The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol

Joël Alwen[2]*, Sandro Coretti[1]**, and Yevgeniy Dodis[1]***

[1] New York University
{corettis,dodis}@nyu.edu
[2] Wickr Inc.
jalwen@wickr.com

Abstract. Signal is a famous secure messaging protocol used by billions of people, by virtue of many secure text messaging applications including Signal itself, WhatsApp, Facebook Messenger, Skype, and Google Allo. At its core it uses the concept of "double ratcheting," where every message is encrypted and authenticated using a fresh symmetric key; it has many attractive properties, such as forward security, post-compromise security, and "immediate (no-delay) decryption," which had never been achieved in combination by prior messaging protocols.

While the formal analysis of the Signal protocol, and ratcheting in general, has attracted a lot of recent attention, we argue that none of the existing analyses is fully satisfactory. To address this problem, we give a clean and general definition of *secure messaging*, which clearly indicates the types of security we expect, including forward security, post-compromise security, and immediate decryption. We are the first to explicitly formalize and model the immediate decryption property, which implies (among other things) that parties seamlessly recover if a given message is permanently lost—a property not achieved by any of the recent "provable alternatives to Signal."

We build a modular "generalized Signal protocol" from the following components: (a) *continuous key agreement (CKA)*, a clean primitive we introduce and which can be easily and generically built from public-key encryption (not just Diffie-Hellman as is done in the current Signal protocol) and roughly models "public-key ratchets;" (b) *forward-secure authenticated encryption with associated data (FS-AEAD)*, which roughly captures "symmetric-key ratchets;" and (c) a two-input hash function that is a pseudorandom function (resp. generator with input) in its first (resp. second) input, which we term PRF-PRNG. As a result, in addition to instantiating our framework in a way resulting in the existing, widely-used Diffie-Hellman based Signal protocol, we can easily get post-quantum security and not rely on random oracles in the analysis.

* Partially supported by the European Research Council under ERC Consolidator Grant (682815 - TOCNeT)

** Supported by NSF grants 1314568 and 1319051.

*** Partially supported by gifts from VMware Labs, Facebook and Google, and NSF grants 1314568, 1619158, 1815546.

1 Introduction

Signal [?] is a famous secure messaging protocol, which is—by virtue of many secure text messaging applications including Signal itself, WhatsApp [?], Facebook Messenger [?], Skype [?] and Google Allo [?]—used by billions of people. At its core it uses the concept of *double ratcheting*, where every message is encrypted and authenticated using a fresh symmetric key. Signal has many attractive properties, such as forward security and post-compromise security, and it supports immediate (no-delay) decryption. Prior to Signal's deployment, these properties had never been achieved in combination by messaging protocols.

Signal was designed by practitioners and was implemented and deployed well before any security analysis was obtained. In fact, a clean description of Signal has been posted by its inventors Marlinspike and Perrin [?] only recently. The write-up does an excellent job at describing the double-ratchet protocol, gives examples of how it is run, and provides security intuition for its building blocks. However, it lacks a formal definition of the secure messaging problem that the double ratchet solves, and, as a result, does not have a formal security proof.

Immediate decryption and its importance. One of the main issues any messaging scheme must address is the fact that messages might arrive out of order or be lost entirely. Additionally, parties can be offline for extended periods of time and send and receive messages asynchronously. Given these inherent constraints, immediate decryption is a very attractive feature. Informally, it ensures that when a legitimate message is (eventually) delivered to the recipient, the recipient can not only immediately decrypt the message but is also able to place it in the correct spot in relation to the other messages already received. Furthermore, immediate decryption also ensures an even more critical liveness property, termed *message-loss resilience (MLR)* in this work: if a message is permanently lost by the network, parties should still be able to communicate (perhaps realizing at some point that a message has never been delivered). Finally, even in settings where messages are eventually delivered (but could come out of order), giving up on immediate decryption seems cumbersome: should out-of-order messages be discarded or buffered by the recipient? If discarded, how will the sender (or the network) know that it should resend the message later? If buffered, how to prevent denial-of-service attacks and distinguish legitimate out-of-order messages (which cannot be immediately decrypted) from fake messages? While these questions could surely be answered (perhaps by making additional timing assumptions about the network), it appears that the simplest answer would be to design a secure messaging protocol which support immediate decryption. Indeed, to the best of our knowledge, all secure messaging services deployed in practice do have this feature (and, hence, MLR).

Additional properties. In practice, parties' states might occasionally leak. To address this concern, a secure messaging protocol should have the following two properties:

- *Forward secrecy (FS):* if the state of a party is leaked, none of the previous messages should get compromised (assuming they are erased from the state, of course).
- *Post-compromise security (PCS) (aka channel healing):* once the exposure of the party's state ends, security is restored after a few communication rounds.

In isolation, fulfilling either of these desirable properties is well understood: FS is achieved by using basic steam ciphers (aka pseudorandom generators (PRGs)) [**?**], while PCS [**?**] is achieved by some form of key agreement executed after the compromise, such as Diffie-Hellman. Unfortunately, these techniques, both of which involve some form of *key evolution*, are clearly at tension with immediate decryption when the network is fully asynchronous. Indeed, the main elegance of Signal, achieved by its *double-ratchet* algorithm, comes from the fact that FS and PCS are not only achieved together, but also *without sacrificing immediate decryption and MLR.*

Goals of this work. One of the main drawbacks of all formal Signal-related papers [**?**,**?**,**?**,**?**], following the initial work of [**?**], is the fact that they all achieve FS and PCS by *explicitly giving up* not only on immediate decryption, but also MLR. (This is not merely a definitional issue as their constructions indeed cease any and all further functionality when, say, a single message is dropped in transit.) While such a drastic weakening of the liveness/correctness property considerably simplifies the algorithmic design for these provably secure alternatives to Signal, it also made them insufficient for settings where message loss is indeed possible. This can occur, in practice, due to a variety of reasons. For example, the protocol may be using an unreliable transport mechanism such as SMS or UDP. Alternatively, traffic may be routed (via more reliable TCP) through a central back-end server so as to facilitate asynchronous communication between end-points (as is very common for secure messaging deployments in practice). Yet, even in this setting, packet losses can still occur as the server itself may end up dropping messages due to a variety of unintended events such as due to outages or being subject to a heavy work/network load (say, because of an ongoing (D)DOS attack, partial outages, or worse yet, an emergency event generating sudden high volumes of traffic). With the goal of providing resilient communication even under these and similar realistic conditions, the main objectives of this work are to:

(a) propose formal definitions of *secure messaging* as a cryptographic primitive that *explicitly mandates immediate decryption and MLR*; and

(b) to provide an analysis of *Signal itself* in a well-defined *general* model for secure messaging.

Our work is the first to address either of these natural goals. Moreover, in order to improve the general understanding of secure messaging and to develop alternative (e.g., post-quantum secure) solutions, this paper aims at

(c) generalizing and abstracting out the reliance on the specific Diffie-Hellman key exchange (making the current protocol insecure in the post-quantum world) as well as clarifying the role of various cryptographic hash functions used inside the current Signal instantiation. That is, the idea is to build a "generalized Signal" protocol of which the current instantiation is a special case, but where other instantiations are possible, including those which are post-quantum secure and/or do not require random oracles.

1.1 Our Results

Addressing the points (a)-(c) above, this paper's main contributions are the following:

- Providing a clean definition of *secure messaging* that clearly indicates the expected types of security, including FS, PCS, and—for the first time—immediate decryption.
- Putting forth a modular *generalized* Signal protocol from several simpler building blocks:
  1. *forward-secure authenticated encryption with associated data (FS-AEAD)*, which can be easily built from a regular PRG and AEAD and roughly models the so-called symmetric-key ratchet of Signal;
  2. *continuous key agreement (CKA)*, which is a clean primitive that can easily be built generically from public-key encryption and roughly models the so-called public-key ratchet of Signal;
  3. a two-input hash function, called *PRF-PRNG*, which is a pseudorandom function (resp. generator) in its first (resp. second) input and helps to "connect" the two ratchet types.
- Instantiating the framework such that we obtain the existing Diffie-Hellman-based protocol and observing that one can easily achieve post-quantum security (by using post-quantum-secure public-key encryption, such as [**?**,**?**,**?**]) and/or not rely on random oracles in the analysis.
- Extending the design to include other forms of "fine-grained state compromise" recently studied by Poettering and Rösler [**?**] and Jaeger and Stepanovs [**?**] but, once more, without sacrificing the immediate decryption property.

The secure messaging definition. The proposed secure messaging (SM) definition encompasses, in one clean game, (Figure 1) all desired properties, including FS as well as PCS and immediate decryption. The attacker in the definition is very powerful, has full control of the order of sending and receiving messages, can corrupt parties' state multiple times, and even controls the randomness used for encryption.[3] In order to avoid trivial and unpreventable attacks, a few restrictions need to be placed on an attacker $\mathcal{A}$. In broad strokes, the definition requires the following properties:

[3] Namely, good randomness is only needed to achieve PCS, while all other security properties hold even with the adversarially controlled randomness (when parties are not compromised).

– When parties are uncompromised, i.e., when their respective states are unknown to $\mathcal{A}$, the protocol is secure against *active* attacks. In particular, the protocol must detect injected ciphertexts (not legitimately sent by one of the parties) and properly handle legitimate ciphertexts delivered in arbitrary order (capturing correctness and immediate decryption).

– When parties are uncompromised, messages are protected even against future compromise of either the sender or the receiver, modeling *forward security.*

– When one or both parties are compromised and the attacker remains *passive*, security is restored "quickly," i.e., within a few rounds of back-and-forth, which models *PCS*.

While the proposed definition is still rather complex, we believe it to be intuitive and *considerably shorter and easier to understand* compared to the recent works of [**?**,**?**], which are discussed in more detail in Section 1.2.

It should be stressed that the basic SM security in this paper only requires PCS against a passive attacker. Indeed, when an active attacker compromises the state of, say, party A, it can always send ciphertexts to the partner B in A's name (thereby even potentially hijacking A's communication with B and removing A from the channel altogether) or decrypt ciphertexts sent by B immediately following state compromise. As was observed by [**?**,**?**] at CRYPTO'18, one might achieve certain limited forms of fine-grained security against active attacks. For example, it is not a priori clear if the attacker should be able to decrypt ciphertexts sent by A to B (if A uses good randomness) or forge legitimate messages from B to A (when A's state is exposed). We comment on these possible extensions in the full version of this paper [**?**] but notice that they are still rather limited, given that the simple devastating attacks mentioned above are inherently non-preventable against active attackers immediately following state compromise. Thus, our main SM security notion simply *disallows all active attacks for* Δ_{SM} *epochs immediately following state compromise* where Δ_{SM} is the number of rounds of communication required to refresh a compromised state.

The building blocks. Since the original Signal protocol is quite subtle and somewhat tricky to understand, one of the main contributions of this work is to distill out three basic and intuitive building blocks used inside the double ratchet.

