USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

How to Compare Bandwidth Constrained Two-Party Secure Messaging Protocols: A Quest for A More Efficient and Secure Post-Quantum Protocol

Benedikt Auerbach, *PQShield;* Yevgeniy Dodis and Daniel Jost, *New York University;*
Shuichi Katsumata, *PQShield and AIST;* Rolfe Schmidt, *Signal Messenger*

https://www.usenix.org/conference/usenixsecurity25/presentation/auerbach

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

How to Compare Bandwidth Constrained Two-Party Secure Messaging Protocols: A Quest for A More Efficient and Secure Post-Quantum Protocol

Benedikt Auerbach
PQShield

Yevgeniy Dodis
New York University

Daniel Jost
New York University

Shuichi Katsumata
PQShield and AIST

Rolfe Schmidt
Signal Messenger

Abstract

Transitioning existing classical two-party secure messaging protocols to post-quantum protocols has been an active movement in practice in recent years: Apple's PQ3 protocol and the recent Triple Ratchet protocol being investigated by the Signal team with academics (Dodis et al. Eurocrypt'25). However, due to the large communication overhead of post-quantum primitives, numerous design choices non-existent in the classical setting are being explored, rendering comparison of secure messaging protocols difficult, if not impossible.

In this work, we thus propose a new pragmatic metric to measure how secure a messaging protocol is given a particular communication pattern, enabling a concrete methodology to compare secure messaging protocols. We uncover that there can be no "optimal" protocol, as different protocols are often incomparable with the respect to worst-case (adversarial) messaging behaviors, especially when faced with real-world bandwidth constraints. We develop a comprehensive framework to experimentally compare various messaging protocols under given bandwidth limits *and* messaging behaviors. Finally, we apply our framework to compare several new and old messaging protocols. Independently, we also uncover untapped optimizations which we call *opportunistic sending*, leading to better post-quantum messaging protocols. To capture these optimizations, we further propose *sparse* continuous key agreement as a fundamental building block for secure messaging protocols, which could be of independent interest.

1 Introduction

Secure Messaging (SM) applications, including WhatsApp, Signal, Google RCS, and Facebook Messenger, have taken over the world. Used by billions of people daily, these applications achieve extremely strong 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) [2], running side-by-side (as "authenticated metadata") with the actual encryption of the message. Hence, CKA forms the main overhead of modern 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 [19] 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 [2]. In principle, this means that one can build post-quantum CKA (and, hence, SM [2]) protocols. Unfortunately, post-quantum encapsulation keys and ciphertexts are much longer than those of Diffie-Hellman (DH) KEM. Concretely, using MLKEM (i.e., Kyber) [21] 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 [18] 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 [3] which provided post-quantum PCS and FS, whose idealized version was analyzed by [22]. At its core, it effectively applied the generic CKA-to-SM compiler of [2], but with one caveat. 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. 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. More details can be found in [14].

Bandwidth limitations. This lead [14] to construct a much leaner SM protocol achieving post-quantum PCS even usable under strict bandwidth constraints. 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 [17]. In essence, instead of sending (long) post-quantum CKA messages in one go, [14] uses *erasure codes* to send CKA messages in small chunks. This not only allows one to decouple CKA overhead *per message* from the native length of a post-quantum CKA message , but also completely eliminates the overhead "spikes" in PQ3; has high resiliency to lost messages; and results in big communication savings over PQ3 in unbalanced communication scenarios. We will call this elegant technique *chunking* from now on.

For the rest of this work, we focus on SM protocols under *bandwidth limitations*.

Security under bandwidth limitations. Of course, while reducing per message overhead, chunking has a different price: PCS now takes much longer. Indeed, a party needs to receive enough chunks to recover a full CKA message. This leads to a natural trade-off between per message overhead and PCS efficiency, but also brings us to the next question. How to define this "PCS efficiency"?

For the concrete chunking-based protocol of [14], the authors developed an ad hoc empirical methodology to convincingly argue the advantages of chunking. However, it did not naturally generalize to other SM protocols, or led to a way to compare such potential protocols. More importantly, it *did not formally correspond to existing security measures of* SM. Let us explain.

In SM literature, PCS speed (and other security measures, such as FS) is counted in terms of *epochs* [2], 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. (Recall, in the CKA-to-SM compiler the party repeats the same CKA message until it receives a new 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.

Epochs vs messages: Type mismatch. Once bandwidth limitations are in place, however, this equality no longer makes any sense. Consider the chunking solution of [14], for example. 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 single meaningful CKA value is finally received. The work of [14], which is the only prior work to explicitly talk about bandwidth limitations, had a limited workaround for the mismatch.

At the messaging level, they still used the notion of "epochs", but they no longer assign any semantic meaning (such as change of direction, etc.) to such epochs. Instead, the SM protocol had to come up with an abstract, protocol-specific "epoch function", mapping party's state to the actual epoch. For a concrete "chunking-CKA-to-SM" compiler, they instantiated this function, and gave an intuitive explanation of what it means. However, while such an interpretation may be useful to understand their SM protocol, this large disconnect between the security definition of CKA and SM hinders us from comparing various SM protocols. Indeed, in the bandwidth limited setting, we can no longer simply look at the quality of the CKA to address the actual question one cares for messaging: *how many messages get exposed during compromise*?

1.1 Our Contributions

In this work we propose a new pragmatic metric that, given a communication pattern, allows us to asses the speed at which bandwidth-limited SM protocols achieve PCS. We design several protocols and experimentally compare them with respect to meaningful communication patterns. To this end we introduce a generalization of CKA that allows us to capture schemes making use of novel optimizations. We discuss our results in more detail below.

A new method to quantify security. The mismatch between epochs and compromised messages discussed above 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. However, there is now a related issue we need to address. For *un*limited bandwidth, we have effectively an *equivalence* between security of CKA-based SM protocols and that of CKA. Thus, we can concentrate on the best CKA protocol in such a setting.

