_{\mathsf{eKE}}$) that we describe below.

Confirm Receiving Epoch On the first activation from Π_{SGNL}, Π_{eKE} follows nearly the same instructions as $\mathcal{F}_{\mathsf{eKE}}$. It retrieves the initialization keys from $\mathcal{F}_{\mathsf{DIR}}$ for starting a conversation between the sender and receiver. It then contacts the long-term memory module $\mathcal{F}_{\mathsf{LTM}}$ associated with its local pid to compute the first sending root key. As described previously, $\mathcal{F}_{\mathsf{LTM}}$ uses the triple Diffie-Hellman key exchange protocol [52]. This binds the conversation to the identities of the two parties.

On each activation, including the first one, Π_{eKE} then computes the new sending chain key (along with the $\mathsf{root\_key}$ and $\mathsf{root\_input}$ using the subroutine **Compute Sending Chain Key** using the Key Derivation Mechanism (KDM). Π_{eKE} then outputs the $\mathsf{epoch\_id}$ to Π_{SGNL}.

Get Sending Key When Π_{mKE} asks for the $\mathsf{sending\_chain\_key}$, Π_{eKE} outputs it, unless it has been erased.

Get Receiving Key When Π_{mKE} sends (`GetRotatingKey`, $\mathsf{epoch\_id}$), Π_{eKE} temporarily stores the $\mathsf{epoch\_id}$ as the sender's epoch id. If this is the first activation, Π_{eKE} contacts $\mathcal{F}_{\mathsf{DIR}}$ and $\mathcal{F}_{\mathsf{LTM}}$ in a similar manner as `ConfirmReceivingEpoch` to compute the initial $\mathsf{root\_key}$. Then, regardless of whether it is the first activation or not, Π_{eKE} runs the subroutine **Compute Receiving Chain Key**, which ratchets the $\mathsf{root\_input}$ and $\mathsf{recv\_chain\_key}$ forward (but stores them in temporary variables). These variables will be stored permanently when `ConfirmReceivingEpoch` is called–otherwise, if decryption is not successful, the temporary variables will be overwritten on future attempts to ratchet forward.

Corrupt On corruption, Π_{eKE} returns its internal state, the following elements: $(\mathsf{epoch\_key}, \mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{partner}}, \mathsf{root\_key})$. Note that, as with the other protocols, the `Corrupt` method is not a "real" interface, but is only there for the purposes of UC-modelling.

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

$\mathit{\Pi}_{\mathsf{eKE}}$

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 $\mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1)$.

keyGen chooses a random Diffie-Hellman exponent $\mathsf{epoch\_key} \overset{\$}{\leftarrow} |G|$ for a known group G and sets $\mathsf{epoch\_id} = g^{\mathsf{epoch\_key}}$.

ConfirmReceivingEpoch: On input $(\mathtt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}^*)$ from $(\mathit{\Pi}_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}')$:

1. If $\mathsf{pid}' \neq \mathsf{pid}$, then end the activation. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathsf{temp\_epoch\_id\_partner}_i = \mathsf{epoch\_id}^*$.
3. If this is the first activation:
   - Initialize state variables $\mathsf{root\_key}, \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Send $(\mathtt{GetInitKeys}, \mathsf{pid}_{1-i}, \mathsf{pid}_i)$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving $(\mathtt{GetInitKeys}, \mathsf{ik}_j^{\mathsf{pk}}, \mathsf{rk}_j^{\mathsf{pk}}, \mathsf{ok}_{j\leftarrow i}^{\mathsf{pk}})$, send input $(\mathtt{ComputeSendingRootKey}, \mathsf{ik}_j^{\mathsf{pk}}, \mathsf{rk}_j^{\mathsf{pk}}, \mathsf{ok}_{j\leftarrow i}^{\mathsf{pk}})$ to $\mathcal{F}_{\mathsf{LTM}}$.
   - Upon receiving $(\mathtt{ComputeSendingRootKey}, k, \mathsf{ek}_i^{\mathsf{pk}})$, set $\mathsf{root\_key} = k$.
   - Do steps 4-6 of **Compute Sending Chain Key**
   - Erase $\mathsf{ek}_i^{\mathsf{pk}}$ and output $(\mathtt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}_{\mathsf{self}} || \mathsf{ek}_i^{\mathsf{pk}} || \mathsf{ok}_{j\leftarrow i}^{\mathsf{pk}})$ to $(\mathit{\Pi}_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$
4. Else (this is not the first activation):
   - **Compute Sending Chain Key.**
   - Output $(\mathtt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}_{\mathsf{self}})$ to $(\mathit{\Pi}_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

GetSendingKey: On receiving input $(\mathtt{GetSendingKey})$ from $(\mathit{\Pi}_{mKE}, \mathsf{sid}.mKE, \mathsf{pid}')$:

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

GetReceivingKey: On receiving input $(\mathtt{GetReceivingKey}, \mathsf{epoch\_id})$ from $(\mathit{\Pi}_{mKE}, \mathsf{sid}, \mathsf{pid}')$:

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

Corrupt: On receiving $(\mathtt{Corrupt})$ from $(\mathit{\Pi}_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$: return $(\mathsf{epoch\_key}, \mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{partner}}, \mathsf{root\_key})$ to $(\mathit{\Pi}_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$

//Note that the Corrupt interface is not part of the “real” protocol; it is only included for UC-modelling purposes.

//Below are subroutines used in the interfaces above.

Compute Sending Chain Key:

1. Compute $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathsf{epoch\_key}_{\mathsf{self}})$.
2. Compute $(\mathsf{root\_key}) = \mathtt{Advance}(\mathsf{root\_key}, \mathsf{root\_input})$.
3. Generate a key pair $(\mathsf{epoch\_key}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{self}}) \leftarrow \text{keyGen}()$.
4. Compute the next input $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{temp\_epoch\_id}_{\mathsf{partner}}, \mathsf{epoch\_key}_{\mathsf{self}})$.
5. Compute $(\mathsf{root\_key}, \mathsf{sending\_chain\_key}) = \mathtt{Compute}(\mathsf{root\_key}, \mathsf{root\_input})$.
6. Finally, advance $(\mathsf{root\_key}) = \mathtt{Advance}(\mathsf{root\_key}, \mathsf{root\_input})$
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 $(\mathsf{root\_key}, \mathsf{temp\_recv\_chain\_key}) = \mathtt{Compute}(\mathsf{root\_key}, \mathsf{root\_input})$.
3. Erase $\mathsf{root\_input}$. //The old root key is overwritten and therefore erased.

Fig. 10. The Epoch Key Exchange Protocol $\mathit{\Pi}_{\mathsf{eKE}}$

6.3 Proof of Theorem 3

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

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

The detailed simulator with all the computations can be found in Figure 6.3 on section 6.3. We first describe the high level intuition behind the simulator and then follow up with a hybrid argument to complete our proof.

When neither party has ever been corrupted, the simulator's actions are simple to describe.

Simulator $\mathcal{S}_{eKE}$ when no corruptions have occurred

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

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

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

When a corruption occurs, the simulator must return the corrupted party's state to simulate a passive corruption. The simulator must then provide $\mathsf{sending\_chain\_key}$'s and $\mathsf{recv\_chain\_key}$'s for the parties to $\mathcal{F}_{\mathsf{eKE}}$ till a compromise of the real protocol would end. (Maybe never if the adversary chooses to man in the middle the parties forever!)

