

Post-Quantum Ratcheting for Signal

Benedikt Auerbach¹, Yevgeniy Dodis², Daniel Jost², Shuichi Katsumata^{1,3}, Thomas Prest¹,
Rolfe Schmidt⁴

¹PQShield

{benedikt.auerbach, shuichi.katsumata, thomas.prest}@pqshield.com

²New York University

dodis@cs.nyu.edu, d.jost@outlook.com

³AIST

⁴Signal Messenger

rolfe@signal.org

June 14, 2025

Abstract

Transitioning existing classical two-party secure messaging protocols to post-quantum protocols has been an active movement in practice in recent years with numerous design choices being explored: Signal Messenger recently deployed the PQXDH protocol for their initial key establishment, and Apple deployed PQ3 which provided post-quantum PCS and FS. However, due to the large communication overhead of post-quantum primitives existing solutions for post-quantum ratcheting are not suited to a setting where bandwidth is limited.

In this work we construct several post-quantum secure-messaging protocols working under strict bandwidth constraints. We evaluate our protocols with respect to meaningful communication patterns which were prepared in discussion with Signal Messenger. Along the way we introduce several new abstractions and uncover untapped optimizations. This work was done in collaboration with Signal as part of their post-quantum migration efforts. The comparison techniques presented here were used by Signal to evaluate candidate protocol updates, and they are bringing a protocol closely related to one we present here into production.

1 Introduction

The Signal protocol, used by Signal, WhatsApp, Google RCS, and Facebook Messenger to protect the communications of billions of people worldwide, has widely been considered to be a benchmark for secure messaging. At its core, it uses the famous *Double Ratchet* protocol [MP16] to provide important security properties, including resilience to compromise captured by *Forward Secrecy* (FS; protecting past messages) and *Post-Compromise Security* (PCS; protecting future messages). This is achieved by ensuring that each message is encrypted with a fresh key, unknown even to the messaging platform. On a technical level, FS is achieved extremely cheaply (and without any interaction) using a stream cipher. In contrast, PCS inherently requires some interaction and public-key cryptography. This is captured by a special protocol, called *Continuous Key Agreement* (CKA) [ACD19], running side-by-side (as “authenticated metadata”) with the actual encryption of the message. Hence, CKA forms the main overhead of modern secure messaging (SM) protocols over archaic encryption-with-a-static-key counterpart.

Post-Quantum Security. Fortunately, in the pre-quantum world, the overhead of CKA is quite minimal. For example, in the famous Double Ratchet [MP16] protocol, it consists of a single 32-byte group element. Moreover, one can build CKA from any key encapsulation mechanism (KEM) with communication complexity equal to the sum of encapsulation key and ciphertext sizes per message [ACD19]. In principle, this means that one can build post-quantum CKA (and, hence, SM [ACD19]) protocols. Unfortunately, post-quantum encapsulation keys and ciphertexts are much longer than those of Diffie-Hellman (DH) KEM. Concretely, using MLKEM (i.e., Kyber) [SAB⁺22] generically would result in sending (1088+1184=2272) bytes compared to 32 bytes, which is 71 times longer than DH!

Indeed, none of the (few) existing post-quantum SM solutions followed this generic approach exactly. For example, Signal Messenger recently deployed the PQXDH [KS23] protocol for their initial key establishment. However, PQXDH only updated the initial protocol handshake, and therefore did not provide any post-quantum PCS (one of the key features of the original Double Ratchet protocol). More recently, Apple deployed PQ3 [App24] which provided post-quantum PCS and FS, whose idealized version was analyzed by [Ste24]. At its core, it effectively applied the generic CKA-to-SM compiler of [ACD19] to both a classically secure ratchet, as well as, a “post quantum” ratchet. However, the post quantum ratchet, which results in long messages, is run approximately every 50 messages (or whenever parties have not sent a fresh MLKEM encapsulation key within a week). While this enables to amortize the large communication cost, such trick may be prohibited when bandwidth is limited. For instance, Signal Messenger aims to limit the overhead of the post-quantum PCS to 40 bytes per message to keep the impact on performance minimal [KAD⁺25]. Even worse, as a party needs to re-send the same CKA message until it hears a response from the other party, a requirement for *immediate decryption*, the same large post-quantum CKA message may be sent over-and-over, negating the benefit of amortization.

1.1 Our contributions

In this work we construct and compare several post quantum secure messaging schemes working under strict bandwidth constraints. Along the way we introduce several new abstractions and optimizations tailored to this setting. Due to page limitations, this work is a shortened version of previous work [DJK⁺25, ADJ⁺25].

Secure messaging from sparse CKA. First, we revisit the CKA to SM compiler from [ACD19] and observe that CKA is not suited to the construction of bandwidth-constrained secure messaging for two reasons. On the one hand, CKA requires the communicating parties to take turns in generating CKA messages, and on the other, it requires a new shared secret be established with every protocol message sent. Note that the latter is not possible, if the underlying public-key primitive has ciphertexts or public keys exceeding the bandwidth limit. Additionally, looking ahead, the former requirement turns out to be too restrictive to capture natural optimizations in the bandwidth constrained setting.

To amend this in Sec. 3.1 we introduce a generalization of CKA, called *Sparse CKA* (SCKA). As with CKA, SCKA produces an ordered sequence of shared keys $I_1, I_2, \dots$. However, the parties are no longer required to take turns in generating protocol messages and the keys are no longer produced with every SCKA message (it’s “sparse” after all). To enable simple key-management at the SM level each key I_t is explicitly associated to its epoch. While we have some subtleties to overcome, in Sec. 3.2 we obtain a compiler from SCKA to SM that is largely based on that of [ACD19]. Importantly, it essentially preserves the bandwidth consumption of the underlying SCKA.

A new method to quantify security. Both SM and SCKA come with a new way of quantifying security due to the following observation. In SM literature, PCS speed (and other security measures, such as FS) is counted in terms of *epochs* [ACD19], where epoch was defined as a change of communication direction between parties. Without bandwidth limitations, this measure had a direct analog in the CKA protocol, as it corresponded to a single (possibly long) CKA message. In turn, security of CKA was naturally counted in terms of individual CKA messages, as each such message produces a fresh CKA key. Turned around, PCS speed of existing CKA-based SM protocols is *semantically* counted in terms of the number of compromised

CKA keys, which happens to be *syntactically* equal to the number of SM epochs it takes to recover from compromise. And since the latter number makes sense for *any* SM protocols, previous work simply adopted counting PCS speed in terms of epochs. In short, without bandwidth constraints, there is a well-defined and easy way to compare SM protocols by simply looking at the underlying CKA. Once bandwidth limitations are in place, however, this equality no longer makes any sense. Looking ahead, the SCKA messages of our bandwidth constrained protocols consist of small “chunks”, where each chunk is not enough to send a full encapsulation key or ciphertext, and parties usually will have many rounds of communications (i.e., “epochs”, as defined before) before a new shared secret is finally established. Thus, simply applying the prior, epoch-based definition of security for SM would lead to a severe mismatch to what users care about in practice, i.e., which sent payload messages are exposed.

This mismatch will have a rather simple solution in this work, where we will define SM security in a way which explicitly looks at the set of exposed messages. Concretely, we require a SM scheme keeps track of the so called *vulnerable message set*, which intuitively corresponds to the messages on which trivial attacks could be mounted would the party be exposed. The security of SCKA is still epoch-based, but with the understanding that an epoch might take more than one message. Additionally to allow for an easy translation from vulnerable SCKA epochs to vulnerable SM messages, we require an SCKA protocol explicitly keeps track of the set of vulnerable epochs.

On hybrid security. We note that in practice one might require post-quantum extensions of Signal should provide what is called hybrid security by meaningfully combining the DH-based Double Ratchet with a post-quantum variant. For ease of exposition in this work we concentrate on the simpler SCKA to SM case and only provide some intuition on how to combine two public-key ratchets (i.e., a CKA and a SCKA) for hybrid security. Note that due to the use of SCKA in hybrid secure a construction the classical and post-quantum public key ratchets are no longer synchronized, and there are situations where one ratchet moves forward and the other does not. As the result, opposed to the construction of [ACD19], one can no longer use a single “root key” to be combined with the new key material whenever one of the ratchets moves forward. This can be resolved, however, by having separate root keys for both public-key ratchet, carefully deriving two separate message keys, and finally combining those message keys when encrypting the payload. For the full details of such a construction we defer to [DJK⁺25].

Bandwidth Constrained SCKA via Chunking. With the construction of bandwidth constrained SM from bandwidth constrained SCKA in place, we can concentrate on building the latter. First, in Sec. 4.1, we observe that we can transform any CKA protocol into a SCKA satisfying any given bandwidth constraint by means of *chunking*: The sending party after having computed a large CKA message, instead of sending it in whole, breaks it into small chunks using an erasure code and sends these one-by-one as their SCKA messages. After the other party has received sufficiently many chunks they are able to recover and process the CKA message and now can take over the role of ‘sender’.

We note that due to the use of erasure codes every message satisfies the bandwidth constraints and, importantly, enables immediate decryption even if messages are dropped or reordered without having to repeat large SCKA messages.

Note that when making use of chunking both a substantially larger amount of messages has to be transmitted between the communicating parties to advance the ratchet and that parties have to keep cryptographic material in state for longer compared to the underlying non-chunked SCKA. As a consequence the chunked scheme achieves PCS and FS at a slower rate compared to the underlying CKA.

Optimizations. To partially amend the issue discussed above, we investigate two orthogonal optimizations for instantiations of SCKA that allow us to improve the rate at which PCS and FS are achieved.

In Sec. 4.2 we define and construct *Ratcheted* KEM (RKEM), a cryptographic primitive to be used as a more efficient replacement for KEMs in the KEM-based CKA construction of [ACD19]. On a high level, RKEM abstracts KEM properties in a way which allows a freshly sampled ciphertext also be used as part of a different KEM public key. In essence, this is precisely why the original DH-based CKA of Signal saved

a factor of 2 in communication, when compared to the generic KEM-based DH construction. And this is why RKEM is precisely fitted for the use inside a CKA. We provide a construction of a RKEM from the hint-MLWE assumption that we call *Katana*. We notice that the construction of CKA from *Katana* is closely related to what previous works called “optimized” lattice-based CKA [ACD19, LKS23], but instantiated with a carefully chosen variant of Kyber. However, we identify a critical flaw in the previous analyses of this “optimized” KEM, and non-trivially fix it with a novel proof relying on the recently introduced hint-MLWE assumption [KLSS23, EEN⁺24].

The generic construction of [ACD19] building SM from CKA applied to Kyber at security level 192 bits, gives CKA messages of size 2272 bytes. In contrast, for the same security level an instantiation from *Katana* uses messages of size 1416 bytes, saving over 37%. As a consequence, when applying chunking to transform the CKA into a bandwidth constrained SCKA substantially fewer chunks have to be sent to transmit a CKA message.

As a second optimization, in Sec. 4.3 we consider SCKA protocols making use of a technique we denote ‘opportunistic sending’. Abstractly, imagine a sequential protocol where Alice sends a long message α , and Bob responds with a long message $\beta = (\beta_0, \beta_1)$, where β_0 is independent of α . As we show, this setting is very common for various post-quantum primitives, such as KEMs. With opportunistic sending, Bob will start sending chunks of β_0 *concurrently* with receiving chunks of α (instead of waiting for receiving α in full). In a balanced communication scenario, this clearly speeds up the exchange of “useful information”, potentially resulting in faster PCS.

Instantiations and Experimental evaluation. The ideas discussed above leave us with a large design space for bandwidth-constrained post-quantum SM protocols. As candidates to be considered for adaption in practice, in Sec. 5 we consider two KEM-based and an RKEM-based protocol making use of opportunistic sending, as well as their ‘non-opportunistic’ counterparts, thus ending up with 6 protocols to compare. We instantiate our protocols with MLKEM or *Katana*, respectively.

We observe that the number of exposed messages (formally *vulnerable message sets*) critically depends on the communication scenario considered. Further, most of the SM protocols we consider are *formally incomparable*, and the selection of such “best” is application-specific (and impossible in general).

As a consequence, we take a pragmatic approach and compare our protocol candidates experimentally. More precisely, in Sec. 6 we evaluate our protocols with respect to several meaningful communication patterns which were prepared in discussion with Signal Messenger. In particular, by distinguishing between primary and linked devices our models aim to reflect two polar user behaviors on the Signal app. We highlight some findings from the simulation results. While there does not seem to present a clear winner with respect to all investigated messaging patterns, protocols with opportunistic sending appear better for relatively balanced communication. Further, protocols making use of unidirectional key generation or advanced cryptographic primitives like RKEM perform well in most of the considered settings while also exhibiting robust behavior across a wide range of messaging patterns.

Signal’s Deployment. This work was done in collaboration with Signal Messenger as part of their design process for a post-quantum update to the Double Ratchet [MP16] protocol. Signal used the evaluation framework and user behavior models presented here to compare candidate protocols, and they are in the process of deploying an MLKEM-based protocol [Mesb] closely related to one we present here.

2 Preliminaries

We recall secure messaging protocols below. Other cryptographic primitives relevant to our constructions can be found in App. A.

2.1 Secure Messaging

A secure messaging protocol consists of the following PPT algorithms: A (two-party) secure messaging protocol enables A and B to send and receive messages. We follow the standard definition of [ACD19], allowing for *immediate decryption*, i.e. the out-of-order receiving of messages. This property is captured in the receive algorithm, where in addition to the plaintext message, it outputs the so-called epoch and period of a message allowing them to be ordered.

Definition 1 *A secure messaging protocol consists of the following PPT algorithms:*

$\text{SM-Init-KeyGen}(1^\lambda) \rightarrow \mathbf{l}_K$: *On input the security parameter 1^λ , it outputs an initial key $\mathbf{l}_K$.*

$\text{SM-Init-A}(\mathbf{l}_K) \rightarrow \mathbf{st}_A$: *On input an initial key $\mathbf{l}_K$, it outputs an initial state $\mathbf{st}_A$ for party A.*

$\text{SM-Send-A}(\mathbf{st}_A, M) \rightarrow (\mathbf{ct}, \mathbf{st}'_A)$: *On input a state $\mathbf{st}$ of party A and a message M , it outputs a ciphertext $\mathbf{ct}$ and an updated state $\mathbf{st}_A$.*

$\text{SM-Rec-A}(\mathbf{st}_A, \mathbf{ct}) \rightarrow (M', \mathbf{t}', i', \mathbf{st}'_A)$: *On input a state $\mathbf{st}$ and a ciphertext $\mathbf{ct}$, it outputs a message M' , an epoch $\mathbf{t}'$, a period i' , and an updated state $\mathbf{st}'_A$. This algorithm is assumed to be deterministic.*

We define algorithms SM-Init-B , SM-Send-B , and SM-Rec-B analogously with roles of parties A and B swapped.

Correctness and security. We defer the formal definition of correctness and security via a security game to App. B and here only provide an intuitive description. As explained in the introduction, we deviate from the prior security definition in [ACD19] by using a finer-grained notion to define trivial attacks. Concretely, we assume that each party P's state has an associated set $\mathbf{st}_P.\text{vuln}$ of messages that will be compromised in case $\mathbf{st}_P$ is leaked to the adversary; we call this a *vulnerable message set*. Each vulnerable message $(P', \mathbf{t}, i) \in \mathbf{st}_P.\text{vuln}$ is identified by its sender $P' \in \{A, B\}$, its epoch $\mathbf{t}$, and its period i . For an execution of the security game we define the vulnerable message *sets* as the set of vulnerable message set $\mathbf{st}_P.\text{vuln}$ for all exposed states $\mathbf{st}_P$ during the game. We then say the secure messaging protocol is secure if the adversary is not able to obtain any information on any message *not* contained in the vulnerable message sets. Intuitively this means that the vulnerable message set contains the messages on which trivial attacks could be mounted would the party be exposed. Other than this modification, the correctness and security guarantees are defined identically to [ACD19].

3 Bandwidth limited secure Messaging from SCKA

3.1 Sparse Continuous Key-Agreement

Continuous key agreement (CKA) was introduced in [ACD19] to abstract the ongoing Diffie-Hellman key exchange of the Double Ratchet protocol, effectively moving the cryptographic complexity of the SM protocol to the CKA layer. Their definition of CKA assumed a restricted setting where two parties take turn in sending a message that each establishes a fresh key. As discussed in the introduction, while this abstraction is reasonable in contexts where each of the classical messages are small, this is no longer the case when we consider a bandwidth limited setting.

We therefore introduce *sparse* CKA (SCKA). At its core, SCKA forgoes the “ping-pong” sending pattern of CKA and instead allows the protocol to optionally output a key I_A for epoch $\mathbf{t}_{I_A}$ whenever ready. Put differently, we allow it outputting $I_A = \perp$, indicating, for instance, that we are still in the process of sending a part of the large post-quantum message. SCKA internalizes the management of keys by having the sender learn the epoch $\mathbf{t}_{I_A}^{\text{snd}}$ of the latest key that is safe to use for the messaging protocol, i.e., for which epoch the receiver is guaranteed to know all keys up to that epoch. Conversely, the recipient of an SCKA message learns the epoch $\mathbf{t}_{I_B}^{\text{rcv}} = \mathbf{t}_{I_A}^{\text{snd}}$ the sender used. As such, SCKA moves the protocol specific complexity of the key management from the SM protocol back into the key agreement abstraction. We emphasize that SCKA is a

generalization of CKA. Indeed, any CKA protocol can be seen as a SCKA by making the implicit information on the current epoch explicit.

Definition 2 A sparse continuous key agreement (SCKA) protocol with initial key space $\mathcal{I}_{\text{CKA}}$, and key space $\mathcal{K}$ consists of the following PPT algorithms:

$\text{CKA-Init-KeyGen}(1^\lambda) \rightarrow l_{\text{CKA}}$: On input the security parameter 1^λ , it outputs an initial key $l_{\text{CKA}} \in \mathcal{I}_{\text{CKA}}$.

$\text{CKA-Init-A}(l_{\text{CKA}}) \rightarrow \bar{\text{st}}_A$: On input an initial key $l_{\text{CKA}} \in \mathcal{I}_{\text{CKA}}$, it outputs an initial state $\bar{\text{st}}_A$ for party A.

$\text{CKA-Send-A}(\bar{\text{st}}_A) \rightarrow ((t_{I_A}, I_A), \rho, t_A^{\text{snd}}, \bar{\text{st}}'_A)$: On input a state $\bar{\text{st}}_A$ of party A, it outputs a pair $(t_{I_A}, I_A) \in (\mathbb{N} \times \mathcal{K}) \cup \{(\perp, \perp)\}$ of epoch counter and key, a message ρ , a sending epoch t_A^{snd} , and an updated state $\bar{\text{st}}'_A$.

$\text{CKA-Rec-A}(\bar{\text{st}}_A, \rho) \rightarrow ((t_{I_B}, I_B), t_A^{\text{rcv}}, \bar{\text{st}}'_A)$: On input a state $\bar{\text{st}}_A$ of party A and a message ρ , it outputs a pair $(t_{I_B}, I_B) \in (\mathbb{N} \times \mathcal{K}) \cup \{(\perp, \perp)\}$ of epoch counter and key, a receiving epoch t_A^{rcv} , and an updated state $\bar{\text{st}}'_A$. This algorithm is assumed to be deterministic.

Above, we define algorithms CKA-Init-B , CKA-Send-B , and CKA-Rec-B analogously with roles of A and B swapped.

Correctness and security. This is defined via an experiment depicted in Fig. 1, following the basic structure to those of CKA's [ACD19]. In line with the CKA security experiment of [ACD19], we consider *passive* adversaries only, which can delay and reorder but not modify SCKA messages.¹

The main difference compared to the security experiment of CKA is two-folds: As a single message may not contain sufficient information to output a key, we only mandate correctness when $(t_{I_P}, I_P) \neq (\perp, \perp)$ during Send-P and $(t_{I_P}, I_P) \neq (\perp, \perp)$ during Receive-P . Furthermore, since SCKA departs from the ping-pong sending pattern of CKA, we accommodate adversarial network behavior. Concretely, Receive-P takes as input the n_P -th message output by Send-P — previously, Receive-P took no input and the messages were assumed to arrive in the order as Send-P output them.

Fine-grained security. We also revisit how to model trivial distinguishing attacks. We assume a SCKA protocol state exposes the *vulnerable epoch set*, denoted as $\bar{\text{st}}_P.\text{vuln}$. Intuitively, this captures the list of epochs for which trivial attacks can be mounted, once corrupted the state $\bar{\text{st}}_P$. As this is protocol dependent, concrete examples are provided in Sec. 5 for each of our protocols. We further define the vulnerable epoch *sets* as the set of vulnerable epoch set $\bar{\text{st}}_P.\text{vuln}$ for all exposed states $\bar{\text{st}}_P$. We then say a key at epoch t is a *valid* challenge iff t is not in the vulnerable epoch sets. It is easy to see that this definition is a more fine-grained notion compared to the previously used forward-secrecy (FS) Δ_{FS} and post-compromise security (PCS) Δ_{PCS} notions [ACD19]. Indeed, we can define Δ_{PCS} and Δ_{FS} to be the smallest numbers such that for any valid attacker interacting with the game, when calling Corr-P for either $P \in \{A, B\}$,

- $t_P^{\text{cur}} + \Delta_{\text{PCS}} > \max(\bar{\text{st}}_P.\text{vuln})$, upon corruption, i.e., no corruption must expose epochs more than Δ_{PCS} into the future.
- $t_A^{\text{cur}} - \Delta_{\text{FS}} \geq \min(\bar{\text{st}}_P.\text{vuln})$, i.e., no corruption must expose epochs more than Δ_{FS} into the past. In particular, for a forward-secure protocol with $\Delta_{\text{FS}} = 0$ even the current epoch is no longer exposed by the protocol's state.