In contrast, we clearly do *not* have this equivalence for the bandwidth-limited setting. First, two epochs are often not enough to get PCS with limited bandwidth. As such, these SM

protocols do not comply with the current definition of CKA, requiring a fresh key to be output with every message. More importantly, the specific "chunking" CKA-to-SM compiler of [14] appears to be quite suboptimal (in terms of number of compromised messages before PCS is restored) for the natural setting where parties largely communicate in a balanced way. We will formally show this later in the paper, but the intuition is as follows. As CKA-messages are now long, Alice might need to send a lot of short chunks of the CKA-message to Bob, before Bob can receive the full CKA message, and in return sending a CKA-message to Alice for the next epoch. In the meanwhile, Bob is sending a lot of normal SM messages to Alice, because we assumed balanced communication. And, from the perspective of CKA, *all these* SM *messages from Bob are largely "wasted", since they do not contain any new* CKA*-messages*. Thus, when the bandwidth is noticeably smaller than the native CKA message, close to half of the SM bandwidth is not really "contributing" to the PCS. As CKA requires the parties to take turns in sending protocol messages, this issue unfortunately seems inherent to any compilation of a CKA protocol to bandwidth-limited messaging by means of chunking.

This leads to the natural question if the original notion of CKA [2] is still "right" for the limited bandwidth setting. As we argue in this work, the answer is negative.

A new abstraction of CKA. We define a generalization of CKA, called *Sparse* CKA (SCKA). SCKA allows for a simple translation of its security to that of SCKA-based SM More importantly, it enables us to explore new optimizations in bandwidth-limited SM that cannot be captured by CKA, namely opportunistic sending and unidirectional key-generation. We address it in detail in Sec. 3, but give (incomplete) highlights here.

As with CKA, SCKA produces an ordered sequence of keys $I_1, I_2, \ldots$. However, the keys are no longer produced with every SCKA message (it's "sparse" after all), and the *epoch t* now corresponds to the period in between outputting key I_t and I_{t+1}. Critically, though, to make SCKA easy to use, its security is still epoch-based, but with the understanding that an epoch might take more than one message. We then show a natural generalization of the CKA-to-SM compiler to the SCKA setting, which effectively preserves the bandwidth constraint of the SCKA (with some minimal accounting). While we have some subtleties to overcome, our compiler is largely based on that of [2]. It entails an explicit translation of the easy-to-use "epoch"-based security of SCKA to the number of exposed (or "vulnerable") messages of the resulting SM.

New SCKA protocols. Of course, in order for SCKA to be useful, we need to exhibit novel limited-bandwidth SCKA protocols. The reference point is the "naive" SCKA protocol implicitly considered by [14]: apply the chunking technique, on a per message basis, to any traditional CKA protocol.[1]

In Sec. 4 we design several new SCKA protocols that we call Opp-UniKEM-CKA (in Sec. 4.1), Opp-BiKEM-CKA (in Sec. 4.2), and Opp-RKEM-CKA (in Sec. 4.3). These protocols are generically based on some other cryptographic primitives (e.g., a KEM), although we envision instantiating them with concrete post-quantum schemes. We defer their detailed descriptions to the corresponding sections, but briefly discuss a novel key feature common to their design: *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. We note that to be able to capture opportunistic sending we have to rely on SCKA. Indeed, on the one hand, it cannot be handled by CKA syntax since new shared keys are no longer established with every message sent, and, on the other, it uses the underlying CKA in a non black-box manner. Thus, implementing the optimization as part of the conversion from key-agreement to SM is not possible.

New comparison framework for SM. Having introduced a variety of SCKA protocols, another contribution of this work is the introduction of a *meaningful framework to compare the resulting* SM *protocols*. This framework is touched upon in Sec. 3.3, and then experimentally applied to a variety of protocols in Sec. 5. So we only give the important highlights.

First, the number of exposed messages (formally *vulnerable message sets*) critically depends on the communication scenario considered. Second, most of SM protocols we consider are *incomparable*, and the selection of such "best" is application-specific (and impossible in general). Thus, any concrete comparison of two protocols should additionally specify the bandwidth limit *and* a particular messaging behavior relevant for the application in hand. We stress that this is not a limitation of our framework, but an actual important result of our work.

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,

[1] For example, in some of our comparisons, we will use the best post-quantum CKA protocol designed by [14], called Katana-CKA.

protocols making use of unidirectional key generation or advanced cryptographic primitives like ratcheting KEMs [14] perform well in most of the considered settings while also exhibiting robust behavior across a wide range of messaging patterns.

1.2 Related Work

The Double Ratchet and more generally PCS for two-party messaging has been studied extensively, e.g., [2–6, 9–11, 14–16, 20]. Most of those works follow the theoretical framework of *epochs* to quantify PCS. The majority of the protocols follow the (implicit) pattern of an epoch increment with every change in communication direction. Some notable exceptions are the work on post-quantum security [3, 14] where bandwidth limitations prevent the establishment of a new epoch with every message. Caforio et al. [8] consider a setting where epochs are incremented on demand.

Some protocols focusing on (almost) optimal security [15, 16, 20] forgo the notion of an epoch. Instead, they characterize PCS as the concrete message pattern that minimally needs to occur after a compromise. This is enabled by the protocols not only not being subject to any bandwidth constraints but also making use of advanced cryptographic primitives.

Blazy et al. [7] introduces a taxonomy of the healing speed for various secure two-party messaging protocols. They characterize protocols by the number of so-called horizontal and vertical evolutions it takes to establish PCS. Roughly speaking, a vertical evolution corresponds to an epoch change, while a horizontal evolution corresponds to a period change, i.e., a symmetric ratchet. As such, their taxonomy does *not* capture the speed at which evolutions occur and, therefore, does not characterize PCS of a protocol in the number of compromised messages. Instead, they utilize their framework to compare protocols' resiliency with respect to attackers with different capabilities. In particular, they consider fine-grained attacks only revealing parts of a party's state — which is outside of the model of this work.

A line of work establishes the impossibility of achieving PCS in the wake of sufficiently powerful adversaries. Cremers et al. [12] show that when taking into account Signal's session-handling layer, then there exist practical scenarios in which the current Signal app does not achieve PCS, despite using the Double Ratchet protocol. In [13], Cremers et al. then show that for any protocol that is resilient against certain types of state loss this is an inherent problem. We remark that in this work we focus on the messaging layer only, considering protocols that do not have resiliency against state loss (i.e., cannot recover if one of the parties' state gets erased). Investigating the effect of session-handling to our methodology remains an interesting open problem.