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 $\mathit{\Pi}_{eKE}$ based on the state that the simulator provides them. If the simulator can produce a 'convincing state' for the corrupted party at the time of corruption, then simply running the parties as in the honest protocol and updating them based on the inputs provided by the environment will look indistinguishable from how an honest corruption would go. To flesh out this intuition, let's start by discussing how the simulator will handle a $\mathtt{ReportState}$ request from $\mathit{\Pi}_{\mathsf{eKE}}$.

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

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

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

GenEpochId: On receiving $(\mathtt{GenEpochId}, i, \mathsf{epoch\_id}^*)$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:
//This call happens at every sending epoch for each party.

1. The variable i denotes the party.
2. If this is the first ever activation set $\mathsf{diverge\_parties} = \mathit{false}$, $\mathsf{compromised} = \mathit{false}$, and initialize an empty dictionary $\mathsf{compromised\_epochs} = \{\}$.
3. If this is the first activation for i then set $\mathsf{epoch\_num}_i = i$.
4. Otherwise, increment $\mathsf{epoch\_num}_i \mathrel{+}= 2$.
5. If $\mathsf{compromised} = \mathit{true}$ or $\mathsf{diverge\_parties} = \mathit{true}$:
   //This step only runs after compromise.
   //Use $\mathcal{V}iew_i$ and run the steps from **Compute Sending Chain Key** in Π_{eKE}.
   (a) Set $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}^*$.
   (b) Compute $\mathcal{V}iew_i.\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}^*, \mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}})$.
   (c) Set $\mathsf{root\_key} = \mathtt{Advance}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
   (d) 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
   (e) If $\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$ or $\mathsf{diverge\_parties} = \mathit{true}$ then:
   //If healing did not occur, continue to simulate by running the instructions from Π_{eKE}.
      - Sample $(\mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}}, \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}}) \xleftarrow{\$}$ keyGen().
      - Compute $\mathcal{V}iew_i.\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}^*, \mathcal{V}iew_i.\mathsf{epoch\_key}_self)$.
      - Set $\mathcal{V}iew_i.\mathsf{sending\_chain\_key} = \mathtt{Compute}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
      - Set $\mathcal{V}iew_i.\mathsf{root\_key} = \mathtt{Advance}(\mathcal{V}iew_i.\mathsf{root\_key}, \mathcal{V}iew_i.\mathsf{root\_input})$.
      - Erase $\mathcal{V}iew_i.\mathsf{root\_input}$, $\mathsf{sending\_chain\_key}$.
      - Output $(\mathtt{GenEpochId}, i, \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

   (f) Otherwise ($\mathsf{epoch\_num}_i \notin \mathsf{compromised\_epochs}$ and $\mathsf{diverge\_parties} \neq \mathit{true}$):
   //If healed, delete the view objects and generate a new epoch id as in the honest case.
      - Set $\mathsf{epoch\_id}_i = \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}}$ and $\mathsf{epoch\_id}_{1-i} = \mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{self}}$.
      - Set $\mathsf{epoch\_key}_i = \mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}}$ and $\mathsf{epoch\_key}_{1-i} = \mathcal{V}iew_{1-i}.\mathsf{epoch\_key}_{\mathsf{self}}$.
      - Delete the objects $\mathcal{V}iew_0, \mathcal{V}iew_1$
6. If $\mathsf{compromised} = \mathit{false}$ and $\mathsf{diverge\_parties} = \mathit{false}$:
   //This is the honest case, also covers first sending epochs after compromise.
   - Store $\mathsf{old\_epoch\_id}_i = \mathsf{epoch\_id}_i$ and $\mathsf{old\_epoch\_key}_i = \mathsf{epoch\_key}_i$. //only store one of the old id-key pairs since there's no need to keep all of them.
   - Sample a new pair $(\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i) \xleftarrow{\$}$ keyGen() and record $\mathsf{epoch\_id}_i$.
   - Output $(\mathtt{GenEpochId}, i, \mathsf{epoch\_id}_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

//The remainder of the interface only gets called on compromise or divergence.
ReportState: On receiving $(\mathtt{ReportState}, \mathsf{pid}_i, \mathsf{recv\_chain\_key}, \mathsf{leak})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:

1. If $\mathsf{compromised} = \mathit{true}$ or $\mathsf{diverge\_parties} = \mathit{true}$: Add $\mathsf{epoch\_num}_i, \mathsf{epoch\_num}_i + 1, \mathsf{epoch\_num}_i + 2, \mathsf{epoch\_num}_i + 3$ to the list $\mathsf{compromised\_epochs}$ and output $(\mathtt{ReportState}, \mathsf{pid}_i, \mathcal{V}iew_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
2. Otherwise, let $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$.
3. If $\mathsf{recv\_chain\_key} = \perp$, choose $\mathcal{V}iew_i.\mathsf{root\_key} \xleftarrow{\$} \{0,1\}^n$.
   //pid_i is in a sending state and pid_{1-i} has not yet started a newer sending epoch.
4. Otherwise, $\mathsf{recv\_chain\_key} \neq \perp$:
   - Compute $\mathsf{leak}' = \{\mathsf{Exp}(\mathsf{epoch\_id}, \mathsf{epoch\_key}_i) \quad | \quad \mathsf{epoch\_id} \in \mathsf{leak}\}$.
   - Let $\mathsf{root\_input} = \mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i)$ and set $T = \{\mathsf{leak}', (\mathsf{root\_input}, \mathsf{recv\_chain\_key})\}$.
   - Finally, set $\mathsf{root\_key} = S(T)$.
5. Create two view objects $\mathcal{V}iew_0, \mathcal{V}iew_1$ and set the following values in these objects:
   - If $\mathsf{epoch\_num}_i > \mathsf{epoch\_num}_{1-i}$: //$\mathsf{pid}_i$ is ahead
     - $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}_{1-i}$
     - $\mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{old\_epoch\_id}_i$
   - Else ($\mathsf{epoch\_num}_i < \mathsf{epoch\_num}_{1-i}$):
     - $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{old\_epoch\_id}_{1-i}$
     - $\mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}_i$
   - $\mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}_i$, $\mathcal{V}iew_i.\mathsf{epoch\_key}_{\mathsf{self}} = \mathsf{epoch\_key}_i$
   - $\mathcal{V}iew_{1-i}.\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}_{1-i}$, $\mathcal{V}iew_{1-i}.\mathsf{epoch\_key}_{\mathsf{self}} = \mathsf{epoch\_key}_{1-i}$.
   - $\mathcal{V}iew_i.\mathsf{root\_key} = \mathsf{root\_key}$, $\mathcal{V}iew_{1-i}.\mathsf{root\_key} = \mathsf{root\_key}^*$.
6. Add $\mathsf{epoch\_num}_i, \mathsf{epoch\_num}_i + 1, \mathsf{epoch\_num}_i + 2, \mathsf{epoch\_num}_i + 3$ to the list $\mathsf{compromised\_epochs}$.
7. Output $(\mathtt{ReportState}, \mathsf{pid}_i, \mathcal{V}iew_i = \{\mathsf{epoch\_key}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{self}}, \mathsf{epoch\_id}_{\mathsf{partner}}, \mathsf{root\_key}\})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

GetSendingKey: On receiving $(\mathtt{GetSendingKey}, i)$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do: //This only happens if the parties are diverged, or if the current epoch is compromised and we're within the quarantine period for the compromise.

