Universally Composable End-to-End Secure Messaging[*]

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

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

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

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

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

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

1 Introduction

Secure communication, namely allowing Alice and Bob to exchange messages securely, over an untrusted communication channel, without having to trust any

[*] 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.

intermediate component or party, is perhaps the quintessential cryptographic problem. Indeed, constructing and breaking secure communication protocols, as well as modeling security concerns and guarantees, providing a security analysis, and then breaking the modeling and analysis, has been a mainstay of cryptography since its early days.

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

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

The Signal protocol has been designed to give a response to these specific challenges of secure messaging, and in doing so it has revolutionized the concept of secure communication over the Internet in many ways. Built on top of predecessors like Off-The-Record [14], the Signal protocol is currently used to transmit hundreds of billions of messages per day [49].

Modeling the requirements of secure messaging in general, and analyzing the security properties of the Signal protocol in particular, has proved to be challenging and has inspired multiple analytical works [1–3, 7, 10, 11, 13, 15, 17, 25–33, 35–37, 46–48, 52–55, 57]. Some of these works directly address the Signal architecture and realization, whereas others propose new cryptographic primitives that are inspired by Signal's various modules.

The need for composable security analysis. Standalone security analyses of the Signal protocol are not always sufficient to capture the security of an entire messaging ecosystem that includes (components of) the Signal protocol. People typically participate concurrently in several conversations spanning several multi-platform chat services (e.g., smartphone and web), and the subtleties between a chat service and the underlying messaging protocol have led to network and systems security issues (e.g., [31, 32, 40]). For example, the Signal protocol is combined with other cryptographic protocols in WhatsApp [56] to perform abuse reporting or Status [50] and Slyo [51] to perform cryptocurrency transactions and Tor-style onion routing.

Moreover, Signal isn't always employed as a single monolithic protocol. Rather, variations and subcomponents of the Signal protocol are used within the Noise protocol family [45], file sharing services like Keybase [38] (which performs less frequent ratcheting), and videoconferencing services like Zoom [39] (which isn't concerned with asynchrony).

This state of affairs seems to call for a security analysis within a framework that allows for modular analysis and composable security guarantees. First steps in this direction were taken by the work of Jost, Maurer, and Mularczyk [37] that defines an abstract ratcheting service within the Constructive Cryptography framework [41, 42], and concurrent work by Bienstock et al. [12] that formulates an ideal functionality of the Signal protocol within the UC framework (see Section 1.5 for details). However, neither of these works give a modular decomposition of Signal into its basic components (as described in [44].)

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

Security of Signal in face of adaptive corruptions. Another potentially thorny aspect of the security of secure messaging protocols (Signal included) is the need to protect against an adversary that decides whom and when to corrupt, adaptively, based on all the communication seen so far. Indeed, not only is standard semantic security not known to imply security in this setting: there exist encryption schemes that are semantically secure (under reasonable intractability assumptions) but completely break in such a setting [34].

1.1 This Work

This work proposes a modular analysis of the Signal protocol and its components using the language of universally composable (UC) security [18, 19]. We focus on modeling Signal at the level specified in their documentation [44] (i.e., not limited to any single choice, of cipher suite), taking care to adhere to the abstractions within the specification.

We provide an ideal functionality, $\mathcal{F}_{\mathsf{SM}}$, for secure messaging along with individual ideal functionalities that capture each module within Signal's architecture. We then compose the modules to realize the top-level secure messaging functionality and demonstrate how to realize the modules in a manner consistent with the Signal specification [44]. Our instantiation achieves adaptive security against transient corruptions while making minimal use of the random oracle model. This combination of composability and modularity makes Signal and its components conveniently plug-and-play: future analyses can easily re-purpose or swap out instantiations of the modules in this work without needing to redo most of the security analysis.

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

The rest of the Introduction is organized as follows. Section 1.2 presents and motivates our formulation of $\mathcal{F}_{\mathsf{SM}}$. Section 1.3 presents and motivates the formulation of the individual modules, and describes how these modules can be realized. Section 1.5 discusses related work.

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

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

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

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

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

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

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

Modeling of PKI and long term keys. We directly model Signal's specific design for the public keys and associated secret keys that are used to identify parties across multiple sessions. Specifically, we formulate a "PKI" functionality $\mathcal{F}_{\mathsf{DIR}}$ that models a public "bulletin board," which stores the long-term, ephemeral, and one-time public keys associated with identities of parties. In addition, we model "long term private key" module $\mathcal{F}_{\mathsf{LTM}}$ for each identity. This module stores the private keys associated with the public keys of the corresponding party. Both functionalities are modeled as *global,* namely they are used as subroutines by multiple instances of $\mathcal{F}_{\mathsf{SM}}$. This modeling is what allows to tie the two participants of a session to long-term identities. Similarily to [16, 24], we treat these modules as incorruptible. It is stressed, however, that, following the Signal architecture, our realization of $\mathcal{F}_{\mathsf{SM}}$ calls the $\mathcal{F}_{\mathsf{LTM}}$ module of each party exactly once, at the beginning of the session.

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

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

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

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

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

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

First, Signal does not give parties a way to detect whether their peers have received forged messages in their name during corruption. (Such situations may occur when either party was corrupted in the past and then recovered.) This represents a known weakness of Signal [15, 31]. Consequently, $\mathcal{F}_{\mathsf{SM}}$ exhibits a similar behavior.

Second, as remarked in the Signal documentation [44], when one of the parties is compromised, an adversary can "fork" the messaging session. That is, the adversary can create a person-in-the-middle situation where both parties believe they are talking with each other in a joint session, and yet they are actually both talking with the adversary. Furthermore, this can remain the case indefinitely, even when no party is compromised anymore. (In fact, we know this situation is inherent in an unauthenticated network with transient attacks, at least without repeated use of a long-term uncompromised public key [20].) While such a situation is mentioned in the Signal design documents, pinpointing and analyzing the conditions under which forking occurs has not been formally done before our work and the concurrent work by Bienstock et al. [12]. In our modeling, $\mathcal{F}_{\mathsf{SM}}$ forks when one of the parties is compromised, and at the same time the other party successfully decapsulates a forged incoming message with an "epoch ID" that is different than the one used by the sender. In that case, $\mathcal{F}_{\mathsf{SM}}$ remains forked indefinitely, without any additional corruptions.