We highlight that the benefit of using vulnerable epochs as opposed to Δ_{FS} and Δ_{PCS} becomes much clearer when we look at the messaging layer. As explained in the introduction, this is because Δ_{FS} and Δ_{PCS} lose practical meanings when considering *bandwidth limited* SM protocols.

¹Analogous to the Double Ratchet protocol, (S)CKA messages are authenticated as part of the messaging layer.

Game $_{\mathcal{A}}^{\text{SCKA}}(1^\lambda)$	Send-P(rleak)	Receive-P(n)
1: $b \xleftarrow{\$} \{0, 1\}$ 2: $\text{Key}[\cdot, \cdot] \leftarrow \perp; \text{Msg}[\cdot, \cdot] \leftarrow \perp$ 3: $\text{Exposed} \leftarrow \emptyset; \text{Challenged} \leftarrow \emptyset$ 4: $\text{l}_{\text{CKA}} \xleftarrow{\$} \text{CKA-Init-KeyGen}(1^\lambda)$ 5: for $P \in \{A, B\}$ 6: $\text{st}_P^{\text{CKA}} \xleftarrow{\$} \text{CKA-Init-P}(\text{l}_{\text{CKA}})$ 7: $\hat{t}_P \leftarrow 0$ 8: $b' \xleftarrow{\$} \mathcal{A}(1^\lambda)^{\mathcal{O}()}$ 9: return $\llbracket b = b' \rrbracket$	1: if $\llbracket \text{rleak} \rrbracket$ then // Leak randomness 2: $\text{rand} \xleftarrow{\$} \mathcal{R}$ 3: $\text{vuln} \leftarrow \text{st}_P^{\text{CKA}}.\text{vuln}$ 4: $((t_{I_P}, I_P), \rho, t_P^{\text{snd}}, \text{st}_P^{\text{CKA}})$ 5: $\xleftarrow{\$} \text{CKA-Send-P}(\text{st}_P^{\text{CKA}}, \text{rand})$ 6: // newly vulnerable epochs 7: $\text{vuln}' \leftarrow \text{st}_P^{\text{CKA}}.\text{vuln} \setminus \text{vuln}$ 8: req $\llbracket \text{vuln}' \cap \text{Challenged} = \emptyset \rrbracket$ 9: $\text{Exposed} \leftarrow \text{vuln}'$ 10: else // Secure randomness 11: $\text{rand} \leftarrow \perp$ 12: $((t_{I_P}, I_P), \rho, t_P^{\text{snd}}, \text{st}_P^{\text{CKA}})$ 13: $\xleftarrow{\$} \text{CKA-Send-P}(\text{st}_P^{\text{CKA}})$ 14: $\text{assert } \llbracket t_P^{\text{snd}} \geq t_P^{\text{cur}} \rrbracket$ 15: $t_P^{\text{cur}} \leftarrow t_P^{\text{snd}}$ 16: if $\llbracket (t_{I_P}, I_P) \neq (\perp, \perp) \rrbracket$ then 17: $\text{assert } \llbracket \text{Key}[P, t_{I_P}] = \perp \rrbracket$ 18: $\text{assert } \llbracket \text{Key}[\bar{P}, t_{I_P}] \in \{I_P, \perp\} \rrbracket$ 19: $\text{Key}[P, t_{I_P}] \leftarrow I_P$ 20: $\text{assert } \llbracket \forall t \leq t_P^{\text{snd}} : \text{Key}[P, t] \neq \perp \rrbracket$ 21: $\text{Msg}[P, ++n_P] \leftarrow (\rho, t_P^{\text{snd}})$ 22: return $(t_P^{\text{snd}}, t_{I_P}, \rho, \text{rand})$	1: req $\llbracket \text{Msg}[\bar{P}, n] \neq \perp \rrbracket$ 2: $(\rho, t_{\bar{P}}^{\text{snd}}) \leftarrow \text{Msg}[\bar{P}, n]$ 3: $((t_{I_{\bar{P}}}, I_{\bar{P}}), t_{\bar{P}}^{\text{rcv}}, \text{st}_{\bar{P}}^{\text{CKA}})$ 4: $\leftarrow \text{CKA-Rec-P}(\text{st}_{\bar{P}}^{\text{CKA}}, \rho)$ 5: $\text{assert } \llbracket t_{\bar{P}}^{\text{rcv}} = t_{\bar{P}}^{\text{snd}} \rrbracket$ 6: $t_{\bar{P}}^{\text{cur}} \leftarrow \max(t_{\bar{P}}^{\text{cur}}, t_{\bar{P}}^{\text{rcv}})$ 7: if $\llbracket (t_{I_{\bar{P}}}, I_{\bar{P}}) \neq (\perp, \perp) \rrbracket$ then 8: $\text{assert } \llbracket \text{Key}[P, t_{I_{\bar{P}}}] = \perp \rrbracket$ 9: $\text{assert } \llbracket \text{Key}[\bar{P}, t_{I_{\bar{P}}}] \in \{I_{\bar{P}}, \perp\} \rrbracket$ 10: $\text{Key}[P, t_{I_{\bar{P}}}] \leftarrow I_{\bar{P}}$ 11: $\text{assert } \llbracket \forall t \leq t_{\bar{P}}^{\text{cur}} : \text{Key}[P, t] \neq \perp \rrbracket$ 12: return $(t_{\bar{P}}^{\text{rcv}}, t_{I_{\bar{P}}})$
Chall(t)		Corr-P()
1: req $\llbracket t \notin \text{Exposed} \cup \text{Challenged} \rrbracket$ 2: if $\llbracket \text{Key}[A, t] \neq \perp \rrbracket$ 3: then $K \leftarrow \text{Key}[A, t]$ 4: else $K \leftarrow \text{Key}[B, t]$ 5: req $\llbracket K \neq \perp \rrbracket$ 6: if $\llbracket b = 1 \rrbracket$ then 7: $K \xleftarrow{\$} \mathcal{K}$ // Replace with random key 8: $\text{Challenged} \leftarrow \perp$ 9: return K		1: req $\llbracket \text{st}_P^{\text{CKA}}.\text{vuln} \cap \text{Challenged} = \emptyset \rrbracket$ 2: $\text{Exposed} \leftarrow \text{st}_P^{\text{CKA}}.\text{vuln}$ 3: return st_P^{CKA}

Figure 1: Correctness and security games for sparse continuous key agreement (SCKA) protocol. $\mathcal{O}()$ denotes the set of oracles $\{\text{Send-P}(), \text{Receive-P}(), \text{Chall}(), \text{Corr-P}()\}$. With an overload of notation, in the above P denotes the variable that can be either A or B. For instance, it is understood that $\mathcal{A}$ is given oracle access to both Send-A and Send-B with the shorthand Send-P .

3.2 Secure Messaging from SCKA

We now present a SM protocol built from SCKA, authenticated encryption AEAD (see App. A.3), and key-derivation function KDF (see App. A.4). The scheme's description can be found in Fig. 2. Our generic construction is almost identical to the simplistic construction by [ACD19] using CKA and importantly essentially preserves the bandwidth of the underlying SCKA. The key differences are that, unlike CKA, SCKA allows for the creation of keys that are not yet usable, and thus require slight key management, and that each party can act as both a sender and receiver for the same epoch. Moreover, we use a protocol specific slack Δ_{Slack} to prevent one party from creating too many keys ahead of the other. Namely, we assume whenever CKA.Send-P outputs sending epoch t_P^{snd} , the other party $\bar{P}$ already processed a message that resulted in $t_{\bar{P}}^{\text{rcv}} \geq t_P^{\text{snd}} - \Delta_{\text{Slack}}$. Looking ahead, for all the protocols we consider, we have $\Delta_{\text{Slack}} \leq 2$.

In more detail, the protocol uses the SCKA as an asymmetric ratcheting layer. The resulting keys are then mixed into the key chain using Mixin-CKA-Keys . The key difference to [ACD19] is that each epoch t has independent sending and receiving key chains KS^t and KR^t , respectively, associated.

When A sends a message, she may not be able to use the latest sending key chain yet. Instead, she uses the one of epoch t^{snd} output by the SCKA so that she is guaranteed that B will have the necessary key to decrypt. B learns the epoch t^{rcv} to decrypt as part of the CKA-Rec-B . Old sending key chains can be discarded as part of SM-Send-A as soon as A moves past that epoch. A must however store counters recording the amount of messages sent during the last Δ_{Slack} many sending epochs. Using these counters, B can then finalize old receiving epochs as part of SM-Rec-B , deriving keys for all pending out-of-order messages. The

SM-Init-KeyGen(1^λ) 1 : $I_{CKA} \xleftarrow{\$} \text{CKA-Init-KeyGen}(1^\lambda)$ 2 : $(K_{root}, K_{CKA}) \xleftarrow{\$} \mathcal{K}_{root} \times \mathcal{K}_{CKA}$ 3 : return $I_K := (I_{CKA}, K_{root}, K_{CKA})$	SM-Send-A(M) 1 : $((t_{IA}, I_A), \rho, t, \overline{st}_A) \xleftarrow{\$} \text{CKA-Send-A}(\overline{st}_A)$ 2 : // Mix CKA key(s) into key chain 3 : if $\llbracket (t_{IA}, I_A) \neq \perp \rrbracket$ then 4 : $\text{Mixin-CKA-Keys}(t_{IA}, I_A)$ 5 : // Bookkeeping: discard old sending state 6 : if $\llbracket t > t^{snd} \rrbracket$ then 7 : for $t' = t^{snd}, \dots, (t-1)$ do 8 : $KS[t'] \leftarrow \perp$ 9 : for $t' = t^{snd}, \dots, (t - \Delta_{Slack} - 1)$ do 10 : $iS[t'] \leftarrow \perp$ 11 : $t^{snd} \leftarrow t$ 12 : // Symmetric ratchet 13 : $(KS[t^{snd}], KS_{aead}) \leftarrow \text{KDF}_2(KS[t^{snd}])$ 14 : $iS[t^{snd}] += 1$ 15 : // Encrypt 16 : $h \leftarrow (\rho, iS)$ 17 : $e \xleftarrow{\$} \text{AEAD.Enc}(KS_{aead}, h, M)$ 18 : return $ct := (h, e)$	SM-Rec-A(ct) 1 : parse $(h, e) \leftarrow ct$ 2 : parse $(\rho, iR') \leftarrow h$ 3 : $((t_{IB}, I_B), t, \overline{st}_A^{CKA}) \leftarrow \text{CKA-Rec-A}(\overline{st}_A, \rho)$ 4 : if $\llbracket t \geq t^{rcv} \rrbracket$ then // skip for out-of-order 5 : // Update CKA state 6 : $\overline{st}_A \leftarrow \overline{st}_A^{CKA}$ 7 : // Mix CKA key(s) into key chain 8 : if $\llbracket (t_{IB}, I_B) \neq \perp \rrbracket$ then 9 : $\text{Mixin-CKA-Keys}(t_{IB}, I_B)$ 10 : // Bookkeeping: expand old receiving epochs 11 : if $\llbracket t > t^{rcv} \rrbracket$ then 12 : for $t' = t^{rcv}, \dots, (t-1)$ do 13 : $\text{Expand-Chain}(t', iR')$ 14 : $iR[t'] \leftarrow \perp$; $KR[t'] \leftarrow \perp$ 15 : $t^{rcv} \leftarrow t$; $iR[t^{rcv}] \leftarrow 0$ 16 : // Symmetric ratchet 17 : if $\llbracket t = t^{rcv} \rrbracket$ then 18 : $\text{Expand-Chain}(t^{rcv}, iR')$ 19 : // Decrypt 20 : $KR_{aead} \leftarrow \text{StoredKeys}[t, iR'[t]]$ 21 : $M \leftarrow \text{AEAD.Dec}(KR_{aead}, h, e)$ 22 : return M
SM-Init-A(I_K) 1 : parse $(I_{CKA}, K_{root}, K_{CKA}) \leftarrow I_K$ 2 : $\overline{st}_A \xleftarrow{\$} \text{CKA-Init-A}(I_{CKA})$ 3 : $(K_{CKA}[\cdot], KS[\cdot], KR[\cdot], iS[\cdot], iR[\cdot], \text{StoredKeys}[\cdot, \cdot]) \leftarrow \perp$ 4 : $(t^{cur}, t^{snd}, t^{rcv}, iS[0], iR[0]) \leftarrow 0$ 5 : $(K_{root}, KS[0], KR[0]) \leftarrow \text{KDF}_1(K_{root}, K_{CKA})$	Mixin-CKA-Keys(t', K') 1 : $K_{CKA}[t'] \leftarrow K'$ 2 : while $\llbracket K_{CKA}[t^{cur} + 1] \neq \perp \rrbracket$ do 3 : $t^{cur} += 1$ 4 : $(K_{root}, KS[t^{cur}], KR[t^{cur}]) \leftarrow \text{KDF}_1(K_{root}, K_{CKA}[t^{cur}])$ 5 : $K_{CKA}[t^{cur}] \leftarrow \perp$	Expand-Chain(t', iR') 1 : while $\llbracket iR[t'] < iR'[t'] \rrbracket$ do 2 : $(KR[t'], KR_{aead}) \leftarrow \text{KDF}_2(KR[t'])$ 3 : $iR[t'] += 1$ 4 : $\text{StoredKeys}[t', iR[t']] \leftarrow KR_{aead}$

Figure 2: An SM protocol based on an SCKA protocol with slack Δ_{Slack} . The algorithms for B are analogous, except for the order of the sending and receiving chains swapped, e.g., line 4 of Mixin-CKA-Keys becoming $(K_{root}, KR[t^{cur}], KS[t^{cur}]) \leftarrow \text{KDF}_1(K_{root}, K_{CKA}[t^{cur}])$.

SCKA message ρ and metadata are authenticated as part of the AEAD encryption.

The correctness and security almost immediately follows from the proof given by [ACD19], replacing the CKA with a SCKA. The only difference is that, for the reduction, we must relate the set of trivial attacks breaking the SM protocol to those of the SCKA — previously, this was implicit as both the SM protocol and CKA used the same forward-secrecy (FS) Δ_{FS} and post-compromise security (PCS) Δ_{PCS} variables to define the set of trivial attacks.

Concretely, using our new terminologies, we must understand how the *vulnerable epoch sets* allowing to break SCKA translates to the *vulnerable message sets* allowing to break the SM protocol. For our SM protocol based on SCKA, this translation is very natural. Essentially, it is any message that was sent or received using AEAD keys generated in epochs included in the vulnerable epoch sets. It further contains additional messages as a result of the key management that the SM protocol has to perform on top of the SCKA layer. More formally, we have the following.

Definition 3 Consider an SCKA with state st_P^{CKA} for $P \in \{A, B\}$. Given vulnerable epoch set $st_P^{CKA}.vuln$, the vulnerable message set $st_P^{SM}.vuln$ for our SM protocol in Fig. 2 is defined as follows:

- For vulnerable SCKA epochs $t \in st_P^{CKA}.vuln$ all messages to be sent (P, t, i) and to be received $(\bar{P}, t, i)$, where $i \in \mathbb{N}$.
- For the current sending epoch $st_P^{SM}.t^{snd}$, all future messages $(P, st_P^{SM}.t^{snd}, i)$, where $i > st_P^{SM}.i^{snd}$, and analogously $(\bar{P}, st_P^{SM}.t^{rcv}, i)$ for all future messages of the receiving epoch.
- For future sending epoch $t > st_P^{SM}.t^{snd}$ for which SCKA already produced a key, all messages (P, t, i) to be sent by P, where $i \in \mathbb{N}$. Analogously, for any such epoch $t > st_P^{SM}.t^{rcv}$ not yet used for receiving, all messages $(\bar{P}, t, i)$ sent by the other party $\bar{P}$.

- Any pending out-of-order message not yet received. That is, $(\bar{P}, t, i)$ for any $(t, i) < (t^{\text{rcv}}, i^{\text{rcv}})$ such that $\bar{P}$ has sent the corresponding message but Receive-P has not output a message for this epoch and period.²

With this, we can formally state the security of our generic SM protocol from SCKA. For the theorem's proof we defer to App. C.

Theorem 1 *Our SM protocol in Fig. 2 is secure w.r.t the vulnerable message sets in Definition 3, if the underlying SCKA is secure w.r.t its vulnerable epoch sets, the AEAD is CCA-secure and unforgeable, and the KDF is PRF-PRNG secure.*

4 Constructing Bandwidth-Limited SCKA

By Sec. 3.2 bandwidth-limited SM can be constructed from bandwidth-limited SCKA, allowing us to focus on the latter.

4.1 Chunking

We observe that we can compile any CKA into a *sparse* CKA satisfying a predetermined bandwidth constraint by breaking protocol messages into small chunks that are sent one-by-one. While one party is sending chunks of their CKA message, the other does not send (chunks of) cryptographic material, instead only acknowledging how many of the other party's chunks they have received so far. To retain immediate decryption, i.e., to ensure functioning of the protocol even if individual SCKA messages are dropped, an *erasure code* (Encode, Decode) (see App. A) is used.

More formally, the initialization of SCKA simply runs the corresponding algorithms of CKA. The first call to CKA-Send-A generates a protocol message ρ_A and epoch key I using the corresponding algorithm of CKA. A initializes a counter $i_{\text{ch}} \leftarrow 1$, and uses an erasure code to generate a chunk $\text{ch} \leftarrow \text{Encode}(\rho_A, i_{\text{ch}})$ that serves as the SCKA message. Subsequent calls to CKA-Send-A increment i and return further chunks $\text{ch} \leftarrow \text{Encode}(\rho_A, i_{\text{ch}})$ of ρ_A .

During this phase CKA-Send-B does not produce any chunks, its protocol message being left empty except for an acknowledgment indicating whether ρ_A was fully transmitted. CKA-Rec-B receiving the chunks ch stores them in a list L_{ch} until it received sufficiently many to recover $\rho_A \leftarrow \text{Decode}(L_{\text{ch}})$. At this point, using its CKA counterpart, it processes ρ_A to recover the epoch key I . Now the parties' roles reverse, the first call to CKA-Send-B generating protocol message ρ_B , setting $i \leftarrow 1$, and starting to send chunks $\text{ch} \leftarrow \text{Encode}(\rho_B, i_{\text{ch}})$.

Impact on PCS. Note that with the approach described above substantially more SCKA messages have to be sent in order to achieve FS and PCS compared to the underlying CKA. Similarly, the parties have to store cryptographic material in state for larger amounts of time, increasing the impact of being compromised.

To partially amend this we explore two orthogonal optimizations in the following. The first introduces *ratcheting KEMs*, a cryptographic primitive that can be used instead to replace KEMs for certain CKA constructions and requires fewer chunks per CKA messages to be sent, as its combined public key and ciphertext size substantially improves on the one of comparable KEMs.

The second, *opportunistic sending*, aims to improve on the issue that under the simple chunking methodology described above the party currently receiving chunks essentially sits idle by not sending any cryptographic material.

4.2 Ratcheting KEMs

On a high level, RKEM abstracts KEM properties in a way which allows a freshly sampled ciphertext also be used as part of a different KEM public key. In essence, this is precisely why the original DH-based CKA of

²In the protocol, P learns the number of messages $\bar{P}$ sent for prior epochs when advancing the epoch. Therefore, this is purely a function of st_P^{SM} .

Signal saved a factor of 2 in communication, when compared to the generic KEM-based DH construction. And this is why RKEM is precisely fitted for the use inside a CKA. Looking ahead, one of our candidate SCKA constructions will be based on RKEM.

Definition 4 (RKEM) A (forward-secure) ratcheting KEM (RKEM) with ratcheting key spaces $\mathcal{RK}_P$ and $\widehat{\mathcal{RK}}_P$ for parties $P \in \{A, B\}$ consists of the following PPT algorithms:

$\text{RSetup}(1^\lambda) \rightarrow \text{par}$: On input the security parameter 1^λ , it outputs a public parameter par .

$\text{RKeyGen-P}(\text{par}, \text{mode}) \rightarrow (\text{ek}_P, \text{dk}_P)$: On input par and mode , it outputs encapsulation and decapsulation keys $(\text{ek}_P, \text{dk}_P) \in \mathcal{RK}_P$ if $\text{mode} = \perp$ and $(\text{ek}_P, \text{dk}_P) \in \widehat{\mathcal{RK}}_P$ if $\text{mode} = \text{updated}$.

$\text{REnc-A}(\widehat{\text{ek}}_B, \text{dk}_A) \rightarrow (\text{ct}_A, I, \widehat{\text{dk}}_A)$: It takes as input an updated encapsulation key $\widehat{\text{ek}}_B$ for party B and a decapsulation key for party A, and outputs a ciphertext ct_A , a shared key I , and an updated decapsulation key $\widehat{\text{dk}}_A$.