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

GetReceivingKey: On receiving $(\mathtt{GetReceivingKey}, i, \mathsf{epoch\_id})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:
//Only happens if the epoch is compromised or the parties have diverged.

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

Simulator $\mathcal{S}_{\mathsf{eKE}}$'s instructions for a ReportState request

ReportState: On receiving $(\texttt{ReportState}, i, \mathsf{recv\_chain\_key})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:

1. If $\mathsf{recv\_chain\_key} = \perp$ then choose a $\mathsf{root\_key}$ at random.
2. Otherwise, choose the root key such that $Compute(\mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i), \mathsf{root\_key}) = (\mathsf{recv\_chain\_key}, \mathsf{root\_key}^*)$.
3. Create emulated party $\mathcal{P}_i$ with state $(\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i, \mathsf{old\_epoch\_id}_{1-i}, \mathsf{root\_key})$.
4. Compute $\mathsf{root\_key}^*$ to be the second half of $Compute(\mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i), \mathsf{root\_key}) = (\mathsf{recv\_chain\_key}, \mathsf{root\_key}^*)$.
5. Create $\mathcal{P}_{1-i}$ with state $(\mathsf{epoch\_key}_{1-i}, \mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_id}_i, \mathsf{root\_key}^*)$
6. Add $\mathsf{epoch\_num}_i, \mathsf{epoch\_num}_i + 1, \mathsf{epoch\_num}_i + 2, \mathsf{epoch\_num}_i + 3$ to the list $\mathsf{compromised\_epochs}$.
7. Output $(\texttt{ReportState}, \mathsf{pid}_i, \mathcal{V}iew_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

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 $\texttt{GenEpochId}$ requests to know how to ratchet forward to the receiving epochs for each party. Once the functionality 'heals' from the corruption, it will stop asking the simulator for keys unless the parties have diverged. If divergence doesn't occur, the simulator must end the execution of the dummy parties and go back to the simulation instructions for the case of no corruptions. However, in the case of divergence, it must continue to run the dummy parties to provide chain keys.

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

Short intuitive version of $\mathcal{S}_{\mathsf{eKE}}$ with corruption handling

The keyGen($\cdot$) operation is the same one as from Π_{eKE}.
GenEpochId: On receiving $(\texttt{GenEpochId}, i, \mathsf{epoch\_id}^*)$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:

1. If neither party is compromised anymore and divergence has not occured, stop the emulations. //$\mathcal{S}_{\mathsf{eKE}}$ keeps track of compromises using the same logic as $\mathcal{F}_{\mathsf{eKE}}$. It can use the confirmed $\mathsf{epoch\_id}^*$ to detect divergence.
   - Store $\mathsf{old\_epoch\_id}_i = \mathsf{epoch\_id}_i$ and $\mathsf{old\_epoch\_key}_i = \mathsf{epoch\_key}_i$.
   - Sample a new pair $(\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i) \overset{\$}{\leftarrow}$ keyGen() and record $\mathsf{epoch\_id}_i$.
2. If the epoch is still compromised or the parties have diverged, ratchet forward emulated party i and let $\mathsf{epoch\_id}_i = \mathcal{V}iew_i.\mathsf{epoch\_id}_{\mathsf{self}}$.
3. Output $(\texttt{GenEpochId}, i, \mathsf{epoch\_id}_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

ReportState: On receiving $(\texttt{ReportState}, i, \mathsf{recv\_chain\_key})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:

1. If $\mathsf{recv\_chain\_key} = \perp$ then choose a $\mathsf{root\_key}$ at random.
2. Otherwise, choose the root key such that $Compute(\mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i), \mathsf{root\_key}) = (\mathsf{recv\_chain\_key}, \mathsf{root\_key}^*)$.
3. Create emulated party $\mathcal{P}_i$ with state $(\mathsf{epoch\_key}_i, \mathsf{epoch\_id}_i, \mathsf{old\_epoch\_id}_{1-i}, \mathsf{root\_key})$.
4. Compute $\mathsf{root\_key}^*$ to be the second half of $Compute(\mathsf{Exp}(\mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_key}_i), \mathsf{root\_key}) = (\mathsf{recv\_chain\_key}, \mathsf{root\_key}^*)$.
5. Create $\mathcal{P}_{1-i}$ with state $(\mathsf{epoch\_key}_{1-i}, \mathsf{epoch\_id}_{1-i}, \mathsf{epoch\_id}_i, \mathsf{root\_key}^*)$
6. Add $\mathsf{epoch\_num}_i, \mathsf{epoch\_num}_i + 1, \mathsf{epoch\_num}_i + 2, \mathsf{epoch\_num}_i + 3$ to the list $\mathsf{compromised\_epochs}$.
7. Output $(\texttt{ReportState}, \mathsf{pid}_i, \mathcal{V}iew_i)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

GetSendingKey: On receiving $(\texttt{GetSendingKey}, i)$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:
//This only happens if the parties are diverged, or if the current epoch is compromised and we're within the quarantine period for the compromise.

1. Output $(\texttt{GetSendingKey}, i, \mathcal{V}iew_i.\mathsf{sending\_chain\_key})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$

GetReceivingKey: On receiving $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ do:
//This will only occur for compromised epochs. we produce the chain key by having the emulated party run Π_{eKE}

1. Have emulated party pid_i run the instructions for $(\texttt{GetReceivingKey}, \mathsf{epoch\_id})$ from Π_{eKE}.
2. Output $(\texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathcal{V}iew_i.\mathsf{temp\_recv\_chain\_key})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.

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

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

Hybrid $\boldsymbol{H_0}$ This is the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ on fig. 6 along with the simulator $\mathcal{S}_{\mathsf{eKE}}$ on section 6.3.

Hybrid $\boldsymbol{H_1}$ This hybrid sets up the format for transitioning from the ideal functionality $\mathcal{F}_{\mathsf{eKE}}$ to the use of a KDM chain while parties are not compromised and have not diverged. We replace the random choice of chain keys in `GetSendingKey` and `GetReceivingKey` with a random choice of function for every epoch, and randomly chosen inputs for these functions.

*We replace **step** 3 of **GetSendingKey** in $\mathcal{F}_{\mathsf{eKE}}$ with:*

1. Choose a random function $\mathsf{sending\_}F_i : G \rightarrow \{0,1\}^n$.
2. Sample a random input $\mathsf{sending\_root\_input}_i \xleftarrow{\$} G$.
3. Set $\mathsf{sending\_chain\_key}_i \leftarrow \mathsf{sending\_}F_i(\mathsf{sending\_root\_input}_i)$.

*We also replace **step** ?? of **GetReceivingKey** in $\mathcal{F}_{\mathsf{eKE}}$ with:*

1. If $\mathsf{diverge\_parties} = \mathit{false}$, $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_{1-i}$, and $\mathsf{epoch\_num}_i + 1 \notin \mathsf{compromised\_epochs}$:
   - Sample a random input $\mathsf{receiving\_root\_input}_i \xleftarrow{\$} G$.
   - Set $\mathsf{receiving\_}F_i = \mathsf{sending\_}F_{1-i}$.
   - Set $\mathsf{recv\_chain\_key}_i \leftarrow \mathsf{receiving\_}F_i(\mathsf{receiving\_root\_input}_i)$.
   - Add $(\mathsf{epoch\_id}, \mathsf{recv\_chain\_key}_i)$ to $\mathsf{receive\_attempts}[\mathsf{epoch\_num}]$.