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

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

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

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

The Signal architecture document [44] de-composes the above mechanism into 3 main cryptographic modules, plus non-cryptographic code used to put these modules together. The modules are: (1) a symmetric authenticated encryption with associated data (AEAD) scheme that is applied to individual messages; (2) a symmetric key *ratcheting* mechanism to evolve the key between messages within an epoch; (3) an asymmetric key *ratcheting* (or "continuous key exchange") mechanism to evolve the "root chain." Since these modules are useful for applications beyond this particular protocol, we follow this partitioning and decompose Signal's protocol into similar components. (Our partitioning into components is also inspired by that of Alwen et al. [1].)

We model the security of each component as an ideal functionality within the UC framework. (These are $\mathcal{F}_{\mathsf{aead}}, \mathcal{F}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{eKE}}$, respectively.) This allows us to distill the properties provided by each module and demonstrate how they can be composed, along with the appropriate management code to obtain the desired functionality—namely to realize $\mathcal{F}_{\mathsf{SM}}$. The management code (specifically, protocols $\Pi_{\mathsf{fs\_aead}}$ and Π_{SGNL}), does not directly access any keying material. Indeed, these protocols realise their respective specifications, namely $\mathcal{F}_{\mathsf{fs\_aead}}$ and $\mathcal{F}_{\mathsf{SM}}$, perfectly—see Theorems 1 and 2.

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

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

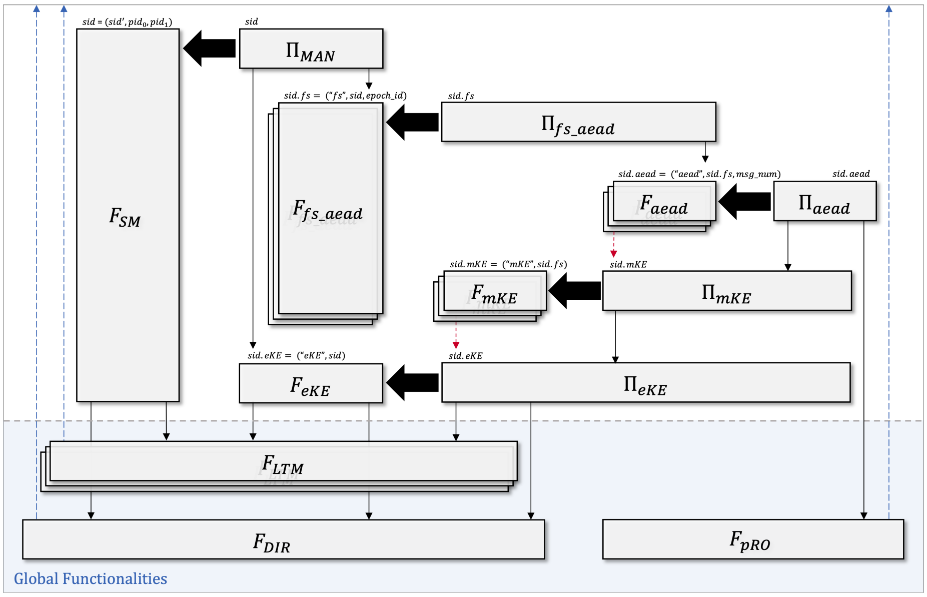
$sid = (sid', pid_0, pid_1)$
sid
Π_{MAN}
F_{SM}
$sid.fs = (\text{"}fs\text{"}, sid, epoch\_id)$
$sid.fs$
F_{fs_aead}
Π_{fs_aead}
$sid.aead = (\text{"}aead\text{"}, sid.fs, msg\_num)$
$sid.aead$
F_{aead}
Π_{aead}
$sid.mKE = (\text{"}mKE\text{"}, sid.fs)$
$sid.mKE$
F_{mKE}
Π_{mKE}
$sid.eKE = (\text{"}eKE\text{"}, sid)$
$sid.eKE$
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 and protocols by Π. 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 corresponding instance of Π_{mKE}) are global for $\mathcal{F}_{\mathsf{aead}}$.

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

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

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

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

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

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

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

$\mathcal{F}_{\mathsf{aead}}$ is realized via protocol Π_{aead}, which employs an authenticated encryption algorithm using a key obtained from $\mathcal{F}_{\mathsf{mKE}}$. If we had opted to assert security against non-adaptive corruptions, any standard AEAD scheme would do. However, we strive to provide simulation-based security in the presence of fully adaptive corruptions, which is provably impossible in the plain model whenever the key is shorter than the plaintext [43]. We get around this issue by realizing $\mathcal{F}_{\mathsf{aead}}$ in the programmable random oracle model. While we provide a very simple AEAD protocol in this model, many common block cipher-based AEADs can also realize $\mathcal{F}_{\mathsf{aead}}$ provided we model the block cipher as a programmable random oracle. It is stressed however that the random oracle is used *only* in the case of short keys and adaptive corruptions. In particular, when corruptions are non-adaptive or the plaintext is sufficiently short, our protocol continues to UC-realize $\mathcal{F}_{\mathsf{aead}}$ even when the random oracle is replaced by the identity function.

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

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

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

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

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

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

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

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

On the other extreme, it can be seen that no plain-model instantiation of the KDF module, with bounded-size local state, can possibly realize $\mathcal{F}_{\mathsf{eKE}}$ in our setting. Indeed, since the adversary can obtain unboundedly many alternative keys for a given epoch, where all keys are generated using the same bounded-size secret state, the Nielsen bound [43] applies.

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

1.4 Streamlining UC analysis

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

Multiple levels of global state. Our analysis makes extensive use of universal composition with global state (UCGS) within the plain UC model, as formulated and proven in [5]. Specifically, we use UCGS to model a global directory that holds the public keys of parties, as well as the long-term storage, within each party, of the secret keys associated with said public keys. Similarly, we use UCGS to model the fact that a single instance of $\mathcal{F}_{\mathsf{eKE}}$ is used by multiple instances of $\mathit{\Pi}_{\mathsf{mKE}}$, and that a single instance of $\mathcal{F}_{\mathsf{mKE}}$ is used by multiple instances of $\mathit{\Pi}_{\mathsf{aead}}$. The random oracle is also modeled as a global functionality.