$\text{RDec-B}(\widehat{\text{dk}}_B, \text{ct}_A, \text{ek}_A) \rightarrow (I, \widehat{\text{ek}}_A)$: It takes as input an updated decapsulation key $\widehat{\text{dk}}_B$ for party B, a ciphertext ct_A and an encapsulation key generated by party A, and outputs a shared key I and an updated encapsulation key $\widehat{\text{ek}}_A$.

Algorithms REnc-B , RDec-A are defined analogously to their counterparts.

We provide intuition on correctness and security of RKEMs and defer their formal definitions to App. B.2. Correctness comes in two flavors. The first requires that for an updated key-pair $(\widehat{\text{ek}}_B, \widehat{\text{dk}}_B)$ if $(\text{ct}_A, I, \widehat{\text{dk}}_A) \xleftarrow{\$} \text{REnc-A}(\widehat{\text{ek}}_B, \text{dk}_A)$ then decapsulation $(K', \widehat{\text{ek}}_A) \xleftarrow{\$} \text{RDec-B}(\widehat{\text{dk}}_B, \text{ct}_A, \text{ek}_A)$ correctly recovers I . The second property essentially requires that the updated keys $\widehat{\text{dk}}_A$ and $\widehat{\text{ek}}_A$ are compatible with each other.

Regarding security we require FS-IND-CPA stating that even given access to $\widehat{\text{ek}}_B$, ct_A , $\widehat{\text{ek}}_A$, and $\widehat{\text{dk}}_A$ no information on I is leaked. Intuitively, this matches IND-CPA security of KEMs with the additional requirement that $\widehat{\text{dk}}_A$ provides forward secrecy, i.e., after updating the decapsulation key can no longer be used to derive I . Further, we require *ratchet simulatability*. It essentially states that, on one hand, updated keys $(\widehat{\text{ek}}_P, \widehat{\text{dk}}_P)$ can be simulated given access to only $(\text{ek}_P, \text{dk}_P)$ but not the additional inputs required by REnc-P and RDec-P . On the other hand, it requires that a ciphertext ct_P together with user $\tilde{P}$'s decapsulation key does not leak any information on P 's decapsulation key.

Instantiation from lattices. In theory, instantiating RKEMs from lattices is trivial, as standard KEMs are special cases of RKEMs. Indeed, the CKAs built from such RKEMs are exactly the generic construction of CKA based on KEMs by [ACD19]. The true strength of RKEM lies in enabling a more efficient CKA construction, like the Double Ratchet protocol used in Signal. Assume Alice holds $a \in \mathbb{Z}_p$ and Bob holds $g^a \in \mathbb{G}$. In Signal, Bob samples $b \xleftarrow{\$} \mathbb{Z}_p$ and sends g^b to Alice. The shared key K is then updated by mixing g^{ab} into K , *ratcheting* the state forward. Importantly, g^b holds two purposes: it acts as an “encryption/ciphertext” for the Diffie-Hellman key exchange while also serving to be a new “public key” for the next ratchet (i.e., Alice will generate $a' \xleftarrow{\$} \mathbb{Z}_p$ and update the state by $g^{a'b}$). While this reusing of g^b for two purposes has an immediate benefit on efficiency, one downside compared to the KEM-based construction is that it achieves a weaker FS guarantee. Recently, [BFG⁺22] showed a simple trick to make it as secure, with almost no overhead.

There have been efforts to port the above efficient classical construction to the post-quantum setting [ACD19, DG19, LKS23]. Notably, [ACD19, LKS23] proposes a lattice-based equivalent to the Double Ratchet protocol used in Signal. At a high level, it goes as follows, where $R_q := \mathbb{Z}_q[X]/(X^n + 1)$ and $\mathbf{D} \in R_q^{k \times k}$ is a public matrix. Assume Alice holds $\mathbf{s}_A \in R_q^k$ and Bob holds $\mathbf{u}_A = \mathbf{D} \cdot \mathbf{s}_A + \mathbf{e}_A \in R_q^k$, where $\mathbf{s}_A$ and $\mathbf{e}_A$ are short. Bob samples short vectors $\mathbf{s}_B, \mathbf{e}_B \in R_q^k$ and $\tilde{\mathbf{e}}_B \in R_q$ from appropriate distributions, and a random $\text{seed} \xleftarrow{\$} \{0, 1\}^n \subset R_q$. It then sends $(\mathbf{u}_B, v_A) := (\mathbf{D}^\top \cdot \mathbf{s}_B + \mathbf{e}_B, \mathbf{u}_A^\top \cdot \mathbf{s}_B + \tilde{\mathbf{e}}_B + \text{seed} \cdot \lfloor q/2 \rfloor)$

to Alice.³ Alice first interprets $(\mathbf{u}_B, v_A)$ as a ciphertext and decrypts `seed` by rounding $v_A - \mathbf{u}_B^\top \cdot \mathbf{s}_A$ to the nearest multiple of $\lfloor q/2 \rfloor$. The `seed` is then mixed into the shared key $\mathbf{K}$ to ratchet the state forward. Alice then interprets part of the ciphertext $\mathbf{u}_B$ as Bob’s public key so that it can perform similar ratcheting. As $\mathbf{u}_B$ is the dominant component in terms of size, this effectively almost halves the communication size, giving us the same benefit as Signal’s Double Ratchet.

Flaw in Previous Analyses. While the construction is intuitive and simple, we observe that the security proof is subtle. Indeed, we identify that both previous works [ACD19, LKS23] contain the same flaw in the CKA security proof, rendering their scheme insecure for certain parameter regime. Recall that Signal’s Double Ratchet was proven to satisfy PCS [ACD19] by arguing two things *even if* Alice’s secret key $a \in \mathbb{Z}_p$ is compromised:

(C.1) g^{ab} can be simulated without Bob’s secret key b .

(C.2) $(g, g^{a'}, g^b, g^{a'b})$ is indistinguishable from $(g, g^{a'}, g^b, g^c)$ for $c \xleftarrow{\$} \mathbb{Z}_p$.

Item (C.2) stipulates that once Alice updates its key to $g^{a'}$, while Bob’s key g^b is uncompromised, then the state *heals* since $g^{a'b}$ is mixed into the shared key. While seemingly unimportant, Item (C.1) is a vital property to formally invoke the DDH assumption in Item (C.2) — if not for Item (C.1), the reduction cannot embed g^b given by the DDH challenge into the CKA protocol. To imitate this proof for the aforementioned lattice-based scheme, we have to argue the following, even if Alice’s secret key $\mathbf{s}_A \in R_q^k$ is compromised:

(L.1) $v_A := \mathbf{u}_A^\top \cdot \mathbf{s}_B + \tilde{e}_B + \text{seed} \cdot \lfloor q/2 \rfloor$ can be simulated without Bob’s secret key $\mathbf{s}_B$.

(L.2) $(\mathbf{u}'_A, \mathbf{u}_B, v'_B) = (\mathbf{D} \cdot \mathbf{s}'_A + \mathbf{e}'_A, \mathbf{D}^\top \cdot \mathbf{s}_B + \mathbf{e}_B, \mathbf{u}_B^\top \cdot \mathbf{s}'_A + \tilde{e}'_A + \text{seed}' \cdot \lfloor q/2 \rfloor)$ is indistinguishable from $(\mathbf{u}'_A, \mathbf{u}_B, v)$ for $v \xleftarrow{\$} R_q$.

It turns out that this Item (L.1) is where the subtlety lies. Unlike in the classical setting, we no longer have clear symmetry. Indeed, observe that v_A is identically expressible as $v_A = (\mathbf{D} \cdot \mathbf{s}_A + \mathbf{e}_A)^\top \cdot \mathbf{s}_B + \tilde{e}_B + \text{seed} \cdot \lfloor q/2 \rfloor = \mathbf{u}_B^\top \cdot \mathbf{s}_A - \mathbf{e}_B^\top \cdot \mathbf{s}_A + \mathbf{e}_A^\top \cdot \mathbf{s}_B + \tilde{e}_B + \text{seed} \cdot \lfloor q/2 \rfloor$, where we plug in $\mathbf{u}_B = \mathbf{D}^\top \cdot \mathbf{s}_B + \mathbf{e}_B$. Denoting the underlined value as h , it is clear that h *cannot* be simulated only using Alice’s secret $\mathbf{s}_A$ (and $\mathbf{e}_A$). In fact, an adversary with $\mathbf{s}_A$ can directly compute h to infer statistical knowledge of $\mathbf{s}_B$ and $\mathbf{e}_B$. Even worse, since the adversary learns slight information about $\mathbf{s}_B$, we can inductively see that the adversary may also learn some information even on the *updated* key $\mathbf{s}'_A$, creating a vicious cycle.

Previous work has overlooked this issue and falsely invoked Item (L.2). We note that technically, we can *statistically* prove Item (L.1) by sampling $\tilde{e}_B$ from a distribution super-polynomially larger than $-\mathbf{e}_B^\top \cdot \mathbf{s}_A + \mathbf{e}_A^\top \cdot \mathbf{s}_B$ (i.e., noise flooding). However, this renders the scheme unusable in practice, and defeats the purpose of using the optimization.

Our Solution. At the core of our technical contribution, we use the recent hint-MLWE problem by [KLSS23, EEN⁺24] to *computationally* prove Item (L.1), and carefully argue Item (L.2). hint-MLWE in essence stipulates that the standard MLWE remains hard even if some *noisy* linear leakage of the secret is given to the adversary. In fact, we go one step further and show that our new proof strategy is essential to make the recent trick by Bienstock et al. [BFG⁺22] improving FS to work in the lattice-setting. The main idea of [BFG⁺22] was for Alice to run the same Signal’s original Double Ratchet protocol, but to store $\hat{a} := a' + H(g^{a'b})$ as opposed to a' . The intuition is that even if $\hat{a}$ is compromised, $g^{a'b}$ remains secure assuming H is a random oracle (or ElGamal encryption is circular secure), hence offering better FS. In the lattice-setting however, the updated $\hat{\mathbf{s}}_A := \mathbf{s}'_A + H(\text{seed}')$ must still remain *short* for decryption to work, and as such, leaking $\hat{\mathbf{s}}_A$ again statistically leaks information on $\mathbf{s}'_A$. For more details on the construction and its security proof, we refer to [DJK⁺25].

³Note that while [ACD19] bases their construction on FrodoKEM, our explanation is based on a Kyber-like KEM as in [LKS23]. These differences will have no importance to our argument.

Concrete instantiation. While hint-MLWE reduces from MLWE, this is not without a slight degradation in the parameters. We wrap up everything by performing cryptanalysis on hint-MLWE based on the reduction from hint-MLWE to MLWE, and set concrete parameters for our RKEM called *Katana*. We obtain an instantiation exhibiting public-key and ciphertext sizes of $|\text{ek}| = 1344\text{B}$ and $|\text{ct}| = 72\text{B}$, respectively. This saves $\approx 40\%$ in terms of the combined ciphertext and public key sizes compared to Kyber-768. For additional details we refer to [DJK⁺25].

4.3 Opportunistic Sending

Note that the compiler from CKA to bandwidth-limited SCKA described in Sec. 4.1 forces the communication to adhere to a ping-pong pattern, with the party currently receiving chunks of a CKA message having to send empty protocol messages for a prolonged amount of time, thus wasting bandwidth. This is an issue, as in secure messaging application we would expect communication to be somewhat balanced, with both parties sending SM (and thus also SCKA) messages.

We observe that the protocol messages of concrete CKA protocols, however, often contain cryptographic material that can be generated independently of any prior message. Exploiting this fact allows us to define *opportunistic SCKA* variants in which both parties are able to concurrently exchange chunks. For example, in one of our protocol candidates of Sec. 5 the party currently receiving a KEM ciphertext will already sample and start sending chunks of the public key for the subsequent epoch, in another the use of an online-offline KEM allows a party currently receiving a public key to already start sampling and sending chunks of a part of the corresponding ciphertext.

Intuitively, using opportunistic sending allows the protocol to progress epochs by exchanging fewer messages and thus increasing speed at which epoch keys are output. On the other hand, sampling key material at an earlier point in time implies the parties have to potentially store it for an extended period, thus making the impact of user compromises more severe.

4.4 Comparing Secure Messaging Protocols

Given constructions of SM from SCKA and SCKA from CKA, as well as several potential optimizations we end up with a large design space of “natural” bandwidth-limited secure messaging protocols. Using the metric of vulnerable message sets to argue how secure a messaging protocol is, we would like to formally compare different protocols. Specifically, given a budget on the bandwidth per message sent (i.e., an upper limit on the size of ct that can be output by the SM protocol), the messaging protocol with the smallest size of vulnerable message sets should be the “best” protocol.

While reasonable, it turns out that such a total ordering of SM protocols is out of our reach. This is because the vulnerable message sets are defined through the security experiment with an adversary, or in other words, they depend on the messaging behavior and the compromise scenario. Thus, for any natural two SM protocols, we can typically prepare (possibly theoretical) messaging behaviors and compromise scenarios where one outperforms the other and vice versa, demonstrating that protocols are generally incomparable.

Looking ahead, this is exactly why in Sec. 6 we compare our proposed SM protocols by simulating realistic messaging behaviors and compromise scenarios. In fact, our results will show that incompatibility of SM protocols already arises for natural messaging behavior (see Fig. 7).

5 Candidate SCKA Constructions

We define three novel bandwidth-limited SCKA protocols, each of which making use of chunking and opportunistic sending. Throughout this section we use superscripts t to mark variables corresponding to the t^{th} epoch key and subscripts A or B to indicate the party generating a particular variable.

5.1 Protocol Opp-UniKEM-CKA.

The idea behind protocol Opp-UniKEM-CKA based on $\text{KEM} = (\text{KeyGen}, \text{Enc}, \text{Dec})$ is as follows. B, after receiving ek_A , generates $(\text{ct}_B, I_B) \leftarrow \text{Enc}(\text{ek}_A)$ and sends ct_B to A. Upon receiving ct_B party A is able to recover the epoch key I_B , deletes decapsulation key dk_A , and moves to the next epoch. Similarly, upon receiving an acknowledgment that ct_B was fully transmitted, B moves on to the next epoch, and the procedure starts anew.

Protocol description. Opp-UniKEM-CKA adapts this idea to the setting of SCKA by means of chunking with erasure code ($\text{Encode}, \text{Decode}$). Further, it allows for opportunistic sending by exploiting the structure of *offline-online* KEMs. I.e., a KEM with ciphertexts consisting of two parts $\text{ct} = (\text{ct}_0, \text{ct}_1)$ such that ct_0 can be generated independently of the public key (see App. A for a formal definition). For the protocols formal description we defer to App. D.

Party A at the beginning of epoch t samples and stores $(\text{ek}_A^t, \text{dk}_A^t) \leftarrow \text{KeyGen}(1^\lambda)$, and initializes counter i_{ch} . Whenever CKA-Send-A is called (and B did not yet fully receive ek_A^t) i_{ch} is incremented and a chunk $\text{ch} \leftarrow \text{Encode}(\text{ek}_A^t, i_{\text{ch}})$ is generated that serves as protocol message ρ .

Party B at the beginning of epoch t generates an offline ciphertext as $(\text{st}_{\text{ct}}^t, \text{ct}_0^t) \leftarrow \text{Enc.Off}(1^\lambda)$ and initializes counter i_{ch} . If CKA-Send-B is called and A did not yet fully receive ct_0^t the protocol message ρ contains a chunk $\text{ch} \leftarrow \text{Encode}(\text{ct}_0^t, i_{\text{ch}})$, i_{ch} being incremented with every subsequent call. If, on the other hand, A already recovered ct_0^t and B fully received ek_A then, the ciphertext is completed as $(\text{ct}_1^t, I_B^t) \leftarrow \text{Enc.On}(\text{st}_{\text{ct}}^t, \text{ek}_A^t)$, epoch key I_B^t returned, i_{ch} reset, and chunks $\text{ch} \leftarrow \text{Encode}(\text{ct}_1^t, i_{\text{ch}})$ are sent in subsequent protocol messages. Algorithms CKA-Rec-B and CKA-Rec-A store the chunks contained in ρ in a list L_{ch} until they are able to recover $\text{ek}_A^t \leftarrow \text{Decode}(L_{\text{ch}})$, $\text{ct}_0^t \leftarrow \text{Decode}(L_{\text{ch}})$, or $\text{ct}_1^t \leftarrow \text{Decode}(L_{\text{ch}})$ respectively. If CKA-Rec-A fully recovers both parts of ct_B^t it computes and outputs the epoch key as $I_B^t \leftarrow \text{Dec}(\text{dk}_A^t, \text{ct}_0^t, \text{ct}_1^t)$, resets L_{ch} , deletes all cryptographic material corresponding to the epoch, and with the next sent message initiates the subsequent epoch. To keep the other party informed about how much of the cryptographic material of the current epoch was already recovered both parties attach acknowledgment flags act.ek-rec and $\text{act.ct}_0\text{-rec}$ that indicate whether B already recovered ek_A^t or whether A recovered ct_0^t , respectively.

Sending/receiving epoch and vulnerable epochs. Consider the situation in which the parties are exchanging the encapsulation key and ciphertext for epoch t . Since A only is able to recover I_B^t after ct_1^t has been fully transmitted, at which point A initiates the next epoch $t + 1$, and since B only learns about this when receiving a protocol message by A we have that $t_B^{\text{snd}} = t - 1 = t_A^{\text{snd}}$. Protocol messages include the epoch t of the sent cryptographic material allowing the processing party to compute the matching t_B^{cv} or t_A^{rcv} respectively.

Regarding the vulnerable epoch set, note that the secret key-material contained in the parties' states at every point in time pertains to a single epoch. Accordingly for $P \in \{A, B\}$, we have

$$\text{st}_P.\text{vuln} = \begin{cases} \{t\} & \text{if } \llbracket P = A \rrbracket \wedge \llbracket \text{dk}_A^t \neq \perp \rrbracket \\ \{t\} & \text{if } \llbracket P = B \rrbracket \wedge \llbracket \text{st}_{\text{ct}}^t \neq \perp \rrbracket \\ \emptyset & \text{else} \end{cases}.$$

Instantiation. We instantiate Opp-UniKEM-CKA with Kyber-768. As we only require IND-CPA security, we forgo the FO transform required for IND-CCA security. As discussed in App. A it has the structure of an online-offline KEM. It exhibits encapsulation key and ciphertext sizes of

$$(|\text{ek}|, |\text{ct}_0|, |\text{ct}_1|) = (1184\text{B}, 960\text{B}, 128\text{B})$$

Looking ahead, our simulations use an erasure code creating three sizes of chunks: (32, 128, 512) bytes. As a consequence, respectively, (37, 10, 3) chunks have to be received to recover an encapsulation key, (30, 8, 2) to recover an offline ciphertext ct_0 , and (4, 1, 1) to recover ct_1 .

5.2 Protocol Opp-BiKEM-CKA.

The CKA from [ACD19] at basis of Opp-BiKEM-CKA works as follows. Opposed to the unidirectional approach of the previous protocol, each party encapsulates epoch keys. In an epoch with party A doing so we may assume (as will become clear below) they already have stored an encapsulation key ek_B . To send a protocol message they, on one hand, create the epoch key I_A by calling $(ct_A, I_A) \leftarrow \text{Enc}(ek_B)$ and, on the other, sample key pair (ek_A, dk_A) to be used in the subsequent epoch. The decapsulation key dk_A is stored in state and the protocol message consists of (ct_A, ek_A) . B does not send any cryptographic material to A. However, after receiving (ct_A, ek_A) they recover I_A from the ciphertext, delete the corresponding decapsulation key, and move to the next epoch, the parties' roles now reversing, i.e. using ek_A to generate epoch key and ciphertext (ct_B, I_B) .

Opp-BiKEM-CKA adapts the this idea to the setting of SCKA using chunking and letting both parties opportunistically exchange encapsulation keys. We defer its formal description to App. D.

Protocol description. The protocol is defined with respect to a (standard) KEM and erasure code (Encode, Decode). For odd epochs $t = 2k + 1$ party A generates epoch keys I_A^t and ciphertexts ct_A^t while B provides encapsulation key ek_B^t . For even epochs $t = 2k$ the roles are reversed, with variables I_B^t, ct_B^t, ek_A^t .

Assume A has just completed sending ciphertext ct_A^{t-2} and fully received user B' ciphertext ct_B^{t-1} generated with respect to A's last encapsulation key ek_A^{t-1} . CKA-Send-A when called for the next time samples a new key pair $(ek_A^{t+1}, dk_A^{t+1}) \leftarrow \text{KeyGen}(1^\lambda)$ and initializes counter i_{ch} to be incremented with every call to CKA-Send-A. Afterward, until A receives an acknowledgment that B has received ek_A^{t+1} every protocol messages generated by CKA-Send-A contains a chunk $ch \leftarrow \text{Encode}(ek_A^{t+1}, i_{ch})$. Upon receiving this acknowledgment and if A additionally has recovered user B's encapsulation key ek_B^t , algorithm CKA-Send-A samples ciphertext and epoch key $(I_A^t, ct_A^t) \leftarrow \text{Enc}(ek_B^t)$, outputs I_A^t , resets i_{ch} , and starts sending chunks $ch \leftarrow \text{Encode}(ct_A^t, i_{ch})$.