Note that when an epoch is compromised or the parties have diverged, all output values are provided to $\mathcal{F}_{\mathsf{eKE}}$ by the simulator who simply creates two emulated parties who run the protocol of Π_{eKE} as specified. (At initial compromise, the state of the emulated parties is chosen specifically so that it is indistinguishable from a state compromise in the real world. To do this $\mathcal{S}_{\mathsf{eKE}}$ take advantage of the simulator used to prove that the KDM protocol is a secure CPRFG. So, all we need to do is argue the indistinguishability of choices made within the functionality without consulting the simulator.)

Hybrid $\boldsymbol{H_2}$ This hybrid transitions from using a version of the ideal CPRFG game to using the actual protocol. In particular, a random shared $\mathsf{root\_key}$ is chosen for the parties at first activation. Then, each ratchet forward chooses random inputs using the code of the simulator (like in the previous hybrid). Unlike the previous hybrid, we now use the `Compute` and `Advance` methods of the KDM to compute the chain keys and update the $\mathsf{root\_key}$ for each party instead of choosing a new random function. This hybrid is indistinguishable from the previous one because our KDM is a secure CPRFG.

Hybrid $\boldsymbol{H_3}$ This hybrid replaces the randomly chosen $\mathsf{root\_input}$'s with the actual Diffie-Hellman keys. This hybrid will be computationally indistinguishable from the previous one because we assume that the DDH assumption holds for group G.

Hybrid $\boldsymbol{H_4}$ In this hybrid we will replace the initial root keys with root keys computed using $\mathcal{F}_{\mathsf{DIR}}$, $\mathcal{F}_{\mathsf{LTM}}$ as in Π_{eKE} instead of being chosen randomly. While the functionality $\mathcal{F}_{\mathsf{eKE}}$ automatically provides a binding between party id's and epoch keys up to the first compromise, the protocol 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.

Note that this hybrid is the real protocol.

7 The Symmetric Key Chain: Realizing $\mathcal{F}_{\mathsf{fs\_aead}}$

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

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

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

Below we describe the ideal message key exchange functionality, $\mathcal{F}_{\mathsf{mKE}}$. This functionality (See Figure 11 on page 39) provides message key management for a single epoch, mainly by keeping track of which message keys have been generated and retrieved and which message keys have been exposed during a state compromise.

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

Below we define an ideal single-message encryption functionality (See Figure 12 on page 40). On receiving an encryption request, $\mathcal{F}_{\mathsf{aead}}$ simply stores the message and adversarially-generated ciphertext in a table and on receiving a decryption request, it looks up the message. $\mathcal{F}_{\mathsf{aead}}$ consults the message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$ associated with the epoch to decide whether to allow or disallow encryption or decryption.

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

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

As outlined in Section 4, protocol $\Pi_{\mathsf{fs\_aead}}$ makes straightforward use of multiple instances of $\mathcal{F}_{\mathsf{aead}}$, along with $\mathcal{F}_{\mathsf{mKE}}$. It is presented in Figure 11 on page 39.

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

Proof. We construct an ideal-process adversary $\mathcal{S}_{\mathsf{fs\_aead}}$ (See Figure 14 on page 44) that interacts with functionality $\mathcal{F}_{\mathsf{fs\_aead}}$. The objective of $\mathcal{S}_{\mathsf{fs\_aead}}$ is to simulate the interactions that would take place between the environment and the protocol $\Pi_{\mathsf{fs\_aead}}$ (together with its subroutine functionalities $\mathcal{F}_{\mathsf{mKE}}$ and $\mathcal{F}_{\mathsf{aead}}$), so that the views of the environment Env are perfectly identical in the real and ideal scenarios. When the simulator $\mathcal{S}_{\mathsf{fs\_aead}}$ receives messages from $\mathcal{F}_{\mathsf{fs\_aead}}$ in the ideal world, it takes the actions specified in the pseudocode on fig. 14.

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

This functionality has a session id $\mathsf{sid}.mKE$ that takes the following format: $\mathsf{sid}.mKE = (\text{“}mKE\text{”}, sid.fs)$. Where $\mathsf{sid}.fs = (\text{“}\textit{fs\_aead}\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$. 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.

This functionality is parametrized by a seed length λ

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

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

StopKeys: On receiving $(\mathtt{StopKeys}, N)$ from $(\mathit{\Pi}_{fs\_aead}, \mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}, b), \mathsf{pid})$,

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

Corruption: On receiving $(\mathtt{Corrupt})$ from $(\mathit{\Pi}_{fs\_aead}, \mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}, b), \mathsf{pid})$:

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

Fig. 11. The Message Key Exchange Functionality $\mathcal{F}_{\mathsf{mKE}}$

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

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

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

1. If this is not the first encryption request or $\mathsf{pid} \notin (\mathsf{pid}_0, \mathsf{pid}_1)$, end the activation. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Provide input $(\mathtt{RetrieveKey}, \mathsf{pid})$ to $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$
3. Upon receiving output $(\mathtt{RetrieveKey}, \mathsf{pid}, k)$ from $(\Pi_{\mathsf{mKE}}, \mathsf{sid}.mKE, \mathsf{pid})$:
   - If $k =\perp$ then end the activation.
   - Else $(k \neq \perp)$:
     - If $\mathtt{IsCorrupt?} = \mathit{true}$, set $\mathsf{leak} = m$.
     - Else $(\mathtt{IsCorrupt?} = \mathit{false})$, set $\mathsf{leak} = |m|$.
   - Send a backdoor message $(\mathtt{Encrypt}, \mathsf{pid}, \mathsf{leak}, N)$ to $\mathcal{A}$.
4. Upon receiving a response $(\mathtt{Encrypt}, \mathsf{pid}, c)$, record the tuple $(c, k, m, N, 1)$ and sender i and set $\mathsf{ready2decrypt} = \mathit{true}$. Output $(\mathtt{Encrypt}, c)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.

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

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

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

1. End the activation if $\mathsf{pid} \notin (\mathsf{pid}_0, \mathsf{pid}_1)$. Otherwise, let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathtt{IsCorrupt?}$ to *true*.
3. If there is a record $(c, k, m, N, 1)$ then set $m^* = m$ otherwise $m^* = \perp$.
4. Send $(\mathtt{ReportState}, \mathsf{pid}, m^*, k)$ to $\mathcal{A}$.
5. Upon receiving a response $(\mathtt{ReportState}, \mathsf{pid}, S)$ from $\mathcal{A}$, send $(\mathtt{Corrupt}, S)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

Fig. 12. The Authenticated Encryption with Associated Data Functionality, $\mathcal{F}_{\mathsf{aead}}$

$\boldsymbol{\Pi}_{\mathsf{fs\_aead}}$

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

Encrypt: On receiving input $(\mathtt{Encrypt}, m, N)$ from $(\Pi_{MAN}, \mathsf{sid}, \mathsf{pid})$ 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 $(\mathtt{Encrypt}, m, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{"}aead\text{"}, \mathsf{sid}.fs, \mathsf{msg\_num}))$ and delete m.
5. Upon receiving $(\mathtt{Encrypt}, c)$, output $(\mathtt{Encrypt}, c)$ to $(\Pi_{MAN}, \mathsf{sid}, \mathsf{pid})$.