To facilitate our multi-layer use of the UCGS theorem we also prove a simple-but-useful lemma that allows us to get around the following difficulty. Recall that the UCGS theorem allows demonstrating that protocol π UC-realizes functionality $\mathcal{F}$ in the presence of some other 'global' functionality $\mathcal{G}$ that takes inputs from π, $\mathcal{F}$, and also potentially directly from the environment. Furthermore, we would like to use *multiple levels* of UCGS: 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 $\mathcal{G}$. However, such implication is not true in general [6, 24].

Lemma 1 in Section 2 asserts that, if γ UC-realizes $\mathcal{G}$ via some simulator $\mathcal{S}$, then for any π that UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}$, it also holds that π UC-realizes $\mathcal{F}$ in the presence of $\mathcal{G}_{\mathcal{S}}$, where $\mathcal{G}_{\mathcal{S}}$ be the functionality that combines $\mathcal{G}$ and $\mathcal{S}$ in the natural way. We then show that, for the protocols in this work, having access to $\mathcal{G}_{\mathcal{S}}$ suffices.

Multiple levels of corruptions. The UC framework allows the adversary to adaptively and individually corrupt each party in each module within a composite protocol. While this is very general, it makes the handling of corruption events (where typically the internal states of multiple modules are exposed together) rather complex. We thus adopt a somewhat simpler modeling of party corruption: We let the environment directly corrupt parties and obtain their local states. Furthermore, a corrupted module forwards the corruption notice to all its subroutines and collects the local states of all to report to the environment. Ideal functionalities operate similarly, except that they ask their respective simulators for the appropriate simulated local states. In addition to being simpler, this modeling provide tighter correspondence between the real and ideal executions and is thus preferable whenever realizable (which is the case in this work).

1.5 Related Work

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

There is a long line of research into the design and analysis of two-party Signal messaging, its subcomponents, and variants of the Signal architecture; this research builds upon decades of study into key exchange protocols (e.g., [8, 9,22,23]) and self-healing after corruption (e.g., [20,28,30]). Some of these secure messaging analyses purposely consider a limited notion of adaptive security in order to analyze instantiations of Signal based on standardized crypto primitives (e.g., [1,10,32,36,57]). Other works consider a strong threat model in which the adversary is malicious, fully adaptive, and can tamper with local state [4,7,35, 37,46], which then intrinsically requires strong HIBE-like primitives that depart from the Signal specification. By contrast, we follow a middle ground in this work: our adversary is fully adaptive and has no restrictions on when it can corrupt a party, yet its corruptions are instantaneous and passive.

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

There are two prior works that perform composable analyses of Signal. In concurrent work to our own, Bienstock et al. [12] provide an alternative modeling of an ideal secure messaging within the UC framework and demonstrate how the Signal protocol can be modeled in a way that is shown to realize their formulation of ideal secure messaging. Like this work, they demonstrate several shortcomings of previous formulations, such as overlooking the effect of choosing keys too early or keeping them around for too long. They also propose and analyze an enhancement of the double ratchet structure that helps parties regain security faster following a compromise event. Additionally, Jost, Maurer, and Mularczyk

[37] conduct an analysis in the constructive cryptography framework. Their work provides a model for message transmission as well as one for ratcheting protocols.

That said, the ideal functionalities in [12] and [37] differ from our $\mathcal{F}_{\mathsf{SM}}$ in a number of ways. First, their modeling does not account for the session initiation process, nor the PKI and long-term key modules that are an integral part of any secure messaging application. Second, they include the communication medium as part of the protocol, which (a) makes it harder to argue about immediate decryption and (b) means that an instantiation of Signal would have to include an entire TCP/IP stack, which weakens modularity and inhibits the use of Signal as a sub-routine within larger functionalities.

Additionally, Bienstock et al. [12] models all key derivation modules as random oracles rather than formalizing the partition of continuous key exchange components within the UC framework as done in this work, and their modeling forces the "calling protocol" to keep track of—and ensure uniqueness for—the message IDs for the Secure Messaging functionality/protocol, which might create a security risk. On the other hand, [12] accounts for adversarial choice of randomness, which our modeling does not account for. Also, Jost et al. [37] requires explicit modeling of a global event history (a list of events having happened at each module), restricts the real-world adversary's events based on this global event history, and employs a HIBE-based implementation that is quite different than that of Signal (and ours) and requires heavier cryptographic primitives.

2 Universally Composable Security: New Capabilities

This work makes extensive use of UC with global subroutines, which allow analysis that a protocol Π UC-realizes functionality $\mathcal{F}$ *in the presence of* a global subroutine G that is not subroutine-respecting. Due to space constraints, we defer a primer of UC security (with global subroutines) to the full version of this work [21]. In this space, we describe two new modeling techniques that simplify our analysis, and may be of more general interest.

The first technique relates to applying the UC theorem to global functionalities. As stated in Section 1.4, the analysis in this work requires the ability to apply composition with global state across multiple layers of Fig. 1. We prove the following lemma in the full version of this work [21].

Lemma 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 $\mathcal{F}_{\mathcal{S}}$.*

The second technique is that, in order to model the immediate encryption and immediate decryption properties of secure messaging, we require the adversary to upload static code (which we call $\mathcal{I}$) to the global $\mathcal{F}_{\mathsf{lib}}$—shown in Fig. 2—that the relevant functionality will run in honest cases of the execution. This static code is specific to the protocol that realizes the functionality, and essentially it

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

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

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

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

Fig. 2. The code library functionality, $\mathcal{F}_{\mathsf{lib}}$

acts as the ideal-world simulator during an honest execution. This ensures that the functionality does not need to wait for the adversary to encrypt or decrypt messages that are not corrupted. In cases where the message or ciphertext is corrupted, the fully adaptive adversary is called for input (for example, asking $\mathcal{A}$ to encrypt a message or decrypt a ciphertext). The state of the static code $\mathcal{I}$ is maintained across calls in a variable $\mathsf{state}_{\mathcal{I}}$, and it is sent to the adversary upon corruption. $\mathcal{F}_{\mathsf{lib}}$ is global because the static code must be defined at the time that the functionality is instantiated; however, the use of $\mathcal{F}_{\mathsf{lib}}$ specifically is mainly a matter of plumbing rather than a topic of conceptual importance.