The first block is *forward-secure authenticated encryption with associated data (FS-AEAD)* and models secure messaging security inside a single so-called *epoch*; an epoch should be thought of as a unidirectional stream of messages sent by one of the parties, ending once a message from the other party is received. As indicated by the name, an FS-AEAD protocol must provide forward secrecy, but also immediate decryption. Capturing this makes the definition of FS-AEAD somewhat non-trivial (cf. Figure 3), but still simpler than that of general SM; in particular, no PCS is required (which allows us to define FS-AEAD as a deterministic primitive and not worry about poor randomness).

Building FS-AEAD turns out to be rather easy: in essence, one uses message counters as associated data for standard AEAD and a PRG to immediately refresh the secret key of AEAD after every message successfully sent or received. This is exactly what is done in Signal.

The second block is a primitive called *continuous key agreement (CKA)* (cf. Figure 2), which could be viewed as an abstraction of the DH-based public-key ratchet in Signal. CKA is a *synchronous* and *passive* primitive, i.e., parties A and B speak in turns, and no adversarial messages or traffic mauling are allowed. With each message sent or received, a party should output a fresh key such that (with "sending" keys generated by A being equal to "receiving" keys generated by B and vice versa). Moreover, CKA guarantees its own PCS, i.e., after a potential state exposure, security is restored within two rounds. Finally, CKA must be forward-secure, i.e., past keys must remain secure when the state is leaked. Forward security is governed by a parameter $\Delta_{\mathsf{CKA}} \geq 0$, which, informally, guarantees that all keys older than Δ_{CKA} rounds remain secure upon state compromise.

Not surprisingly, minimizing Δ_{CKA} results in faster PCS for secure messaging.[4] Fortunately, optimal CKA protocols achieving optimal $\Delta_{\mathsf{CKA}} = 0$ can be built generically from key-encapsulation mechanisms. Interestingly, the elegant DH-based CKA used by Signal achieves slightly sub-optimal $\Delta_{\mathsf{CKA}} = 1$, which is due to how long parties need to hold on to their secret exponents. However, the Signal CKA saves about a factor of 2 in communication complexity, which makes it a reasonable trade-off in practice.

The third and final component of the generalized Signal protocol is a two-argument hash function P, called a PRF-PRNG, which is used to produce secret keys for FS-AEAD epochs from an entropy pool refreshed by CKA keys. More specifically, with each message exchanged using FS-AEAD, the parties try to run the CKA protocol "on the side," by putting the CKA messages as associated data. Due to asynchrony, the party will repeat a given CKA message until it receives a legitimate response from its partner, after which the CKA moves forward with the next message. Each new CKA key is absorbed into the state of the PRF-PRNG, which is then used to generate a new FS-AEAD key.

Informally, a PRF-PRNG takes as inputs a state σ and a CKA key I and produces a new state σ' and a (FS-AEAD) key k. It satisfies a *PRF* property saying that if σ is random, then $\mathsf{P}(\sigma, \cdot)$ acts like a PRF (keyed by σ) in that outputs (σ', k) on *adversarially chosen* inputs I are random. Moreover, it also acts like a PRNG in that, if the input I is random, then so are the resulting state σ' and key k. Observe that standard hash functions are assumed to satisfy this notion; alternatively, one can also very easily build a PRF-PRNG from any PRG and a pseudorandom permutation (cf. Section 4.3).

Generalized signal. Putting the above blocks together properly yields the generalized Signal protocol (cf. Figure 6). As a special case, one can easily obtain the existing Signal implementation[5] by using the standard way of building FS-AEAD from PRG and AEAD, CKA using the Diffie-Hellman based public-key

[4] Specifically, the healing time of the generic Signal protocol presented in this work is $\Delta_{\mathsf{SM}} = 2 + \Delta_{\mathsf{CKA}}$.

[5] For syntactic reasons having to do with our abstractions, our protocol is a minor variant of Signal, but is logically equivalent to Signal in every aspect.

ratchet mentioned above, and an appropriate cryptographic hash function in place of PRF-PRNG. However, many other variants become possible. For example, by using a generic CKA from DH-KEM, one may trade communication efficiency (worse by a factor of 2) for a shorter healing period Δ_{SM} (from 3 rounds to 2). More interestingly, using any post-quantum KEM, such as [**?**,**?**,**?**] results in a *post-quantum secure variant of Signal*. Finally, we also believe that our generalized double ratcheting scheme is much more intuitive than the existing DH-based variant, as it abstracts precisely the cryptographic primitives needed, including the two types ratchets, and what security is needed from each primitive.

Beyond double ratcheting to full signal. Following most of the prior (and concurrent) work [**?**,**?**,**?**,**?**] (discussed in the next section), this paper primarily concerned with formalizing the double-ratchet aspect of the Signal protocol. This assumes that any set of two parties can correctly and securely agree on the initial secret key. The latter problem is rather non-trivial, especially (a) in the multi-user setting, when a party could be using a global public key to communicate with multiple recipients, some of which might be malicious, (b) when the initial secret key agreement is required to be non-interactive, and (c) when state compromise (including that of the master secret for the PKI) is possible, and even frequent. Some of those subtleties are discussed and analyzed by Cohn-Gordon *et al.* [**?**], but, once again, in a manner specific to the existing Signal protocol (rather than a general secure messaging primitive). Signal also suggests using the X3DH protocol [**?**] as one particular way to generate the initial shared key. Certainly, studying (and even appropriately defining) secure messaging *without idealized setup*, and analyzing "full Signal" in this setting, remains an important area for future research.

1.2 Related Work

The OTR (off-the-record) messaging protocol [**?**] is an influential predecessor of Signal, which was the first to introduce the idea of the DH-based double ratchet to derive fresh keys for each encrypted message. However, it was mainly suitable for synchronous back-and-fourth conversations, so Signal's double ratchet algorithm had to make a number of non-trivial modifications to extend the beautiful OTR idea into a full-fledged asynchronous messaging protocol.

Following the already discussed rigorous description of DH-based double ratcheting by Marlinspike and Perrin [**?**], and the protocol-specific analysis by Cohn-Gordon et al. [**?**], several formal analyses of ratcheting have recently appeared [**?**,**?**,**?**,**?**]; they design definitions of various types of ratcheting and provide schemes meeting these definitions. As previously mentioned, all these works have the drawback of no longer satisfying immediate decryption.

Bellare et al. [**?**] looked at the question of *unidirectional ratcheting*. In this simplified variant of double (or bidirectional) ratcheting, the receiver is never corrupted, and never needs to update its state. Coupled with giving up immediate decryption, this allowed the authors to obtain a rather simple solution

Unfortunately, extending their ideas to the case of bidirectional communication appeared non-trivial and was left to future work.

Bidirectionality has been achieved in work by Jaeger and Stepanovs [?] and Poettering and Rösler [?]. The papers differ in syntax (one treats secure messaging while the other considers key exchange) and hence use different definitions. However, in spirit both papers attempt to model a bidirectional channel satisfying FS and PCS (but not immediate decryption). Moreover, both consider "fine-grained" PCS requirements which are not met by Signal's double ratchet protocol (and not required by the SM definition in this work). The extra security appears to come at a steep price: both papers use growing (and potentially unbounded) state as well as heavy techniques from public-key cryptography, including hierarchical identity-based encryption [?] (HIBE). More discussion can be found in the full version of this paper [?], including an (informally stated) extension to Signal which achieves a slightly weaker form of fine-grained compromise than [?,?], yet still using only constant sized states, bandwidth and computation as well as comparatively lightweight primitives.

Finally, the notion of immediate decryption is reminiscent in spirit to the zero round trip time (0-RTT) communication with forward secrecy which was recently studied by [?,?]. However, the latter primitive is stateless on the sender side, making it more difficult to achieve (e.g., the schemes of [?,?] use a heavy tool called *puncturable encryption* [?]).

Concurrent and Independent Work. We have recently become aware of two concurrent and independent works by Durak and Vaudenay [?] and Jost, Maurer and Mularczyk [?]. Like other prior works, these works (1) designed their own protocols and did not analyze Signal; and (2) do not satisfy immediate decryption or even message-loss resilience (in fact, they critically rely on receiving messages from one party in order). Both works also provide formal notions of security, including privacy, authenticity, and a new property called unrecoverability by [?] and post-impersonation authentication by [?]: if an active attacker sends a fake message to the recipient immediately following state compromise of the sender, the sender can, by design, never recover (and, thus, will notice the attack by being unable to continue the conversation).

2 Preliminaries

2.1 Game-Based Security and Notation

All security definitions in this work are game-based, i.e., they consider games executed between a challenger and an adversary. The games have one of the following formats:

– *Unpredictability games:* First, the challenger executes the special init procedure, which sets up the game. Subsequently, the attacker is given access to a set of oracles that allow it to interact with the scheme in question. The goal of the adversary is to provoke a particular, game-specific *winning* condition. The *advantage* of an adversary $\mathcal{A}$ against construction C in an unpredictability game Γ^C is

$$\mathrm{Adv}_{\Gamma}^{C}(\mathcal{A}) \;:=\; \mathsf{P}[\mathcal{A} \text{ wins } \Gamma^C] \;.$$

– *Indistinguishability games:* In addition to setting up the game, the init procedure samples a secret bit $b \in \{0,1\}$. The goal of the adversary is to determine the value of b. Once more, upon completion of init, the attacker interacts arbitrarily with all available oracles up to the point where it outputs a guess bit b'. The adversary *wins* the game if $b = b'$. The *advantage* of an adversary $\mathcal{A}$ against construction C in an indistinguishability game Γ is

$$\mathrm{Adv}_{\Gamma}^{C}(\mathcal{A}) \;:=\; 2 \cdot \left|\mathsf{P}[\mathcal{A} \text{ wins } \Gamma^C] - 1/2\right| \;.$$

With the above in mind, to describe a any security (or correctness) notion, one need only specify the init oracle and the oracles available to $\mathcal{A}$. The following special keywords are used to simplify the exposition of the security games:

– **req** is followed by a condition; if the condition is not satisfied, the oracle/procedure containing the keyword is exited and all actions by it are undone.

– **win** is used to declare that the attacker has won the game; it can be used for both types of games above.

– **end** disables all oracles and returns all values following it to the attacker.

Moreover, the descriptions of some games/schemes involve *dictionaries*. For ease of notation, these dictionaries are described with the *array-notation* described next, but it is important to note that they are to be implemented by a data structure whose size grows (linearly) with the number of elements *in* the dictionary (unlike arrays):

– *Initialization:* The statement $D[\cdot] \leftarrow \lambda$ initializes an *empty* dictionary D.

– *Adding elements:* The statement $D[i] \leftarrow v$ adds a value v to dictionary D with key i, overriding the value previously stored with key i if necessary.