Algorithm CKA-Rec-A, in case of the starting conditions above (ct_A^{t-2} delivered and ct_B^{t-1} received), expects to receive B's encapsulation key ek_B^t . Accordingly, it stores chunks ch contained in protocol messages sent by B in a list L_{ch} until it is able to recover $ek_B^t \leftarrow \text{Decode}(L_{ch})$. After acknowledging this to B and resetting L_{ch} it expects to receive ciphertext chunks and at some point recovers $ct_B^{t+1} \leftarrow \text{Decode}(L_{ch})$. It outputs the epoch key $I_B^{t+1} \leftarrow \text{Dec}(dk_A^{t+1}, ct_B^{t+1})$, and deletes the decapsulation key. If B also fully received ct_A^t , the working epoch t is incremented by 2 and the procedure starts anew.

Algorithms CKA-Send-B and CKA-Rec-B are defined to match the above. I.e., B upon completed reception of ct_A^{t-2} and an acknowledgment of ct_B^{t-1} generates (ek_B^t, dk_B^t) and starts sending chunks of the encapsulation key. After receiving ek_A^{t+1} and an acknowledgment that ek_B^t was fully transmitted CKA-Send-B generates $(I_B^{t+1}, ct_B^{t+1}) \leftarrow \text{Enc}(ek_A^{t+1})$ and starts sending chunks of ct_B^{t+1} . CKA-Rec-B processes the chunks sent by A to recover ek_A^{t+1}, ct_A^t , and in turn I_A^t .

Sending/receiving epoch and vulnerable epochs. The parties inform each other whether ciphertexts ct_P^t with $P \in \{A, B\}$ were fully transmitted, and in turn whether the epoch key was received, by exchanging boolean acknowledgment $act^t.ct-rec$. Since epoch keys need be included in the symmetric ratchet in order and the keys I_A^t, I_B^{t+1} might be recovered in reversed order we define $t_P^{snd} = \max(t : \llbracket act^t.ct-rec \rrbracket \wedge \llbracket act^{t-1}.ct-rec \rrbracket)$. P includes a copy of t_P^{snd} in protocol messages which is output as t_P^{cv} by receiving party $\bar{P}$.

Regarding the vulnerable epoch set, note that the only secret key material kept in state are decapsulation keys dk_A^t or dk_B^t respectively. Accordingly, for $P \in \{A, B\}$ we have

$$st_P.vuln = \begin{cases} \{t\} & \text{if } \llbracket dk_P^t \neq \perp \rrbracket \\ \emptyset & \text{else.} \end{cases}$$

Instantiation. We instantiate Opp-BiKEM-CKA using Kyber-768. Similarly to Opp-UniKEM-CKA, we only require the IND-CPA variant. Encapsulation keys and ciphertexts are of sizes

$$(|ek|, |ct|) = (1184B, 1088B).$$

Since our simulations use chunks of size (32, 128, 512) bytes, (37, 10, 3) chunks have to be received to recover an encapsulation key and (34, 9, 3) to recover a ciphertext.

5.3 Protocol Opp-RKEM-CKA.

Opp-RKEM-CKA also follows the structure of BiKEM-CKA. However, parties use a *ratcheting* KEM to reuse the first part of an epoch's ciphertext as the subsequent epoch's encapsulation key (see ??). Loosely, it can be viewed as a lattice variant of Double Ratchet. Further, the protocol makes use of chunking and opportunistic sending. We defer its formal description to App. D.

Protocol description. The protocol is defined with respect to ratcheting KEM RKEM and erasure code (Encode, Decode). For odd epochs $t = 2k+1$ party A generates epoch key I_A^t and communicates it to B by transmitting ct_A^t and ek_A^{t+1} . Here, the latter intuitively serves as the ciphertext's first, ek_B^t -independent part. The ciphertext is generated as $(ct_A^t, I_A^t, dk_A^{t+1}) \leftarrow \text{REnc-A}(ek_B^t, dk_A^{t+1})$, i.e., with respect to B's current encapsulation key and A's subsequent decapsulation key. Note that the latter is updated by performing the operation. Analogously, B for even epochs $t = 2k$ generates epoch keys as $(ct_B^t, I_B^t, dk_B^{t+1}) \leftarrow \text{REnc-B}(ek_A^t, dk_B^{t+1})$ and transmits the ciphertext as well as ek_B^{t+1} to A.

Assume A received an acknowledgment that B recovered both ct_A^{t-2} and ek_A^{t-1} , the values A generated for epoch $t-2$. Now, the first call to CKA-Send-A generates a new RKEM key pair (dk_A^{t+1}, ek_A^{t+1}) , stores them in state, and initializes counter i_{ch} to be incremented with every subsequent call. At this point CKA-Send-A generates protocol messages containing chunks $ch \leftarrow \text{Encode}(ek_A^{t+1}, i_{ch})$ until A receives an acknowledgment that the encapsulation key was received by B. In order to proceed by generating ciphertext ct_A^t party A must already have recovered and updated the corresponding encapsulation key ek_B^t . The latter is done by calling $(I_B^{t-1}, ek_B^t) \leftarrow \text{RDec-A}(ct_B^{t-1}, dk_A^{t-1}, ek_B^t)$. Thus, after B acknowledged receiving ek_A^{t+1} , sending algorithm CKA-Send-A generates $(ct_A^t, I_A^t, dk_A^{t+1}) \leftarrow \text{REnc-A}(ek_B^t, dk_A^{t+1})$ in the first call to the algorithm after receiving both ek_B^t and ct_B^{t-1} . After resetting i_{ch} , CKA-Send-A from this point on generates protocol messages containing chunks $ch \leftarrow \text{Encode}(ct_A^t, i_{ch})$. After receiving an acknowledgment of B recovering ct_A^t party A is able to restart the procedure, generating a key pair for epoch $t+3$.

Receiving algorithm CKA-Rec-B stores chunks contained in protocol messages in a list L_{ch} until it is able to first recover ek_A^{t+1} then, after resetting the list, ct_A^t . When it has done so, it recovers epoch key $(I_A^t, ek_A^{t+1}) \leftarrow \text{RDec-B}(ct_A^t, dk_B^t, ek_A^{t+1})$ also updating A's encapsulation key in the process and deletes dk_B^t . Algorithms CKA-Send-B and CKA-Rec-A are defined analogously, however with the roles of A and B reversed and B generating the epoch keys I_B^t for even epochs.

Sending/receiving epoch and vulnerable epochs. We set $t_P^{snd} = \max(t : \llbracket \text{act}^t.\text{ct-rec} \rrbracket \wedge \llbracket \text{act}^{t-1}.\text{ct-rec} \rrbracket)$ for user $P \in \{A, B\}$ where acknowledgment $\text{act}^t.\text{ct-rec}$ included in the users' protocol messages indicates whether the ciphertext for epoch t was fully transmitted. P includes a copy of t_P^{snd} in protocol messages which is output as t_P^{rcv} by receiving party $\bar{P}$.

Regarding the vulnerable epoch set, note that the only secret key material contained in users' states are the decapsulation keys. If a key dk_P^t was already updated by creating a ciphertext it reveals the epoch secret of epoch t . If the key was not updated, on the other hand, it reveals both $t-1$ and t , as it is used to create the ciphertext of the former epoch. Finally, we point out that a party's state may contain up to 2 decapsulation keys at any point in time, since, for example, A might already generate dk_A^{t+1} before receiving and processing ciphertext ct_B^{t-1} and deleting dk_A^{t-1} . Accordingly, the vulnerable epoch set is given by

$$st_P.vuln = \{t : \llbracket dk_P^t \neq \perp \rrbracket\} \cup \{t-1 : \llbracket \perp \neq dk_P^t \text{ not updated} \rrbracket\}.$$

Instantiation. We instantiate Opp-RKEM-CKA with the lattice-based RKEM Katana of Sec. 4.2. It exhibits encapsulation key and ciphertext sizes of

$$(|ek|, |ct|) = (1344B, 72B).$$

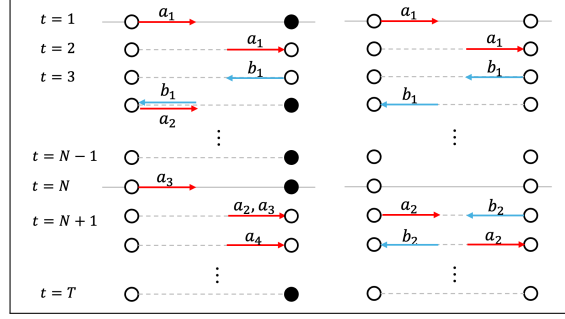


Figure 3: Messaging behavior of a primary-desktop model (left) and a primary-primary model (right). The white (resp. black) circles indicate that a device is online (resp. offline), i.e., $\iota_P^{\text{on}}(t \bmod N) = 1$ ($\iota_P^{\text{on}}(t \bmod N) = 0$). Arrows coming out (resp. in) a circle indicate the message that is being sent (resp. received) at a given time t .

Since our simulations use chunks of size (32, 128, 512) bytes, (42, 11, 3) chunks have to be received to recover an encapsulation key and (3, 1, 1) to recover a ciphertext.

6 Experimental Evaluation

The aim of our experiments is, given a fixed bandwidth, to determine which SM protocol is the most secure under realistic messaging behaviors. During the preparation of the experimental setup we reached out to Signal. While they do not keep data on users' messaging behavior, we prepared our probabilistic messaging behavior models in discussion with them. In particular, the primary vs desktop device and online vs offline distinctions detailed below are based on the two polar user behaviors on the Signal app. We compare six SM protocols obtained by instantiating the generic SM protocol in Sec. 3.2 with either the SCKAs presented in Sec. 5 or their counterparts not making use of opportunistic sending (see App. E for additional details).

6.1 Experimental Setup

Messaging behavior models. To understand the practical security of our SM protocols we use a randomized model of messaging behavior. The Signal Messenger app has two device types: a *primary* device is a user's phone that was used to create an account, where a *linked device*, such as a desktop, is authorized by the primary device. Although the user does not always have the application open, we model primary devices to be always *online* in the sense that, as long as they have power and network access, push notifications will trigger them to receive incoming messages. Linked devices such as desktops, however, are often powered off and completely *offline*. A typical behavior of a desktop user is to open the application during work hours and then close it.

Motivated by that background, we create a model of messaging behavior that breaks a day into N discrete timesteps and provides each party P , with an *online indicator function*, $\iota_P^{\text{on}} : \mathbb{Z}_N \rightarrow \{0, 1\}$ that determines when that party can receive messages. To capture the fact that devices receive messages regularly when online, but mostly send messages in response to user activity, we also give each party a parameter p_P^{send} which gives the probability that they will send a message at any given timestep where they are online.

To simulate a session between two parties, we generate a randomized communication pattern proceeding for $T \gg N$ timesteps (i.e., T/N days) as follows. For each timestep t and for each party P we evaluate whether P is online at the corresponding day and time by computing $\iota_P^{\text{on}}(t \bmod N)$. If it returns 0 (i.e., party is offline), no action is taken for that party. Otherwise, the online party receives all incoming messages and then sends a single message with probability p_P^{send} .

For all experiments reported, we choose $N = 240$, translating to one timestep being 6 minutes long. For most experiments, we then take $p_A^{\text{send}} = p_B^{\text{send}} = 0.4$ to model balanced communications, where parties send about 4 messages per hour. We also experiment with unbalanced communications with $(p_A^{\text{send}}, p_B^{\text{send}}) =$

(0.04, 0.36) and (0.36, 0.04).⁴ We then run each experiment for $T = 10^6$ timesteps corresponding to a conversation lasting for roughly 11 years. With this we define 4 message behavior models:

- *primary-primary* models interaction between two primary devices that are always online to receive messages. For all $t \in \mathbb{Z}_N$, $\iota_A^{\text{on}}(t) = \iota_B^{\text{on}}(t) = 1$.
- *primary-desktop* models interaction between a primary device and a desktop device that is online from 6h to 18h daily. User A has the same parameters as in the primary-primary model.
- *overlapping desktop-desktop* models interaction between two desktop devices with overlapping online-time windows — user A is online from 0h to 12h, and user B is online from 6h to 18h daily.
- *disjoint desktop-desktop* models interaction between two desktop devices that are never online at the same time — user A is online from 0h to 12h, and user B is online from 12h to 24h daily.

For a visualization of the messaging behavior, see Fig. 3.

Corruption model. We consider *single-party corruptions* allowing us to draw conclusions on the impact of one party’s state being exposed. To this end, we track the vulnerable message set as the parties’ states evolve during the experiment. More precisely, for every timestep t and each party P we check whether P ’s state changed as a result of processing one or more messages and/or sending a message. If so, we record the vulnerable message set $\text{st}_P.\text{vuln}$. Note that this can only occur if the party is online, and avoids over-tracking the vulnerable message sets of an offline party.

As in some of the considered protocols the parties have asymmetric roles, we track their vulnerable message sets separately. That is, we present our results on the size of the recorded statistics for each party P separately. Lastly, we do not explicitly consider both-party corruptions as they can be inferred from the single-party corruption for two parties.

Implementation. Simulations were implemented in Rust. We consider three bandwidth upper bounds: 32, 128, and 512 bytes. Accordingly, all six protocols use Reed-Solomon erasure codes based on $GF(2^{16})^{w/2}$ to transmit messages in w byte chunks. Opp-BiKEM-SM and BiKEM-SM, using standard KEMs, are fully functional and use libcrux [ONL] to implement the KEM using MLKEM [SAB⁺22]. Opp-UniKEM-SM and UniKEM-SM, using online-offline KEMs, use mocked implementations of MLKEM allowing correctness tests. Lastly, Opp-RKEM-SM and RKEM-SM, using ratcheting KEMs, use mocked implementations of *Katana*. All protocols are subjected to a test suite that uses different messaging behaviors, drops messages, and delivers messages out of order.

6.2 Comparison Methodology

The most important metric of a SM protocol is the size of the Vulnerable Message sets (VulM), i.e., the number of messages exposed by a compromise, computed via Sec. 3.2. A well-designed SM protocol should offer security for all realistic cases, rather than only excelling at one, e.g., in securing the communication between primary devices. To be considered for adoption, we thus require that the protocol performs well across all different scenarios. Furthermore, a good protocol should balance average-case and worst-case guarantees. In particular, in addition to recover fast on average, it should exhibit the following properties.

- The variance of $|\text{VulM}|$ should be small. Otherwise, there might be a substantial chance of the party’s compromise lasting much longer, even though recovery is fast on average.
- $|\text{VulM}|$ should be similar for both parties. Otherwise, one would need to know each party’s chance of compromise in order to optimally assign their roles.

⁴The concrete values of p_A^{send} and p_B^{send} do not matter too much as long as the ratio is fixed; a higher value means more sent messages but the time it takes to recover becomes that much shorter.

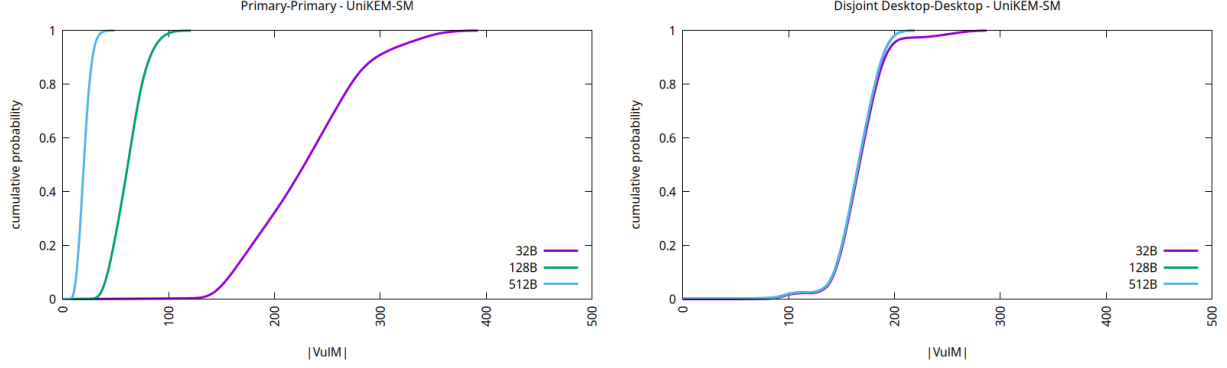


Figure 4: Cumulative probability of $|\text{VuIM}|$ for UniKEM-SM with chunk sizes 32, 128, 512 bytes. Left (resp. right) is primary-primary (resp. disjoint desktop-desktop) model.

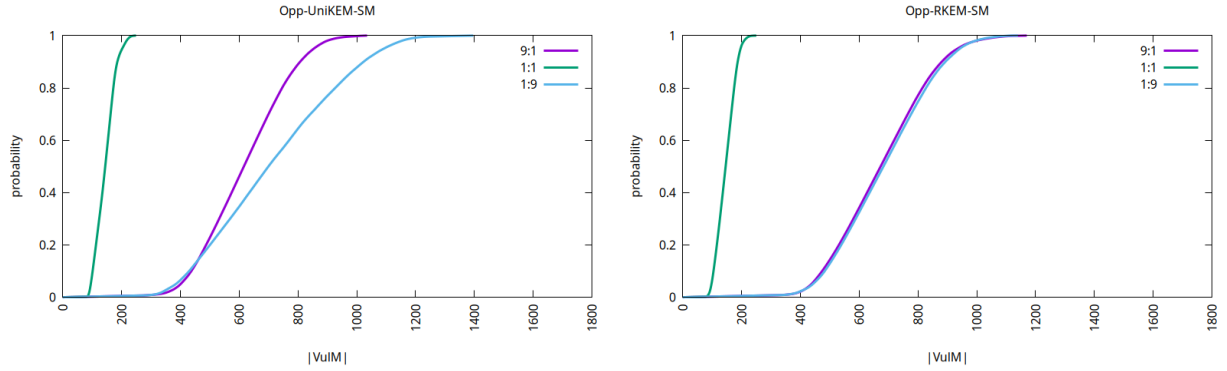


Figure 5: Cumulative probability of $|\text{VuIM}|$ for Opp-UniKEM-SM (left) and Opp-RKEM-SM (right) with chunk size 32 bytes. $(p_A^{\text{send}}, p_B^{\text{send}})$ is set to $(0.36, 0.04)$ (ratio 9 : 1), $(0.4, 0.4)$ (ratio 1 : 1), and $(0.04, 0.36)$ (ratio 1 : 9).

6.3 Protocol Independent Phenomena

We exhibit some general properties that hold irrespective of the protocol. As it is also irrespective of who the compromised party is, we only consider party A corruption.

Small vs large bandwidth limit. As SM protocols supporting larger bandwidth allow larger chunks to be sent, $|\text{VuIM}|$ becomes smaller. However, since the protocol cannot become secure while one party is offline, the benefit of allowing larger chunks diminishes as the overlap of the parties' online time becomes small. Fig. 4 illustrates this.

Balanced vs unbalanced communication. The security of a SM protocol hinges on the most insecure party; if one party is compromised, then all messages sent during that time become vulnerable, regardless of the peer's corruption state. As such, unbalanced communications amplify the time it takes to recover from a compromise, making the average and variance of $|\text{VuIM}|$ larger. Fig. 5 illustrates this.

Offline vs online parties. Similarly to above, a compromised party that is offline cannot recover, and thus, the average and variance of $|\text{VuIM}|$ increase. This is illustrated in Fig. 6 (left). Note that the primary-primary model and the overlapping desktop-desktop model produce similar cumulative probability of $|\text{VuIM}|$ as both send a similar number of messages and are online at the same time enough to avoid blocking; For

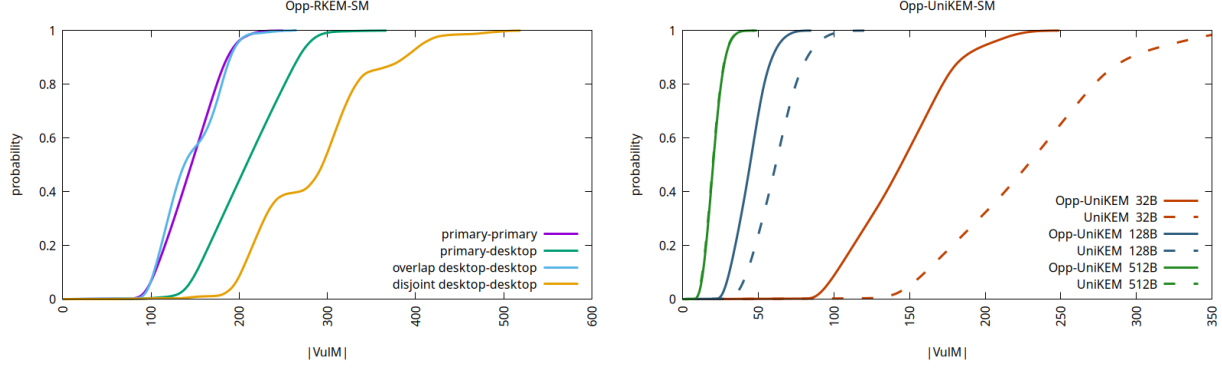


Figure 6: Cumulative probability of $|\text{VuIM}|$ (Left), and (Right) all protocols with chunk size 32 bytes in primary-primary model.

primary-desktop we see a higher average of $|\text{VuIM}|$ because the communication is unbalanced and the online party will continue sending vulnerable messages while the other party is offline.