3 Formal Modeling and Analysis

In this section, we showcase our modular, iterative process for decomposing the ideal secure messaging functionality $\mathcal{F}_{\mathsf{SM}}$ into a collection of functionalities and protocols that each address one specific purpose. After fully specifying $\mathcal{F}_{\mathsf{SM}}$ itself, we present its realization at the "second level" of Fig. 1 by Π_{SGNL}, $\mathcal{F}_{\mathsf{fs\_aead}}$, and $\mathcal{F}_{\mathsf{eKE}}$, and at the "third level" by functionalities $\mathcal{F}_{\mathsf{aead}}$ and $\mathcal{F}_{\mathsf{mKE}}$.

Due to space constraints, we only give brief descriptions of these functionalities here and defer more exposition to the full version of this work [21]. The full version also contains rigorous specifications of the remaining protocols (on the far right of Fig. 1) and all underlying global functionalities (in blue at the bottom of Fig. 1), along with proofs of all theorems in this section.

Secure Messaging Functionality. Our top-level functionality $\mathcal{F}_{\mathsf{SM}}$ can be found in Figs. 3 to 4. It takes two types of inputs: `SendMessage` is used to encapsulate a message for sending to the peer, whereas `ReceiveMessage` is used to decapsulate a received message. We also have a `Corrupt` input; this is a 'modeling input' that is used to capture party corruption. In addition, $\mathcal{F}_{\mathsf{SM}}$ takes a number of 'side channel' messages from the adversary which are used to fine-tune the security guarantees. It relies on three global functionalities whose specifications are provided in the full version of this work [21]: $\mathcal{F}_{\mathsf{DIR}}$ representing the directory of public keys, $\mathcal{F}_{\mathsf{LTM}}$ representing the long-term key storage within a party, and a programmable random oracle $\mathcal{F}_{\mathsf{pRO}}$.

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}.

There are three primary takeaways from the design of Π_{SGNL} (Fig. 5): 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.

The epoch key exchange functionality $\mathcal{F}_{\mathsf{eKE}}$ (Figs. 6 to 7) 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. It also provides recovery from a state compromise (aka, post-compromise security).

The forward secure authenticated encryption functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ (Figs. 8 to 9) 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 $\Pi_{\mathsf{fs\_aead}}$ realizes $\mathcal{F}_{\mathsf{fs\_aead}}$ by outsourcing authenticated encryption and decryption of each message to separate $\mathcal{F}_{\mathsf{aead}}$ instances, described below.

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

The Symmetric Ratchet: Realizing $\mathcal{F}_{\mathsf{fs\_aead}}$. Next, we decompose the symmetric key component of Signal into two smaller pieces: a one-time-use authenticated encryption routine $\mathcal{F}_{\mathsf{aead}}$, and a message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$

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

The local session ID is parsed as $\mathsf{sid} = (sid', \mathsf{pid}_0, \mathsf{pid}_1)$. Inputs arriving from machines whose identity is neither pid_0 nor pid_1 are ignored. //For notational simplicity we assume some fixed interpretation of pid_0 and pid_1 as complete identities of the two calling machines.
It also has *internal adversary code* $\mathcal{I} = \mathcal{I}_{\mathsf{sm}}$. We initialize the state for $\mathcal{I}$ to be $\mathsf{state}_{\mathcal{I}} = \perp$.

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 $\mathsf{advControl} = \{\}$, $\mathsf{id\_dict} = \{\}$, and $\mathsf{N\_dict} = \{\}$. Initialize $\mathsf{advControl}[\mathsf{epoch\_num}_0] = \perp$ and $\mathsf{advControl}[e] = \infty$ for all $e \geq 0$. //$\mathsf{advControl}$ will record which parties are adversarially controlled in each epoch, $\mathsf{id\_dict}$ maps epoch id's to epoch numbers, and $\mathsf{N\_dict}$ will hold the number of messages sent in each epoch.
   - Call $\mathcal{F}_{\mathsf{lib}}$ with input $\mathcal{F}_{\mathsf{SM}}$ to obtain the internal code $\mathcal{I}$.
2. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$. Increment $\mathsf{sent\_msgnum}_i$ by 1.
3. If $\mathtt{leak} \in \mathsf{advControl}[\mathsf{epoch\_num}_i]$ or $\mathsf{diverge\_parties} = \mathit{true}$: Send a backdoor message $(\mathsf{state}_{\mathcal{I}}, \mathsf{SendMessage}, \mathsf{pid}, m)$ to $\mathcal{A}$.
4. Else ($\mathtt{leak} \notin \mathsf{advControl}[\mathsf{epoch\_num}_i]$ and $\mathsf{diverge\_parties} = \mathit{false}$): Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \mathsf{SendMessage}, \mathsf{pid}, |m|)$
5. Upon obtaining $(\mathsf{state}'_{\mathcal{I}}, \mathsf{SendMessage}, \mathsf{pid}, \mathsf{epoch\_id}, c)$ from $\mathcal{A}$ or $\mathcal{I}$ do:
   - Update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$.
   - If $\mathsf{sent\_msgnum}_i == 1$: If $\mathsf{epoch\_id}$ equals any of the keys in the dictionary $\mathsf{id\_dict}$ then end the activation. Else record $\mathsf{id\_dict}[\mathsf{epoch\_id}] = \mathsf{epoch\_num}_i$.
   - Set $h = (\mathsf{epoch\_id}, \mathsf{sent\_msgnum}_i, N\_\mathsf{self}_i)$. //$N\_\mathsf{self}_i$ holds the # of messages sent by 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.

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 $(\mathsf{state}_{\mathcal{I}}, \mathtt{ReportState}, \mathsf{pid}_i, \mathsf{pending\_msgs})$ to $\mathcal{A}$.
5. On receiving a state $(\mathtt{ReportState}, \mathsf{pid}_i, S)$ from $\mathcal{A}$, send S to Env.

Fig. 3. The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$

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

The local session ID is parsed as $\mathsf{sid} = (sid', \mathsf{pid}_0, \mathsf{pid}_1)$. Inputs arriving from machines whose identity is neither pid_0 nor pid_1 are ignored. //For notational simplicity we assume some fixed interpretation of pid_0 and pid_1 as complete identities of the two calling machines.
It also has *internal adversary code* $\mathcal{I} = \mathcal{I}_{\mathsf{sm}}$. We initialize the state for $\mathcal{I}$ to be $\mathsf{state}_{\mathcal{I}} = \bot$.