– *Retrieval:* The expression $D[i]$ returns the value v with key i in the dictionary; if there are no values with key i, the value λ is returned.

– *Deletion:* The statement $D[i] \leftarrow \lambda$ *deletes* the value v corresponding to key i.

Finally, sometimes the random coins of certain probabilistic algorithms are made explicit. For example, $y \leftarrow A(x; r)$ means that A, on input x and with random tape r, produces output y. If r is not explicitly stated, is assumed to be chosen uniformly at random; in this case, the notation $y \leftarrow\!\!\$\; A(x)$ is used.

2.2 Cryptographic Primitives

This paper makes use of the following cryptographic primitives:

AEAD. An *authenticated encryption with associated data (AEAD) scheme* is a pair of algorithms $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ with the following syntax:

- *Encryption:* Enc takes a key K, associated data a, and a message m and produces a ciphertext $e \leftarrow \mathsf{Enc}(K, a, m)$.
- *Decryption:* Dec takes a key K, associated data a, and a ciphertext e and produces a message $m \leftarrow \mathsf{Dec}(K, a, e)$.

All AEAD schemes in this paper are assumed to be deterministic, i.e., all randomness stems from the key K.

KEMs. A *key-encapsulation mechanism* (KEM) is a public-key primitive consisting of three algorithms $\mathsf{KEM} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ with the following syntax:

- *Key generation:* KG takes a (implicit) security parameter and outputs a fresh key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow\$ \mathsf{KG}$.
- *Encapsulation:* Enc takes a public key pk and produces a ciphertext and a symmetric key $(c, k) \leftarrow\$ \mathsf{Enc}(\mathsf{pk})$.
- *Decapsulation:* Dec takes a secret key sk and a ciphertext c and recovers the symmetric key $k \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$.

3 Secure Messaging

A *secure messaging (SM)* scheme allows two parties A and B to communicate securely bidirectionally and is expected to satisfy the following informal requirements:

- *Correctness:* If no attacker interferes with the transmission, B outputs the messages sent by A in the correct order and vice versa.
- *Immediate decryption and message-loss resilience (MLR):* Messages must be decrypted as soon as they arrive and may not be buffered; if a message is lost, the parties do not stall.
- *Authenticity:* While the parties' states are uncompromised (i.e., unknown to the attacker), the attacker cannot change the messages sent by them or inject new ones.
- *Privacy:* While the parties' states are uncompromised, an attacker obtains no information about the messages sent.
- *Forward secrecy (FS):* All messages sent and received prior to a state compromise of either party (or both) remain hidden to an attacker.
- *Post-compromise security (PCS, aka "healing"):* If the attacker remains passive (i.e., does not inject any corrupt messages), the parties recover from a state compromise (assuming each has access to fresh randomness).
- *Randomness leakage/failures:* While the parties' states are uncompromised, all the security properties above except PCS hold even if the attacker completely

controls the parties' local randomness. That is, good randomness is only required for PCS.

This section presents the syntax of and a formal security notion for SM schemes.

3.1 Syntax

Formally, an SM scheme consists of two initialization algorithms, which are given an initial shared key k, as well as a sending algorithm and a receiving algorithm, both of which keep (shared) state across invocations. The receiving algorithm also outputs a so-called epoch number and an index, which can be used to determine the order in which the sending party transmitted their messages.

Definition 1. *A* secure-messaging (SM) scheme *consists of four probabilistic algorithms* $\mathsf{SM} = (\mathsf{Init\text{-}A}, \mathsf{Init\text{-}B}, \mathsf{Send}, \mathsf{Rcv})$*, where*

- $\mathsf{Init\text{-}A}$ *(and similarly* $\mathsf{Init\text{-}B}$*) takes a key* k *and outputs a state* $s_{\mathsf{A}} \leftarrow \mathsf{Init\text{-}A}(k)$*,*
- Send *takes a state* s *and a message* m *and produces a new state and a ciphertext* $(s', c) \leftarrow\!\!\$\ \mathsf{Send}(s, m)$*, and*
- Rcv *takes a state* s *and a ciphertext* c *and produces a new state, an epoch number, an index, and a message* $(s', t, i, m) \leftarrow \mathsf{Rcv}(s, c)$*.*

3.2 Security

Basics. The security notion for SM schemes considered in this paper is intuitive in principle. However, formalizing it is non-trivial and somewhat cumbersome due to a number of subtleties that naturally arise and cannot be avoided if the criteria put forth at the beginning of Section 3 are to be met. Therefore, before presenting the definition itself, this section introduces some basic concepts that will facilitate understanding of the definition.

EPOCHS. SM schemes proceed in so-called *epochs*, which roughly correspond the "back-and-forth" between the two parties A and B. By convention, odd epoch numbers t are associated with A sending and B receiving, and the other way around for even epochs. Note, however, that SM schemes are completely asynchronous, and, hence, epochs overlap to a certain extent. Correspondingly, consider two epoch counters t_{A} and t_{B} for A and B, respectively, satisfying the following properties:

- The two counters are never more than one epoch apart, i.e., $|t_{\mathsf{A}} - t_{\mathsf{B}}| \leq 1$ at all times.
- When A receives an epoch-t message from B for $t = t_{\mathsf{A}} + 1$, it sets $t_{\mathsf{A}} \leftarrow t$ (even). The next time A sends a message, t_{A} is incremented again (to an odd value).
- Similarly, when B receives an epoch-t message from A for $t = t_{\mathsf{B}} + 1$, it sets $t_{\mathsf{B}} \leftarrow t$ (odd). The next time B sends a message, t_{B} is incremented again (to an even value).

Message indices. Within an epoch, messages are identified by a simple counter. To capture the property of immediate decryption and MLR, the receive algorithm of an SM scheme is required to output the correct epoch number and index *immediately* upon reception of a ciphertext, even when messages arrive out of order.

Corruptions and their consequences. Since SM schemes are required to be forward-secure and to recover from state compromise, any SM security game must allow the attacker to learn the state of either party at any given time. Moreover, to capture authenticity and privacy, the attacker should be given the power to inject malicious ciphertexts and to call a (say) left-or-right challenge oracle, respectively. These requirements, however, interfere as follows:

– When either party is in a compromised state, the attacker cannot invoke the challenge oracle since this would allow him to trivially distinguish.
– When either party is in a compromised state, the attacker can trivially forge ciphertexts and must therefore be barred from calling the inject oracle.
– When the receiver of messages in transmission is compromised, these messages lose all security, i.e., the attacker learns their content and can replace them by a valid forgery. Consequently, while any challenge ciphertext is in transmission, the recipient may not be corrupted. Similarly, an SM scheme must be able to deal with forgeries of compromised messages (once the parties have healed).

These issues require that the security definition keep track of ciphertexts *in transmission*, of *challenge* ciphertexts, and of *compromised* ciphertexts; this will involve some (slightly cumbersome) record keeping.

Natural SM schemes. For simplicity, SM schemes in this work are assumed to satisfy the natural requirements below.[6]

Definition 2. *An SM scheme* $\mathsf{SM} = (\mathsf{Init\text{-}A}, \mathsf{Init\text{-}B}, \mathsf{Send}, \mathsf{Rcv})$ *is* natural *if the following criteria are satisfied:*

(A) Whenever Rcv *outputs* $m = \perp$*, the state remains unchanged.*

(B) Any given ciphertext corresponds to an epoch t *and an index* i*, i.e., the values* (t, i) *output by* Rcv *are an (efficiently computable) function of* c*.*

(C) Algorithm Rcv *never accepts two messages corresponding to the same pair* (t, i)*.*

(D) A party always rejects ciphertexts corresponding to an epoch in which the party does not act as receiver

(E) If a party, say A*, accepts a ciphertext corresponding to an epoch* t*, then* $t_{\mathsf{A}} \geq t - 1$*.*

The security game. The security game, which is depicted in Figure 1, consists of an initialization procedure **init** and of

[6] The reader may skip over this definition on first read. The properties are referenced where they are needed.

– two "send" oracles, **transmit-A** (normal transmission) and **chall-A** (challenge transmission);
– two "receive" oracles, **deliver-A** (honest delivery) and **inject-A** (for forged ciphertexts); and
– a corrupt oracle **corr-A**

pertaining to party A, and of the corresponding oracles pertaining to B. Moreover, Figure 1 also features an epoch-management function **ep-mgmt**, a function **sam-if-nec** explained below, and two record-keeping functions **record** and **delete**; these functions cannot be called by the attacker. The game is parametrized by Δ_{SM}, which relates to how fast parties recover from a state compromise. All components are explained in detail below, following the intuition laid out above.

The advantage of $\mathcal{A}$ against an SM scheme SM is denoted by $\mathrm{Adv}^{\mathsf{SM}}_{\mathrm{sm},\Delta_{\mathsf{SM}}}(\mathcal{A})$. The attacker is parameterized by its running time t, the total number of queries q it makes, and the maximum number of epochs q_{ep} it runs for.

Definition 3. *A secure-messaging scheme* SM *is* $(t, q, q_{\mathsf{ep}}, \Delta_{\mathsf{SM}}, \varepsilon)$-secure *if for all* (t, q, q_{ep})*-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\mathsf{SM}}_{\mathrm{sm},\Delta_{\mathsf{SM}}}(\mathcal{A}) \;\leq\; \varepsilon \; .$$

Initialization and state. The initialization procedure chooses a random key and initializes the states s_{A} and s_{B} of A and B, respectively. Moreover, it defines several variables to keep track of the execution: (1) t_{A} and t_{B} are the epoch counters for A and B, respectively; (2) variables i_{A} and i_{B} count how many messages have been sent by each party in their respective current epochs; (3) t_{L} records the last time either party's state was leaked to the attacker and is used, together with t_{A} and t_{B}, to preclude trivial attacks; (4) the sets trans, chall, and comp will contain records and allow to track ciphertexts in transmission, challenge ciphertexts, and compromised ciphertexts, respectively; (5) the bit b is used to create the challenge.

Sampling if necessary. The send oracles **transmit-A** and **chall-A** allow the attacker to possibly control the random coins r of Send. If $r = \perp$, the function samples $r \leftarrow\!\!\$\; \mathcal{R}$ (from some appropriate set $\mathcal{R}$), and returns (r, good), where good indicates that fresh randomness is used. If, on the other hand, $r \neq \perp$, the function returns (r, bad), indicating, via bad, that adversarially controlled randomness is used.

Epoch management. The epoch management function **ep-mgmt** advances the epoch of the calling party if that party's epoch counter has a "receiving value" (even for A; odd for B) and resets the index counter. The flag argument is to indicate whether fresh or adversarial randomness is used. If a currently corrupted party starts a new epoch with bad randomness, the new epoch is considered corrupted. However, if it does not start a new epoch, bad randomness does not make the ciphertext corrupted. This captures that randomness should only be used for PCS (but for none of the other properties mentioned above).