Opportunistic vs non-opportunistic sending. Opportunistic sending is consistently better (i.e., small average and variance of $|\text{VuIM}|$) for small chunks, suggesting that wasting bandwidth is more harmful than sampling key material in advance. However, when the chunks become larger the benefit starts to diminish as they behave similarly, except for the disjoint desktop-desktop model. This is because as long as the party is offline, no recovery can happen — even if the online party finishes sending all information required to proceed to the next epoch secret. This is illustrated in Fig. 6 (right).

6.4 Comparing Opportunistic Protocols

We now compare Opp-UniKEM-SM, Opp-BiKEM-SM, and Opp-RKEM-SM in Fig. 7 as they perform consistently better than their non-opportunistic counterparts, and make the following observations.

- Opp-UniKEM-SM and Opp-RKEM-SM overall perform consistently close across all messaging behaviors.
- Opp-BiKEM-SM, on the other hand, has substantially larger $|\text{VuIM}|$ and variance in most settings (with the noticeable exception of the disjoint desktop-desktop setting for corruptions of A).
- Opp-BiKEM-SM exhibits a striking asymmetry between corruptions of A and corruptions of B, with compromises of A being significantly less harmful.
- Opp-UniKEM-SM also exhibits some asymmetry, but to a significantly smaller degree.

As a consequence, when comparing the opportunistic protocols only — at chunk sizes comparable to the overhead of the Double Ratchet — then Opp-RKEM-SM and Opp-UniKEM-SM both perform well, slightly favoring the former for its symmetric behavior.

6.5 Conclusion

We observe that both Opp-UniKEM-SM and Opp-RKEM-SM perform well with similar performance. Both their behavior is robust across a wide range of messaging behaviors and other parameter choices such as the chosen per-message overhead. Overall, Opp-RKEM-SM seems to have a small edge in some settings (but not all) and also is the most symmetric one with respect to the impact a corruption of either of the users has.

In summary, for bandwidth-limited post-quantum SM, there does not seem to present a clear winner with different protocols emerging as the winner depending on the messaging pattern. Therefore, to make an

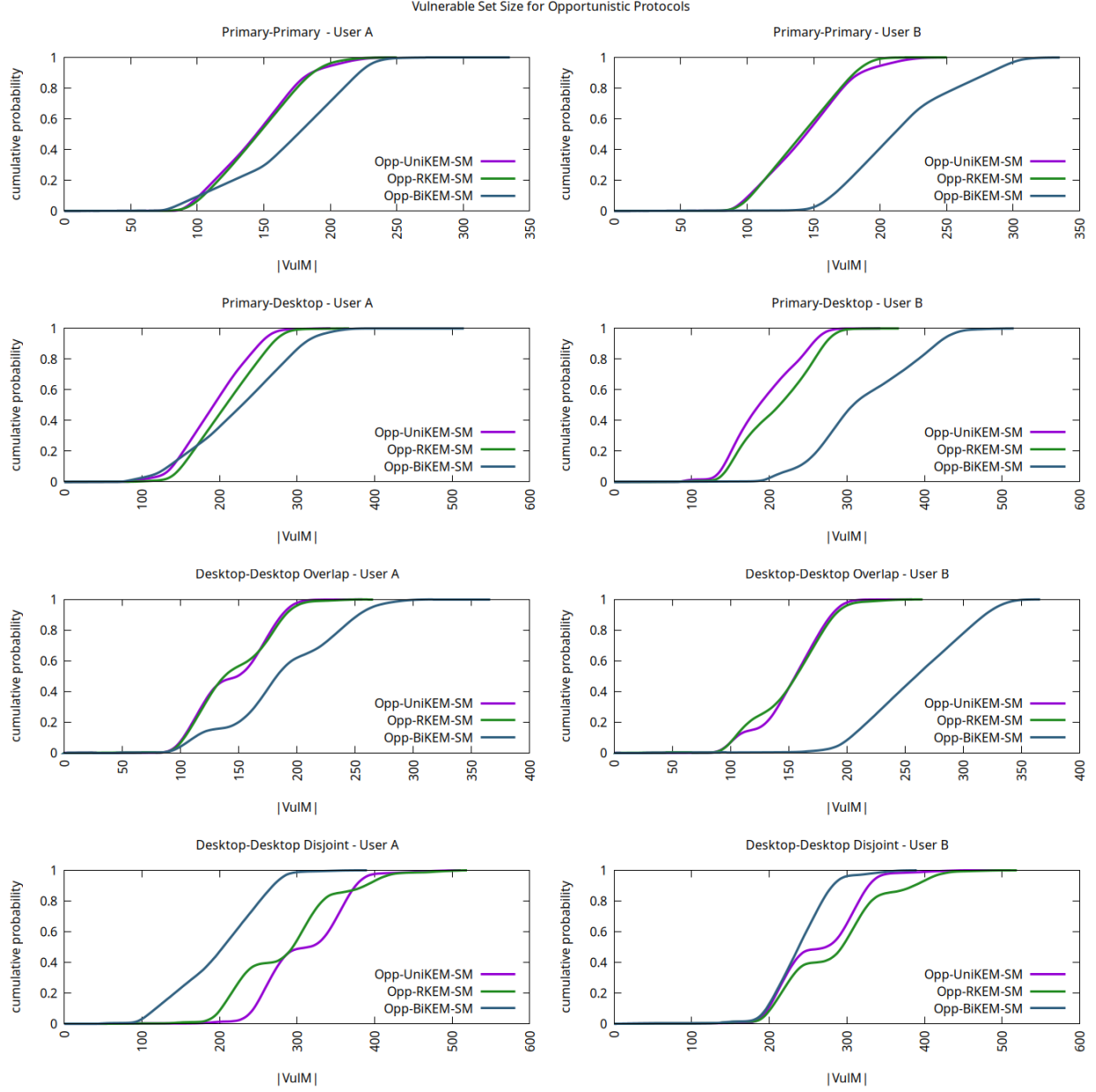


Figure 7: Cumulative probability of $|\text{VuIM}|$ for the opportunistic protocol variants using 32 byte chunks under all four message behavior models.

informed choice, it is vital for practitioners to understand the messaging behavior of their users and client devices.

Based on this analysis combined with their desire to use standardized primitives, Signal has modified Opp-UniKEM-SM to use the specified IND-CCA variant of MLKEM and hybridized it with the existing Double Ratchet [MP16] using the Triple Ratchet construction. The resulting protocol is in the process of deployment and its source code can be found at [Mesb, Mesa].

References

- [ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158, Darmstadt, Germany, May 19–23, 2019. Springer, Cham, Switzerland.
- [ADJ⁺25] Benedikt Auerbach, Yevgeniy Dodis, Daniel Jost, Shuichi Katsumata, and Rolfe Schmidt. How to compare bandwidth constrained two-party secure messaging protocols: A quest for a more efficient and secure post-quantum protocol. In *USENIX Security 2025*. (forthcoming), 2025.
- [App24] Apple Security Engineering and Architecture (SEAR). iMessage with PQ3: The new state of the art in quantum-secure messaging at scale, 2 2024. Available at <https://security.apple.com/blog/imessage-pq3/>.
- [BFG⁺22] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 784–813, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.
- [DG19] Nir Drucker and Shay Gueron. Continuous key agreement with reduced bandwidth. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 33–46. Springer, 2019.
- [DJK⁺25] Yevgeniy Dodis, Daniel Jost, Shuichi Katsumata, Thomas Prest, and Rolfe Schmidt. Triple ratchet: A bandwidth efficient hybrid-secure signal protocol. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - Madrid, Spain, May 4-8, 2025, Proceedings, Part VIII*, volume 15608 of *Lecture Notes in Computer Science*, pages 302–331. Springer, 2025.
- [EEN⁺24] Muhammed F. Esgin, Thomas Espitau, Guilhem Niot, Thomas Prest, Amin Sakzad, and Ron Steinfeld. Plover: Masking-friendly hash-and-sign lattice signatures. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VII*, volume 14657 of *LNCS*, pages 316–345, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.
- [KAD⁺25] Shuichi Katsumata, Benedikt Auerbach, Yevgeniy Dodis, Daniel Jost, Thomas Prest, and Rolfe Schmidt. The triple ratchet protocol: A bandwidth efficient hybrid-secure signal protocol. Real World Crypto Symposium 2025, 2025.
- [KLSS23] Duhyeong Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Toward practical lattice-based proof of knowledge from hint-MLWE. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 549–580, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Cham, Switzerland.
- [KS23] Ehren Kret and Rolfe Schmidt. The pqxdh key agreement protocol, 2023. Available at <https://signal.org/docs/specifications/pqxdh/>.
- [LKS23] Joohee Lee, Jihoon Kwon, and Ji Sun Shin. Efficient continuous key agreement with reduced bandwidth from a decomposable kem. *IEEE Access*, 11:33224–33235, 2023.
- [Mesa] Signal Messenger. libsignal. Available at <https://github.com/signalapp/libsignal>.
- [Mesb] Signal Messenger. Spqr - the sparse post quantum ratchet. Available at <https://github.com/signalapp/SparsePostQuantumRatchet>.
- [MP16] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm, 2016. Available at <https://signal.org/docs/specifications/doubleratchet/>.

- [ONL] libcrux - the formally verified crypto library. Available at <https://github.com/cryspen/libcrux>.
- [SAB⁺22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [Ste24] Douglas Stebila. Security analysis of the iMessage PQ3 protocol. Cryptology ePrint Archive, Report 2024/357, 2024.

A Omitted Preliminaries

A.1 Key-encapsulation mechanisms (KEMs).

Some of our constructions make use of a special type of KEM for which the encapsulation algorithm Enc can generate ciphertexts $\text{ct} = (\text{ct}_0, \text{ct}_1)$ in an offline and online phase.

Definition 5 ((Online-Offline) KEM) *We say a key-encapsulation mechanism KEM is an online-offline KEM if it consists of the following PPT algorithms:*

$\text{Setup}(1^\lambda) \rightarrow \text{par}$: On input the security parameter 1^λ , it returns a public parameter par . We assume all algorithms to take par as input and may omit it for simplicity.

$\text{KeyGen}(\text{par}) \rightarrow (\text{ek}, \text{dk})$: On input par , it returns an encapsulation key ek and a decapsulation key dk .

$\text{Enc.Off}(\text{par}) \rightarrow (\text{st}_{\text{ct}}, \text{ct}_0)$: On input par , it returns state st_{ct} and offline ciphertext ct_0 .

$\text{Enc.On}(\text{st}_{\text{ct}}, \text{ek}) \rightarrow (\text{ct}_1, I)$: On input the state st_{ct} and encapsulation key ek , it returns the encapsulated key I and the online ciphertext ct_1 .

$\text{Dec}(\text{dk}, \text{ct}) \rightarrow I$: On input a decapsulation key dk and a ciphertext ct , it returns I or the special symbol $\perp$ indicating a decryption failure.

A regular KEM merges Enc.Off and Enc.On into a single algorithm $\text{Enc}(\text{par}, \text{ek}) \rightarrow (\text{ct}, I)$, where intuitively $\text{ct} = (\text{ct}_0, \text{ct}_1)$, and par may be omitted if subsumed by ek .

We require an (online-offline) KEM to be correct and only require it to be IND-CPA secure.

As a concrete example, MLKEM (i.e., Kyber) [SAB⁺22] can be viewed as an online-offline KEM by seeing the public matrix $\mathbf{D}$ defined in the encapsulation key as a public parameter par . Ignoring optimization details for now, $\text{ct}_0 = \mathbf{D} \cdot \mathbf{r} + \mathbf{z}$ and $\text{ct}_1 = \text{ek}^\top \cdot \mathbf{r} + z' + \lfloor (q/2) \rfloor \cdot I$, where ek is the user-specific encapsulation key. Looking ahead, this allows a user to compute the offline ciphertext ct_0 while still waiting to receive the peer's encapsulation key ek .

A.2 Erasure codes

We define erasure codes. For a more convenient presentation of our protocols we allow the decoding procedure to fail, outputting a special symbol $\perp$ in this case.

Definition 6 *An erasure code for a set of symbols Σ , a block length N , and a message size n_{chunk} consists of PPT algorithms $\text{Encode}, \text{Decode}$ defined as follows:*

$\text{Encode}(M, i) \rightarrow c$: It takes as input a message $M \in \Sigma^{n_{\text{chunk}}}$, and an integer $i \in \mathbb{Z}_N$ and outputs symbol $c \in \Sigma$.

$\text{Decode}(\mathbf{L}) \rightarrow M$: It takes as input a set $\mathbf{L} \subset \mathbb{Z}_N \times \Sigma$ and outputs either a message $M \in \Sigma^{n_{\text{chunk}}}$ or the symbol $\perp$.

An erasure code is said to be correct if for all messages $M \in \Sigma^{n_{\text{chunk}}}$, for all $I \subset \mathbb{Z}_N$ and $\mathbf{L} = \{(i, \text{Encode}(M, i)) \mid i \in I\}$ we have that $\text{Decode}(\mathbf{L}, n_{\text{chunk}}) = M$ if $|I| = n_{\text{chunk}}$ and $\text{Decode}(\mathbf{L}, n_{\text{chunk}}) = \perp$ if $|I| < n_{\text{chunk}}$.

A.3 Authenticated Encryption

The secure messaging protocol uses a standard AEAD to send the payload.

Definition 7 (Authenticated Encryption) *An authenticated encryption with associated data (AEAD) scheme is a pair of algorithms $\text{AEAD} := (\text{Enc}, \text{Dec})$ with the following syntax:*

$\text{Enc}(K, h, M) \rightarrow e$: It takes a key K , authenticated data h , and a message M , and produces a ciphertext e .

$\text{Dec}(K, h, c) \rightarrow M'$: It takes a key K , authenticated data h , and a ciphertext c , and outputs a plaintext M' .

We assume all algorithms to be deterministic, i.e., all randomness to be based off the key.

We say that an AEAD scheme is correct, if for all keys K and all pairs (h, M) ,

$$\text{Dec}(K, h, \text{Enc}(K, h, m)) = m.$$

We require AEAD to be one-time IND-CCA secure, formalized by the game in Fig. 8, and define the following advantage

$$\text{Adv}_{\mathcal{A}}^{\text{AEAD}}(1^\lambda) := \left| \Pr[\text{Game}_{\mathcal{A}}^{\text{AEAD}}(1^\lambda)] - \frac{1}{2} \right|$$

and say the scheme to be secure iff every PPT $\mathcal{A}$ has negligible advantage.

$\text{Game}_{\mathcal{A}}^{\text{AEAD}}(1^\lambda)$	$\text{enc}(h, M)$
1 : $b \xleftarrow{\$} \{0, 1\}$	1 : if $\llbracket b = 0 \rrbracket$ then
2 : $K \xleftarrow{\$} \{0, 1\}^\lambda$	2 : $e^* \leftarrow \text{Enc}(K, h, M)$
3 : $e^* \leftarrow \perp$	3 : else
4 : $b' \xleftarrow{\$} \mathcal{A}(1^\lambda)^{\text{enc}(), \text{dec}()}$	4 : $e^* \xleftarrow{\$} \mathcal{C}$
5 : return $\llbracket b = b' \rrbracket$	5 : return e^*
	$\text{dec}(h, e)$
	1 : if $\llbracket e = e^* \rrbracket \vee \llbracket b = 1 \rrbracket$ then
	2 : return $\perp$
	3 : return $\text{Dec}(K, h, e)$

Figure 8: The one-time IND-CCA game of an AEAD scheme (Enc, Dec) with ciphertext space $\mathcal{C}$, where encrypt is a one-time oracle.

A.4 PRF-PRNG

Lastly, we use a special type of KDF to glue a (sparse) CKA and a secure messaging protocol together, an abstraction introduced in [ACD19]. Essentially, it says that a KDF can be safely used as both a PRF and a PRNG, a property that trivially holds once instantiating a KDF with a hash function modeled as a random oracle.

Definition 8 (PRF-PRNG) A key derivation function KDF with two input arguments is said to have PRF-PRNG security, if

$$\text{Adv}_{\mathcal{A}}^{\text{PRF-PRNG}}(1^\lambda) := \left| \Pr[\text{Game}_{\mathcal{A}}^{\text{PRF-PRNG}}(1^\lambda)] - \frac{1}{2} \right|$$

is negligible for every PPT $\mathcal{A}$, for the game from Fig. 9.

B Omitted Definitions

B.1 Security of Secure Messaging Protocols

We define correctness and security of a secure messaging (SM) scheme. The combined security and correctness game is depicted in Fig. 10. Recall from Sec. 2.1 that we reuse the notion from [ACD19] adapting it to fine-grained security. To this end, we replace the epoch-based forward-secrecy (FS) and post-compromise (PCS) notion using our notion of *vulnerable message sets*.

$\text{Game}_{\mathcal{A}}^{\text{PRF-PRNG}}(1^\lambda)$	$\text{sample-if-nec}(I)$	$\text{chall-prf}(I)$
1 : $b \xleftarrow{\$} \{0, 1\}$	1 : if $\llbracket I = \perp \rrbracket$ then	1 : req $\llbracket \neg \text{corr} \rrbracket \wedge \llbracket \neg \text{prng} \rrbracket$
2 : $\sigma \xleftarrow{\$} \{0, 1\}^\lambda$	2 : $I \xleftarrow{\$} \mathcal{I}$	2 : $\text{prf} \leftarrow \text{true}$
3 : $\text{corr}, \text{prng}, \text{prf} \leftarrow \text{false}$	3 : $\text{corr} \leftarrow \text{false}$	3 : $(\sigma', R) \leftarrow \text{KDF}(\sigma, I)$
4 : $\mathcal{F}[\cdot] \leftarrow \perp$	4 : return I	4 : if $\llbracket b = 1 \rrbracket$ then
5 : $b' \xleftarrow{\$} \mathcal{A}(1^\lambda)^\mathcal{O}$	$\text{eval}(I)$	5 : $R \leftarrow \text{eval}(I)$
6 : return $\llbracket b = b' \rrbracket$	1 : if $\llbracket \mathcal{F}[I] = \perp \rrbracket$ then	6 : return (σ', R)
$\text{corr}()$	2 : $\mathcal{F}[I] \xleftarrow{\$} \mathcal{R}$	$\text{chall-prng}(I)$
1 : req $\llbracket \neg \text{prf} \rrbracket$	3 : return $\mathcal{F}[I]$	1 : $I \leftarrow \text{sample-if-nec}(I)$
2 : $\text{corr} \leftarrow \text{true}$	$\text{process}(I)$	2 : req $\llbracket \neg \text{corr} \rrbracket \wedge \llbracket \neg \text{prf} \rrbracket$
3 : return σ	1 : $I \leftarrow \text{sample-if-nec}(I)$	3 : $\text{prng} \leftarrow \text{true}$
	2 : $(\sigma, R) \leftarrow \text{KDF}(\sigma, I)$	4 : $(\sigma, R) \leftarrow \text{KDF}(\sigma, I)$
	3 : return R	5 : if $\llbracket b = 1 \rrbracket$ then
		6 : $R \xleftarrow{\$} \mathcal{R}$
		7 : return R

Figure 9: PRF-PRNG security of a two-input KDF. $\mathcal{O}$ denotes the set of oracles $\{\text{process}, \text{chall-prf}, \text{chall-prng}, \text{corr}\}$. If KDF expands to more than two return values, then the first one is σ and the remaining tuple of outputs should be considered R over an appropriate composite space $\mathcal{R}$.

We now provide some additional intuition on the security experiment. The goal of the adversary is to either break confidentiality by distinguish between challenge messages M_0 and M_1 , break authenticity by having a party output a message not sent by the other party, or to break correctness an honest ciphertext not decrypting correctly. Success is measured as the guessing probability exceeding $\frac{1}{2}$, plus the probability of breaking authenticity or correctness.

Security. We assume that for each party $P \in \{A, B\}$, the protocol state contains a field `stp.vuln` containing tuples (P', t, i) meaning that leaking the state to $\mathcal{A}$ would trivially expose the i^{th} message P' sent during epoch t .

Upon each corruption of a party, the currently vulnerable message set is then added to the exposed message set `Exposed`. In addition, if `Send-P` is used with bad randomness, then any newly vulnerable messages are also added to `Exposed`. The game keeps also track of the set of challenges, where again $(P', t, i) \in \text{Challenged}$ means that P' sent a challenge as the i^{th} of epoch t . The game then ensures that `Exposed` and `Challenged` are disjoint as part of `safe-chall` and `safe-corr`, respectively. To break security, the goal of the adversary is to distinguish between challenge messages M_0 and M_1 .

Authenticity. In line with [ACD19], we consider a semi-active adversary Adv that is restricted as follows: As long as neither of the parties is corrupted, the adversary is active and has full control over the network; however, after a corruption, the adversary must behave passively and only deliver sent network traffic until both parties have recovered from the corruption using PCS. (This restriction reflects the limited resilience of the Double Ratchet protocol against active attackers.)

The `Inject-P` oracle allows the adversary to try to break authenticity by having P process an arbitrary ciphertext `ct`. To capture our notion of semi-active adversaries, however, this is only permitted if `safe-inj` deems that both parties have recovered since the last exposure by checking that their receiving epoch and period is past any vulnerable message for the communication direction from $\bar{P}$ to P .