2 Preliminary

We recall some cryptographic primitives below. Other standard building blocks for constructing secure messaging protocols introduced in [2] are provided in the full version due to space limitations.

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 $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$ in an offline and online phase.

Definition 2.1 ((Online-Offline) KEM). *We say a* key-encapsulation mechanism KEM *is an* online-offline KEM *if it consists of the following PPT algorithms:*

$\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: 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.*

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

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

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

$\mathsf{Dec}(\mathsf{dk}, \mathsf{ct}) \to I$*: 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* $\mathsf{Enc}(\mathsf{par}, \mathsf{ek}) \to (\mathsf{ct}, I)$*, where intuitively* $\mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1)$*, 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) [21] 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, $\mathsf{ct}_0 = \mathbf{D} \cdot \mathbf{r} + \mathbf{z}$ and $\mathsf{ct}_1 = \mathsf{ek}^\top \cdot \mathbf{r} + z' + \lfloor (q/2) \rceil \cdot I$, 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.

Ratcheting KEMs. Recently, Dodis et al. [14] introduced a KEM variant tailored to Signal's Double Ratchet specifics. This allows reusing part of the ciphertext as an encapsulation key, minimizing the cost of sending both components. Continuing with the above Kyber example, by tweaking the parameters, we can see that ct_0 can be viewed as a valid *transposed* encapsulation key — the encapsulation key is defined as $\mathsf{ek} = \mathbf{D}^\top \cdot \mathbf{s} + \mathbf{e}$ for decryption to work, i.e., $\mathsf{ek}^\top \cdot \mathbf{r} \approx \mathbf{s}^\top \cdot \mathsf{ct}_0$. This is why we will have two algorithms for each party role below. For more details, we refer to [14].

Definition 2.2 (RKEM)**.** *A* (forward-secure) ratcheting KEM (RKEM) *with ratcheting key spaces* $\mathcal{RK}_\mathsf{P}$ *and* $\widehat{\mathcal{RK}}_\mathsf{P}$ *for parties* $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ *consists of the following PPT algorithms:*

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

$\mathsf{RKeyGen\text{-}P}(\mathsf{par},\mathsf{mode}) \to (\mathsf{ek_P},\mathsf{dk_P})$ *: On input* par *and* mode*, it outputs encapsulation and decapsulation keys* $(\mathsf{ek_P},\mathsf{dk_P}) \in \mathcal{RK}_\mathsf{P}$ *if* $\mathsf{mode} = \bot$ *and* $(\mathsf{ek_P},\mathsf{dk_P}) \in \widehat{\mathcal{RK}}_\mathsf{P}$ *if* $\mathsf{mode} = \mathtt{updated}$*.*[2]

$\mathsf{REnc\text{-}A}(\widehat{\mathsf{ek}}_\mathsf{B},\mathsf{dk_A}) \to (\mathsf{ct_A}, I, \widehat{\mathsf{dk}}_\mathsf{A})$ *: It takes as input an updated encapsulation key* $\widehat{\mathsf{ek}}_\mathsf{B}$ *for party* B *and a decapsulation key for party* A*, and outputs a ciphertext* $\mathsf{ct_A}$*, a shared key* I*, and an updated decapsulation key* $\widehat{\mathsf{dk}}_\mathsf{A}$*.*

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

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

Correctness requires that for an updated key-pair $(\widehat{\mathsf{ek}}_\mathsf{B},\widehat{\mathsf{dk}}_\mathsf{B})$ decapsulation RDec-B recovers I, and that the updated keys $\widehat{\mathsf{dk}}_\mathsf{A}$ and $\widehat{\mathsf{ek}}_\mathsf{A}$ are compatible. Security requires that even given access to $\widehat{\mathsf{ek}}_\mathsf{B}$, $\mathsf{ct_A}$, $\widehat{\mathsf{ek}}_\mathsf{A}$, and $\widehat{\mathsf{dk}}_\mathsf{A}$ no information on I is leaked. Intuitively, this matches IND-CPA security of KEMs with the additional requirement that $\widehat{\mathsf{dk}}_\mathsf{A}$ provides forward secrecy, i.e., after updating the decapsulation key can no longer be used to derive I. For formal definitions see the full version.

Erasure codes. We define erasure codes, used by the chunking optimization [14]. For a more convenient presentation of our protocols we allow the decoding procedure to fail, outputting a special symbol $\bot$ in this case.

Definition 2.3. *An* erasure code *for a set of symbols* Σ*, a block length* N*, and a message size* n_chunk *consists of PPT algorithms* Encode, Decode *defined as follows:*

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

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

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

[2] RKeyGen-P with $\mathsf{mode} = \mathtt{updated}$ was mainly used for security analysis, and as such, we will typically omit mode outside this section.

Secure Messaging Protocols. A (two-party) secure messaging protocol enables A and B to send and receive messages. We follow the standard definition of [2], 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 2.4. *A secure messaging protocol consists of the following PPT algorithms:*

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

$\mathsf{SM\text{-}Init\text{-}A}(\mathsf{I_K}) \to \mathsf{st_A}$ *: On input an initial key* $\mathsf{I_K}$*, it outputs an initial state* $\mathsf{st_A}$ *for party* A*.*

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

$\mathsf{SM\text{-}Rec\text{-}A}(\mathsf{st_A},\mathsf{ct}) \to (\mathsf{M}',\mathsf{t}',\mathsf{i}',\mathsf{st}'_\mathsf{A})$ *: On input a state* st *and a ciphertext* ct*, it outputs a message* M′*, an epoch* t′*, a period* i′*, and an updated state* 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.*

As explained in the introduction, we deviate from the prior security definition in [2] by using a finer-grained notion to define trivial attacks. Concretely, we assume that each party's state has an associated set $\mathsf{st_P}.\mathsf{vuln}$ of messages that will be compromised in case $\mathsf{st_P}$ is leaked to the adversary; we call this a *vulnerable message set*. Each vulnerable message $(\mathsf{P}',\mathsf{t},\mathsf{i}) \in \mathsf{st_P}.\mathsf{vuln}$ is identified by its sender $\mathsf{P}' \in \{\mathsf{A},\mathsf{B}\}$, its epoch t, and its period i. We further define the vulnerable message *sets* as the set of vulnerable message set $\mathsf{st_P}.\mathsf{vuln}$ for all exposed states $\mathsf{st_P}$. We then say the secure messaging protocol is secure if the adversary is not able to obtain any information on the messages not contained in the vulnerable message sets . Other than this modification, the correctness and security guarantees are defined identically to [2, 14], and we defer the details to the full version. We postpone the implication of using vulnerable message sets to Secs. 3.2 and 3.3.