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

1. Check that 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 $(\mathtt{Decrypt}, c, N)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{"}aead\text{"}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
4. Upon receiving $(\mathtt{Decrypt}, v)$, if $v = \mathtt{Fail}$ then output $(\mathtt{Decrypt}, \mathtt{Fail})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.
5. Otherwise $(v \neq \mathtt{Fail})$:
   - While $\mathsf{curr\_msg\_num} < \mathsf{msg\_num}$:
     - Increment $\mathsf{curr\_msg\_num} += 1$.
     - Add $\mathsf{curr\_msg\_num}$ to $\mathsf{pending\_msgs}$.
   - Remove $\mathsf{msg\_num}$ from $\mathsf{pending\_msgs}$ and output $(\mathtt{Decrypt}, v)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

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

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

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

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

1. Check that 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 $(\mathtt{Corrupt})$ to $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{"}mKE''\!, \mathsf{sid}.fs))$.
4. On receiving $(\mathtt{Corrupt}, S_{mKE})$, add it to S and do the following.
5. For each record $\mathsf{msg\_num} \in \mathsf{pending\_msgs}$:
   - Send $(\mathtt{Corrupt}, \mathsf{pid}_i)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{"}aead''\!, \mathsf{sid}.fs, \mathsf{msg\_num}))$
   - On receiving a response $(\mathtt{Corrupt}, \mathsf{pid}_i, S_{\mathsf{msg\_num}})$, add $S_{\mathsf{msg\_num}}$ to S.
6. Output $(\mathtt{Corrupt}, S)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

Fig. 13. The Forward-Secure Encryption $\Pi_{\mathsf{fs\_aead}}$

Observe that the APIs of $\mathcal{F}_{\mathsf{fs\_aead}}$ and $\Pi_{\mathsf{fs\_aead}}$ are identical: they each have 5 methods. In the remainder of the proof, we describe the actions taken by $\Pi_{\mathsf{fs\_aead}}$ in the real world and $\mathcal{F}_{\mathsf{fs\_aead}}$ together with $\mathcal{S}_{\mathsf{fs\_aead}}$ and explain why they maintain the property that the view of the real environment 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 $\mathcal{F}_{\mathsf{aead}}$. The proof proceeds by induction over the steps of the simulator.

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

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

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

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

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

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

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

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 recieving a state S_{mKE} from $\mathcal{A}$, $\mathcal{F}_{\mathsf{mKE}}$ reports a list of all messages still in transit and S_{mKE}. Then, $\Pi_{\mathsf{fs\_aead}}$ corrupts each $\mathcal{F}_{\mathsf{aead}}$ instance that has a pending message, each of which leaks its message and key seed to $\mathcal{A}$. When $\mathcal{F}_{\mathsf{aead}}$ gets a state $S_{\mathsf{msg\_num}}$ back from $\mathcal{A}$, it reports this to $\Pi_{\mathsf{fs\_aead}}$. $\Pi_{\mathsf{fs\_aead}}$ then reports all of the states S_{mKE} and $S_{\mathsf{msg\_num}}$'s along with the list of pending messages and current message number to Π_{SGNL}.

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

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

8 Realizing Ideal Message Key Exchange

As outlined in Section 4, protocol Π_{mKE} for realizing $\mathcal{F}_{\mathsf{mKE}}$ implements the symmetric keychain for the epoch specified in its session ID. This is done by obtaining the base key for the chain from $\mathcal{F}_{\mathsf{eKE}}$ and then extending the chain as needed. Importantly, the keys on the main symmeric chain are never directly given as output; rather the outputs are keys derived from the chain keys. This structure allows the key derivation function to only be a (length doubling) pseudorandom number generator. Protocol Π_{mKE} is presented in Figure 11 on page 39.

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* $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$*, and* $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$*.*

Proof. We construct an ideal-process adversary $\mathcal{S}_{\mathsf{mKE}}$ in the figure on fig. 16 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 $\mathcal{F}_{\mathsf{eKE}} + \mathcal{S}_{\mathsf{eKE}}$), so that the views of the environment Env are computationally indistinguishable in the real and ideal scenarios. When the simulator $\mathcal{S}_{\mathsf{mKE}}$ receives messages from $\mathcal{F}_{\mathsf{mKE}}$ in the ideal world, it takes the actions specified in the pseudocode on fig. 16.

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

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

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 requiers 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

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

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

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

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

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

    //End simulation of $\texttt{Decrypt}$ from $\mathcal{F}_{\mathsf{aead}}$
13. While $\mathcal{V}iew_i.\mathsf{curr\_msg\_num} \leq \mathsf{msg\_num}$ do: //we only get to this step if decryption succeeds.
    - Increment $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$.
    - $(\mathcal{V}iew_i.\mathsf{curr\_chain\_key}, \mathcal{V}iew_i.k) = PRG(\mathcal{V}iew_i.\mathsf{curr\_chain\_key})$.
    - Store $\mathcal{V}iew_i.\mathsf{missed\_msgs}[\mathcal{V}iew_i.\mathsf{curr\_msg\_num}] = \mathcal{V}iew_i.k$.
14. Let $k = \mathcal{V}iew_i.\mathsf{missed\_msgs}[\mathsf{msg\_num}]$,and erase $\mathcal{V}iew_i.\mathsf{missed\_msgs}[\mathsf{msg\_num}]$ if the entry doesn't exist then output $(\texttt{inject}, \bot)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.
15. Otherwise, output $(\texttt{inject}, v)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

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

1. Initialize $\mathsf{pending\_msgs} = []$. If $\mathcal{V}iew_i.\mathsf{missed\_msgs}$ exists, set $\mathsf{pending\_msgs} = \mathcal{V}iew_i.\mathsf{missed\_msgs}.keys$.
2. Initialize a state object S and add $\mathcal{V}iew_i.\mathsf{curr\_msg\_num}$ and $\mathsf{pending\_msgs}$ to it.
   //Simulating $\texttt{Corrupt}$ from $\mathcal{F}_{\mathsf{mKE}}$
3. Set $\texttt{IsCorrupt?} = true$, set $\mathsf{chain\_key} = \mathcal{V}iew_i.\mathsf{curr\_chain\_key}$ and let $\mathsf{keys\_in\_transit} = \mathcal{V}iew_i.\mathsf{missed\_msgs}$.
4. If there is a record $(\texttt{StopKeys}, i, N)$ then let $\mathsf{chain\_key} = \bot$.
5. Send $(\texttt{ReportState}, i, \mathsf{keys\_in\_transit}, \mathsf{curr\_msg\_num}_i, \mathsf{chain\_key})$ to $\mathcal{A}$ on behalf of $(\mathcal{F}_{\mathsf{mKE}}, \mathsf{sid}.mKE = (\text{“}mKE\text{”}, \mathsf{sid}.fs))$
6. On receiving a response $(\texttt{ReportState}, i, S_{mKE})$ from $\mathcal{A}$, add it to the state S.
   //End simulation of $\texttt{Corrupt}$ from $\mathcal{F}_{\mathsf{mKE}}$
7. For each $\mathsf{msg\_num} \in \mathsf{pending\_msgs}$:
   //Simulate $\texttt{Corrupt}$ from $\mathcal{F}_{\mathsf{aead}}$.
   - If there is an entry $\mathsf{records}[\mathsf{msg\_num}]$, then get m from the entry and set $m^* = m$ otherwise $m^* = \bot$.
   - Set $k = \mathcal{V}iew_i.\mathsf{missed\_msgs}[\mathsf{msg\_num}]$.
   - Send $(\texttt{ReportState}, \mathsf{pid}, m^*, k)$ to $\mathcal{A}$ on behalf of $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead = (\text{“}aead\text{”}, \mathsf{sid}.fs, \mathsf{msg\_num}))$.
   - Upon receiving a response $(\texttt{ReportState}, \mathsf{pid}, S_{\mathsf{msg\_num}})$ from $\mathcal{A}$, add it to S.