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 $(\mathsf{GetResponseKeys}, \mathsf{pid}_0, \mathsf{pid}_1)$ to $\mathcal{F}_{\mathsf{DIR}}$.
   - Upon receiving output $(\mathsf{GetResponseKeys}, \mathsf{pid}_0, \mathsf{ik}_0^{\mathsf{pk}})$ 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 $\mathsf{sent\_msgnum}_i = 0$, 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. If ($\mathsf{diverge\_parties} = \mathit{false}$ and $\mathtt{inject} \notin \mathsf{advControl}[\mathsf{epoch\_num}]$): Run $\mathcal{I}(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, \mathsf{pid}, h, c)$ //Honest Case
7. Else: send backdoor message $(\mathsf{state}_{\mathcal{I}}, \mathtt{inject}, \mathsf{pid}, h, c)$ to $\mathcal{A}$ // $\mathcal{F}_{\mathsf{SM}}$ is asking the adversary for advice on how to decrypt c.
8. On receiving $(\mathsf{state}'_{\mathcal{I}}, \mathtt{inject}, h, c, v)$ from $\mathcal{A}$ or $\mathcal{I}$ update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$ and:
   If $(\mathsf{sender}, h, c, m)$ is recorded then record $(\mathtt{Authenticate}, \mathsf{pid}, h, c, 1)$ and set $m^* = m$. Else:
   - If $v = \bot$: record $(\mathtt{Authenticate}, \mathsf{pid}, h, c, 0)$ and output $(\mathtt{ReceiveMessage}, h, c, \mathtt{Fail})$.
   - If $v \neq \bot$ 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 $v \neq \bot$ 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.
9. If $\mathsf{epoch\_num}_i < \mathsf{epoch\_num}$, do: //we only get to this step if decryption is successful
   - Set $\mathsf{N\_dict}[\mathsf{epoch\_num} - 2] = N$, $\mathsf{epoch\_num}_i \mathrel{+}= 2$, $N\mathsf{\_self}_i = \mathsf{sent\_msgnum}_i$, and $\mathsf{sent\_msgnum}_i = 0$.
   - if $\mathsf{diverge\_parties} = \mathit{false}$ 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 $\mathsf{advControl}[\mathsf{epoch\_num}] = \bot$.
10. Output $(\mathtt{ReceiveMessage}, h, m^*)$ to pid.

Fig. 4. The Secure Messaging Functionality $\mathcal{F}_{\mathsf{SM}}$

Π_{SGNL}

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

1. If this is the first activation do: //initialization for the initiator of the session
   - Parse the local session id sid to retrieve the party identifiers $(\mathsf{pid}_0, \mathsf{pid}_1)$ for the initiator and responder. If pid_0 is different from either the local party identifier pid, or the party identifier of pid, end the activation.
   - Initialize $\mathsf{epoch\_id}_{\mathsf{self}} = \perp$, $\mathsf{epoch\_id}_{\mathsf{partner}} = \perp$, $\mathsf{sent\_msg\_num} = 0$, $N_{\mathsf{last}} = 0$.
   - Provide input $(\texttt{ConfirmReceivingEpoch}, \perp)$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
   - On receiving $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id})$ from $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$, set $\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}$.
   - Initialize a list $\mathsf{receiving\_epochs} = []$.
2. Provide input $(\texttt{Encrypt}, m, N_{\mathsf{last}})$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$, where $\mathsf{sid}.fs = (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{self}})$. //$\mathcal{F}_{\mathsf{fs\_aead}}$ already knows epoch_id and 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 $\mathsf{epoch\_id}_{\mathsf{self}} = \perp$, $\mathsf{epoch\_id}_{\mathsf{partner}} = \perp$, $\mathsf{sent\_msg\_num} = 0$ and $N_{\mathsf{last}} = 0$, $\mathsf{received\_msg\_num} = 0$.
   - Initialize a dictionary $\mathsf{missed\_msgs} = \{\}$ and a list $\mathsf{receiving\_epochs} = []$.
2. Provide input $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs = (\mathsf{sid}, \mathsf{epoch\_id}))$.
3. Upon receiving $(\texttt{Decrypt}, c, \mathsf{msg\_num}, N, v)$ from $(\mathcal{F}_{\mathsf{fs\_aead}}, \mathsf{sid}.fs)$: if $v = \texttt{Fail}$ then send $(\texttt{ReceiveMessage}, h, \mathsf{ad}, \texttt{Fail})$ to pid. //Otherwise, v is the decrypted message
4. While $\mathsf{msg\_num} > \mathsf{received\_msg\_num}$:
   //note down any expected messages
   - Append $\mathsf{received\_msg\_num}$ to the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$.
   - Increment $\mathsf{received\_msg\_num} + = 1$.
5. If $\mathsf{msg\_num}$ is in the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$:
   - remove it from the list.
   - If the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$ is now an empty list then remove $\mathsf{epoch\_id}$ from $\mathsf{missed\_msgs}.keys$.
6. Else $(\mathsf{msg\_num} \notin \mathsf{missed\_msgs}[\mathsf{epoch\_id}])$:
   - If $\mathsf{epoch\_id} = \mathsf{epoch\_id}_{\mathsf{partner}}$ or $\mathsf{sent\_msg\_num} = 0$, output $(\texttt{ReceiveMessage}, h, c, \perp)$. Otherwise continue. //Starting new epoch–ratchet forward
   - Append the numbers $\mathsf{received\_msg\_num}, \ldots, N$ to the entry $\mathsf{missed\_msgs}[\mathsf{epoch\_id}]$.
   - Send $(\texttt{StopDecrypting}, N)$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{partner}}))$. //'Closing' the $\mathcal{F}_{\mathsf{fs\_aead}}$ for the last epoch.
   - On receiving $(\texttt{StopDecrypting}, \texttt{Success})$, update $\mathsf{epoch\_id}_{\mathsf{partner}} = \mathsf{epoch\_id}$, and send $(\texttt{StopEncrypting})$ to $(\mathcal{F}_{\mathsf{fs\_aead}}, (\mathsf{sid}, \mathsf{epoch\_id}_{\mathsf{self}}))$.
   - On receiving $(\texttt{StopEncrypting}, \texttt{Success})$, send $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id})$ to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE)$.
   - On receiving $(\texttt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}^*)$, update $\mathsf{epoch\_id}_{\mathsf{self}} = \mathsf{epoch\_id}^*$, $N_{\mathsf{last}} = \mathsf{sent\_msg\_num}$, and $\mathsf{sent\_msg\_num} = 0$.
7. Output $(\texttt{ReceiveMessage}, h, c, v)$ to pid while deleting the decrypted message v.