Notational conventions. When writing (stateful) algorithms, we assume that if any statement, such as parsing an input or invoking a fallible subroutine, fails that the algorithm discards any changes to its state before terminating with an error. The keyword **req** enforces a boolean condition, causing the algorithm to fail if violated. For security games, the same keyword is used to denote conditions that, if violated, causes the oracle to abort, unwinding all changes to the game's state. The keyword **assert** indicates special winning conditions.

3 From Sparse Continuous Key Agreement to Secure Messaging

We define a generalization of the continuous key agreement (CKA) put forth by Alwen, Coretti, and Dodis [2], called *sparse* CKA (SCKA), enabling to capture new optimization tricks such as opportunistic sending and unidirectional key generation, formally explained in Sec. 4. We then show a generic compiler from an SCKA to a SM protocol, and discuss that in general, there can be no "optimal" SM protocol.

3.1 Definition of Sparse CKA

Continuous key agreement (CKA) was introduced 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. While this abstraction is reasonable in contexts where each of the classical messages are small, this is no longer the case when we consider post-quantum messages. Indeed, [14] applies chunking to CKA messages when turning it into a SM protocol, since in practice, we may only have limited bandwidth and sending a post-quantum CKA in one message is costly. However, such an ad-hoc generic construction introduces complexity outside of CKA. In fact, our new optimization trick uses the underlying CKA in a non-black box manner that would render the SM construction even more ad-hoc and complex.

We therefore introduce *sparse* CKA (SCKA), allowing to handle these optimizations at the CKA layer. 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 t_{I_A} whenever ready. Put differently, we allow it outputting $\perp$, 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 $t^{snd}_{I_A}$ 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 $t^{rcv}_{I_B} = t^{snd}_{I_A}$ 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 3.1. *A* sparse continuous key agreement (SCKA) *protocol with initial key space I_{CKA}, and key space I consists of the following PPT algorithms:*

CKA-Init-KeyGen$(1^\lambda) \to \mathsf{I}_{CKA}$: *On input the security parameter 1^λ, it outputs an* initial key $\mathsf{I}_{CKA} \in I_{CKA}$.

CKA-Init-A$(\mathsf{I}_{CKA}) \to \mathsf{st}_A$: *On input an initial key $\mathsf{I}_{CKA} \in I_{CKA}$, it outputs an initial state st_A for party* A.

CKA-Send-A$(\mathsf{st}_A) \to ((\mathsf{t}_{I_A}, I_A), \rho, \mathsf{t}^{snd}_A, \mathsf{st}'_A)$: *On input a state st_A of party* A*, it outputs a pair $(\mathsf{t}_{I_A}, I_A) \in (\mathbb{N} \times I) \cup \{(\perp, \perp)\}$ of epoch counter and key, a message ρ, a sending epoch t^{snd}_A, and an updated state st'_A.*

CKA-Rec-A$(\mathsf{st}_A, \rho) \to ((\mathsf{t}_{I_B}, I_B), \mathsf{t}^{rcv}_A, \mathsf{st}'_A)$: *On input a state st_A of party* A *and a message ρ, it outputs a pair $(\mathsf{t}_{I_B}, I_B) \in (\mathbb{N} \times I) \cup \{(\perp, \perp)\}$ of epoch counter and key, a receiving epoch t^{rcv}_A, and an updated state 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 [2]. In line with the CKA security experiment of [2], we consider *passive* adversaries only, which can delay and reorder but not modify SCKA messages.[3]

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 $(\mathsf{t}_{I_P}, I_P) \neq (\perp, \perp)$ during Send-P and $(\mathsf{t}_{I_{\bar{P}}}, I_{\bar{P}}) \neq (\perp, \perp)$ 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 $\mathsf{n}_{\bar{P}}$-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. Below, we explain them in slightly more detail.

Oracles. The basic experiment initializes both parties by first executing CKA-Init-KeyGen to obtain I_{CKA}, and then initializes both parties using CKA-Init-A and CKA-Init-B. Afterward, for each user $P \in \{A, B\}$, the experiment offers the following oracles:

Send-P**:** Runs the respective CKA-Send-P algorithm. Records $(P, \mathsf{n}_P, \rho, \mathsf{t}^{snd}_P)$ where n_P is a counter of the number of messages P sent. If I_P is set, then also records $(P, \mathsf{t}_{I_P}, I_P)$. It returns the epochs t_{I_P} and t^{snd}_P as well as the message ρ to the adversary.

Receive-P$(\mathsf{n}_{\bar{P}})$**:** Delivers the $\mathsf{n}_{\bar{P}}$-th message by executing CKA-Rec-P(ρ) where $(\bar{P}, \mathsf{n}_{\bar{P}}, \rho)$ has been previously recorded. If I_P is set, then also records $(P, \mathsf{t}_{I_P}, I_P)$. The adversary is given t_{I_P} and t^{rcv}_P.

Correctness. For simplicity, we assume that the game furthermore maintains t^{cur}_P as the maximal value of any t^{snd}_A and

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

$\mathsf{t}_{\mathsf{A}}^{\mathsf{rcv}}$ output by Send-P and Receive-P, respectively. Correctness is then checked by enforcing the following invariants as part of the game.

Consistent keys: If both a tuple $(\mathsf{A},\mathsf{t},I_{\mathsf{t}})$ and $(\mathsf{B},\mathsf{t},I'_{\mathsf{t}})$ is recorded, then it must hold that $I_{\mathsf{t}} = I'_{\mathsf{t}}$.

Unique epochs: For each party $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$, at most one tuple $(\mathsf{P},\mathsf{t}_{\mathsf{P}},I_{\mathsf{P}})$ can be recorded for every epoch t_{P}.