   //End simulation of $\texttt{Corrupt}$ from $\mathcal{F}_{\mathsf{aead}}$.
8. Output $(\texttt{ReportState}, S)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$.

Fig. 14. Forward Secure Authenticated Encryption Simulator, $\mathcal{S}_{\mathsf{fs\_aead}}$

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 $(\mathtt{RetrieveKey}, \mathsf{pid})$ 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 $(\mathtt{GetSendingKey})$ to $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid}))$.

3. Upon receiving $(\mathtt{GetSendingKey}, \mathsf{sending\_chain\_key})$ from $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE)$ set $\mathsf{curr\_chain\_key} = \mathsf{sending\_chain\_key}$.
4. While $\mathsf{curr\_msg\_num} \leq \mathsf{msg\_num}$ do:
   - Increment $\mathsf{curr\_msg\_num}$.
   - $(\mathsf{curr\_chain\_key}, k) = PRG(\mathsf{curr\_chain\_key})$.
   - Store $\mathsf{missed\_msgs}[\mathsf{curr\_msg\_num}] = k$.
5. Output $(\mathtt{RetrieveKey}, \mathsf{pid}, \mathsf{missed\_msgs}[\mathsf{msg\_num}])$ to $(\Pi_{\mathsf{aead}}, \mathsf{sid}.aead, \mathsf{pid})$ while erasing the entry $\mathsf{missed\_msgs}[\mathsf{msg\_num}]$.

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

1. If $\mathtt{StopKeys}$ has already been called, return $(\mathtt{StopKeys}, \mathtt{Success})$.
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 $(\mathtt{StopKeys}, \mathtt{Success})$.

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

//Note that the $\mathtt{Corrupt}$ 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 $(\mathtt{Corrupt}, S)$ to $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid})$.

Fig. 15. The Message Key Exchange Protocol Π_{mKE}

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

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 ($\mathtt{ReportState}$, pid_i, keys_in_transit, $\mathsf{msg\_num}_i$, chain_key) from ($\mathcal{F}_{\mathsf{mKE}}$, sid.$mKE$) do:

```
1. Initialize a dictionary seeds = {}.
2. For (msg_num, $k$) $\in$ keys_in_transit such that msg_num $<$ $\mathsf{msg\_num}_i$:
    – Store seeds[msg_num] $= k$
3. If chain_key $= \bot$, output (ReportState, $S = \bot$) to $\mathcal{F}_{\mathsf{mKE}}$. //This happens when the party has already run StopKeys for this epoch.
4. Let curr_msg_num $=$ $\mathsf{msg\_num}_i$, and let the curr_chain_key $=$ chain_key.
5. While there is some (msg_num, $k$) $\in$ keys_in_transit such that msg_num $>$ $\mathsf{msg\_num}_i$:
    – curr_msg_num$+= 1$
    – Let (curr_chain_key, key_seed) $= PRG$(curr_chain_key).
    – Store seeds[msg_num] $=$ key_seed.
    – Set latest_seed_num $=$ curr_msg_num.
    – Delete (msg_num, $k$) from keys_in_transit.
6. Delete local variable curr_msg_num.
7. Output (ReportState, $S = \{$chain_key$\}$).
```

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

```
1. If msg_num $\leq$ latest_seed_num then output (RetrieveKey, pid, $k =$ seeds[msg_num]) to ($\mathcal{F}_{\mathsf{mKE}}$, 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 msg_num $>$ latest_seed_num, initialize $j =$ latest_seed_num $+ 1$.
3. While $j <$ msg_num:
    – (curr_chain_key, key_seed) $\leftarrow PRG$(curr_chain_key).
    – Store seeds[$j$] $=$ key_seed, $j+= 1$.
4. Output (RetrieveKey, pid, $k =$ seeds[msg_num]) to ($\mathcal{F}_{\mathsf{mKE}}$, sid.$mKE$).
```

Fig. 16. Message Key Exchange Simulator, $\mathcal{S}_{\mathsf{mKE}}$

$Dec(k,c) = m$. 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 [56].

We circumvent this impossibility by resorting to the random oracle model (again, using ideas from [56]). Specifically, we employ a simple Encrypt-then-MAC scheme [46] 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 17 on page 48. We show:

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* $\mathcal{F}_{\mathsf{pRO}}$*, as well as* $F^{\Pi}_{mKE} = (\mathcal{S}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{mKE}}), \mathcal{F}^{\Pi_{\mathsf{eKE}}}_{\mathsf{eKE}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}}), \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}$.

Proof. The ideal-process adversary $\mathcal{S}_{\mathsf{aead}}$ that interacts with functionality $\mathcal{F}_{\mathsf{aead}}$ is presented in Figure 18 on page 49. The objective of $\mathcal{S}_{\mathsf{aead}}$ is to simulate the interactions that would take place between the environment and the protocol Π_{aead} (in the presence of the global random oracle $\mathcal{F}_{\mathsf{pRO}}$), so that the views of the environment Env are exactly identical in the real and ideal scenarios. When the simulator $\mathcal{S}_{\mathsf{aead}}$ receives messages from $\mathcal{F}_{\mathsf{aead}}$ in the ideal world, it takes the actions specified in the pseudocode on fig. 18. Additionally, on corruption, $\mathcal{S}_{\mathsf{aead}}$ equivocates on ciphertext, message pairs that are available to the adversary by programming the random oracle with the correct message key.

We will argue that, conditioned on three bad events not happening, the simulation is perfect. Then, to finish proving Theorem 6, we will show that the bad events happen with negligible probability. The first bad event Forge_1 is the environment being able to produce a verifying message tag pair (m', t') for a fresh message m'. This corresponds to the standard security game for message authentication codes, it happens with negligible probability for exactly this reason. (Note that our functionality can be realized without using a strong Mac) The second bad event Forge_2 requires an environment with a $\mathsf{MAC}(\cdot)$ oracle for a key k chosen u.a.e being able to produce a message tag pair (m', t') that verifies with respect to a different key k' sampled independently. Note that the security of the MAC in the first forgery game implies its security against the second game. This is because an adversary in the Forge_1 game can simulate the Forge_2 game to a different adversary by sampling its own key k' and acting as the Verify oracle. The last bad event $\mathsf{Collision}$ is the event that the simulator attempts to program the random oracle at a point already programmed before by the simulator or the environment, since the seeds provided by F^{Π}_{mKE} are uniformly random and independent, then the probability that two of them collide is negligible.