Game_ASM(1^λ) 1: $b \xleftarrow{\$} \{0, 1\}$ 2: $l_K \xleftarrow{\$} \text{SM-Init-KeyGen}(1^\lambda)$ 3: for $P \in \{A, B\}$ 4: $st_P \xleftarrow{\$} \text{SM-Init-P}(l_K)$ 5: $L_{trans}, L_{chall}, L_{comp} \leftarrow \emptyset$ 6: $\text{Exposed}, \text{Challenged} \leftarrow \emptyset$ 7: $b' \xleftarrow{\$} \mathcal{A}(1^\lambda)^{\mathcal{O}()}$ 8: return $\llbracket b = b' \rrbracket$ Chall-A (M ₀ , M ₁ , leak) 1: $\text{rand} \xleftarrow{\$} \mathcal{R}$ 2: req $\llbracket M_0 = M_1 \rrbracket$ 3: $(ct, st_A) \xleftarrow{\$} \text{SM-Send-A}(st'_A, M_b; \text{rand})$ 4: $\text{epoch-mgmt}(A, st'_A, \text{chall}, \text{leak})$ 5: req $\llbracket \text{safe-chall}(A) \rrbracket$ 6: $\text{record} := (A, M_b, ct, st_A.t^{snd}, st_A.i^{snd})$ 7: $L_{trans}, L_{chall}, L_{comp} \stackrel{+}{\leftarrow} \text{record}$ 8: if $\llbracket \neg \text{leak} \rrbracket$ then $\text{rand} \leftarrow \perp$ 9: return (ct, rand)	Inject-A (ct) 1: req $\llbracket (B, \neg, ct, \neg, \neg) \notin L_{trans} \rrbracket \wedge \llbracket \text{safe-inj}() \rrbracket$ 2: $(M', t', i', st'_A) \leftarrow \text{SM-Rec-A}(st'_A, ct)$ 3: $\text{epoch-mgmt}(A, st'_A, \text{receive}, \text{false})$ // Authenticity guarantee 4: if $\llbracket M' \neq \perp \rrbracket$ then 5: assert $\llbracket (B, \neg, \neg, t', i') \in L_{comp} \rrbracket$ 6: if $\llbracket (B, \neg, \neg, t', i') \in L_{trans} \rrbracket$ then 7: $L_{trans}, L_{chall}, L_{comp} \stackrel{+}{\leftarrow} (B, \neg, \neg, t', i')$ 8: return (M', t', i') Send-A (M, leak) 1: $\text{rand} \xleftarrow{\$} \mathcal{R}$ 2: $(ct, st_A) \xleftarrow{\$} \text{SM-Send-A}(st'_A, M; \text{rand})$ 3: $\text{epoch-mgmt}(A, st'_A, \text{send}, \text{leak})$ 4: $\text{record} := (A, M_b, ct, st_A.t^{snd}, st_A.i^{snd})$ 5: $L_{trans} \stackrel{+}{\leftarrow} \text{record}$ 6: if $\llbracket \neg \text{safe-chall}(A) \rrbracket$ then 7: $L_{comp} \stackrel{+}{\leftarrow} \text{record}$ 8: if $\llbracket \neg \text{leak} \rrbracket$ then $\text{rand} \leftarrow \perp$ 9: return (ct, rand)	Receive-A (ct) 1: req $\llbracket (B, \neg, ct, \neg, \neg) \in L_{trans} \rrbracket$ 2: $(M', t', i', st'_A) \leftarrow \text{SM-Rec-A}(st'_A, ct)$ 3: $\text{epoch-mgmt}(A, st'_A, \text{receive}, \text{false})$ 4: $\text{record} := (B, M', ct, t', i')$ // Correctness guarantee 5: assert $\llbracket \text{record} \in L_{trans} \rrbracket$ 6: if $\llbracket \text{record} \in L_{chall} \rrbracket$ then 7: $M' \leftarrow \perp$ 8: $L_{trans}, L_{chall}, L_{comp} \stackrel{+}{\leftarrow} \text{record}$ 9: return (M', t', i') Corr-A () 1: req $\llbracket (B, \neg, \neg, \neg, \neg) \notin L_{chall} \rrbracket \wedge \llbracket \text{safe-corr}(A) \rrbracket$ 2: foreach $(B, M', ct', t', i') \in L_{trans}$ 3: $L_{comp} \stackrel{+}{\leftarrow} (B, M', ct', t', i')$ 4: Exposed $\stackrel{+}{\leftarrow} st'_A.vuln$ 5: return st_A
epoch-mgmt (P, st'_P, act, leak) 1: if $\llbracket \text{act} \in \{\text{send}, \text{chall}\} \rrbracket$ then 2: assert $\llbracket (st'_P.t^{snd}, st'_P.i^{snd}) > (st_P.t^{snd}, st_P.i^{snd}) \rrbracket$ 3: assert $\llbracket (st'_P.t^{rcv}, st'_P.i^{rcv}) = (st_P.t^{rcv}, st_P.i^{rcv}) \rrbracket$ 4: else 5: assert $\llbracket (st'_P.t^{rcv}, st'_P.i^{rcv}) > (st_P.t^{rcv}, st_P.i^{rcv}) \rrbracket$ 6: assert $\llbracket (st'_P.i^{snd}) = (st_P.t^{snd}, st_P.i^{snd}) \rrbracket$ 7: $\text{corr-mgmt}(P, st'_P, \text{act}, \text{leak})$ 8: $st_P \leftarrow st'_P$	corr-mgmt (P, st'_P, act, leak) 1: if $\llbracket \text{act} = \text{chall} \rrbracket$ then 2: Challenged $\stackrel{+}{\leftarrow} (P, st'_P.t^{rcv}, st'_P.i^{rcv})$ 3: if $\llbracket \text{bad-rand}_P \rrbracket$ then 4: $vuln' \leftarrow st'_P.vuln \setminus st_P.vuln$ 5: req $\llbracket vuln' \cap \text{Challenged} = \emptyset \rrbracket$ 6: Exposed $\stackrel{+}{\leftarrow} vuln'$	safe-chall (P) // Non-trivial challenge 1: return $\llbracket (P, st_P.t^{snd}, st_P.i^{snd}) \notin \text{Exposed} \rrbracket$ safe-inj () // Once both parties healed 1: return $\forall (\bar{P}, t, i) \in \text{Exposed} :$ $\llbracket (st_A.t^{rcv}, st_A.i^{rcv}) > (t, i) \rrbracket$ $\wedge \llbracket (st_B.t^{rcv}, st_B.i^{rcv}) > (t, i) \rrbracket$ safe-corr (P) // No challenge revealed 1: return $\llbracket st_P.vuln \cap \text{Challenged} = \emptyset \rrbracket$

Figure 10: The security and correctness game for a secure messaging scheme. The oracles for B are defined analogously and $\mathcal{O}()$ denotes the set of oracles $\{\text{Send-P}(), \text{Receive-P}(), \text{Inject-P}(), \text{Chall-P}(), \text{Corr-P}()\}_{P \in \{A, B\}}$. The text highlighted in blue denotes the main differences to the respective game from [ACD19]. We use lexicographic ordering for epoch-period pairs, i.e., $(t, i) > (t', i') \Leftrightarrow t > t' \vee (t = t' \wedge i > i')$.

If permitted, ct is then delivered to P using Receive-P with the following outcome:

- If $\bar{P}$ sent a message with epoch t' and i' that has not yet been delivered to P, i.e., is still in L_{trans} , then the injection is considered *benign*, and the adversary is given the output. Afterward, the message is removed from the set of messages in transmission.
- Otherwise, the injection is considered *preventable*, and the adversary wins the game.

Correctness. Finally, correctness is enforced by recording each message sent in L_{trans} and then checking in Receive-P that the correct plaintext as well as epoch and period numbers are output.

Distribution $\mathcal{D}_{A,0}^{\text{KeyBaseSim}}$	
1 :	$(\widehat{\text{ek}}_A, \widehat{\text{dk}}_A) \xleftarrow{\$} \widehat{\mathcal{D}}_{\text{RKeyGen-A}}$
2 :	return $(\widehat{\text{ek}}_A, \widehat{\text{dk}}_A)$
Distribution $\mathcal{D}_{A,1}^{\text{KeyBaseSim}}$	
1 :	$(\text{ek}_A, \text{dk}_A) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-A}}$
2 :	$(\widehat{\text{ek}}_A, \widehat{\text{dk}}_A, _) \xleftarrow{\$} \text{RSimKey-A}_1(\text{ek}_A, \text{dk}_A)$
3 :	return $(\widehat{\text{ek}}_A, \widehat{\text{dk}}_A)$

Figure 11: Base Key simulatability.

B.2 Security for ratcheting KEMs

Definition 9 We say a ratcheting RKEM is correct if it satisfies two properties. The first property, correctness with updated keys, requires the following. If $(\text{ek}_A, \text{dk}_A) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-A}}$, $(\widehat{\text{ek}}_B, \widehat{\text{dk}}_B) \xleftarrow{\$} \widehat{\mathcal{D}}_{\text{RKeyGen-B}}$, $(\text{ct}_A, K, \widehat{\text{dk}}_A) \xleftarrow{\$} \text{REnc-A}(\widehat{\text{ek}}_B, \text{dk}_A)$, and $(K', \widehat{\text{ek}}_A) \xleftarrow{\$} \text{RDec-B}(\widehat{\text{dk}}_B, \text{ct}_A, \text{ek}_A)$ then it holds that

$$\Pr[K = K'] = 1 - \text{negl}(\lambda).$$

We require the above to hold with the roles of parties A and B swapped. We denote the marginal distribution of $(\widehat{\text{ek}}_A, \widehat{\text{dk}}_A)$ generated through the above process as $\mathcal{D}'_{\text{RKeyGen-A}}$, and define $\mathcal{D}'_{\text{RKeyGen-B}}$ similarly. The second property, correctness of update key distribution, then requires that $\mathcal{D}'_{\text{RKeyGen-P}}$ is statistically close to $\widehat{\mathcal{D}}_{\text{RKeyGen-P}}$ for $P \in \{A, B\}$.

Regarding security, a ratcheting KEM must satisfy FS-IND-CPA security, as well as the notion of *ratchet simulatability*.

Definition 10 (FS-IND-CPA Security) We say a ratcheting KEM Π_{RKEM} is forward-secure IND-CPA (FS-IND-CPA) secure if the advantages $\text{Adv}_{\mathcal{A}}^{\text{FS-IND-CPA-A}}(1^\lambda)$ and $\text{Adv}_{\mathcal{A}}^{\text{FS-IND-CPA-B}}$ defined as follows are negligible.

$$\text{Adv}_{\mathcal{A}}^{\text{FS-IND-CPA-A}}(1^\lambda) := \left| \Pr \left[\begin{array}{c} b \xleftarrow{\$} \{0, 1\} \\ b' \xleftarrow{\$} \mathcal{A}(\text{ek}_A, \widehat{\text{ek}}_A, \text{ek}_B, \text{ct}_A, \widehat{\text{dk}}_A, K_b) \end{array} : b = b' \right] - \frac{1}{2} \right|,$$

where $K_1 \xleftarrow{\$} \mathcal{K}$, $(\text{ek}_A, \text{dk}_A) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-A}}$, $(\widehat{\text{ek}}_B, \widehat{\text{dk}}_B) \xleftarrow{\$} \widehat{\mathcal{D}}_{\text{RKeyGen-B}}$, $(\text{ct}_A, K_0, \widehat{\text{dk}}_A) \xleftarrow{\$} \text{REnc-A}(\widehat{\text{ek}}_B, \text{dk}_A)$, and $(\cdot, \widehat{\text{ek}}_A) \xleftarrow{\$} \text{RDec-B}(\widehat{\text{dk}}_B, \text{ct}_A, \text{ek}_A)$.

$\text{Adv}_{\mathcal{A}}^{\text{FS-IND-CPA-B}}$ is defined analogously with the roles of parties A and B swapped. We denote $\text{Adv}_{\mathcal{A}}^{\text{FS-IND-CPA}} := \max_{P \in \{A, B\}} (\text{Adv}_{\mathcal{A}}^{\text{FS-IND-CPA-P}}(1^\lambda))$.

Definition 11 (Ratchet Simulatability) For $b \in \{0, 1\}$, let $\mathcal{D}_{A,b}^{\text{KeyBaseSim}}$ be the distributions as defined in Fig. 11, $\mathcal{D}_{A,b}^{\text{KeyUpdSim}}$ be the distributions as defined in Fig. 12, and $\mathcal{D}_{B,b}^{\text{CtxtSim}}$ be the distributions as defined in Fig. 13. Moreover, let $\mathcal{D}_{B,b}^{\text{KeyBaseSim}}$ and $\mathcal{D}_{B,b}^{\text{KeyUpdSim}}$ and $\mathcal{D}_{B,b}^{\text{CtxtSim}}$ be defined analogously with the roles of the two parties swapped in the respective experiments. We say a ratcheting KEM Π_{RKEM} is ratchet simulatable if there exists efficient simulators $(\text{RSimKey-P}_1, \text{RSimKey-P}_2, \text{RSimCtxt-P})_{P \in \{A, B\}}$ such that the advantage against base-key simulatability

$$\text{Adv}_{\mathcal{A}}^{\text{KeyBaseSim-P}}(1^\lambda)$$

Distribution $\mathcal{D}_{A,0}^{\text{KeyUpdSim}}$	
1 :	$(ek_B, dk_B) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-B}}\{\text{rand}_0\}$
2 :	$(\widehat{ek}_B, \widehat{dk}_B, aux_0) \xleftarrow{\$} \text{RSimKey-B}_1(ek_B, dk_B)$
3 :	$(ek_A, dk_A) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-A}}\{\text{rand}_1\}$
4 :	$(ct_B, K, \widehat{dk}_A) \xleftarrow{\$} \text{REnc-A}(\widehat{ek}_B, dk_A; \text{rand}_2)$
5 :	$(K', \widehat{ek}_A) \xleftarrow{\$} \text{RDec-B}(\widehat{dk}_B, ct_A, ek_A)$
6 :	return $\left((\widehat{ek}_B, \widehat{dk}_B), (\widehat{ek}_A, \widehat{dk}_A), ct_A, K, K', \right.$ $\left. aux_0, \text{rand}_0, \text{rand}_1, \text{rand}_2 \right)$
Distribution $\mathcal{D}_{A,1}^{\text{KeyUpdSim}}$	
1 :	$(ek_B, dk_B) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-B}}\{\text{rand}_0\}$
2 :	$(\widehat{ek}_B, \widehat{dk}_B, aux_0) \xleftarrow{\$} \text{RSimKey-B}_1(ek_B, dk_B)$
3 :	$(ek_A, dk_A) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-A}}\{\text{rand}_1\}$
4 :	$(\widehat{ek}_A, \widehat{dk}_A, aux_1) \xleftarrow{\$} \text{RSimKey-A}_1(ek_A, dk_A)$
5 :	$(ct_B, K, K', \text{rand}_2) \xleftarrow{\$} \text{RSimKey-A}_2(\widehat{ek}_B, \widehat{dk}_B, aux_1)$
6 :	return $\left((\widehat{ek}_B, \widehat{dk}_B), (\widehat{ek}_A, \widehat{dk}_A), ct_A, K, K', \right.$ $\left. aux_0, \text{rand}_0, \text{rand}_1, \text{rand}_2 \right)$

Figure 12: Updated Key simulatability. The text highlighted in blue denotes the main differences between the two distributions. Recall $D\{\text{rand}\}$ denotes the process of sampling from the distribution D with randomness rand . Above, we assume rand (except for those output by RSimKey-A_2) to be distributed uniformly over their respective domain.

$$:= \left| \Pr \left[\begin{array}{c} b \xleftarrow{\$} \{0, 1\}, x \xleftarrow{\$} \mathcal{D}_{P,b}^{\text{KeyBaseSim}}, \\ b' \xleftarrow{\$} \mathcal{A}(x) \end{array} : b' = b \right] - \frac{1}{2} \right|,$$

the advantage against updated key simulatability

$$\text{Adv}_{\mathcal{A}}^{\text{KeyUpdSim-P}}(1^\lambda) := \left| \Pr \left[\begin{array}{c} b \xleftarrow{\$} \{0, 1\}, x \xleftarrow{\$} \mathcal{D}_{P,b}^{\text{KeyUpdSim}}, \\ b' \xleftarrow{\$} \mathcal{A}(x) \end{array} : b' = b \right] - \frac{1}{2} \right|,$$

and the advantage against ciphertext simulatability

$$\text{Adv}_{\mathcal{A}}^{\text{CtxtSim-P}}(1^\lambda) := \left| \Pr \left[\begin{array}{c} b \xleftarrow{\$} \{0, 1\}, x \xleftarrow{\$} \mathcal{D}_{P,b}^{\text{CtxtSim}}, \\ b' \xleftarrow{\$} \mathcal{A}(x) \end{array} : b' = b \right] - \frac{1}{2} \right|$$

for both $P \in \{A, B\}$ are negligible. We denote $\text{Adv}_{\mathcal{A}}^{\text{KeyBaseSim}} := \max_{P \in \{A, B\}} \left(\text{Adv}_{\mathcal{A}}^{\text{KeyBaseSim-P}}(1^\lambda) \right)$, $\text{Adv}_{\mathcal{A}}^{\text{KeyUpdSim}} := \max_{P \in \{A, B\}} \left(\text{Adv}_{\mathcal{A}}^{\text{KeyUpdSim-P}}(1^\lambda) \right)$ and $\text{Adv}_{\mathcal{A}}^{\text{CtxtSim}} := \max_{P \in \{A, B\}} \left(\text{Adv}_{\mathcal{A}}^{\text{CtxtSim-P}}(1^\lambda) \right)$.

Distribution $\mathcal{D}_{B,0}^{\text{CtxtSim}}$	
1 :	$(ek_A, dk_A) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-A}}\{\text{rand}\}$
2 :	$(\widehat{ek}_A, \widehat{dk}_A, \text{aux}) \xleftarrow{\$} \text{RSimKey-A}_1(ek_A, dk_A)$
3 :	$(ek_B, dk_B) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-B}}$
4 :	$(ct_A, K, \widehat{dk}_B) \xleftarrow{\$} \text{REnc-B}(\widehat{ek}_A, dk_B)$
5 :	$(K', \widehat{ek}_B) \xleftarrow{\$} \text{RDec-A}(\widehat{dk}_A, ct_B, ek_B)$
6 :	return $(\text{aux}, \text{rand}, (\widehat{ek}_A, \widehat{dk}_A), ct_B,$ $(ek_B, \widehat{ek}_B), (K, K'))$
Distribution $\mathcal{D}_{B,1}^{\text{CtxtSim}}$	
1 :	$(ek_A, dk_A) \xleftarrow{\$} \mathcal{D}_{\text{RKeyGen-A}}\{\text{rand}\}$
2 :	$(\widehat{ek}_A, \widehat{dk}_A, \text{aux}) \xleftarrow{\$} \text{RSimKey-A}_1(ek_A, dk_A)$
3 :	$(\widehat{ek}_B, \widehat{dk}_B) \xleftarrow{\$} \widehat{\mathcal{D}}_{\text{RKeyGen-B}}$
4 :	$(ct_A, ek_B, K, K') \xleftarrow{\$} \text{RSimCtxt-B}(\widehat{ek}_B, \widehat{ek}_A, \widehat{dk}_A)$
5 :	return $(\text{aux}, \text{rand}, (\widehat{ek}_A, \widehat{dk}_A), ct_A,$ $(ek_B, \widehat{ek}_B), (K, K'))$

Figure 13: Ciphertext simulatability. The text highlighted in blue denotes the main differences between the two distributions.

C Proof of Theorem 1

We provide a proof sketch of Sec. 3.2, focusing on the main differences to the proof of the Double Ratchet [ACD19]. In particular, note that the symmetric ratcheting layer and the AEAD encryption are essentially the same as in those protocols. As a result, we omit the reduction to the underlying PRF-PRNG security and AEAD security and focus on the effects of using SCKA instead of CKA instead.

Correctness. Note that the main difference to the Double Ratchet [ACD19] is the use of the SCKA. Ignoring injections for now, we observe however that by correctness of the SCKA both parties will mix in the same sequence of keys into their chains. This therefore produces matching sending and receiving chains $KS[t]$ and $KR[t]$ for each epoch t . Correctness of the SCKA moreover ensures that CKA-Send-A and CKA-Rec-B, run as part of SM-Send-A and SM-Rec-B, will produce the matching epoch numbers $t_A = t_B$ that A and B will use to encrypt/decrypt under. Moreover, correctness ensures that whenever B receives ρ as part of M , he will know all keys up to t_B , ensuring the availability of the decryption key.

Confidentiality. The secrecy of messages is implied by the security of the AEAD. For more details about the security of the key derivation and the encryption we refer to [ACD19]. (Note that deriving both a sending and receiving chain per epoch does not affect security. See [DJK⁺25] for additional details.)

As such, it remains to convince ourselves that there are no trivial distinguishing attacks by having the key for a challenge leaked. To this end, observe that `stp.vuln` as defined in Sec. 3.2 directly mirrors the keys kept as part of the state:

- A corruption exposes the SCKA state $\overline{\text{stp}}^{\text{CKA}}$. This in turn will reveal the keys for epochs $t \in \overline{\text{stp.vuln}}$ and therefore sending and receiving chains for those epochs.