Security Game for Secure Messaging

```
init
    $k \leftarrow\$ \mathcal{K}$
    $s_{\mathsf{A}} \leftarrow \mathsf{Init\text{-}A}(k)$
    $s_{\mathsf{B}} \leftarrow \mathsf{Init\text{-}B}(k)$
    $(t_{\mathsf{A}}, t_{\mathsf{B}}) \leftarrow (0, 0)$
    $i_{\mathsf{A}}, i_{\mathsf{B}} \leftarrow 0$
    $t_{\mathsf{L}} \leftarrow -\infty$
    $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \leftarrow \emptyset$
    $b \leftarrow\$ \{0, 1\}$

corr-A
    req $\mathsf{B} \notin \mathsf{chall}$
    $\mathsf{comp} \overset{+}{\leftarrow} \mathsf{trans}(\mathsf{B})$
    $t_{\mathsf{L}} \leftarrow \max(t_{\mathsf{A}}, t_{\mathsf{B}})$
    return $s_{\mathsf{A}}$
```

```
transmit-A $(m, r)$
    $(r, \mathsf{flag}) \leftarrow$
        sam-if-nec$(r)$
    ep-mgmt$(\mathsf{A}, \mathsf{flag})$
    $i_{\mathsf{A}}$ ++
    $(s_{\mathsf{A}}, c) \leftarrow \mathsf{Send}(s_{\mathsf{A}}, m; r)$
    record$(\mathsf{A}, \mathsf{norm}, m, c)$
    return $c$

chall-A $(m_0, m_1, r)$
    $(r, \mathsf{flag}) \leftarrow$
        sam-if-nec$(r)$
    ep-mgmt$(\mathsf{A}, \mathsf{flag})$
    req $\mathsf{safe\text{-}ch}_{\mathsf{A}}$ and
        $|m_0| = |m_1|$
    $i_{\mathsf{A}}$ ++
    $(s_{\mathsf{A}}, c) \leftarrow \mathsf{Send}(m_b; r)$
    record$(\mathsf{A}, \mathsf{chall}, m_b, c)$
    return $c$
```

```
deliver-A $(c)$
    req $(\mathsf{B}, t, i, m, c) \in \mathsf{trans}$
        for some $t, i, m$
    $(s_{\mathsf{A}}, t', i', m') \leftarrow \mathsf{Rcv}(s_{\mathsf{A}}, c)$
    if $(t', i', m') \neq (t, i, m)$
        win
    if $(t, i, m) \in \mathsf{chall}$
        $m' \leftarrow \perp$
    $t_{\mathsf{A}} \leftarrow \max(t_{\mathsf{A}}, t)$
    delete$(t, i)$
    return $(t', i', m')$

inject-A $(c)$
    req $(\mathsf{B}, c) \notin \mathsf{trans}$ and $\mathsf{safe\text{-}inj}$
    $(s_{\mathsf{A}}, t', i', m') \leftarrow \mathsf{Rcv}(s_{\mathsf{A}}, c)$
    if $m' \neq \perp$ and $(\mathsf{B}, t', i') \notin \mathsf{comp}$
        win
    $t_{\mathsf{A}} \leftarrow \max(t_{\mathsf{A}}, t')$
    delete$(t', i')$
    return $(t', i', m')$
```

```
ep-mgmt $(\mathsf{P}, \mathsf{flag})$
    if $\mathsf{P} = \mathsf{A}$ and $t_{\mathsf{P}}$ even or
      $\mathsf{P} = \mathsf{B}$ and $t_{\mathsf{P}}$ odd
        if $\mathsf{flag} = \mathsf{bad}$ and
          $\neg\mathsf{safe\text{-}ch}_{\mathsf{P}}$
            $t_{\mathsf{L}} \leftarrow t_{\mathsf{P}} + 1$
        $t_{\mathsf{P}}$ ++
        $i_{\mathsf{P}} \leftarrow 0$

sam-if-nec $(r)$
    $\mathsf{flag} \leftarrow \mathsf{bad}$
    if $r = \perp$
        $r \leftarrow\$ \mathcal{R}$
        $\mathsf{flag} \leftarrow \mathsf{good}$
    return $(r, \mathsf{flag})$
```

```
record $(\mathsf{P}, \mathsf{flag}, m, c)$
    $\mathsf{rec} \leftarrow (\mathsf{P}, t_{\mathsf{P}}, i_{\mathsf{P}}, m, c)$
    $\mathsf{trans} \overset{+}{\leftarrow} \mathsf{rec}$
    if $\neg\mathsf{safe\text{-}ch}_{\mathsf{P}}$
        $\mathsf{comp} \overset{+}{\leftarrow} \mathsf{rec}$
    if $\mathsf{flag} = \mathsf{chall}$
        $\mathsf{chall} \overset{+}{\leftarrow} \mathsf{rec}$
```

```
delete $(t, i)$
    $\mathsf{rec} \leftarrow (\mathsf{P}, t, i, m, c)$
        for some $\mathsf{P}, m, c$
    $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \overset{-}{\leftarrow} \mathsf{rec}$

$\mathsf{safe\text{-}ch}_{\mathsf{P}} \;:\Longleftrightarrow\; t_{\mathsf{P}} \geq t_{\mathsf{L}} + \Delta_{\mathsf{SM}}$

$\mathsf{safe\text{-}inj}$
  $:\Longleftrightarrow\; \min(t_{\mathsf{A}}, t_{\mathsf{B}}) \geq t_{\mathsf{L}} + \Delta_{\mathsf{SM}}$
```

Fig. 1. Oracles corresponding to party A of the SM security game for a scheme $\mathsf{SM} = (\mathsf{Init\text{-}A}, \mathsf{Init\text{-}B}, \mathsf{Send}, \mathsf{Rcv})$; the oracles for B are defined analogously.

Record keeping. The game keeps track of ciphertexts in transmission, of challenge ciphertexts, and of compromised ciphertexts. Records have the format $(\mathsf{P}, t_{\mathsf{P}}, i_{\mathsf{P}}, m, c)$, where P is the sender, t_{P} the epoch in which the message was sent, i_{P} the index within the epoch, m the message itself, and c the ciphertext.

Whenever **record** is called, the new record is added to the set trans. If a party is not in a safe state, the record is also added to the set of compromised ciphertexts comp. If the function is called with $\mathsf{flag} = \mathsf{chall}$, the record is added to

chall. The function **delete** takes an epoch number and an index and removes the corresponding record from all three record keeping sets trans, chall, and comp.

Sometimes, it is convenient to refer to a particular record (or a set thereof) by only specifying parts of it. For example, the expression $\mathsf{B} \notin \mathsf{chall}$ is equivalent to there not being any record $(\mathsf{B}, *, *, *, *)$ in the set chall. Similarly, $\mathsf{trans}(\mathsf{B})$ is the set of all records of this type in trans.

Send oracles. Both send oracles, **transmit-A** and **chall-A**, begin with **sam-if-nec**, which samples fresh randomness if necessary, followed by a call to **ep-mgmt**. Observe that the flag argument is set to $\mathsf{flag} \leftarrow \mathsf{good}$ by **sam-if-nec** if fresh randomness is used, and to $\mathsf{flag} \leftarrow \mathsf{bad}$ otherwise. Subsequently:

– **transmit-A** increments i_A, executes Send, and creates a record using $\mathsf{flag} = \mathsf{norm}$, indicating that this is not a challenge ciphertext. Observe that if A is not currently in a safe state, the record is added to comp.
– **chall-A** works similarly to **transmit-A**, except that one of the two inputs is selected according to b, and the record is saved with $\mathsf{flag} = \mathsf{chall}$, which will cause it to be added to the challenges chall. Note that **chall-A** can only be called when A is not in a compromised state, which is captured by the statement **req** $\mathsf{safe\text{-}ch}_\mathsf{A}$.

The oracles for B are defined analogously.

Receive oracles. Two oracles are available by which the attacker can get A to receive a ciphertext: **deliver-A** is intended for honest delivery, i.e., to deliver ciphertexts created by B, whereas **inject-A** is used to inject forgeries. These rules are enforced by checking (via **req**) the set trans.

– **deliver-A**: The ciphertext is first passed through Rcv, which must correctly identify the values t, i, and m recorded when c was created; if it fails to do so, the correctness property is violated and the attacker immediately wins the game. In case c was a challenge, the decrypted message is replaced by $\perp$ in order to avoid trivial attacks. Before returning the output of Rcv, t_A is incremented if t is larger than t_A, and the record corresponding to c is deleted.
– **inject-A**: Again, the ciphertext is first passed through Rcv. Unless the ciphertext corresponds[7] to $(t, i) \in \mathsf{comp}$, algorithm Rcv must reject it; otherwise, authenticity is violated and the attacker wins the game. The final instructions are as in **deliver-A**. Oracle **inject-A** may only be called if neither party is currently recovering from state compromise, which is taken care of by flag $\mathsf{safe\text{-}inj}$.

The oracles for B are defined analogously.

By deleting records at the end of **deliver-A** and **inject-A**, the game enforces that no replay attacks take place. For example, if a ciphertext c that at some point is in trans is accepted twice, the second time counts as a forgery. Similarly,

[7] cf. Property (B) in Definition 2.

if two forgeries for a compromised pair (t, i) are accepted, the attacker wins as well. Note, however, that natural schemes do not allow replay attacks (cf. Property (C) in Definition 2).

Corruption oracles. The corruption oracle for A, **corr-A**, can be called whenever no challenges are in transit from B to A, i.e., when $\mathsf{B} \notin \mathsf{chall}$. If corruption is allowed, all ciphertexts in transit sent by B become compromised. Before returning A's state, the oracle updates the time of the most recent corruption. The corruption oracle **chall-B** for B is defined similarly.

4 Building Blocks

The SM scheme presented in this work is a modular construction and uses three components: continuous key-agreement (CKA), forward-secure authenticated encryption with associated data (FS-AEAD) and—for lack of a better name—PRF-PRNGs. These primitives are presented in isolation in this section before combining them into an SM scheme in Section 5.

4.1 Continuous Key Agreement

This work distills out the public-ratchet part of the Signal protocol and casts it as a separate primitive called *continuous key agreement (CKA)*. This step is not only useful to improve the intuitive understanding of the various components of the Signal protocol and their interdependence, but it also increases modularity, which, for example, would—once the need arises—allow to replace the current CKA mechanism based on DDH by one that is post-quantum secure.

Defining CKA. At a high level, CKA is a synchronous two-party protocol between A and B. Odd rounds i consist of A sending and B receiving a message T_i, whereas in even rounds, B is the sender and A the receiver. Each round i also produces a key I_i, which is output by the sender upon sending T_i and by the receiver upon receiving T_i.