Known prefix: For all epoch $\mathsf{t}_{\mathsf{P}} \leq \mathsf{t}_{\mathsf{P}}^{\mathsf{cur}}$, a tuple $(\mathsf{P},\mathsf{t}_{\mathsf{P}},I_{\mathsf{P}})$ has been recorded, i.e., P has output keys for all epochs up to and including $\mathsf{t}_{\mathsf{P}}^{\mathsf{cur}}$.

Monotonicity: For the value $\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}}$ output by CKA-Send-P, $\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}} \geq \mathsf{t}_{\mathsf{P}}^{\mathsf{cur}}$ (before updating $\mathsf{t}_{\mathsf{P}}^{\mathsf{cur}}$).

Matching epoch: For the output $\mathsf{t}_{\mathsf{P}}^{\mathsf{rcv}}$ of Receive-P(n) it holds that $\mathsf{t}_{\mathsf{P}}^{\mathsf{rcv}} = \mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{snd}}$, where $(\bar{\mathsf{P}},n_{\bar{\mathsf{P}}},\rho,\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{snd}})$ is the respective send action recorded. In other words, the recipient of ρ outputs the same epoch that the sender output when producing ρ.

Fine-grained security. We also revisit how to model trivial distinguishing attacks. In our work, we assume a protocol state exposes the *vulnerable epoch set*, denoted as $\mathsf{st}_{\mathsf{P}}.\mathsf{vuln}$. Intuitively, this captures the list of epochs for which trivial attacks can be mounted, once corrupted the state st_{P}. As this is protocol dependent, concrete examples are provided in Sec. 4 for each of our protocols. We further define the vulnerable epoch *sets* as the set of vulnerable epoch set $\mathsf{st}_{\mathsf{P}}.\mathsf{vuln}$ for all exposed states 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 [2]. 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 $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$,

- $\mathsf{t}_{\mathsf{P}}^{\mathsf{cur}} + \Delta_{\mathsf{PCS}} > \max(\mathsf{st}_{\mathsf{P}}.\mathsf{vuln})$, upon corruption, i.e., no corruption must expose epochs more than Δ_{PCS} into the future.
- $\mathsf{t}_{\mathsf{A}}^{\mathsf{cur}} - \Delta_{\mathsf{FS}} \geq \min(\mathsf{st}_{\mathsf{P}}.\mathsf{vuln})$, i.e., no corruption must expose epochs more than Δ_{FS} into the past. In particular, for a forward-secure protocol with $\Delta_{\mathsf{FS}} = 0$ 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.

3.2 Sparse CKA to Secure Messaging

We now present a SM protocol built from SCKA in Fig. 3. As we implicitly capture all the optimization tricks at the SCKA layer, our generic construction is almost identical to the simplistic construction by [2] using CKA. The key differences are that, unlike CKA, SCKA allows 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 $\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}}$, the other party $\bar{\mathsf{P}}$ already processed a message that resulted in $\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{rcv}} \geq \mathsf{t}_{\mathsf{P}}^{\mathsf{snd}} - \Delta_{\mathsf{Slack}}$. Looking ahead, for all the protocols we consider, we have $\Delta_{\mathsf{Slack}} \leq 2$.

In more detail, the protocol uses the SCKA as an asymmetric ratcheting layer. The resulting keys are then mixed into the

$\mathsf{Game}_{\mathcal{A}}^{\mathsf{SCKA}}(1^{\lambda})$

$b \xleftarrow{\$} \{0,1\}$
$\mathsf{Key}[\cdot,\cdot] \leftarrow \bot$; $\mathsf{Msg}[\cdot,\cdot] \leftarrow \bot$
$\mathsf{Exposed} \leftarrow \emptyset$; $\mathsf{Challenged} \leftarrow \emptyset$
$\mathsf{I}_{\mathsf{CKA}} \xleftarrow{\$} \text{CKA-Init-KeyGen}(1^{\lambda})$
for $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$
  $\mathsf{st}_{\mathsf{P}} \xleftarrow{\$} \text{CKA-Init-P}(\mathsf{I}_{\mathsf{CKA}})$
  $\widehat{\mathsf{t}}_{\mathsf{P}} \leftarrow 0$
$b' \xleftarrow{\$} \mathcal{A}(1^{\lambda})^{\mathcal{O}()}$
return $[\![b = b']\!]$

Send-P(rleak)

if $[\![\mathsf{rleak}]\!]$ **then** / Leak randomness
  $\mathsf{rand} \xleftarrow{\$} \mathcal{R}$
  $\mathsf{vuln} \leftarrow \mathsf{st}_{\mathsf{P}}.\mathsf{vuln}$
  $((\mathsf{t}_{I_{\mathsf{P}}},I_{\bar{\mathsf{P}}}),\rho,\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{snd}},\mathsf{st}_{\mathsf{P}}) \xleftarrow{\$} \text{CKA-Send-P}(\mathsf{st}_{\mathsf{P}};\mathsf{rand})$
  / newly vulnerable epochs
  $\mathsf{vuln}' \leftarrow \mathsf{st}_{\mathsf{P}}.\mathsf{vuln} \setminus \mathsf{vuln}$
  req $[\![\mathsf{vuln}' \cap \mathsf{Challenged} = \emptyset]\!]$
  $\mathsf{Exposed} \xleftarrow{+} \mathsf{vuln}'$
else / Secure randomness
  $\mathsf{rand} \leftarrow \bot$
  $((\mathsf{t}_{I_{\mathsf{P}}},I_{\bar{\mathsf{P}}}),\rho,\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{snd}},\mathsf{st}_{\mathsf{P}}) \xleftarrow{\$} \text{CKA-Send-P}(\mathsf{st}_{\mathsf{P}})$
assert $[\![\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}} \geq \mathsf{t}_{\mathsf{P}}^{\mathsf{cur}}]\!]$
$\mathsf{t}_{\mathsf{P}}^{\mathsf{cur}} \leftarrow \mathsf{t}_{\mathsf{P}}^{\mathsf{snd}}$
if $[\![(\mathsf{t}_{I_{\mathsf{P}}},I_{\bar{\mathsf{P}}}) \neq (\bot,\bot)]\!]$ **then**
  assert $[\![\mathsf{Key}[\mathsf{P},\mathsf{t}_{I_{\mathsf{P}}}] = \bot]\!]$
  assert $[\![\mathsf{Key}[\bar{\mathsf{P}},\mathsf{t}_{I_{\mathsf{P}}}] \in \{I_{\bar{\mathsf{P}}},\bot\}]\!]$
  $\mathsf{Key}[\mathsf{P},\mathsf{t}_{I_{\mathsf{P}}}] \leftarrow I_{\bar{\mathsf{P}}}$
assert $[\![\forall \mathsf{t} \leq \mathsf{t}_{\mathsf{P}}^{\mathsf{snd}} : \mathsf{Key}[\mathsf{P},\mathsf{t}] \neq \bot]\!]$
$\mathsf{Msg}[\mathsf{P},{+}{+}\mathsf{n}_{\mathsf{P}}] \leftarrow (\rho,\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}})$
return $(\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}},\mathsf{t}_{I_{\mathsf{P}}},\rho,\mathsf{rand})$