- A corruption will furthermore reveal any keys stored in $K_{CKA}[t]$ for $t > t_P^{cur}$. By our definition of $st_P.vuln$, all messages within those epochs are considered vulnerable.
- Moreover, a corruption will reveal $KS[t]$ and $KR[t]$ for all epochs where stored. For $t > t_P^{snd}$ and $t > t_P^{rcv}$, respectively, this is again covered by $st_P.vuln$ treating the entire epoch as vulnerable for sending or receiving, respectively. For the current sending and receiving epochs, all future messages are considered vulnerable as $KS[t_P^{snd}]$ and $KR[t_P^{rcv}]$ are updated in a forward secure manner using the KDF.
- Finally, a corruption exposed **StoredKeys**. This part of the state exactly contains the decryption keys for pending out-of-order message not yet received. Again, this matches $st_P.vuln$ considering those incoming messages as insecure, disallowing challenges.

Authenticity. Whenever **safe-inj** is true, we know that no current (or future key) of P is vulnerable. Therefore, if the ciphertext M is with respect to an epoch $t \geq t_P^{rcv}$, then the unforgeability of the AEAD will cause M to be rejected.

On the other hand, for epochs $t < t_P^{rcv}$, the KR_{aead} must come from **StoredKeys**. Note however, that the definition of Δ_{slack} ensures that P has learnt the number of messages sent for all prior epochs, in particular epoch t . Therefore, **StoredKeys** will only store keys for a period i such that $(\bar{P}, _, _, t, i) \in L_{trans}$. Finally, note that for epochs $t < t_P^{rcv}$ the SCKA message ρ part of M does not affect the SCKA state (or the keys mixed into the chain).

D Formal Protocol Descriptions

Protocol Opp-UniKEM-CKA. We provide pseudo-code for Opp-UniKEM-CKA in Fig. 14. Party A's state contains the following values that are all defined with respect to the epoch the parties currently exchange key material for, i.e.,

dk_A	decapsulation key of A
ek_A	encapsulation key of A
ct_0	offline ciphertext
t	epoch
i_{ch}	chunk counter (w.r.t. ek_A)
L_{ch}	chunk list (w.r.t. ct_B)
act	acknowledgments

User B's state contains the following.

ek_A	encapsulation key of A
$ct_B = (ct_0, ct_1)$	ciphertext created by B
st_{ct}	ciphertext state
t	epoch
i_{ch}	chunk counter (w.r.t. ct_B)
L_{ch}	chunk list (w.r.t. ek_A)
act	acknowledgments

Protocol Opp-BiKEM-CKA. We provide pseudo-code for Opp-BiKEM-CKA in Fig. 15 (algorithms of user A) and Fig. 16 (algorithms of user B). The protocol does not use initialization keys, i.e., $\perp = l_{CKA} \leftarrow CKA\text{-Init-KeyGen}$. Party P's state contains an array of acknowledgments **ACK** as well as responder state st_{res} and requestor state st_{req} which contain the variables relevant to P acting as responder or requestor, respectively. The parties keep track of responder epoch t_{res-P} and requestor epoch t_{req-P} corresponding to the epoch

CKA-Init-KeyGen(1^λ) 1: return $l_{CKA} := (\perp)$	CKA-Send-A(st_A) 1: parse $(dk_A, ek_A, ct_0, t, i_{ch}, L_{ch}, act) \leftarrow st_A$ 2: $ch \leftarrow \perp$ 3: if $\llbracket dk_A = \perp \rrbracket$ 4: $(ek_A, dk_A) \leftarrow \text{KeyGen}$ 5: $i_{ch} \leftarrow 0$ 6: if $\neg \llbracket act.ek-rec \rrbracket$ 7: $i_{ch} += 1$ 8: $ch \leftarrow \text{Encode}(ek_A, i_{ch})$ 9: $\rho \leftarrow (ch, act, t)$ 10: $st_A \leftarrow (dk_A, ek_A, ct_0, t, i_{ch}, L_{ch}, act)$ 11: return $((\perp, \perp), \rho, t - 1, st_A)$	CKA-Send-B(st_B) 1: parse $(ek_A, ct_0, ct_1, st_{ct}, t, i_{ch}, L_{ch}, act) \leftarrow st_B$ 2: $I_B \leftarrow \perp, t_{I_B} \leftarrow \perp, ch \leftarrow \perp$ 3: if $\llbracket ct_0 = \perp \rrbracket$ 4: $(st_{ct}, ct_0) \leftarrow \text{Enc.Off}(1^\lambda)$ 5: $i_{ch} \leftarrow 0$ 6: if $\neg \llbracket act.ct_0-rec \rrbracket$ 7: $i_{ch} += 1, b \leftarrow 0$ 8: $ch \leftarrow \text{Encode}(ct_0, i_{ch})$ 9: elseif $\llbracket ek_A \neq \perp \rrbracket$ 10: if $\llbracket ct_1 = \perp \rrbracket$ 11: $(ct_1, I_B) \leftarrow \text{Enc.On}(st_{ct}, ek_A)$ 12: $t_{I_B} \leftarrow t, i_{ch} \leftarrow 0$ 13: $i_{ch} += 1, b \leftarrow 1$ 14: $ch \leftarrow \text{Encode}(ct_1, i_{ch})$ 15: $\rho \leftarrow (ch, act, t, b)$ 16: $st_B \leftarrow (ek_A, ct_0, ct_1, st_{ct}, t, i_{ch}, L_{ch}, act)$ 17: return $((t_{I_B}, I_B), \rho, t - 1, st_B)$
CKA-Init-A(l_{CKA}) 1: $(act.ek-rec, act.ct_0-rec) \leftarrow (false, false)$ 2: $st_A \leftarrow (\perp, \perp, \perp, \perp, 1, 0, \emptyset, act)$ 3: return st_B	CKA-Rec-A((st_A, ρ)) 1: parse $(dk_B, ek_B, ct_0, t, i_{ch}, L_{ch}, act) \leftarrow st_A$ 2: parse $(ch, act', t', b) \leftarrow \rho$ 3: $I_B \leftarrow \perp, t_{I_B} \leftarrow \perp$ 4: if $\llbracket t = t' \rrbracket$ 5: if $\llbracket ct_0 = \perp \rrbracket \wedge \llbracket b = 0 \rrbracket$ 6: $L_{ch} \xleftarrow{\cup} \{ch\}$ 7: $ct_0 \leftarrow \text{Decode}(L_{ch})$ 8: if $\llbracket ct_0 \neq \perp \rrbracket$ 9: $act.ct_0-rec \leftarrow true$ 10: $L_{ch} \leftarrow \emptyset$ 11: elseif $\llbracket b = 1 \rrbracket$ 12: $L_{ch} \xleftarrow{\cup} \{ch\}$ 13: $ct_1 \leftarrow \text{Decode}(L_{ch})$ 14: if $\llbracket ct_1 \neq \perp \rrbracket$ 15: $I_B \leftarrow \text{Dec}(dk_A, (ct_0, ct_1))$ 16: $t_{I_B} \leftarrow t$ 17: $t += 1$ 18: $L_{ch} \leftarrow \emptyset$ 19: $(dk_A, ek_A, ct_0) \leftarrow (\perp, \perp, \perp)$ 20: $(act.ek-rec, act.ct_0-rec) \leftarrow (false, false)$ 21: if $\llbracket act'.ek-rec \rrbracket$ 22: $act.ek-rec \leftarrow true$ 23: $st_A \leftarrow (dk_A, ek_A, ct_0, t, i_{ch}, L_{ch}, act)$ 24: return $((I_B, t_{I_B}), t' - 1, st_A)$	CKA-Rec-B(st_B, ρ) 1: parse $(ek_A, ct_0, ct_1, st_{ct}, i_{ch}, L_{ch}, act) \leftarrow st_B$ 2: parse $(ch, act', t') \leftarrow \rho$ 3: if $\llbracket t < t' \rrbracket$ 4: $t += 1$ 5: $(ct_0, ct_1, st_{ct}) \leftarrow (\perp, \perp, \perp)$ 6: $(ek_A, L_{ch}) \leftarrow (\perp, \perp)$ 7: $(act.ek-rec, act.ct_0-rec) \leftarrow (false, false)$ 8: if $\llbracket t = t' \rrbracket \wedge \llbracket ek_A = \perp \rrbracket$ 9: $L_{ch} \xleftarrow{\cup} \{ch\}$ 10: $ek_A \leftarrow \text{Decode}(L_{ch})$ 11: $act.ek-rec \leftarrow \llbracket ek_A \neq \perp \rrbracket$ 12: if $\llbracket act'.ct_0-rec \rrbracket$ 13: $act.ct_0-rec \leftarrow true$ 14: $st_B \leftarrow (ek_A, ct_0, ct_1, st_{ct}, t, i_{ch}, L_{ch}, act)$ 15: return $((\perp, \perp), t - 1, st)$
CKA-Init-B(l_{CKA}) 1: $(act.ek-rec, act.ct_0-rec) \leftarrow (false, false)$ 2: $st_B \leftarrow (\perp, \perp, \perp, \perp, 1, 0, \emptyset, act)$ 3: return st_A		
$st_A.vuln$ 1: if $\llbracket dk_A \neq \perp \rrbracket$ 2: $st_A.vuln = \{t\}$ 3: else $st_A.vuln = \emptyset$		
$st_B.vuln$ 1: if $\llbracket st_{ct} \neq \perp \rrbracket$ 2: $st_B.vuln = \{t\}$ 3: else $st_B.vuln = \emptyset$		

Figure 14: Opp-UniKEM-CKA constructed from offline-online KEM $KEM = (\text{KeyGen}, \text{Enc.Off}, \text{Enc.Off}, \text{Dec})$ and erasure code $(\text{Encode}, \text{Decode})$.

of keys I_P and $I_{\bar{P}}$ currently being transmitted and received, respectively. In more detail, st_{res} contains the following.

$t_{\text{res-P}}$	responder epoch
$\text{EK}_{\bar{P}}$	sparse array of $\bar{P}$'s encapsulation keys
ct_P	ciphertext created by P
i_{ch}	chunk counter

Further, requestor state st_{req} contains

$t_{\text{req-P}}$	requestor epoch
DK_P	sparse array of P 's decapsulation keys
ek_A	A 's encapsulation key
L_{ch}	list of chunks

Protocol Opp-RKEM-CKA. We provide pseudo-code for Opp-RKEM-CKA in Fig. 17. The parties' states are structured similarly to Opp-BiKEM-CKA, consisting of a responder state and a requestor state. As additional values the parties keep track of RKEM parameters par set up by CKA-Init-KeyGen, and the requestor state contains an array UP keeping track whether the parties decapsulation keys were already updated, which is used in the computation of vulnerable epochs set $\text{st}_P.\text{vuln}$.

E Description of Non-opportunistic Protocol Variants

As discussed in Sec. 4.1, any CKA can be compiled into a bandwidth constraint SCKA variant by making use of chunking (but not opportunistic sending). To not introduce additional notation we simply denote the SCKAs considered in our experiments by the same name as their CKA counterparts, i.e., UniKEM-CKA, BiKEM-CKA, and RKEM-CKA.

E.1 UniKEM-CKA

We provide a description of UniKEM-CKA an SCKA variant of the unidirectional CKA.

Protocol description. The protocol is defined with respect to KEM KEM and erasure code (Encode, Decode). For epoch t user CKA-Send-A generates key-pair $(\text{ek}_A^t, \text{dk}_A^t) \leftarrow \text{KeyGen}(1^\lambda)$ and starts sending chunks of ek_A^t . During this phase CKA-Send-B does not send cryptographic material, its protocol messages only containing acknowledgments indicating whether the encapsulation key was fully transmitted.

After the transmission completes CKA-Send-B generates (ct_B^t, I_B^t) and starts sending chunks of ct_B^t , now CKA-Send-A being silent except for sending acknowledgments. After recovering the ciphertext from the sent chunks CKA-Rec-A computes the epoch key, deletes dk_A^t , and the procedure starts anew.

Sending/receiving epoch and vulnerable epochs. In a phase where the parties are exchanging cryptographic material pertaining to epoch t we have $t_A^{\text{snd}} = t - 1 = t_B^{\text{snd}}$. Protocol messages include the epoch t of the sent cryptographic material allowing the processing party to compute the matching t_A^{cv} or t_B^{cv} respectively.

Regarding the vulnerable epoch set note that the only secret stored in the users' state is dk_A^t . We thus have $\text{st}_B.\text{vuln} = \emptyset$ and

$$\text{st}_A.\text{vuln} = \begin{cases} \{t\} & \text{if } \llbracket \text{dk}_A^t \neq \perp \rrbracket \\ \emptyset & \text{else} \end{cases}.$$

CKA-Init-A(I_{CKA}) <pre> 1 : initialize $EK_B[t] = \perp, DK_A[t] = \perp$ for all t 2 : initialize $(ACK[t].ek-rec, ACK[t].ct-rec) = (false, false)$ for all t 3 : $(ACK[-1].ct-rec, ACK[0].ct-rec) \leftarrow (true, true)$ 4 : $st_{res} \leftarrow (-1, EK_B, \perp, 0)$ 5 : $st_{req} \leftarrow (0, DK_A, \perp, \emptyset)$ 6 : return $(st_{res}, st_{req}, ACK)$ </pre> CKA-Send-A(st_{res}, st_{req}, ACK) <pre> 1 : parse $(t_{res-A}, EK_B, ct_A, i_{ch}) \leftarrow st_{res}$ // responder state 2 : parse $(t_{req-A}, DK_A, ek_A, L_{ch}) \leftarrow st_{req}$ // requestor state 3 : $I_A \leftarrow \perp, t_{I_A} \leftarrow \perp, ch \leftarrow \perp, b \leftarrow \perp$ 4 : if $\llbracket ek_A = \perp \rrbracket \wedge \llbracket ACK[t_{res-A}].ct-rec \rrbracket \wedge \llbracket ACK[t_{res-A} + 1].ct-rec \rrbracket$ 5 : $t_{res-A} += 2$ 6 : $i_{ch} \leftarrow 0$ 7 : $(dk_A, ek_A) \leftarrow KeyGen(1^\lambda)$ 8 : $DK_A[t_{res-A} + 1] \leftarrow dk_A$ 9 : if $\neg \llbracket ACK[t_{res-A} + 1].ek-rec \rrbracket$ // ek_A not yet received by B 10 : $i_{ch} += 1$ 11 : $b \leftarrow 0$ 12 : $ch \leftarrow Encode(ek_A, i_{ch})$ 13 : elseif $\neg \llbracket ACK[t_{res-A}].ct-rec \rrbracket$ // ct_A not yet received by B 14 : if $\llbracket ct_A = \perp \rrbracket \wedge \llbracket ACK[t_{res-A}].ek-rec \rrbracket$ 15 : $t_{I_A} \leftarrow t_{res-A} \quad i_{ch} \leftarrow 0$ 16 : $(ct_A, I_A) \leftarrow Enc(EK_B[t_{res-A}])$ 17 : $i_{ch} += 1$ 18 : $b \leftarrow 1$ 19 : $ch \leftarrow Encode(ct_A, i_{ch})$ 20 : $t_A^{snd} \leftarrow \max(t : \llbracket ACK[t].ct-rec \rrbracket \wedge \llbracket ACK[t-1].ct-rec \rrbracket)$ 21 : $act \leftarrow (ACK[t_{req-A} + 1].ek-rec, ACK[t_{req-A}].ct-rec)$ 22 : $\rho \leftarrow (ch, t_{res-A}, t_{req-A}, t_A^{snd}, act, b)$ 23 : $st_{res} \leftarrow (t_{res-A}, EK_B, ct_A, i_{ch})$ 24 : $st_{req} \leftarrow (t_{req-A}, DK_A, ek_A, L_{ch})$ 25 : return $((t_{I_A}, I_A), \rho, t_A^{snd}, (st_{res}, st_{req}, ACK))$ </pre> $st_A.vuln$ <pre> 1 : $st_A.vuln = \{t : DK_A[t] \neq \perp\}$ </pre>	CKA-Rec-A($(st_{res}, st_{req}), \rho$) <pre> 1 : parse $(t_{res-A}, EK_B, ct_A, i_{ch}) \leftarrow st_{res}$ 2 : parse $(t_{req-A}, DK_A, ek_A, L_{ch}) \leftarrow st_{req}$ 3 : parse $(ch, t_{res-B}, t_{req-B}, t_B^{snd}, (act.ek-rec, act.ct-rec), b) \leftarrow \rho$ 4 : $I_B \leftarrow \perp, t_{I_B} \leftarrow \perp$ 5 : if $\llbracket act.ct-rec \rrbracket$ // incorporate acknowledgments 6 : $ACK[t_{req-B}].ct-rec \leftarrow true$ 7 : if $\llbracket act.ek-rec \rrbracket$ 8 : $ACK[t_{req-B} - 1].ek-rec \leftarrow true$ 9 : if $\llbracket t_{res-B} < t_{req-A} \rrbracket$ // outdated message 10 : break 11 : elseif $\llbracket t_{res-B} > t_{req-A} \rrbracket$ // first message of new epoch 12 : $t_{req-A} += 2$ 13 : if $\llbracket EK[t_{req-A} - 1] = \perp \rrbracket \wedge \llbracket b = 0 \rrbracket$ // ek_A not yet recovered 14 : $L_{ch} \leftarrow \bigcup \{ch\}$ 15 : $ek_B \leftarrow Decode(L_{ch})$ 16 : if $ek_B \neq \perp$ 17 : $L_{ch} \leftarrow \emptyset$ 18 : $EK_B[t_{req-A} - 1] \leftarrow ek_B$ 19 : $ACK[t_{req-A} - 1].ek-rec \leftarrow true$ 20 : elseif $\neg \llbracket ACK[t_{req-A}].ct-rec \rrbracket \wedge \llbracket b = 1 \rrbracket$ // ct_B not yet recovered 21 : $L_{ch} \leftarrow \bigcup \{ch\}$ 22 : $ct_A \leftarrow Decode(L_{ch})$ 23 : if $ct_B \neq \perp$ 24 : $L_{ch} \leftarrow \emptyset$ 25 : $ACK[t_{req-A}].ct-rec \leftarrow true$ 26 : $t_{I_B} \leftarrow t_{req-A}$ 27 : $I_B \leftarrow Dec(ct_B, DK_A[t_{req-A}])$ 28 : $DK_A[t_{req-A}] \leftarrow \perp$ 29 : if $\llbracket ACK[t_{res-A}].ct-rec \rrbracket$ // delete fully delivered variables 30 : $ct_A \leftarrow \perp$ 31 : if $\llbracket ACK[t_{res-A} + 1].ek-rec \rrbracket$ 32 : $ek_A \leftarrow \perp$ 33 : $st_{res} \leftarrow (t_{res-A}, EK_B, ct_A, i_{ch})$ 34 : $st_{req} \leftarrow (t_{req-A}, DK_A, ek_A, L_{ch})$ 35 : return $((t_{I_B}, I_B), t_B^{snd}, (st_{res}, st_{req}, ACK))$ </pre>
---	--

Figure 15: Opp-BiKEM-CKA constructed from standard KEM KEM and erasure code (Encode, Decode). Algorithms of user A.