Definition 4. *A* continuous-key-agreement (CKA) scheme *is a quadruple of algorithms* $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}A}, \mathsf{CKA\text{-}Init\text{-}B}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$*, where*

– $\mathsf{CKA\text{-}Init\text{-}A}$ *(and similarly* $\mathsf{CKA\text{-}Init\text{-}B}$*) takes a key k and produces an initial state* $\gamma^{\mathsf{A}} \leftarrow \mathsf{CKA\text{-}Init\text{-}A}(k)$ *(and γ^{B}),*

– $\mathsf{CKA\text{-}S}$ *takes a state γ, and produces a new state, message, and key* $(\gamma', T, I) \leftarrow\$\ \mathsf{CKA\text{-}S}(\gamma)$*, and*

– $\mathsf{CKA\text{-}R}$ *takes a state γ and message T and produces new state and a key* $(\gamma', I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$.

Denote by $\mathcal{K}$ the space of initialization keys k and by $\mathcal{I}$ the space of CKA keys I.

Security Game for CKA

```
init $(t^*)$
    $k \leftarrow\$ \mathcal{K}$
    $\gamma^{\mathsf{A}} \leftarrow \mathsf{CKA\text{-}Init\text{-}A}(k)$
    $\gamma^{\mathsf{B}} \leftarrow \mathsf{CKA\text{-}Init\text{-}B}(k)$
    $t_{\mathsf{A}}, t_{\mathsf{B}} \leftarrow 0$
    $b \leftarrow\$ \{0,1\}$

corr-A
    req $\mathsf{allow\text{-}corr}$ or $\mathsf{finished}_{\mathsf{A}}$
    return $\gamma^{\mathsf{A}}$

send-A
    $t_{\mathsf{A}}$ ++
    $(\gamma, T_{t_{\mathsf{A}}}, I_{t_{\mathsf{A}}}) \leftarrow\$ \mathsf{CKA\text{-}S}(\gamma)$
    return $(T_{t_{\mathsf{A}}}, I_{t_{\mathsf{A}}})$

send-A' $(r)$
    $t_{\mathsf{A}}$ ++
    req $\mathsf{allow\text{-}corr}$
    $(\gamma, T_{t_{\mathsf{A}}}, I_{t_{\mathsf{A}}}) \leftarrow \mathsf{CKA\text{-}S}(\gamma; r)$
    return $(T_{t_{\mathsf{A}}}, I_{t_{\mathsf{A}}})$

receive-A
    $t_{\mathsf{A}}$ ++
    $(\gamma^{\mathsf{A}}, *) \leftarrow \mathsf{CKA\text{-}R}(\gamma^{\mathsf{A}}, T_{t_{\mathsf{A}}})$

chall-A
    $t_{\mathsf{A}}$ ++
    req $t_{\mathsf{A}} = t^*$
    $(\gamma, T_{t_{\mathsf{A}}}, I_{t_{\mathsf{A}}}) \leftarrow\$ \mathsf{CKA\text{-}S}(\gamma)$
    if $b = 0$
        return $(T_{t_{\mathsf{A}}}, I_{t_{\mathsf{A}}})$
    else
        $I \leftarrow\$ \mathcal{I}$
        return $(T_{t_{\mathsf{A}}}, I)$
```

$$\mathsf{allow\text{-}corr}_{\mathsf{P}} \;:\Longleftrightarrow\; \max(t_{\mathsf{A}}, t_{\mathsf{B}}) \leq t^* - 2$$

$$\mathsf{finished}_{\mathsf{P}} \;:\Longleftrightarrow\; t_{\mathsf{P}} \geq t^* + \Delta_{\mathsf{CKA}}$$

Fig. 2. Oracles corresponding to party A of the CKA security game for a scheme $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}A}, \mathsf{CKA\text{-}Init\text{-}B}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$; the oracles for B are defined analogously.

Correctness. A CKA scheme is correct if in the security game in Figure 2 (explained below), A and B always, i.e., with probability 1, output the same key in every round.

Security. The basic property a CKA scheme must satisfy is that conditioned on the transcript $T_1, T_2, \ldots$, the keys $I_1, I_2, \ldots$ look uniformly random and independent. An attacker against a CKA scheme is required to be passive, i.e., may not modify the messages T_i. However, it is given the power to possibly (1) control the random coins used by the sender and (2) leak the current state of either party. Correspondingly, the keys I_i produced under such circumstances need not be secure. The parties are required to recover from a state compromise *within* 2 *rounds*.[8]

The formal security game for CKA is provided in Figure 2. It begins with a call to the **init** oracle, which samples a bit b, initializes the states of both parties, and defines epoch counters t_{A} and t_{B}. Procedure **init** takes a value t^*, which determines in which round the challenge oracle may be called.

Upon completion of the initialization procedure, the attacker gets to interact arbitrarily with the remaining oracles, as long as *the calls are in a "ping-pong" order*, i.e., a call to a send oracle for A is followed by a receive call for B, then by a send oracle for B, etc. The attacker only gets to use the challenge oracle

[8] Of course, one could also parametrize the number of rounds required to recover (all CKA schemes in this work recover within two rounds, however).

for epoch t^*. No corruption or using bad randomness (**send-A'** and **send-B'**) is allowed less than two epochs before the challenge is sent ($\mathsf{allow\text{-}corr}$).

The game is parametrized by Δ_{CKA}, which stands for the number of epochs that need to pass after t^* until the states do not contain secret information pertaining to the challenge. Once a party reaches epoch $t^* + \Delta_{\mathsf{CKA}}$, its state may be revealed to the attacker (via the corresponding corruption oracle). The game ends (not made explicit) once both states are revealed after the challenge phase. The attacker wins the game if it eventually outputs a bit $b' = b$.

The advantage of an attacker $\mathcal{A}$ against a CKA scheme CKA with $\Delta_{\mathsf{CKA}} = \Delta$ is denoted by $\mathrm{Adv}^{\mathsf{CKA}}_{\mathrm{ror},\Delta}(\mathcal{A})$. The attacker is parameterized by its running time t.

Definition 5. *A CKA scheme* CKA *is* (t, Δ, ε)-secure *if for all t-attackers $\mathcal{A}$,*

$$\mathrm{Adv}^{\mathsf{CKA}}_{\mathrm{ror},\Delta}(\mathcal{A}) \;\leq\; \varepsilon \;.$$

Instantiating CKA. This paper presents several instantiations of CKA: First, a generic CKA scheme with $\Delta = 0$ based on any key-encapsulation mechanism (KEM). Then, by considering the ElGamal KEM and observing that an encapsulated key can be "reused" as public key, one obtains a CKA scheme based on the decisional Diffie-Hellman (DDH) assumption, where the scheme saves a factor of 2 in communication compared to a straight-forward instantiation of the generic scheme. However, the scheme has $\Delta = 1$.

CKA FROM KEMS. A CKA scheme with $\Delta = 0$ can be built from a KEM in natural way: in every epoch, one party sends a public key pk of a freshly generated key pair and an encapsulated key under the key pk' received from the other party in the previous epoch. Specifically, consider a CKA scheme $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}A}, \mathsf{CKA\text{-}Init\text{-}B}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$ that is obtained from a KEM KEM as follows:

- The initial shared state $k = (\mathsf{pk}, \mathsf{sk})$ consists of a (freshly generated) KEM key pair. The initialization for A outputs $\mathsf{pk} \leftarrow \mathsf{CKA\text{-}Init\text{-}A}(k)$ and that for B outputs $\mathsf{sk} \leftarrow \mathsf{CKA\text{-}Init\text{-}B}(k)$.
- The send algorithm $\mathsf{CKA\text{-}S}$ takes as input the current state $\gamma = \mathsf{pk}$ and proceeds as follows: It
  1. encapsulates a key $(c, I) \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk})$;
  2. generates a new key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KG}$;
  3. sets the CKA message to $T \leftarrow (c, \mathsf{pk})$;
  4. sets the new state to $\gamma \leftarrow \mathsf{sk}$; and
  5. returns (γ, T, I).
- The receive algorithm $\mathsf{CKA\text{-}R}$ takes as input the current state $\gamma = \mathsf{sk}$ as well as a message $T = (c, \mathsf{pk})$ and proceeds as follows: It
  1. decapsulates the key $I \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$;
  2. sets the new state to $\gamma \leftarrow \mathsf{pk}$; and
  3. returns (γ, I).

The full version of this paper [**?**] shows that the above scheme is a secure CKA protocol by reducing its security to that of the underlying KEM.

CKA FROM DDH. Observe that if one instantiates the above KEM-based CKA scheme with the ElGamal KEM over some group G, both the public key and the encapsulated key are elements of G. Hence, the Signal protocol uses an optimization of the ElGamal KEM where a single group element first serves as an encapsulated key sent by, say, A and then as the public key B uses to encapsulate his next key. Interestingly, this comes at the price of having $\Delta = 1$ (as opposed to $\Delta = 0$) due to the need for parties to hold on to their exponents (which serve both as secret keys and encapsulation randomness) longer.

Concretely, a CKA scheme $\mathsf{CKA} = (\mathsf{CKA\text{-}Init\text{-}A}, \mathsf{CKA\text{-}Init\text{-}B}, \mathsf{CKA\text{-}S}, \mathsf{CKA\text{-}R})$ can be obtained from the DDH assumption[9] in a cyclic group $G = \langle g \rangle$ as follows:

- The initial shared state $k = (h, x_0)$ consists of a (random) group element $h = g^{x_0}$ and its discrete logarithm x_0. The initialization for A outputs $h \leftarrow \mathsf{CKA\text{-}Init\text{-}A}(k)$ and that for B outputs $x_0 \leftarrow \mathsf{CKA\text{-}Init\text{-}B}(k)$.
- The send algorithm $\mathsf{CKA\text{-}S}$ takes as input the current state $\gamma = h$ and proceeds as follows: It
  1. chooses a random exponent x;
  2. computes the corresponding key $I \leftarrow h^x$;
  3. sets the CKA message to $T \leftarrow g^x$;
  4. sets the new state to $\gamma \leftarrow x$; and
  5. returns (γ, T, I).
- The receive algorithm $\mathsf{CKA\text{-}R}$ takes as input the current state $\gamma = x$ as well as a message $T = h$ and proceeds as follows: It
  1. computes the key $I = h^x$;
  2. sets the new state to $\gamma \leftarrow h$; and
  3. returns (γ, I).

The full version of this paper [**?**] shows that the above scheme is a secure CKA protocol if the DDH assumption holds in group G.

4.2 Forward-Secure AEAD

Defining FS-AEAD. *Forward-secure authenticated encryption with associated data* is a stateful primitive between a sender A and a receiver B and can be considered a single-epoch variant of an SM scheme, a fact that is also evident from its security definition, which resembles that of SM schemes.

Definition 6. Forward-secure authenticated encryption with associated data (FS-AEAD) *is a tuple of algorithms* $\mathsf{FS\text{-}AEAD} = (\mathsf{FS\text{-}Init\text{-}S}, \mathsf{FS\text{-}Init\text{-}R}, \mathsf{FS\text{-}Send}, \mathsf{FS\text{-}Rcv})$*, where*

[9] The DDH assumption states that it is hard to distinguish DH triples (g^a, g^b, g^{ab}) from random triples (g^a, g^b, g^c), where a, b, and c are uniformly random and independent exponents.