Chall(t)

req $[\![\mathsf{t} \notin \mathsf{Exposed} \cup \mathsf{Challenged}]\!]$
if $[\![\mathsf{Key}[\mathsf{A},\mathsf{t}] \neq \bot]\!]$
  then $\mathsf{K} \leftarrow \mathsf{Key}[\mathsf{A},\mathsf{t}]$
else $\mathsf{K} \leftarrow \mathsf{Key}[\mathsf{B},\mathsf{t}]$
req $[\![\mathsf{K} \neq \bot]\!]$
if $[\![b = 1]\!]$ **then**
  $\mathsf{K} \xleftarrow{\$} I$ / Replace with random key
$\mathsf{Challenged} \xleftarrow{+} \mathsf{t}$
return K

Receive-P(n)

req $[\![\mathsf{Msg}[\bar{\mathsf{P}},n] \neq \bot]\!]$
$(\rho,\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{snd}}) \leftarrow \mathsf{Msg}[\bar{\mathsf{P}},n]$
$((\mathsf{t}_{I_{\bar{\mathsf{P}}}},I_{\bar{\mathsf{P}}}),\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{rcv}},\mathsf{st}_{\mathsf{P}}) \leftarrow \text{CKA-Rec-P}(\mathsf{st}_{\mathsf{P}},\rho)$
assert $[\![\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{rcv}} = \mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{snd}}]\!]$
$\mathsf{t}_{\mathsf{P}}^{\mathsf{cur}} \leftarrow \max(\mathsf{t}_{\mathsf{P}}^{\mathsf{cur}},\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{rcv}})$
if $[\![(\mathsf{t}_{I_{\bar{\mathsf{P}}}},I_{\bar{\mathsf{P}}}) \neq (\bot,\bot)]\!]$ **then**
  assert $[\![\mathsf{Key}[\mathsf{P},\mathsf{t}_{I_{\bar{\mathsf{P}}}}] = \bot]\!]$
  assert $[\![\mathsf{Key}[\bar{\mathsf{P}},\mathsf{t}_{I_{\bar{\mathsf{P}}}}] \in \{I_{\bar{\mathsf{P}}},\bot\}]\!]$
  $\mathsf{Key}[\mathsf{P},\mathsf{t}_{I_{\bar{\mathsf{P}}}}] \leftarrow I_{\bar{\mathsf{P}}}$
assert $[\![\forall \mathsf{t} \leq \mathsf{t}_{\mathsf{P}}^{\mathsf{cur}} : \mathsf{Key}[\mathsf{P},\mathsf{t}] \neq \bot]\!]$
return $(\mathsf{t}_{\mathsf{P}}^{\mathsf{rcv}},\mathsf{t}_{I_{\bar{\mathsf{P}}}})$

Corr-P()

/ No challenge of a vulnerable epoch
req $[\![\mathsf{st}_{\mathsf{P}}.\mathsf{vuln} \cap \mathsf{Challenged} = \emptyset]\!]$
$\mathsf{Exposed} \xleftarrow{+} \mathsf{st}_{\mathsf{P}}.\mathsf{vuln}$
return st_{P}

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.

key chain using Mixin-CKA-Keys. The key difference to [2] is that each epoch t has independent sending and receiving key chains KS^t and KR^t, respectively, associated. We refer to Fig. 2 for a graphical depiction of the key schedule, where $\mathsf{K}^t_{\mathsf{root}}$ denotes the root key of epoch t, $\mathsf{K}^t_{\mathsf{CKA}}$ the key from the SCKA freshly mixed in, and $\mathsf{KS}^{t,i}$ and $\mathsf{KR}^{t,i}$ the i^{th} state of the sending and receiving key chains, respectively.

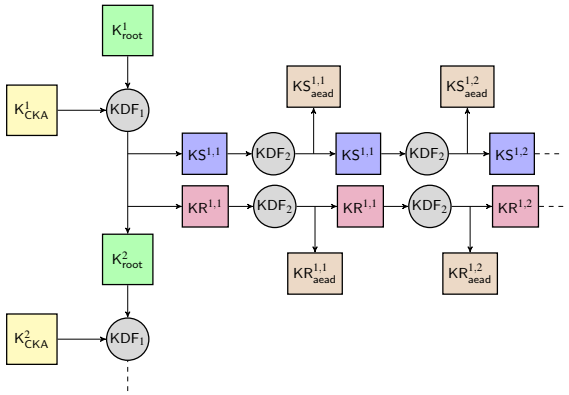

Figure 2: A schematic representation of the key schedule. While we make the indices clear, they are dropped from Fig. 3 for readability. For instance, $\mathsf{KS}^{1,2}_{\mathsf{aead}}$ denotes the second sender AEAD key generated by user A in the first epoch.