Π_{aead}

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

Fig. 17. The Authenticated Encryption with Associated Data Protocol, Π_{aead}

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

The global session id sid encodes a ciphersuite, including the MAC protocol $(\mathsf{MAC}, \mathsf{Verify})$, the encryption protocol OTP, and the length of the MAC keys λ.

On receiving $(\mathtt{Encrypt}, \mathsf{pid}, \mathsf{leak}, N)$ from $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$ do:

```
1. If $\mathtt{IsCorrupt?} = \mathit{false}$:
     – Set $\ell = \mathsf{leak}$ and $m = 0^{\ell}$.
     – Choose a random message key $\mathsf{msg\_key} \xleftarrow{\$} \{0,1\}^{\ell+\lambda}$.
2. Else ($\mathtt{IsCorrupt?} = \mathit{true}$),
     //no ciphertext was released yet so we simply run the instructions for $\Pi_{\mathsf{aead}}$.
     – Set $\ell = |\mathsf{leak}|$ and $m = \mathsf{leak}$.
     – Send $(\mathtt{HashQuery}, k, \ell + \lambda)$ to $\mathcal{F}_{\mathsf{pRO}}$.   //Note that $k$ is sent to $\mathcal{S}_{\mathsf{aead}}$ during $\mathtt{ReportState}$
     – Upon receiving the output $(\mathtt{HashQuery}, \mathsf{msg\_key})$, continue.
3. Parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = \ell$ and $|\mathsf{k_{mac}}| = \lambda$.
4. Compute ciphertext $c' = \mathsf{k_{otp}} \oplus m$ and tag $t = \mathsf{MAC}(\mathsf{k_{mac}}, (c', \mathsf{sid}.aead, N))$.
5. Finally, set $c = (c', t)$.
6. Record $(\mathsf{msg\_key}, m, c, N)$ and output $(\mathtt{Encrypt}, \mathsf{pid}, c)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$.
```

On receiving $(\mathtt{inject}, \mathsf{pid}, c^*, N)$ from $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$:

```
1. Parse $c^* = (c', t')$
2. Let $\ell = |c'|$.
3. If $\mathtt{IsCorrupt?} = \mathit{false}$:
     – If there is no record $(\mathsf{msg\_key}, m, c = (c', t), N)$ then end the activation.
     – Else, parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = |m|$ and $|\mathsf{k_{mac}}| = \lambda$.
     – If $\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 0$: set $v = \perp$.
     – Else $(\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 1)$: set $v = m$.
4. Otherwise ($\mathtt{IsCorrupt?} = \mathit{true}$),
     – Send $(\mathtt{HashQuery}, k, \ell + \lambda)$ to $\mathcal{F}_{\mathsf{pRO}}$. //Note that $k$ is sent to $\mathcal{S}_{\mathsf{aead}}$ during $\mathtt{ReportState}$
     – Upon receiving the output $(\mathtt{HashQuery}, \mathsf{msg\_key}')$, continue.
     – Parse $\mathsf{msg\_key}' = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = \ell$, $|\mathsf{k_{mac}}| = \lambda$.
     – If $\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 0$: set $v = \perp$.
     – Else $(\mathsf{Verify}(\mathsf{k_{mac}}, t', (c', \mathsf{sid}.aead, N)) = 1)$: compute $v = \mathsf{k_{otp}} \oplus c'$.
5. Output $(\mathtt{inject}, \mathsf{pid}, c, v)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$.
```

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

```
1. End the activation if $\mathsf{pid} \notin (\mathsf{pid}_0, \mathsf{pid}_1)$.
2. Initialize state $S = \perp$.
3. If there is a record $(\mathsf{msg\_key}, m, c = (c', t))$:
     – Parse $\mathsf{msg\_key} = \mathsf{k_{otp}}||\mathsf{k_{mac}}$ where $|\mathsf{k_{otp}}| = |m^*|$.
     – Let $\mathsf{k_{otp}}^* = c' \oplus m^*$ and $\mathsf{msg\_key}^* = \mathsf{k_{otp}}^*||\mathsf{k_{mac}}$.
     – Send a backdoor message $(\mathtt{Program}, k, \mathsf{msg\_key}^*)$ to $\mathcal{F}_{\mathsf{pRO}}$.
     – On receiving $(\mathtt{Program})$, continue.
4. Store the variable $k$.
5. Set $\mathtt{IsCorrupt?} = \mathit{true}$
6. Send $(\mathtt{ReportState}, \mathsf{pid}, S)$ to $(\mathcal{F}_{\mathsf{aead}}, \mathsf{sid}.aead)$.
```

Fig. 18. Authenticated Encryption with Associated Data Simulator, $\mathcal{S}_{\mathsf{aead}}$

MAC Forgery Games

We will define two kinds of forgery events. The first type of MAC forgery will be 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_1. 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 $\mathsf{Collision}$ to be the event that Env already programmed $\mathcal{F}_{\mathsf{pRO}}$ on input k.

Lemma 1. *If none of the events Forge_1, Forge_2, $\mathsf{Collision}$ happen during the executions, then the the simulation by $\mathcal{S}_{\mathsf{aead}}$ is perfect.*

Proof. Assume that the bad events $\mathsf{Forge}_1, \mathsf{Forge}_2$ and $\mathsf{Collision}$ do not occur during the executions. We will prove that, for all environments Env and adversaries $\mathcal{A}$, the simulator $\mathcal{S}_{\mathsf{aead}}$ together with $\mathcal{F}_{\mathsf{aead}}$ *perfectly* simulate the real-world views of Env and $\mathcal{A}$ when they interact with $\mathit{\Pi}_{\mathsf{aead}}$.

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

In the ideal world, $\mathcal{F}_{\mathsf{aead}}$ retrieves the message key from $\mathit{\Pi}_{\mathsf{mKE}}$ and if the key is available, $\mathcal{F}_{\mathsf{aead}}$ returns the message m that it encrypted to $c = (c', t)$ in the case that the other party asks to decrypt c. This case is identical to the real world, by definition. If, on the other hand, $\mathcal{F}_{\mathsf{aead}}$ gets a different ciphertext $c^* \neq c$, then $\mathcal{F}_{\mathsf{aead}}$ sends the ideal-process adversary $\mathcal{S}_{\mathsf{aead}}$ $(\mathtt{inject}, c^*)$ to see if it wants to inject a potentially different message and waits for a response value v. $\mathcal{S}_{\mathsf{aead}}$ uses the $\mathsf{k_{mac}}$ 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 $\mathcal{S}_{\mathsf{aead}}$'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, $\mathcal{S}_{\mathsf{aead}}$ uses the message key seed k that it received from $\mathcal{F}_{\mathsf{aead}}$ to query the random oracle $\mathcal{F}_{\mathsf{pRO}}$ to expand the key. It then verifies the tag and outputs the decryption of the ciphertext (which may be different from m). In this case, $\mathcal{F}_{\mathsf{aead}}$ outputs the injected message from $\mathcal{S}_{\mathsf{aead}}$. This is identical to the powers of the real-world adversary after a state compromise.