– $\mathsf{FS\text{-}Init\text{-}S}$ *(and similarly* $\mathsf{FS\text{-}Init\text{-}R}$*) takes a key k and outputs a state* $v_{\mathsf{A}} \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$*,*

– $\mathsf{FS\text{-}Send}$ *takes a state v, associated data a, and a message m and produces a new state and a ciphertext* $(v', e) \leftarrow \mathsf{FS\text{-}Send}(v, a, m)$*, and*

– $\mathsf{FS\text{-}Rcv}$ *takes a state v, associated data a, and a ciphertext e and produces a new state, an index, and a message* $(v', i, m) \leftarrow \mathsf{FS\text{-}Rcv}(v, a, e)$.

Observe that all algorithms of an FS-AEAD scheme are deterministic.

Memory management. In addition to the basic syntax above, it is useful to define the following two functions $\mathsf{FS\text{-}Stop}$ (called by the sender) and $\mathsf{FS\text{-}Max}$ (called by the receiver) for memory management:

– $\mathsf{FS\text{-}Stop}$, given an FS-AEAD state v, outputs how many messages have been received and then "erases" the FS-AEAD session corresponding to v form memory; and

– $\mathsf{FS\text{-}Max}$, given a state v and an integer ℓ, remembers ℓ internally such that the session corresponding to v is erased from memory as soon as ℓ messages have been received.

These features will be useful in the full protocol (cf. Section 5) to be able to terminate individual FS-AEAD sessions when they are no longer needed. Providing a formal requirement for these additional functions is omitted. Moreover, since an attacker can infer the value of the message counter from the behavior of the protocol anyway, there is no dedicated oracle included in the security game below.

Correctness and security. Both correctness and security are built into the security game depicted in Figure 3. The game is the single-epoch analogue of the SM security game (cf. Figure 1) and therefore has similarly defined oracles and similar record keeping. A crucial difference is that as soon as the receiver B is compromised, the game ends with a full state reveal as no more security can be provided. If only the sender A is compromised, the game continues and uncompromised messages must remain secure.

The advantage of an attacker $\mathcal{A}$ against an FS-AEAD scheme $\mathsf{FS\text{-}AEAD}$ is denoted by the expression $\mathrm{Adv}^{\mathsf{FS\text{-}AEAD}}_{\text{fs-aead}}(\mathcal{A})$. The attacker is parameterized by its running time t and the total number of queries q it makes.

Definition 7. *An FS-AEAD scheme* $\mathsf{FS\text{-}AEAD}$ *is* (t, q, ε)-secure *if for all* (t, q)*-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\mathsf{FS\text{-}AEAD}}_{\text{fs-aead}}(\mathcal{A}) \leq \varepsilon \ .$$

Instantiating FS-AEAD An FS-AEAD scheme can be easily constructed from two components:

– an AEAD scheme $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$, and

– a PRG $G : \mathcal{W} \to \mathcal{W} \times \mathcal{K}$, where $\mathcal{K}$ is the key space of the AEAD scheme.

Security Game for FS-AEAD

```
init
    $k \leftarrow\$ \mathcal{K}$
    $v_{\mathsf{A}} \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$
    $v_{\mathsf{B}} \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
    $i_{\mathsf{A}} \leftarrow 0$
    $\mathsf{corr}_{\mathsf{A}} \leftarrow \mathsf{false}$
    $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \leftarrow \emptyset$
    $b \leftarrow\$ \{0,1\}$

corr-A
    $\mathsf{corr}_{\mathsf{A}} \leftarrow \mathsf{true}$
    return $v_{\mathsf{A}}$

corr-B
    req $\mathsf{chall} = \emptyset$
    end $(v_{\mathsf{A}}, v_{\mathsf{B}})$

transmit-A $(a, m)$
    $i_{\mathsf{A}}$ ++
    $(v_{\mathsf{A}}, e) \leftarrow \mathsf{FS\text{-}Send}(v_{\mathsf{A}}, a, m)$
    record$(\mathsf{good}, a, m, e)$
    return $c$

chall-A $(a, m_0, m_1)$
    req $\neg\mathsf{corr}_{\mathsf{A}}$ and $|m_0| = |m_1|$
    $i_{\mathsf{A}}$ ++
    $(v_{\mathsf{A}}, e) \leftarrow \mathsf{FS\text{-}Send}(v_{\mathsf{A}}, a, m_b)$
    record$(\mathsf{chall}, a, m_b, e)$
    return $e$

deliver-A $(a, e)$
    req $(i, a, m, e) \in \mathsf{trans}$
        for some $i, m$
    $(v_{\mathsf{A}}, i', m') \leftarrow \mathsf{FS\text{-}Rcv}(v_{\mathsf{A}}, a, e)$
    if $(i', m') \neq (i, m)$
        win
    if $(i, m) \in \mathsf{chall}$
        $m' \leftarrow \bot$
    delete$(i)$
    return $(i', m')$

inject-A $(a, e)$
    req $(a, e) \notin \mathsf{trans}$
    $(v_{\mathsf{A}}, i', m') \leftarrow \mathsf{FS\text{-}Rcv}(v_{\mathsf{A}}, a, e)$
    if $m' \neq \bot$ and $(\mathsf{B}, i') \notin \mathsf{comp}$
        win
    delete$(i')$
    return $(i', m')$
```

```
record $(\mathsf{flag}, a, m, e)$
    $\mathsf{rec} \leftarrow (i_{\mathsf{A}}, a, m, e)$
    $\mathsf{trans} \xleftarrow{+} \mathsf{rec}$
    if $\mathsf{flag} = \mathsf{bad}$ or $\mathsf{corr}_{\mathsf{A}}$
        $\mathsf{comp} \xleftarrow{+} \mathsf{rec}$
    if $\mathsf{flag} = \mathsf{chall}$
        $\mathsf{chall} \xleftarrow{+} \mathsf{rec}$

delete $(i)$
    $\mathsf{rec} \leftarrow (i, a, m, e)$ for $m, a, e$
        s.t. $(i, a, m, e) \in \mathsf{trans}$
    $\mathsf{trans}, \mathsf{chall}, \mathsf{comp} \xleftarrow{-} \mathsf{rec}$
```

Fig. 3. Oracles corresponding to party A of the FS-AEAD security game for a scheme FS-AEAD = (FS-Init-S, FS-Init-R, FS-Send, FS-Rcv); the oracles for B are defined analogously.

The scheme is described in Figure 4. For simplicity the states of sender A and receiver B are is not made explicit; it consists of the variables set during initialization. The main idea of the scheme, is that the A and B share the state w of a PRG G. State w is initialized with a pre-shared key $k \in \mathcal{W}$, which is assumed to be chosen uniformly at random. Both parties keep local counters i_{A} and i_{B}, respectively.[10] A, when sending the i^{th} message m with associated data (AD) a, uses G to expand the current state to a new state and an AEAD key $(w, K) \leftarrow G(w)$ and computes an AEAD encryption under K of m with AD $h = (i, a)$.

Since B may receive ciphertext out of order, whenever he receives a ciphertext, he first checks whether the key is already stored in a dictionary $\mathcal{D}$. If the index of the message is higher than expected (i.e., larger than $i_{\mathsf{B}} + 1$), B skips the PRG

[10] For ease of description, the FS-AEAD state of the parties is not made explicit as a variable v.

Forward-Secure AEAD

```
Init-A $(k)$
    $w \leftarrow k$
    $i_A \leftarrow 0$

Init-B $(k)$
    $w \leftarrow k$
    $i_B \leftarrow 0$
    $\mathcal{D}[\cdot] \leftarrow \lambda$

try-skipped $(i)$
    $K \leftarrow \mathcal{D}[i]$
    $\mathcal{D}[i] \leftarrow \bot$
    return $K$

FS-Send $(a, m)$
    $i_A$ ++
    $(w, K) \leftarrow G(w)$
    $h \leftarrow (i_A, a)$
    $e \leftarrow \mathsf{Enc}(K, h, m)$
    return $(i_A, e)$

skip $(u)$
    while $i_B < u - 1$
        $i_B$ ++
        $(w, K) \leftarrow G(w)$
        $\mathcal{D}[u] \leftarrow K$

FS-Rcv $(a, c)$
    $(i, e) \leftarrow c$
    $K \leftarrow$ try-skipped$(i)$
    if $K = \bot$
        skip$(i)$
        $(w, K) \leftarrow G(w)$
        $i_B \leftarrow i$
    $h \leftarrow (i, a)$
    $m \leftarrow \mathsf{Dec}(K, h, e)$
    if $m = \bot$
        error
    return $(i, m)$
```

Fig. 4. FS-AEAD scheme based on AEAD and a PRG.

ahead and stores the skipped keys in $\mathcal{D}$. In either case, once the key is obtained, it is used to decrypt. If decryption fails, FS-Rcv throws an exception (**error**), which causes the state to be rolled back to where it was before the call to FS-Rcv.

In the full version of this work [**?**], it is shown that, based on the security of the AEAD scheme and the PRG, the above yields a secure FS-AEAD scheme.

4.3 PRF-PRNGs

Defining PRF-PRNGs. A *PRF-PRNG* resembles both a pseudo-random function (PRF) and a pseudorandom number generator with input (PRNG)—hence the name. On the one hand, as a PRNG would, a PRF-PRNG (1) repeatedly accepts inputs I and uses them to refresh its state σ and (2) occasionally uses the state, provided it has sufficient entropy, to derive a pseudo-random pair of output R and new state; for the purposes of secure messaging, it suffices to combine properties (1) and (2) into a single procedure. On the other hand, a PRF-PRNG can be used as a PRF in the sense that if the state has high entropy, the answers to various inputs I *on the same state* are indistinguishable from random and independent values.