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

Correctness and security. The correctness and security almost immediately follows from the proof given by [2], 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 3.2. *Consider an* SCKA *with state* $\mathsf{st}^{\mathsf{CKA}}_{\mathsf{P}}$ *for* $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$. *Given vulnerable epoch set* $\mathsf{st}^{\mathsf{CKA}}_{\mathsf{P}}.\mathsf{vuln}$ *as in Sec. 3.1, the vulnerable message set* $\mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}.\mathsf{vuln}$ *for our* SM *protocol in Fig. 3 is defined as follows:*

- *For vulnerable* SCKA *epochs* $t \in \mathsf{st}^{\mathsf{CKA}}_{\mathsf{P}}.\mathsf{vuln}$ *all messages to be sent* (P,t,i) *and to be received* $(\bar{\mathsf{P}},t,i)$, *where* $i \in \mathbb{N}$.
- *For the current sending epoch* $\mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}.t^{\mathsf{snd}}$, *all future messages* $(\mathsf{P},\mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}.t^{\mathsf{snd}},i)$, *where* $i > \mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}.i^{\mathsf{snd}}$, *and analogously* $(\bar{\mathsf{P}},\mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}.t^{\mathsf{rcv}},i)$ *for all future messages of the receiving epoch.*
- *For future sending epoch* $t > \mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}.t^{\mathsf{snd}}$ *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* $t > \mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}.t^{\mathsf{rcv}}$ *not yet used for receiving, all messages* $(\bar{\mathsf{P}},t,i)$ *sent by the other party* $\bar{\mathsf{P}}$.
- *Any pending out-of-order message not yet received. That is,* $(\bar{\mathsf{P}},t,i)$ *for any* $(t,i) < (t^{\mathsf{rcv}},i^{\mathsf{rcv}})$ *such that* $\bar{\mathsf{P}}$ *has sent the corresponding message but* Receive-P *has not output a message for this epoch and period.*[4]

With this, we can formally state the security of our generic SM protocol from SCKA.

Theorem 3.3. *Our* SM *protocol in Fig. 3 is secure w.r.t the vulnerable message sets in Def. 3.2, if the underlying* SCKA *is secure w.r.t the vulnerable epoch sets in Sec. 3.1, the* AEAD *is CCA-secure and unforgeable, and the* KDF *is PRF-PRNG secure.*

Proof Sketch. We focus on the main differences to the proof of the Double Ratchet [2] and Triple Ratchet [14] protocols, respectively. 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 [2] 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 $\mathsf{KS}[t]$ and $\mathsf{KR}[t]$ 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

[4] In the protocol, P learns the number of messages $\bar{\mathsf{P}}$ sent for prior epochs when advancing the epoch. Therefore, this is purely a function of $\mathsf{st}^{\mathsf{SM}}_{\mathsf{P}}$.

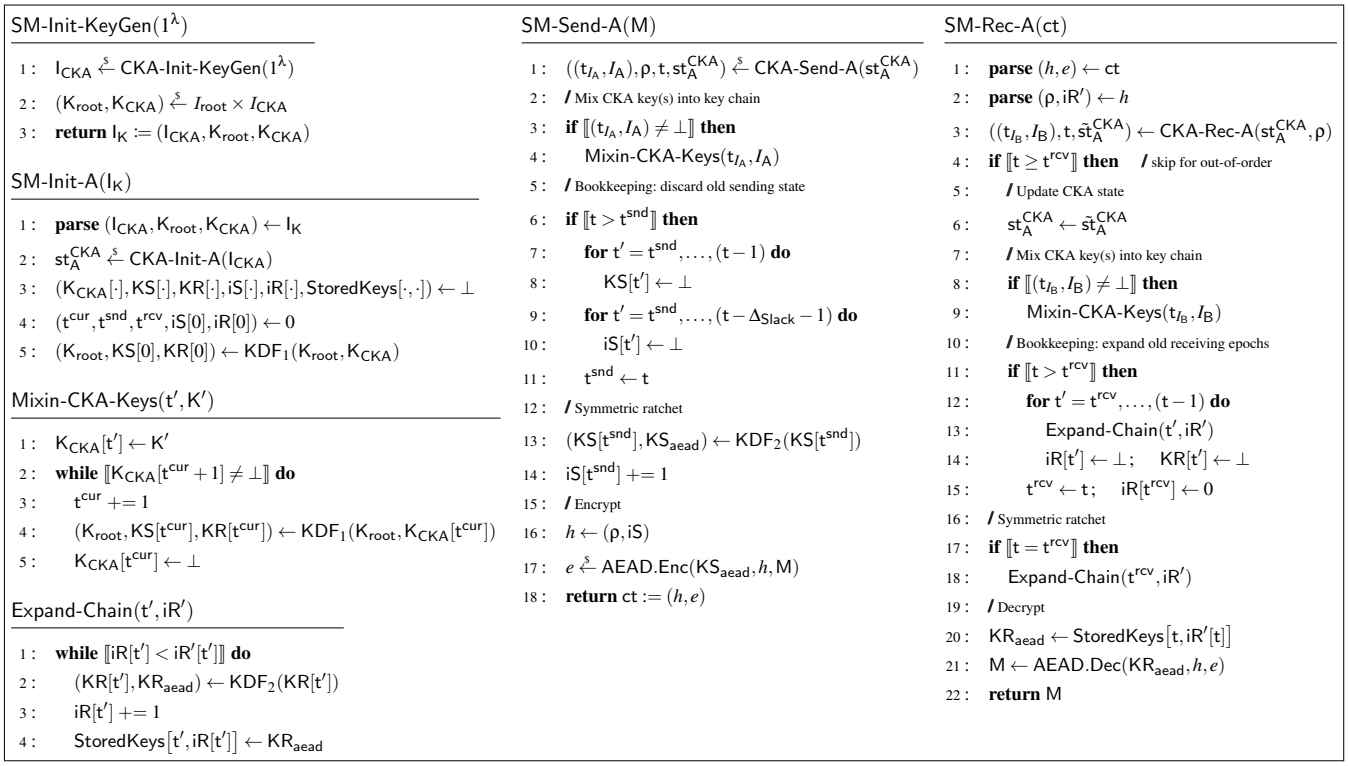

Figure 3: An SM protocol based on an SCKA protocol with slack Δ_{Slack}. The algorithms for B are analogous, except for the order of the sending and receiving chains swapped, e.g., line 4 of Mixin-CKA-Keys becoming $(K_{root}, KR[t^{cur}], KS[t^{cur}]) \leftarrow KDF_1(K_{root}, K_{CKA}[t^{cur}])$.

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 [2]. (Note that deriving both a sending and receiving chain per epoch is analogous to [14] and does not otherwise affect security.)