CKA-Init-B (I_{CKA}) <pre> 1 : initialize $EK_A[t] = \perp, DK_B[t] = \perp$ for all t 2 : initialize $(ACK[t].ek-rec, ACK[t].ct-rec) = (false, false)$ for all t 3 : $(ACK[-1].ct-rec, ACK[0].ct-rec) \leftarrow (true, true)$ 4 : $st_{res} \leftarrow (0, EK_A, \perp, 0)$ 5 : $st_{req} \leftarrow (-1, DK_B, \perp, \emptyset)$ 6 : return $(st_{res}, st_{req}, ACK)$ </pre>	CKA-Rec-B ($(st_{res}, st_{req}), \rho$) <pre> 1 : parse $(t_{res-B}, EK_A, ct_B, i_{ch}) \leftarrow st_{res}$ 2 : parse $(t_{req-B}, DK_B, ek_B, L_{ch}) \leftarrow st_{req}$ 3 : parse $(ch, t_{res-A}, t_{req-A}, t_A^{snd}, (act.ek-rec, act.ct-rec), b) \leftarrow \rho$ 4 : $I_A \leftarrow \perp, t_{I_A} \leftarrow \perp$ 5 : if $[act.ct-rec]$ // incorporate acknowledgments 6 : $ACK[t_{req-A}].ct-rec \leftarrow true$ 7 : if $[act.ek-rec]$ 8 : $ACK[t_{req-A} + 1].ek-rec \leftarrow true$ 9 : if $[t_{res-A} < t_{req-B}]$ // outdated message 10 : break 11 : elseif $[t_{res-A} > t_{req-B}]$ // first message of new epoch 12 : $t_{req-B} += 2$ 13 : if $[EK[t_{req-B} + 1] = \perp] \wedge [b = 0]$ // ek_A not yet recovered 14 : $L_{ch} \leftarrow \bigcup \{ch\}$ 15 : $ek_A \leftarrow Decode(L_{ch})$ 16 : if $ek_A \neq \perp$ 17 : $L_{ch} \leftarrow \emptyset$ 18 : $EK_A[t_{req-B} + 1] \leftarrow ek_A$ 19 : $ACK[t_{req-B} + 1].ek-rec \leftarrow true$ 20 : elseif $\neg[ACK[t_{req-B}].ct-rec] \wedge [b = 1]$ // ct_A not yet recovered 21 : $L_{ch} \leftarrow \bigcup \{ch\}$ 22 : $ct_A \leftarrow Decode(L_{ch})$ 23 : if $ct_A \neq \perp$ 24 : $L_{ch} \leftarrow \emptyset$ 25 : $ACK[t_{req-B}].ct-rec \leftarrow true$ 26 : $t_{I_A} \leftarrow t_{req-B}$ 27 : $I_A \leftarrow Dec(ct_A, DK_B[t_{req-B}])$ 28 : $DK_B[t_{req-B}] \leftarrow \perp$ 29 : if $[ACK[t_{res-B}].ct-rec]$ // delete fully delivered variables 30 : $ct_B \leftarrow \perp$ 31 : if $[ACK[t_{res-B} - 1].ek-rec]$ 32 : $ek_B \leftarrow \perp$ 33 : $st_{res} \leftarrow (t_{res-B}, EK_A, ct_B, i_{ch})$ 34 : $st_{req} \leftarrow (t_{req-B}, DK_B, ek_B, L_{ch})$ 35 : return $((t_{I_A}, I_A), \rho, t_A^{snd}, (st_{res}, st_{req}, ACK))$ </pre>
CKA-Send-B (st_{res}, st_{req}, ACK) <pre> 1 : parse $(t_{res-B}, EK_A, ct_B, i_{ch}) \leftarrow st_{res}$ // responder state 2 : parse $(t_{req-B}, DK_B, ek_B, L_{ch}) \leftarrow st_{req}$ // requestor state 3 : $I_B \leftarrow \perp, t_{I_B} \leftarrow \perp, ch \leftarrow \perp, b \leftarrow \perp$ 4 : if $[ek_B = \perp] \wedge [ACK[t_{res-B}].ct-rec] \wedge [ACK[t_{res-B} - 1].ct-rec]$ 5 : $t_{res-B} += 2$ 6 : $i_{ch} \leftarrow 0$ 7 : $(dk_B, ek_B) \leftarrow KeyGen(1^\lambda)$ 8 : $DK_B[t_{res-A} - 1] \leftarrow dk_B$ 9 : if $\neg[ACK[t_{res-A} - 1].ek-rec]$ // ek_B not yet received by A 10 : $i_{ch} += 1$ 11 : $b \leftarrow 0$ 12 : $ch \leftarrow Encode(ek_B, i_{ch})$ 13 : elseif $\neg[ACK[t_{res-A}].ct-rec]$ // ct_B not yet received by A 14 : if $[ct_B = \perp] \wedge [ACK[t_{res-B}].ek-rec]$ 15 : $t_{I_B} \leftarrow t_{res-B}, i_{ch} \leftarrow 0$ 16 : $(ct_B, I_B) \leftarrow Enc(EK_A[t_{res-B}])$ 17 : $i_{ch} += 1$ 18 : $b \leftarrow 1$ 19 : $ch \leftarrow Encode(ct_B, i_{ch})$ 20 : $t_B^{snd} \leftarrow \max(t : [ACK[t].ct-rec] \wedge [ACK[t - 1].ct-rec])$ 21 : $act \leftarrow (ACK[t_{req-B} - 1].ek-rec, ACK[t_{req-B}].ct-rec)$ 22 : $\rho \leftarrow (ch, t_{res-B}, t_{req-B}, t_B^{snd}, act, b)$ 23 : $st_{res} \leftarrow (t_{res-B}, EK_A, ct_B, i_{ch})$ 24 : $st_{req} \leftarrow (t_{req-B}, DK_B, ek_B, L_{ch})$ 25 : return $((t_{I_B}, I_B), \rho, t_B^{snd}, (st_{res}, st_{req}, ACK))$ </pre>	
st_B.vuln <pre> 1 : $st_B.vuln = \{t : DK_B[t] \neq \perp\}$ </pre>	

Figure 16: Opp-BiKEM-CKA constructed from standard KEM KEM and erasure code (Encode, Decode). Algorithms of user B.

CKA-Init-KeyGen(1^λ) <pre> 1: $\text{par} \xleftarrow{\\$} \text{RSetup}(1^\lambda)$ 2: $(\text{ek}_B, \text{dk}_B) \leftarrow \text{RKeyGen-B}(\text{par}, \text{updated})$ 3: return $\text{l}_{\text{CKA}} := (\text{par}, \text{ek}_B, \text{dk}_B)$ </pre> CKA-Init-A(l_{CKA}) <pre> 1: parse $(\text{par}, \text{ek}_B, \text{dk}_B) \leftarrow \text{l}_{\text{CKA}}$ 2: initialize $\text{EK}_B[t] = \perp, \text{DK}_A[t] = \perp$ for all t 3: initialize $(\text{ACK}[t].\text{ek-rec}, \text{ACK}[t].\text{ct-rec}) = (\text{false}, \text{false})$ for all t 4: initialize $\text{UP}[t] \leftarrow \text{false}$ for all t 5: $(\text{ACK}[0].\text{ek-rec}, \text{ACK}[-1].\text{ct-rec}) \leftarrow (\text{true}, \text{true})$ 6: $(\text{ACK}[1].\text{ek-rec}, \text{ACK}[0].\text{ct-rec}) \leftarrow (\text{true}, \text{true})$ 7: $\text{EK}_B[1] \leftarrow \text{ek}_B$ 8: $\text{st}_{\text{res}} \leftarrow (-1, \text{EK}_B, \perp, 0)$ 9: $\text{st}_{\text{req}} \leftarrow (0, \text{DK}_A, \perp, \emptyset)$ 10: return $(\text{st}_{\text{res}}, \text{st}_{\text{req}}, \text{ACK}, \text{par})$ </pre> CKA-Send-A($\text{st}_{\text{res}}, \text{st}_{\text{req}}, \text{ACK}, \text{par}$) <pre> 1: parse $(\text{t}_{\text{res-A}}, \text{EK}_B, \text{ct}_A, i_{\text{ch}}) \leftarrow \text{st}_{\text{res}}$ // responder state 2: parse $(\text{t}_{\text{req-A}}, \text{DK}_A, \text{ek}_A, L_{\text{ch}}, \text{UP}) \leftarrow \text{st}_{\text{req}}$ // requestor state 3: $I_A \leftarrow \perp, t_{I_A} \leftarrow \perp, \text{ch} \leftarrow \perp$ 4: if $[\text{ek}_A = \perp] \wedge [\text{ACK}[\text{t}_{\text{res-A}}].\text{ct-rec}] \wedge [\text{ACK}[\text{t}_{\text{res-A}} + 1].\text{ek-rec}]$ 5: $\text{t}_{\text{res-A}} += 2$ 6: $i_{\text{ch}} \leftarrow 0$ 7: $(\text{dk}_A, \text{ek}_A) \leftarrow \text{RKeyGen-A}(\text{par})$ 8: $\text{DK}_A[\text{t}_{\text{res-A}} + 1] \leftarrow \text{dk}_A$ 9: if $\neg[\text{ACK}[\text{t}_{\text{res-A}} + 1].\text{ek-rec}]$ // ek_A not yet received by B 10: $i_{\text{ch}} += 1$ 11: $b \leftarrow 0$ 12: $\text{ch} \leftarrow \text{Encode}(\text{ek}_A, i_{\text{ch}})$ 13: elseif $\neg[\text{ACK}[\text{t}_{\text{res-A}}].\text{ct-rec}]$ // ct_A not yet received by B 14: if $[\text{ct}_A = \perp] \wedge [\text{ACK}[\text{t}_{\text{res-A}}].\text{ek-rec}] \wedge [\text{ACK}[\text{t}_{\text{res-A}} - 1].\text{ct-rec}]$ 15: $i_{\text{ch}} \leftarrow 0$ 16: $t_{I_A} \leftarrow \text{t}_{\text{res-A}}$ 17: $(\text{ct}_A, I_A, \text{DK}_A[\text{t}_{\text{res-A}} + 1]) \leftarrow \text{REnc-A}(\text{EK}_B[\text{t}_{\text{res-A}}], \text{DK}_A[\text{t}_{\text{res-A}} + 1])$ 18: $\text{UP}[\text{t}_{\text{res-A}} + 1] \leftarrow \text{true}$ 19: $i_{\text{ch}} += 1$ 20: $b \leftarrow 1$ 21: $\text{ch} \leftarrow \text{Encode}(\text{ct}_A, i_{\text{ch}})$ 22: $t_A^{\text{snd}} \leftarrow \max(t : [\text{ACK}[t].\text{ct-rec}] \wedge [\text{ACK}[t - 1].\text{ct-rec}])$ 23: $\text{act} \leftarrow (\text{ACK}[\text{t}_{\text{req-A}} + 1].\text{ek-rec}, \text{ACK}[\text{t}_{\text{req-A}}].\text{ct-rec})$ 24: $\rho \leftarrow (\text{ch}, \text{t}_{\text{res-A}}, \text{t}_{\text{req-A}}, t_A^{\text{snd}}, \text{act}, b)$ 25: $\text{st}_{\text{res}} \leftarrow (\text{t}_{\text{res-A}}, \text{EK}_B, \text{ct}_A, i_{\text{ch}})$ 26: $\text{st}_{\text{req}} \leftarrow (\text{t}_{\text{req-A}}, \text{DK}_A, \text{ek}_A, L_{\text{ch}}, \text{UP})$ 27: return $((t_{I_A}, I_A), \rho, t_A^{\text{snd}}, (\text{st}_{\text{res}}, \text{st}_{\text{req}}, \text{ACK}, \text{par}))$ </pre> $\text{st}_A.\text{vuln}$ <pre> 1: $\text{st}_A.\text{vuln} = \{t : [\text{DK}_A[t] \neq \perp]\} \cup \{t - 1 : [\text{DK}_A[t] \neq \perp] \wedge \neg[\text{UP}[t]]\}$ </pre> $\text{st}_B.\text{vuln}$ <pre> 1: $\text{st}_B.\text{vuln} = \{t : [\text{DK}_B[t] \neq \perp]\} \cup \{t - 1 : [\text{DK}_B[t] \neq \perp] \wedge \neg[\text{UP}[t]]\}$ </pre>	CKA-Init-B(l_{CKA}) <pre> 1: parse $(\text{par}, \text{ek}_B, \text{dk}_B) \leftarrow \text{l}_{\text{CKA}}$ 2: initialize $\text{EK}_A[t] = \perp, \text{DK}_B[t] = \perp$ for all t 3: initialize $(\text{ACK}[t].\text{ek-rec}, \text{ACK}[t].\text{ct-rec}) = (\text{false}, \text{false})$ for all t 4: $(\text{ACK}[0].\text{ek-rec}, \text{ACK}[-1].\text{ct-rec}) \leftarrow (\text{true}, \text{true})$ 5: $(\text{ACK}[1].\text{ek-rec}, \text{ACK}[0].\text{ct-rec}) \leftarrow (\text{true}, \text{true})$ 6: $\text{DK}_B[1] \leftarrow \text{dk}_B$ 7: $\text{st}_{\text{res}} \leftarrow (0, \text{EK}_A, \perp)$ 8: $\text{st}_{\text{req}} \leftarrow (-1, \text{DK}_B, \perp)$ 9: return $(\text{st}_{\text{res}}, \text{st}_{\text{req}}, \text{ACK}, \text{par})$ </pre> CKA-Rec-B($(\text{st}_{\text{res}}, \text{st}_{\text{req}}, \text{par}), \rho$) <pre> 1: parse $(\text{t}_{\text{res-B}}, \text{EK}_A, \text{ct}_B, i_{\text{ch}}) \leftarrow \text{st}_{\text{res}}$ 2: parse $(\text{t}_{\text{req-B}}, \text{DK}_B, \text{ek}_B, L_{\text{ch}}, \text{UP}) \leftarrow \text{st}_{\text{req}}$ 3: parse $(\text{ch}, \text{t}_{\text{res-A}}, \text{t}_{\text{req-A}}, t_A^{\text{snd}}, (\text{act.ek-rec}, \text{act.ct-rec})) \leftarrow \rho$ 4: $I_A \leftarrow \perp, t_{I_A} \leftarrow \perp$ 5: $I_A \leftarrow \perp, t_{I_A} \leftarrow \perp$ 6: if $[\text{act.ct-rec}]$ // incorporate acknowledgments 7: $\text{ACK}[\text{t}_{\text{req-A}}].\text{ct-rec} \leftarrow \text{true}$ 8: if $[\text{act.ek-rec}]$ 9: $\text{ACK}[\text{t}_{\text{req-A}} + 1].\text{ek-rec} \leftarrow \text{true}$ 10: if $[\text{t}_{\text{res-A}} < \text{t}_{\text{req-B}}]$ // outdated message 11: break 12: elseif $[\text{t}_{\text{res-A}} > \text{t}_{\text{req-B}}]$ // first message of new epoch 13: $\text{t}_{\text{req-B}} += 2$ 14: if $[\text{EK}[\text{t}_{\text{req-B}} + 1] = \perp] \wedge [b = 0]$ // ek_A not yet recovered 15: $L_{\text{ch}} \xleftarrow{\cup} \{\text{ch}\}$ 16: $\text{ek}_A \leftarrow \text{Decode}(L_{\text{ch}})$ 17: if $\text{ek}_A \neq \perp$ 18: $L_{\text{ch}} \leftarrow \emptyset$ 19: $\text{EK}_A[\text{t}_{\text{req-B}} + 1] \leftarrow \text{ek}_A$ 20: $\text{ACK}[\text{t}_{\text{req-B}} + 1].\text{ek-rec} \leftarrow \text{true}$ 21: elseif $\neg[\text{ACK}[\text{t}_{\text{req-B}}].\text{ct-rec}] \wedge [b = 1]$ // ct_A not yet recovered 22: $L_{\text{ch}} \xleftarrow{\cup} \{\text{ch}\}$ 23: $\text{ct}_A \leftarrow \text{Decode}(L_{\text{ch}})$ 24: if $\text{ct}_A \neq \perp$ 25: $L_{\text{ch}} \leftarrow \emptyset$ 26: $\text{ACK}[\text{t}_{\text{req-B}}].\text{ct-rec} \leftarrow \text{true}$ 27: $t_{I_A} \leftarrow \text{t}_{\text{req-B}}$ 28: $(I_A, \text{EK}_A[\text{t}_{\text{req-B}} + 1]) \leftarrow \text{RDec-B}(\text{ct}_A, \text{DK}_B[\text{t}_{\text{req-B}}], \text{EK}_A[\text{t}_{\text{req-B}} + 1])$ 29: $\text{DK}_B[\text{t}_{\text{req-B}}] \leftarrow \perp$ 30: if $[\text{ACK}[\text{t}_{\text{res-B}}].\text{ct-rec}]$ // delete fully delivered variables 31: $\text{ct}_B \leftarrow \perp$ 32: if $[\text{ACK}[\text{t}_{\text{res-B}} + 1].\text{ek-rec}]$ 33: $\text{ek}_B \leftarrow \perp$ 34: $t_B^{\text{cv}} \leftarrow \max(\text{t}_{\text{res-B}}, \text{t}_{\text{req-B}}) - 1$ 35: $\text{st}_{\text{res}} \leftarrow (\text{t}_{\text{res-B}}, \text{EK}_A, \text{ct}_B, i_{\text{ch}})$ 36: $\text{st}_{\text{req}} \leftarrow (\text{t}_{\text{req-B}}, \text{DK}_B, \text{ek}_B, L_{\text{ch}}, \text{UP})$ 37: return $((t_{I_A}, I_A), t_B^{\text{cv}}, (\text{st}_{\text{res}}, \text{st}_{\text{req}}, \text{ACK}, \text{par}))$ </pre>
--	---

Figure 17: Opp-RKEM-CKA constructed from ratcheting KEM RKEM and erasure code (Encode, Decode). Algorithms CKA-Send-B and CKA-Rec-A are defined analogously to CKA-Send-A and CKA-Rec-B, the roles of A and B reversed for every variable.

Instantiations. We instantiate UniKEM-CKA with Kyber-768. It exhibits encapsulation key and ciphertext sizes of

$$(|ek|, |ct|) = (1184B, 1088B).$$

Since our simulations use chunks of size (32, 128, 512) bytes, (37, 10, 3) chunks have to be received to recover an encapsulation key and (34, 9, 3) to recover a ciphertext.

E.2 BiKEM-CKA

We provide a description of BiKEM-CKA an SCKA variant of the bidirectional KEM based CKA.

Protocol description. In an odd epoch t party A generates and transmits the epoch key. We may assume (as will become clear below) they already have stored an encapsulation key ek_B^t . CKA-Send-A samples key pair (ek_A^{t+1}, dk_A^{t+1}) , to be used in the subsequent epoch, stores the decapsulation key dk_A^t in state, and generates $(ct_A^t, I_A^t) \leftarrow \text{Enc}(ek_B^t)$. It then starts sending chunks of (ek_A^{t+1}, ct_A^t) as part of the protocol message. During this phase CKA-Send-A does not send cryptographic material, its protocol messages only containing an acknowledgment indicating whether (ek_A^{t+1}, ct_A^t) was fully transmitted.

If this is the case, CKA-Rec-B uses dk_B^t to recover I_A^t , and deletes the decapsulation key. At this point the roles of A and B reverse, the latter sending $((ek_B^{t+2}, ct_B^{t+1}))$ in chunks.

Sending/receiving epoch and vulnerable epochs. In a phase where the parties are exchanging cryptographic material pertaining to epoch key I_P^t we have

$$t_A^{\text{snd}} = t - 1 = t_B^{\text{snd}}.$$

Protocol messages include the epoch t of the sent cryptographic material allowing the processing party to compute the matching t_A^{rcv} or t_B^{rcv} respectively.

Regarding the vulnerable epoch set note that the only secret stored in the users' state is dk_P^t . Accordingly, for $P \in \{A, B\}$ we have

$$\text{st.vuln}_P = \begin{cases} \{t\} & \text{if } \llbracket dk_P^t \neq \perp \rrbracket \\ \emptyset & \text{else} \end{cases}.$$

Instantiations. We instantiate UniKEM-CKA with Kyber-768. It exhibits encapsulation key and ciphertext sizes of

$$(|ek|, |ct|) = (1184B, 1088B).$$

Since our simulations use chunks of size (32, 128, 512) bytes, (37, 10, 3) chunks have to be received to recover an encapsulation key and (34, 9, 3) to recover a ciphertext.

E.3 RKEM-CKA

We provide an overview on RKEM-CKA.

Protocol description. The protocol is defined with respect to ratcheting KEM RKEM and erasure code (Encode, Decode). Assume party A fully received encapsulation key ek_B^t and ciphertext ct_B^{t-1} from B, the former being already updated. The next call to CKA-Send-A initiates epoch t by sampling $(dk_A^{t+1}, ek_A^{t+1}) \leftarrow \text{RKeyGen-A}(\text{par})$, generating $(ct_A^t, I_A^t, dk_A^{t+1}) \leftarrow \text{REnc-A}(ek_B^t, dk_A^{t+1})$, also updating dk_A in the process, and outputting the epoch key I_A^t . CKA-Send-A then starts sending chunks of (ek_A^{t+1}, ct_A^t) .

During this phase protocol messages output by CKA-Send-B only contain an acknowledgment indicating whether B fully received (ek_A^{t+1}, ct_A^t) . If this is the case CKA-Rec-B computes and outputs the epoch

key $(I_A^t, ek_A^{t+1}) \leftarrow \text{RDec-B}(ct_A^t, dk_B^t, ek_A^{t+1})$, and deletes dk_B^t . We point out that this operation updates the following epoch's encapsulation key ek_A^{t+1} . At this point the parties' roles reverse, with B generating ek_B^{t+2}, I_B^{t+1} , and ct_B^{t+1} and sending encapsulation key and ciphertext to A in chunks.

Sending/receiving epoch and vulnerable epochs. In a phase with party $P \in \{A, B\}$ sending ct_P^t and ek_P^{t+1} the sending epochs are given by

$$t_A^{\text{snd}} = t - 1 = t_B^{\text{snd}}.$$

Protocol messages include the epoch t of the sent cryptographic material allowing the processing party to compute the matching t_A^{rcv} or t_B^{rcv} respectively.

Regarding the vulnerable epochs, note that the only secret key material contained in users' states are the decapsulation keys. If a key dk_P^t was already updated by creating a ciphertext it reveals the epoch secret of epoch t . If the key was not updated, on the other hand, it reveals both $t - 1$ and t , as it is used to create the ciphertext of the former epoch. Opposed to **Opp-RKEM-CKA**, at most one decapsulation key is contained in any party's state at any point in time. We obtain

$$st_P.\text{vuln} = \begin{cases} \{t - 1, t\} & \text{if } [\perp \neq dk_P^t \text{ not updated}] \\ \{t\} & \text{if } [\perp \neq dk_P^t \text{ updated}] \\ \emptyset & \text{else} \end{cases}$$

Instantiations. We instantiate **RKEM-CKA** with the lattice based ratcheting KEM **Katana**. It exhibits encapsulation key and ciphertext sizes of

$$(|ek|, |ct|) = (1344\text{B}, 72\text{B}).$$

Since our simulations use chunks of size $(32, 128, 512)$ bytes, $(42, 11, 3)$ chunks have to be received to recover an encapsulation key and $(3, 1, 1)$ to recover a ciphertext.