Definition 8. *A* PRF-PRNG *is a pair of algorithms* $\mathsf{P} = (\mathsf{P\text{-}Init}, \mathsf{P\text{-}Up})$*, where*

- $\mathsf{P\text{-}Init}$ *takes a key* k *and produces a state* $\sigma \leftarrow \mathsf{P\text{-}Init}(k)$*, and*
- $\mathsf{P\text{-}Up}$ *takes a state* σ *and an input* I *and produces a new state and an output* $(\sigma', R) \leftarrow \mathsf{P\text{-}Up}(\sigma, I)$*.*

SECURITY. The simple intuitive security requirement for a double-seed PRG is that $\mathsf{P\text{-}Init}(\sigma, I)$ produce a pseudorandom value if the state σ is uncorrupted (i.e., has high entropy) or the input I is chosen uniformly from some set $\mathcal{S}$. Moreover,

Security Game for PRF-PRNG

```
init
    $k \leftarrow\$ \mathcal{K}$
    $\sigma \leftarrow \mathsf{P\text{-}Init}(k)$
    $\mathsf{corr} \leftarrow \mathsf{false}$
    $\mathsf{prng}, \mathsf{prf} \leftarrow \mathsf{false}$
    $b \leftarrow\$ \{0,1\}$

corr
    req $\neg\mathsf{prf}$
    return $\sigma$

process $(I)$
    $I \leftarrow$ sam-if-nec$(I)$
    $(\sigma, R) \leftarrow \mathsf{P\text{-}Up}(\sigma, I)$
    return $R$

chall-prf $(I)$
    req $\neg\mathsf{corr}$ and $\neg\mathsf{prng}$
    $\mathsf{prf} \leftarrow \mathsf{true}$
    $(\sigma', R) \leftarrow \mathsf{P\text{-}Up}(\sigma, I)$
    if $b = 1$
        $R \leftarrow\$ \mathcal{R}$
    return $(\sigma', R)$

chall-prng $(I)$
    $I \leftarrow$ sam-if-nec$(I)$
    req $\neg\mathsf{corr}$ and $\neg\mathsf{prf}$
    $\mathsf{prng} \leftarrow \mathsf{true}$
    $(\sigma, R) \leftarrow \mathsf{P\text{-}Up}(\sigma, I)$
    if $b = 1$
        $R \leftarrow\$ \mathcal{R}$
    return $R$

sam-if-nec $(I)$
    if $I = \perp$
        $I \leftarrow\$ \mathcal{I}$
        $\mathsf{corr} \leftarrow \mathsf{false}$
    return $I$
```

Fig. 5. Oracles of the PRF-PRNG security game for a scheme $\mathsf{P} = (\mathsf{P\text{-}Init}, \mathsf{P\text{-}Up})$.

if the state is uncorrupted, it should have the PRF property described above. This is captured by the security definition described by Figure 5:

– *Initialization:* Procedure **init** chooses a random bit b, initializes the PRF-PRNG with a random key, and sets two flags prng and prf to false: the PRNG and PRF modes are mutually exclusive and only one type of challenge may be called; the flags keep track of which.
– *PRNG mode:* The oracle **process** can be called in two ways: either I is an input specified by the attacker and is simply absorbed into the state, or $I = \perp$, in which case the game chooses it randomly (inside **sam-if-nec**) and absorbs it into the state, which at this point becomes uncorrupted. Oracle **chall-prng** is works in the same fashion but creates a challenge.
– *PRF mode:* Once the state is uncompromised the attacker can decide to obtain PRF challenges by calling **chall-prf**, which simply evaluates the (adversarially chosen) input on the current state without updating it and creates a challenge.
– *Corruption:* At any time, except after asking for PRF challenges, the attacker may obtain the state by calling **corr**.

The advantage of $\mathcal{A}$ in the PRF-PRNG game is denoted by $\mathrm{Adv}^{\mathsf{P}}_{\mathrm{PP}}(\mathcal{A})$. The attacker is parameterized by its running time t.

Definition 9. *An PRF-PRG* P *is* (t, ε)-secure *if for all t-attackers $\mathcal{A}$,*

$$\mathrm{Adv}^{\mathsf{P}}_{\mathrm{PP}}(\mathcal{A}) \;\le\; \varepsilon \;.$$

Instantiating PRF-PRNGs. Being a PRF-PRNG is a property the HKDF function used by Signal is assumed to have; in particular, Marlinspike and Perrin [**?**] recommend the primitive be implemented with using HKDF [**?**] with SHA-256 or SHA-512 [**?**] where the state σ is used as HKDF salt and I as HKDF input key material. This paper therefore merely reduces the security of the presented schemes to the PRF-PRNG security of whatever function is used to instantiate it.

Alternatively, a simple standard-model instantiation (whose rather immediate proof is omitted) can be based on a pseudorandom *permutation* (PRP) $\Pi : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ and a PRG $G : \{0,1\}^n \to \{0,1\}^n \times \mathcal{K}$ by letting the state be the PRP key $s \in \{0,1\}^n$ and

$$(s', R) \leftarrow \mathsf{P\text{-}Up}(s, I) \;=\; G(\Pi_s(I)) \ .$$

5 Secure Messaging Scheme

This section presents a Signal-based secure messaging (SM) scheme and establishes its security under Definition 3. The scheme suitably and modularly combines continuous key-agreement (CKA), forward-secure authenticated encryption with associated data (FS-AEAD), and PRF-PRNGs; these primitives are explained in detail in Section 4.

5.1 The Scheme

The scheme is inspired by the Signal protocol, but differs from it in a few points, as explained in Section 5.2. The main idea of the scheme is that the parties A and B keep track of the same PRF-PRG (aka the "root RNG"), which they use to generate keys for FS-AEAD instances as needed. The root RNG is continuously refreshed by random values output by a CKA scheme that is run "in parallel."

State. Scheme SM keeps an internal state s_A (resp. s_B), which is initialized by $\mathsf{Init\text{-}A}$ (resp. $\mathsf{Init\text{-}B}$) and used as well as updated by Send and Rcv. The state s_A of SM consists of the following values:

– an ID field with $\mathsf{id} = \mathsf{A}$,
– the state σ_{root} of the root RNG,
– states $v[0], v[1], v[2], \ldots$ of the various FS-AEAD instances,
– the state γ of the CKA scheme,
– the current CKA message T_{cur}, and
– an epoch counter t_A.

In order to remove expired FS-AEAD sessions from memory, there is also a variable ℓ_{prv} that remembers the number of messages sent in the second most recent epoch. Recall (cf. Section 4.2) that once the maximum number of messages has been set via $\mathsf{FS\text{-}Max}$, a session "erases" itself from the memory, and similarly for calling $\mathsf{FS\text{-}Stop}$ on a particular FS-AEAD session. For simplicity, removing

Signal-Based Secure-Messaging Scheme

```
Init-A (k)
    id ← A
    $(k_{\mathsf{root}}, k_{\mathsf{CKA}}) \leftarrow k$
    $\sigma_{\mathsf{root}} \leftarrow \mathsf{P\text{-}Init}(k_{\mathsf{root}})$
    $v[\cdot] \leftarrow \lambda$
    $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{P\text{-}Up}(\sigma_{\mathsf{root}}, \lambda)$
    $v[0] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
    $\gamma \leftarrow \mathsf{CKA\text{-}Init\text{-}A}(k_{\mathsf{CKA}})$
    $T_{\mathsf{cur}} \leftarrow \lambda$
    $\ell_{\mathsf{prv}} \leftarrow 0$
    $t_{\mathsf{A}} \leftarrow 0$

Send-A (m)
    if $t_{\mathsf{A}}$ is even
        $\ell_{\mathsf{prv}} \leftarrow \mathsf{FS\text{-}Stop}(v[t_{\mathsf{A}} - 1])$
        $t_{\mathsf{A}}$ ++
        $(\gamma, T_{\mathsf{cur}}, I) \leftarrow\$ \mathsf{CKA\text{-}S}(\gamma)$
        $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{P\text{-}Up}(\sigma_{\mathsf{root}}, I)$
        $v[t_{\mathsf{A}}] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(k)$
    $h \leftarrow (t_{\mathsf{A}}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$
    $(v[t_{\mathsf{A}}], e) \leftarrow \mathsf{FS\text{-}Send}(v[t_{\mathsf{A}}], h, m)$
    return $(h, e)$

Rcv-A (c)
    $(h, e) \leftarrow c$
    $(t, T, \ell) \leftarrow h$
    req $t$ even and $t \leq t_{\mathsf{A}} + 1$
    if $t = t_{\mathsf{A}} + 1$
        $t_{\mathsf{A}}$ ++
        $\mathsf{FS\text{-}Max}(v[t-2], \ell)$
        $(\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$
        $(\sigma_{\mathsf{root}}, k) \leftarrow \mathsf{P\text{-}Up}(\sigma_{\mathsf{root}}, I)$
        $v[t] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(k)$
    $(v[t], i, m) \leftarrow \mathsf{FS\text{-}Rcv}(v[t], h, e)$
    if $m = \perp$
        error
    return $(t, i, m)$
```

Fig. 6. Secure-messaging scheme based on a FS-AEAD, a CKA scheme, and a PRF-PRNG.

the corresponding $v[t]$ from memory is not made explicit in either case. The state s_{B} is defined analogously.

The algorithms. The algorithms of scheme SM are depicted in Figure 6 and described in more detail below. For ease of description, the algorithms Send and Rcv are presented as Send-A and Rcv-A, which handle the case where id = A; the case id = B works analogously. Moreover, to improve readability, the state s_{A} is not made explicit in the description: it consists of the variables set by the initialization algorithm.

- *Initialization:* The initialization procedure Init-A expects a key k shared between A and B; k is assumed to have been created at some point before the execution during a trusted setup phase and to consist of initialization keys k_{root} and k_{CKA} for the root RNG and the CKA scheme, respectively. In a second step, the root RNG is initialized with k. Then, it is used to generate a key for FS-AEAD epoch $v[0]$; A acts as receiver in $v[0]$ and all subsequent even epochs and as sender in all subsequent odd epochs. Furthermore, Init-A also initializes the CKA scheme and sets the initial epoch $t_{\mathsf{A}} \leftarrow 0$ and T_{cur} to a default value.[11]

As pointed out above, scheme SM runs a CKA protocol in parallel to sending its messages. To that end, A's first message includes the first message T_1 output by CKA-S. All subsequent messages sent by A include T_1 until some message received from B includes T_2. At that point A would run CKA-S again and include T_3 with all her messages, and so on (cf. Section 4.1).

[11] B also starts in epoch $t_{\mathsf{B}} \leftarrow 0$.

Upon either sending or receiving T_i for odd or even i, respectively, the CKA protocol also produces a random value I_i, which A absorbs into the root RNG. The resulting output k is used as key for a new FS-AEAD epoch.

– *Sending messages:* Procedure Send-A allows A to send a message to B. As a first step, Send-A determines whether it is A's turn to send the next CKA message, which is the case if t_{A} is even. Whenever it is A's turn, Send-A runs CKA-S to produce the her next CKA message T and key I, which is absorbed into the root RNG. The resulting value k is used as a the key for a new FS-AEAD epoch, in which A acts as sender. The now old epoch is terminated by calling FS-Stop and the number of messages in the old epoch is stored in ℓ_{prv}, which will be sent along inside the header for every message of the new epoch.
  Irrespective of whether it was necessary to generate a new CKA message and generate a new FS-AEAD epoch, Send-A creates a header $h = (t_{\mathsf{A}}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$, and uses the current epoch $v[t_{\mathsf{A}}]$ to get a ciphertext for (h, m) (where h is treated as associated data).
– *Receiving messages:* When a ciphertext $c = (h, e, \ell)$ with header $h = (t, T, \ell)$ is processed by Rcv-A, there are two possibilities:
  - $t \leq t_{\mathsf{A}}$ (and t even): In this case, ciphertext c pertains to an existing FS-AEAD epoch, in which case FS-Send is simply called on $v[t]$ to process e. If the maximum number of messages has been received for session $v[t]$, the session is removed from memory.
  - $t = t_{\mathsf{A}} + 1$ and t_{A} odd: Here, the receiver algorithm advances t_{A} by incrementing it and processes T with CKA-R. This produces a key I, which is absorbed into the PRF-PRG to obtain a key k with which to initialize a new epoch $v[t_{\mathsf{A}}]$ as receiver. Then, e is processed by FS-Rcv on $v[t_{\mathsf{A}}]$. Note that Rcv also uses FS-Max to store ℓ as the maximum number of messages in the previous receive epoch.