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

1. Initialize a list S and send $(\texttt{Corrupt})$ as input to $(\mathcal{F}_{\mathsf{eKE}}, \mathsf{sid}.eKE = \text{“}eKE\text{”}, \mathsf{sid})$.
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. 5. The Signal Protocol, Π_{SGNL}

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

This functionality has a session id $\mathsf{sid}.eKE$ that takes the following format: $\mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid})$. Inputs arriving from machines whose identity is neither 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 $(\mathtt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}^*)$ from $(\mathit{\Pi}_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$:

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

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

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

that interfaces with the epoch key exchange to produce the symmetric chain keys.

Each authenticated encryption functionality instance $\mathcal{F}_{\mathsf{aead}}$ (Fig. 10) handles the encryption, decryption, and authentication of a particular message for a particular epoch. It hands the ciphertext or message back to $\mathit{\Pi}_{\mathsf{fs\_aead}}$.

Theorem 2. *Protocol $\mathit{\Pi}_{\mathsf{fs\_aead}}$ (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 $\mathcal{F}_{\mathsf{eKE}}^{\mathit{\Pi}_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.*

Note that the simulator $\mathcal{S}_{\mathsf{eKE}}$, along with a proof of this theorem, are deferred to the full version of this work [21].

Each instance of the message key exchange functionality $\mathcal{F}_{\mathsf{mKE}}$ (Fig. 11) handles the key derivation for the symmetric ratchet for a particular epoch. Specifically, it provides $\mathsf{key\_seed}$'s to $\mathit{\Pi}_{\mathsf{aead}}$ instances that are then expanded to any length using the global random oracle $\mathcal{F}_{\mathsf{pRO}}$. When instructed, it also closes

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

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

1. Set i such that $\mathsf{pid} = \mathsf{pid}_i$.
2. If $\texttt{ConfirmReceivingEpoch}$ has never been run successfully (i.e $\mathsf{epoch\_id\_self}_0$ hasn't been initialized) or $\mathsf{got\_sending\_key}_i = \mathit{true}$, then end the activation. //the functionality isn't initialized or the sending key for the current epoch has already been retrieved
3. Sample $\mathsf{sending\_chain\_key}_i \xleftarrow{\$} \mathcal{K}_{ep}$ from the key distribution. //In the honest case, the key is not known to the adversary. Otherwise the key will get overwritten in the following step.
4. If $\mathsf{diverge\_parties} = \mathit{true}$, or $\mathsf{epoch\_num}_i \in \mathsf{compromised\_epochs}$, then:
   - Send backdoor message $(\mathsf{state}_{\mathcal{I}}, \texttt{GetSendingKey}, i)$ to $\mathcal{A}$
   - On receiving backdoor message $(\mathsf{state}'_{\mathcal{I}}, \texttt{GetSendingKey}, i, K_{\mathsf{send}})$ from $\mathcal{A}$, update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$ and set $\mathsf{sending\_chain\_key}_i = K_{\mathsf{send}}$.
5. Set $\mathsf{got\_sending\_key}_i = \mathit{true}$ 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 $(\texttt{ComputeReceivingRootKey}, k)$, call $\mathcal{F}_{\mathsf{lib}}$ to obtain internal code $\mathcal{I}$. Initialize the state for $\mathcal{I}$ to be $\mathsf{state}_{\mathcal{I}} = \perp$.
4. If $\mathsf{diverge\_parties} = \mathit{true}$ or $\mathsf{epoch\_id} \neq \mathsf{epoch\_id\_self}_{1-i}$: //Let $\mathcal{A}$ choose key
   - Send $(\mathsf{state}_{\mathcal{I}}, \texttt{GetReceivingKey}, i, \mathsf{epoch\_id})$ to $\mathcal{A}$
   - Upon receiving $(\mathsf{state}'_{\mathcal{I}}, \texttt{GetReceivingKey}, i, \mathsf{epoch\_id}, \mathsf{recv\_chain\_key}^*)$ from $\mathcal{A}$, update $\mathsf{state}_{\mathcal{I}} \leftarrow \mathsf{state}'_{\mathcal{I}}$.
   - If $\mathsf{diverge\_parties} = \mathit{false}$ and $\mathsf{epoch\_num}_i + 1 \notin \mathsf{compromised\_epochs}$, add $\mathsf{epoch\_id}$ to $\mathsf{receive\_attempts}[\mathsf{epoch\_num}]$.
5. Else ($\mathsf{diverge\_parties} = \mathit{false}$ 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 $\mathsf{epoch\_num}_{1-i} \in \mathsf{receive\_attempts}.\mathit{keys}$ then set $\mathsf{leak} = \mathsf{receive\_attempts}[\mathsf{epoch\_num}_{1-i}]$
- Send $(\texttt{ReportState}, \mathsf{state}_{\mathcal{I}}, i, \mathsf{recv\_chain\_key}_i, \mathsf{leak})$ to $\mathcal{A}$.
- Upon receiving $(\texttt{ReportState}, i, S)$ from $\mathcal{A}$, output $(\texttt{Corrupt}, S)$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

Fig. 7. The Epoch Key Exchange Functionality, $\mathcal{F}_{\mathsf{eKE}}$ (continued)

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

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

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

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

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

Fig. 8. The Forward-Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$

epochs at a certain message number N by generating all $\mathsf{key\_seed}$'s up to N and later disallowing the generation of any further key seeds for its epoch.

Theorem 3. *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* $\mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}})$.

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. We defer a description of Π_{aead} to the full version [21] and show only the result here.

Theorem 4. *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* $F_{mKE}^{\Pi} = (\mathcal{S}_{\mathsf{mKE}}, \mathcal{F}_{\mathsf{mKE}}), \mathcal{F}_{\mathsf{eKE}}^{\Pi_{\mathsf{eKE}}} = (\mathcal{S}_{\mathsf{eKE}}, \mathcal{F}_{\mathsf{eKE}}), \mathcal{F}_{\mathsf{DIR}}, \mathcal{F}_{\mathsf{LTM}}, \mathcal{F}_{\mathsf{lib}}$.