Corruption Notice that the protocol 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 $\mathsf{Collision}$ does not occur. The space of inputs to $\mathcal{F}_{\mathsf{pRO}}$ is exponential in the security parameter while the simulator and environment run in only polynomial time. Note that the key seeds that the simulator programs $\mathcal{F}_{\mathsf{pRO}}$ on are either chosen completely at random or using a PRG. The probability that these seeds collide is therefore negligible. Furthermore, since the environment can't guess the seeds that the simulator will need to program, and since it can only program a polynomial number of keys, the probability that it can cause a collision is also negligible.

10 Putting It All Together: Composition Theorems

It is left to put the pieces together and assert the security of the composed protocol (see Figure 1 on page 8 for a gaphical depiction of the various components). 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 1. *Assume that the KDM module used by Π_{eKE} is a CPRFG, that the DDH assumption holds for the group 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 $\Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}$.*

Next we use Theorem 6 together with Corollary 1 and Proposition 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 2. *Assume that* $(\mathsf{MAC}, \mathsf{Verify})$ *used in Π_{aead} is unforgeable, that the KDM module used by Π_{eKE} is a CPRFG, that the DDH assumption holds for the group 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 $\Pi_{\mathsf{mKE}}, \Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{pRO}}$.*

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 2 and the UC with Global Subroutines (UCGS) theorem says that:

Corollary 3. *Assume that* $(\mathsf{MAC}, \mathsf{Verify})$ *used in Π_{aead} is unforgeable, that the KDM module used by Π_{eKE} is a CPRFG, that the DDH assumption holds for the group 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 $\Pi_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{pRO}}$.*

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 1, together with Corrolary 3 and the UCGS theorem says that:

Corollary 4. *Assume that* $(\mathsf{MAC}, \mathsf{Verify})$ *used in Π_{aead} is unforgeable, that the KDM module used by Π_{eKE} is a CPRFG, that the DDH assumption holds for the group 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 $\mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{pRO}}$.*

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.: The double ratchet: security notions, proofs, and modularization for the signal protocol. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 129–158. Springer (2019)
3. 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)
4. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: CCS. ACM (2021)
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: TCC (2). Lecture Notes in Computer Science, vol. 13043, pp. 626–657. Springer (2021)
7. Balli, F., Rösler, P., Vaudenay, S.: Determining the core primitive for optimally secure ratcheting. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part III. LNCS, vol. 12493, pp. 621–650. Springer, Heidelberg (Dec 2020)
8. Bellare, M., Canetti, R., Krawczyk, H.: Pseudorandom functions revisited: The cascade construction and its concrete security. In: Proceedings of 37th Conference on Foundations of Computer Science. pp. 514–523. IEEE (1996)
9. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO'93. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (Aug 1994)
10. Bellare, M., Rogaway, P.: Provably secure session key distribution: The three party case. In: 27th ACM STOC. pp. 57–66. ACM Press (May / Jun 1995)
11. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 619–650. Springer, Heidelberg (Aug 2017)
12. Bienstock, A., Dodis, Y., Rösler, P.: On the price of concurrency in group ratcheting protocols. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 198–228. Springer, Heidelberg (Nov 2020)
13. Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the signal double ratchet algorithm. Cryptology ePrint Archive, Report 2022/355 (2022), https://ia.cr/2022/355
14. Blake-Wilson, S., Johnson, D., Menezes, A.: Key agreement protocols and their security analysis. In: Darnell, M. (ed.) 6th IMA International Conference on Cryptography and Coding. LNCS, vol. 1355, pp. 30–45. Springer, Heidelberg (Dec 1997)
15. 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)
16. 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)
17. Borisov, N., Goldberg, I., Brewer, E.A.: Off-the-record communication, or, why not to use PGP. In: WPES. pp. 77–84. ACM (2004)
18. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 501–519. Springer, Heidelberg (Mar 2014)
19. Brendel, J., Fischlin, M., Günther, F., Janson, C., Stebila, D.: Towards post-quantum security for signal's X3DH handshake. In: SAC. Lecture Notes in Computer Science, vol. 12804, pp. 404–430. Springer (2020)
20. 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)
21. 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://ia.cr/2018/165
22. 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)
23. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001)
24. Canetti, R.: Universally composable security. J. ACM 67(5), 28:1–28:94 (2020)
25. 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)

26. Canetti, R., Halevi, S., Herzberg, A.: Maintaining authenticated communication in the presence of break-ins. J. Cryptol. 13(1), 61–105 (2000)
27. Canetti, R., Herzog, J.: Universally composable symbolic security analysis. J. Cryptol. 24(1), 83–147 (2011), https://doi.org/10.1007/s00145-009-9055-0
28. 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)
29. 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)
30. 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)
31. 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)
32. 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)
33. 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)
34. 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)
35. 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)
36. 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)
37. 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)
38. Drucker, N., Gueron, S.: Continuous key agreement with reduced bandwidth. Cryptology ePrint Archive, Report 2019/088 (2019), https://eprint.iacr.org/2019/088
39. 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)
40. 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)
41. 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)
42. 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)
43. 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)
44. Keybase blog: New cryptographic tools on keybase. https://keybase.io/blog/crypto (2020)
45. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 669–684. ACM Press (Nov 2013)
46. 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
47. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 546–566. Springer, Heidelberg (Aug 2005)
48. Krohn, M.: Zoom rolling out end-to-end encryption offering. https://blog.zoom.us/zoom-rolling-out-end-to-end-encryption-offering/ (2020)
49. Kudla, C., Paterson, K.G.: Modular security proofs for key agreement protocols. In: Roy, B.K. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 549–565. Springer, Heidelberg (Dec 2005)
50. LaMacchia, B.A., Lauter, K., Mityagin, A.: Stronger security of authenticated key exchange. In: Susilo, W., Liu, J.K., Mu, Y. (eds.) ProvSec 2007. LNCS, vol. 4784, pp. 1–16. Springer, Heidelberg (Nov 2007)

51. Lauter, K., Mityagin, A.: Security analysis of KEA authenticated key exchange protocol. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 378–394. Springer, Heidelberg (Apr 2006)
52. Marlinspike, M., Perrin, T.: The X3DH key agreement protocol. https://signal.org/docs/specifications/x3dh/ (2016)
53. Martiny, I., Kaptchuk, G., Aviv, A.J., Roche, D.S., Wustrow, E.: Improving signal's sealed sender. In: NDSS. The Internet Society (2021)
54. Maurer, U.: Constructive cryptography - a primer (invited paper). In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, p. 1. Springer, Heidelberg (Jan 2010)
55. 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)
56. 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)
57. Open Whisper Systems: Technical information: Specifications and software libraries for developers. https://signal.org/docs/ (2016)
58. 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)
59. 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)
60. 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)
61. 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)
62. 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)
63. 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)
64. Unger, N., Goldberg, I.: Improved strongly deniable authenticated key exchanges for secure messaging. PoPETs 2018(1), 21–66 (Jan 2018)
65. Ustaoglu, B.: Obtaining a secure and efficient key agreement protocol from (H)MQV and NAXOS. Des. Codes Cryptogr. 46(3), 329–342 (2008)
66. 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)
67. WhatsApp LLC: About end-to-end encryption. https://faq.whatsapp.com/general/security-and-privacy/end-to-end-encryption/ (2021)
68. 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)