  Irrespective of whether a new CKA message was received and a new epoch created, if e is rejected by FS-Rcv, the algorithm raises an exception (**error**), which causes the entire state s_{A} to be rolled back to what it was before Rcv-A was called.

5.2 Differences to Signal

By instantiating the building blocks as shown below, one obtains an SM scheme that is very close to the actual Signal protocol (cf. [**?**, Section 5.2] for more details):

– *CKA:* the DDH-based CKA scheme from Section 4.1 using Curve25519 or Curve448 as specified in [**?**];
– *FS-AEAD:* FS-AEAD scheme from Section 4.2 with HMAC [**?**] with SHA-256 or SHA-512 [**?**] for the PRG, and an AEAD encryption scheme based on either SIV or a composition of CBC with HMAC [**?**,**?**];

– *PRF-PRNG:* HKDF [?] with SHA-256 or SHA-512 [?], used as explained in Section 4.3.

We now detail the main differences:

Deferred randomness for sending. Deployed Signal implementations generate a new CKA message and absorb the resulting key into the RNG in Rcv, as opposed to taking care of this inside Send, as done here. The way it is done here is advantageous in the sense that the new key is not needed until the Send operation is actually initiated, so there is no need to risk its exposure unnecessarily (in case the state is compromised in between receiving and sending). Indeed, this security enhancement to Signal was explicitly mentioned by Marlinspike and Perrin [?] (cf. Section 6.5), and we simply follow this suggestion for better security.

Epoch indexing. In our scheme we have an explicit epoch counter t to index a given epoch. In Signal, one uses the uniqueness of latest CKA message (of the form g^x) to index an epoch. This saves an extra counter from each party's state, but we find our treatment of having explicit epoch counters much more intuitive, and not relying on any particular structure of CKA messages. In fact, indexing a dictionary becomes slightly more efficient when using a simple counter than the entire CKA message (which could be long for certain CKA protocols; e.g., post-quantum from lattices).

FS-AEAD abstraction. Unlike the SM proposed from this section, Signal does not use the FS-AEAD abstraction. Instead, each party maintains a sending and a receiving PRG that are kept in sync with the other party's receiving and sending PRG, respectively. Moreover, when receiving the first message of a new epoch, the current receive PRG is skipped ahead appropriately depending on the value ℓ, and the skipped keys are stored in a *single*, global dictionary. The state of the receive PRG is then overwritten with the new state output by the root RNG. Then, upon the next send operation new randomness for the CKA message is generated, and the sending RNG is also overwritten by the state output from updating the root RNG again. This is logically equivalent to our variant of Signal with the particular FS-AEAD implementation in Figure 4, except we will maintain multiple dictionaries (one for each epoch t). However, merging these dictionaries into one global dictionary (indexed by epoch counter in addition to the message count within epoch) becomes a simple efficiency optimization of the resulting scheme. Moreover, once this optimization is done, there is no need to store an array of FS-AEAD instances $v[t]$. Instead, we can only remember the latest sending and receiving FS-AEAD instance, overwriting them appropriately with each new epoch. Indeed, storing old message keys from not-yet-delivered messages is the only information one needs to remember from the prior FS-AEAD instances. So once this information is stored in the global dictionary, we can simply overwrite the remaining information when moving to the new epoch. With these simple efficiency optimizations, we arrive to (almost) precisely what is done by Signal (cf. Figure 7).

Signal Scheme

```
Init-A $(k)$
    $\mathsf{id} \leftarrow \mathsf{A}$
    $(k_{\mathsf{root}}, k_{\mathsf{CKA}}) \leftarrow k$
    $\sigma_{\mathsf{root}} \leftarrow \mathsf{P\text{-}Init}(k_{\mathsf{root}})$
    $(\sigma_{\mathsf{root}}, \sigma_{\mathsf{B}}) \leftarrow \mathsf{P\text{-}Up}(\sigma_{\mathsf{root}}, \lambda)$
    $\gamma \leftarrow \mathsf{CKA\text{-}Init\text{-}A}(k_{\mathsf{CKA}})$
    $T_{\mathsf{cur}} \leftarrow \lambda$
    $\ell_{\mathsf{prv}} \leftarrow 0$
    $t_{\mathsf{A}}, i_{\mathsf{A}}, i_{\mathsf{B}} \leftarrow 0$
    $\mathcal{D}[\cdot] \leftarrow \lambda$

skip $(t, u)$
    while $i_{\mathsf{B}} < u$
        $i_{\mathsf{B}}$ ++
        $(\sigma_{\mathsf{B}}, K) \leftarrow G(\sigma_{\mathsf{B}})$
        $\mathcal{D}[t, i_{\mathsf{B}}] \leftarrow K$
```

```
Send-A $(m)$
    if $t_{\mathsf{A}}$ is even
        $t_{\mathsf{A}}$ ++
        $\ell_{\mathsf{prv}} \leftarrow i_{\mathsf{A}}$
        $i_{\mathsf{A}} \leftarrow 0$
        $(\gamma, T_{\mathsf{cur}}, I) \leftarrow\$ \mathsf{CKA\text{-}S}(\gamma)$
        $(\sigma_{\mathsf{root}}, \sigma_{\mathsf{A}}) \leftarrow \mathsf{P\text{-}Up}(\sigma_{\mathsf{root}}, I)$
    $i_{\mathsf{A}}$ ++
    $h \leftarrow (t_{\mathsf{A}}, i_{\mathsf{A}}, T_{\mathsf{cur}}, \ell_{\mathsf{prv}})$
    $(\sigma_{\mathsf{A}}, K) \leftarrow G(\sigma_{\mathsf{A}})$
    $e \leftarrow \mathsf{Enc}(K, h, m)$
    return $(h, e)$

try-skipped $(t, i)$
    $K \leftarrow \mathcal{D}[t, i]$
    $\mathcal{D}[t, i] \leftarrow \perp$
    return $K$
```

```
Rcv-A $(c)$
    $(h, e) \leftarrow c$
    $(t, i, T, \ell) \leftarrow h$
    req $t$ even and $t \leq t_{\mathsf{A}} + 1$
    if $t = t_{\mathsf{A}} + 1$
        $\mathsf{skip}(t - 2, \ell_{\mathsf{prv}})$
        $t_{\mathsf{A}}$ ++, $i_{\mathsf{B}} \leftarrow 0$
        $(\gamma, I) \leftarrow \mathsf{CKA\text{-}R}(\gamma, T)$
        $(\sigma_{\mathsf{root}}, \sigma_{\mathsf{B}}) \leftarrow \mathsf{P\text{-}Up}(\sigma_{\mathsf{root}}, I)$
    $K \leftarrow \mathsf{try\text{-}skipped}(t, i)$
    if $K = \perp$
        $\mathsf{skip}(t, i - 1)$
        $i_{\mathsf{B}}$ ++
        $(\sigma_{\mathsf{B}}, K) \leftarrow G(\sigma_{\mathsf{B}})$
    $m \leftarrow \mathsf{Dec}(K, h, e)$
    if $m = \perp$
        error
    return $(t, i, m)$
```

Fig. 7. Signal scheme without the FS-AEAD abstraction, based on a CKA scheme, a PRF-PRNG, authenticated encryption, and a regular PRG. The figure only shows the algorithms for A; B's algorithms are analogous, with the roles of i_{A} and i_{B} switched.

To sum up, blindly using the FS-AEAD abstraction results in a slightly less efficient scheme, but (1) we feel our treatment is more modular and intuitive; (2) when using a concrete FS-AEAD scheme from Section 4.2, getting actual Signal becomes a simple efficiency optimization of the resulting scheme. In particular, the security of Signal itself still follows from our framework.

Initial key agreement. As mentioned in the introduction, our modeling only addresses the double-ratchet aspect of the Signal protocol, and does not tackle the challenging problem of the generation of the initial shared key k. One thing this also allows us to do is to elegantly side-step the issue that natural CKA protocols are *unkeyed*, and do not generate shared a shared key I_0 from the initial message T_0. Instead, we model CKA as a *secret key* primitive, where the initial key k_{CKA} effectively generates the first message T_0 of "unkeyed CKA" protocol, but now shared keys $I_1, I_2, \ldots$ get generated right away from subsequent messages $T_1, T_2, \ldots$. In other words, rather than having k only store the root key k_{root}, in our protocol we let it store a tuple $(k_{\mathsf{root}}, k_{\mathsf{CKA}})$, and then use k_{CKA} to solve the syntactic issue of having a special treatment for the first CKA message T_0.

In most actual Signal implementations, the initial shared key k will only contain the value k_{root}, and it is the receiver B who stores several initial CKA messages T_0 (called "one-time prekeys") on the Signal server for new potential senders A. When such A comes along, A would take one such one-time prekey value T_0 from the Signal server, and (optionally) *use it* to generate the initial shared key k_{root} using the X3DH Key Agreement Protocol [?]. This creates slight circularity, and we leave it to the future work to properly model and analyze such generation of the initial key k_{root}.

5.3 Security of the SM Scheme

The proof of the following main theorem can be found in the full version of this paper [?].

Theorem 1. *Assume that*

- CKA *is a* $(t', \Delta_{\mathsf{CKA}}, \varepsilon_{\text{cka}})$*-secure CKA scheme,*
- $\mathsf{FS\text{-}AEAD}$ *is a* $(t', q, \varepsilon_{\text{fs-aead}})$*-secure FS-AEAD scheme, and*
- P *is a* $(t', \varepsilon_{\text{p}})$*-secure PRF-PRNG.*

Then, the SM *construction above is* $(t, q, q_{\mathsf{ep}}, \Delta_{\mathsf{SM}}, \varepsilon)$*-SM-secure for* $t \approx t'$, $\Delta_{\mathsf{SM}} = 2 + \Delta_{\mathsf{CKA}}$, *and*

$$\varepsilon \;\leq\; 2q_{\mathsf{ep}}^2 \cdot (\varepsilon_{\text{cka}} + q \cdot \varepsilon_{\text{fs-aead}} + \varepsilon_{\text{p}}) \;.$$