Instantiating $\mathcal{F}_{\mathsf{eKE}}$ via Π_{eKE} (Figs. 12 to 13) is subtle. 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

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

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

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

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

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, $\mathsf{pid} \neq \mathsf{pid}_{1-\mathsf{sender}}$, or no messages have been successfully decrypted by pid_i: end the activation.
2. Mark all $\mathsf{msg\_num} > \mathsf{msg\_num}^*$ as inaccessible, and output $(\mathtt{StopDecrypting}, \mathtt{Success})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid})$.

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

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

Fig. 9. The Forward-Secure Encryption Functionality $\mathcal{F}_{\mathsf{fs\_aead}}$ (continued)

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

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

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

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

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

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

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

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

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

Fig. 10. The Authenticated Encryption with Associated Data Functionality, $\mathcal{F}_{\mathsf{aead}}$

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

This functionality has a session id $\mathsf{sid}.mKE$ that takes the following format: $\mathsf{sid}.mKE = (\text{“}mKE\text{”}, sid.fs)$. Where $\mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id})$. The local session ID is parsed as $\mathsf{sid} = (sid', \mathsf{pid}_0, \mathsf{pid}_1)$. Inputs arriving from machines whose identity is neither pid_0 nor pid_1 are ignored.

This functionality is parametrized by a seed length λ

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

1. If this is the first activation,
   - Initialize dictionary $\mathsf{key\_dict}$ and variables $\mathtt{IsCorrupt?} = false$, $\mathsf{msg\_num}_0, \mathsf{msg\_num}_1 = 0$.
   - 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 $\mathtt{IsCorrupt?} = false$:
   - 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 ($\mathtt{IsCorrupt?} = true$):
   - 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 $\mathsf{msg\_num}_i = \mathsf{msg\_num}$. //$\mathsf{msg\_num}_i$ is the largest successfully retrieved message by party i.
8. Record $(\mathtt{Retrieved}, i, \mathsf{msg\_num})$ and output $(\mathtt{RetrieveKey}, \mathsf{pid}, k)$ to $(\Pi_{\mathsf{aead}}, \mathsf{sid}.aead)$.

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

- 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 $(\Pi_{fs\text{-}aead}, \mathsf{sid}.fs = (\text{“}fs\_aead\text{”}, \mathsf{sid}, \mathsf{epoch\_id}, b), \mathsf{pid})$:

1. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathtt{IsCorrupt?} = true$, 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 $(\Pi_{\mathsf{eKE}}, \mathsf{sid}.eKE, \mathsf{pid})$.
   //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 Π_{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 $\Longrightarrow$ 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 $(\Pi_{\mathsf{fs\_aead}}, \mathsf{sid}.fs, \mathsf{pid}_i)$.

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

Π_{eKE}

This protocol has a party id pid and session id $\mathsf{sid}.eKE$ of the form: $\mathsf{sid}.eKE = (\text{“}eKE\text{”}, \mathsf{sid})$ where $\mathsf{sid} = (\mathsf{sid}', \mathsf{pid}_0, \mathsf{pid}_1)$.

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

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

1. If $\mathsf{pid}' \neq \mathsf{pid}$, then end the activation. Let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathsf{temp\_epoch\_id\_partner}_i = \mathsf{epoch\_id}^*$.
3. If this is the first activation:
   - Initialize state variables $\mathsf{root\_key}, \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Send $(\mathtt{GetInitKeys}, \mathsf{pid}_{1-i}, \mathsf{pid}_i)$ 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 $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$
4. Else (this is not the first activation):
   - **Compute Sending Chain Key.**
   - Output $(\mathtt{ConfirmReceivingEpoch}, \mathsf{epoch\_id}_{\mathsf{self}})$ to $(\Pi_{\mathsf{SGNL}}, \mathsf{sid}, \mathsf{pid}_i)$.

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

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

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

1. If $\mathsf{pid}' \neq \mathsf{pid}$, then end the activation. Otherwise, let i be such that $\mathsf{pid} = \mathsf{pid}_i$.
2. Set $\mathsf{temp\_epoch\_id\_partner} = \mathsf{epoch\_id}$.
3. If this is the first activation:
   - Initialize state variables $\mathsf{root\_key}, \mathsf{epoch\_id}, \mathsf{epoch\_key}, \mathsf{sending\_chain\_key} = \perp$.
   - Parse $\mathsf{epoch\_id} = (\mathsf{epoch\_id}', \mathsf{ek}_j^{\mathsf{pk}}, \mathsf{ok}_{i\leftarrow j}^{\mathsf{pk}})$ 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 $\mathsf{temp\_recv\_chain\_key}$.

(The rest of this protocol is in Fig. 13)

Fig. 12. The Epoch Key Exchange Protocol Π_{eKE}

Π_{eKE} continued...

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

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

//Below are subroutines used in the interfaces above. The calls to `Advance` and `Compute` use Π_{KDF} that is a cascaded $\mathsf{PRF} - \mathsf{PRNG}$

Compute Sending Chain Key:

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

Fig. 13. The Epoch Key Exchange Protocol Π_{eKE} (continued)

order to maintain security against adaptive adversaries. Second, we provide an instantiation in the plain model based on punctured PRFs.

Theorem 5. *Assume that $KDF : \{0,1\}^n \rightarrow \{0,1\}^{2n}$ 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 $\mathcal{F}_{\mathsf{LTM}}$.*

Cascaded PRF-PRG. The goal of this primitive is to formalize the requirements required of a key derivation function (KDF) to adhere to the Signal specification [44] in the adaptive setting. We consider a stateful key derivation function (KDF) with two algorithms. 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 $\mathsf{root\_key}'$ (discarding the old state). 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.

Our *cascaded PRF-PRG* definition requires that the KDF be secure against an adversary who can repeatedly execute methods ($\mathsf{Compute}$) and ($\mathsf{Advance}$) at will, and who can also obtain the module's local state at any time. More specifically, we require existence of a simulator such that no adversary can distinguish an interaction with the scheme from an ideal interaction (Fig. 14) where the keys are truly random and the exposed state is generated by the simulator.

Cascaded PRF-PRG Security Game

Security game for a KDF $(\mathsf{Compute}, \mathsf{Advance})$ 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 $\mathcal{S}$:

Real game:

- Oracle O is initialized with random state $s \leftarrow \{0,1\}^n$.
- On input $(\mathsf{Compute}, \mathsf{root\_input})$: O runs $\mathsf{Compute}(s, \mathsf{root\_input}) = (\mathsf{chain\_key}, \mathsf{root\_key}')$, outputs $\mathsf{chain\_key}$ and changes state $s = \mathsf{root\_key}'$.
- On input $(\mathsf{Advance})$: O chooses $\mathsf{root\_input}' \leftarrow R_n$ at random, runs $\mathsf{Advance}(s, \mathsf{root\_input}') = \mathsf{root\_key}'$, and changes state $s = \mathsf{root\_key}'$.
- On input $(\mathsf{Expose\text{-}Advance})$: O outputs the old state s, chooses $\mathsf{root\_input}' \leftarrow R_n$ at random, computes $\mathsf{Advance}(s, \mathsf{root\_input}') = \mathsf{root\_key}'$, 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 $(\mathsf{Compute}, \mathsf{root\_input})$: O outputs $F(\mathsf{root\_input}) = \mathsf{chain\_key}$.
- On input $(\mathsf{Advance})$: O updates its state to a new random function $F : R_n \to \{0,1\}^m$.
- On input $(\mathsf{Expose\text{-}Advance})$: O outputs $\mathcal{S}((\mathsf{root\_input}_1, F(\mathsf{root\_input}_1)), \ldots, (\mathsf{root\_input}_k, F(\mathsf{root\_input}_k))$, where $\mathsf{root\_input}_1, \ldots, \mathsf{root\_input}_k$ are all the queries made by $\mathcal{A}$ since the last $\mathsf{Advance}$ query and F is the currently used random function. Finally O updates its state to a new random function $F : R_n \to \{0,1\}^m$.

Fig. 14. Cascaded PRF-PRG Security Game

Definition 1 (Cascaded PRF-PRG (CPRFG)). *A KDF $(\mathtt{Compute}, \mathtt{Advance})$ 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 in Fig. 14 only with advantage that is negligible in n.*

In the full version of this work [21], we show two constructions of a Cascaded PRF-PRG: a straightforward one based on a programmable random oracle, and a non-trivial construction in the plain model based on a puncturable PRF and a PRF-PRG (in the style of Alwen et al. [1]).

References

1. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019)
2. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Heidelberg (Aug 2020)
3. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 1463–1483. ACM Press (Nov 2021)

4. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Heidelberg (Nov 2020)
5. Badertscher, C., Canetti, R., Hesse, J., Tackmann, B., Zikas, V.: Universal composition with global subroutines: Capturing global setup within plain UC. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part III. LNCS, vol. 12552, pp. 1–30. Springer, Heidelberg (Nov 2020)
6. Badertscher, C., Hesse, J., Zikas, V.: On the (ir)replaceability of global setups, or how (not) to use a global ledger. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part II. LNCS, vol. 13043, pp. 626–657. Springer, Heidelberg (Nov 2021)
7. Balli, F., Rösler, P., Vaudenay, S.: Determining the core primitive for optimally secure ratcheting. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part III. LNCS, vol. 12493, pp. 621–650. Springer, Heidelberg (Dec 2020)
8. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO'93. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (Aug 1994)
9. Bellare, M., Rogaway, P.: Provably secure session key distribution: The three party case. In: 27th ACM STOC. pp. 57–66. ACM Press (May / Jun 1995)
10. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 619–650. Springer, Heidelberg (Aug 2017)
11. Bienstock, A., Dodis, Y., Rösler, P.: On the price of concurrency in group ratcheting protocols. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 198–228. Springer, Heidelberg (Nov 2020)
12. Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the signal double ratchet algorithm. Cryptology ePrint Archive, Report 2022/355 (2022), https://eprint.iacr.org/2022/355
13. 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)
14. Borisov, N., Goldberg, I., Brewer, E.A.: Off-the-record communication, or, why not to use PGP. In: WPES. pp. 77–84. ACM (2004)
15. 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)
16. Camenisch, J., Drijvers, M., Lehmann, A.: Universally composable direct anonymous attestation. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) PKC 2016, Part II. LNCS, vol. 9615, pp. 234–264. Springer, Heidelberg (Mar 2016)
17. 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)
18. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001)
19. Canetti, R.: Universally composable security. J. ACM 67(5), 28:1–28:94 (2020)
20. Canetti, R., Halevi, S., Herzberg, A.: Maintaining authenticated communication in the presence of break-ins. J. Cryptol. 13(1), 61–105 (2000)
21. Canetti, R., Jain, P., Swanberg, M., Varia, M.: Universally composable end-to-end secure messaging. Cryptology ePrint Archive, Report 2022/376 (2022), https://eprint.iacr.org/2022/376

22. 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)
23. 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)
24. 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)
25. 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)
26. 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)
27. 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)
28. 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)
29. 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)
30. 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)
31. 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)
32. 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)
33. 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)
34. Hofheinz, D., Rao, V., Wichs, D.: Standard security does not imply indistinguishability under selective opening. In: Hirt, M., Smith, A.D. (eds.) TCC 2016-B, Part II. LNCS, vol. 9986, pp. 121–145. Springer, Heidelberg (Oct / Nov 2016)
35. 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)
36. 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)

37. 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)
38. Keybase blog: New cryptographic tools on keybase. https://keybase.io/blog/crypto (2020)
39. Krohn, M.: Zoom rolling out end-to-end encryption offering. https://blog.zoom.us/zoom-rolling-out-end-to-end-encryption-offering/ (2020)
40. Martiny, I., Kaptchuk, G., Aviv, A.J., Roche, D.S., Wustrow, E.: Improving signal's sealed sender. In: NDSS. The Internet Society (2021)
41. Maurer, U.: Constructive cryptography - a primer (invited paper). In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, p. 1. Springer, Heidelberg (Jan 2010)
42. 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)
43. 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)
44. Open Whisper Systems: Technical information: Specifications and software libraries for developers. https://signal.org/docs/ (2016)
45. Perrin, T.: The noise protocol framework. https://noiseprotocol.org/noise.html (2018)
46. 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)
47. 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)
48. 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)
49. 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)
50. Status: Private, secure communication. https://status.im (2022)
51. Sylo: Comms for the metaverse. https://sylo.io (2022)
52. 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)
53. 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)
54. Unger, N., Goldberg, I.: Improved strongly deniable authenticated key exchanges for secure messaging. PoPETs 2018(1), 21–66 (Jan 2018)
55. 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)
56. WhatsApp LLC: About end-to-end encryption. https://faq.whatsapp.com/general/security-and-privacy/end-to-end-encryption/ (2021)
57. 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)