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 st_P.vuln as defined in Sec. 3.2 directly mirrors the keys kept as part of the state:

- A corruption exposes the SCKA state st_P^{CKA}. This in turn will reveal the keys for epochs $t \in st.vuln$ 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 unforgeablility 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).

3.3 How to Compare Secure Messaging?

Now that we defined a pragmatic metric 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. For reference, in App. A, we include one such example using our proposed SM protocols.

Looking ahead, this is exactly why in Sec. 5 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 also arises for natural messaging behavior (see Fig. 8). In the following section, we detail these SM protocols. As there is a concrete translation between SCKA to SM protocols, established in Thm. 3.3, we can simply focus on SCKA next. □

4 Candidate SCKA Constructions

We define three novel bandwidth-limited SCKA protocols. 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.

Compiling CKA to SCKA by chunking. Following [14] we can compile any CKA into a *sparse* CKA satisfying a predetermined bandwidth constraint by breaking protocol messages into small chunks. As explained in Sec. 3.1, [14] does not go through this abstraction, and performs chunking when building a SM protocol from CKA.

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. It initializes a counter $i_{\mathsf{ch}} \leftarrow 1$, and uses an erasure code $(\mathsf{Encode}, \mathsf{Decode})$ to generate a chunk $\mathsf{ch} \leftarrow \mathsf{Encode}(\rho_{\mathsf{A}}, i_{\mathsf{ch}})$ that serves as the protocol message. Subsequent calls to CKA-Send-A increment i and return further chunks $\mathsf{ch} \leftarrow \mathsf{Encode}(\rho_{\mathsf{A}}, i_{\mathsf{ch}})$ 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 $\rho_{\mathsf{A}} \leftarrow \mathsf{Decode}(L_{\mathsf{ch}})$. 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 $\mathsf{ch} \leftarrow \mathsf{Encode}(\rho_{\mathsf{B}}, i_{\mathsf{ch}})$.

Note that the compiler described above forces the communication to adhere to a ping-pong pattern, with one party having to send empty protocol messages for a prolonged amount of time, thus wasting bandwidth. 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.

Intuitively, this approach 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. We qualitatively analyze the effect of opportunistic sending in Sec. 5, revealing that in most realistic communications, it performs better compared to its non-opportunistic variant.

Our candidate protocols can be seen as chunked versions of (base) protocols UniKEM-CKA, BiKEM-CKA [2], and RKEM-CKA [14] that additionally make use of opportunistic sending. For completeness, we also consider the base protocols in our simulations. For an overview on them see App. B.

4.1 Protocol Opp-UniKEM-CKA

At a high level, unidirectional base protocol UniKEM-CKA works as follows. B, after receiving ek_{A}, generates $(\mathsf{ct}_{\mathsf{B}}, I_{\mathsf{B}}) \leftarrow \mathsf{Enc}(\mathsf{ek}_{\mathsf{A}})$ 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 is an adaption of UniKEM-CKA to the setting of SCKA that makes use of chunking with erasure code $(\mathsf{Encode}, \mathsf{Decode})$ and further allows for opportunistic sending by exploiting the structure of *offline-online* KEMs. Its formal description can be found in the full version.

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

Party B at the beginning of epoch t generates an offline ciphertext ciphertext as $(\mathsf{st}_{\mathsf{ct}}^{\mathsf{t}}, \mathsf{ct}_0^{\mathsf{t}}) \leftarrow \mathsf{Enc.Off}(1^{\lambda})$ and initializes counter i_{ch}. If CKA-Send-B is called and A did not yet fully receive $\mathsf{ct}_0^{\mathsf{t}}$ the protocol message ρ contains a chunk

$\mathsf{ch} \leftarrow \mathsf{Encode}(\mathsf{ct}_0^t, i_{\mathsf{ch}})$, 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 $(\mathsf{ct}_1^t, I_B^t) \leftarrow \mathsf{Enc.On}(\mathsf{st}_{\mathsf{ct}}^t, \mathsf{ek}_A^t)$, epoch key I_B^t returned, i_{ch} reset, and chunks $\mathsf{ch} \leftarrow \mathsf{Encode}(\mathsf{ct}_1^t, i_{\mathsf{ch}})$ 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 $\mathsf{ek}_A^t \leftarrow \mathsf{Decode}(L_{\mathsf{ch}})$, $\mathsf{ct}_0^t \leftarrow \mathsf{Decode}(L_{\mathsf{ch}})$, or $\mathsf{ct}_1^t \leftarrow \mathsf{Decode}(L_{\mathsf{ch}})$ respectively. If CKA-Rec-A fully recovers both parts of ct_B^t it computes and outputs the epoch key as $I_B^t \leftarrow \mathsf{Dec}(\mathsf{dk}_A^t, \mathsf{ct}_0^t, \mathsf{ct}_1^t)$, 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 ack.ek-rec and ack.ct_0-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 $t_B^{\mathsf{snd}} = t - 1 = t_A^{\mathsf{snd}}$. Protocol messages include the epoch t of the sent cryptographic material allowing the processing party to compute the matching t_B^{rcv} 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

$$\mathsf{st}_P.\mathsf{vuln} = \begin{cases} \{t\} & \text{if } [\![P = A]\!] \wedge [\![\mathsf{dk}_A^t \neq \bot]\!] \\ \{t\} & \text{if } [\![P = B]\!] \wedge [\![\mathsf{st}_{\mathsf{ct}}^t \neq \bot]\!] \\ \emptyset & \text{else} \end{cases}.$$

Instantiations. 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 Sec. 2 it has the structure of an online-offline KEM. It exhibits encapsulation key and ciphertext sizes of

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

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.

4.2 Protocol Opp-BiKEM-CKA

BiKEM-CKA [2], the CKA at basis of Opp-BiKEM-CKA works as follows. Opposed to the unidirectional approach outlined in Section 4.1, 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 $(\mathsf{ct}_A, I_A) \leftarrow \mathsf{Enc}(\mathsf{ek}_B)$ and, on the other, sample key pair $(\mathsf{ek}_A, \mathsf{dk}_A)$ to be used in the subsequent epoch. The decapsulation key dk_A is stored in state and the protocol message consists of $(\mathsf{ct}_A, \mathsf{ek}_A)$. B does not send any cryptographic material to A. However, after receiving $(\mathsf{ct}_A, \mathsf{ek}_A)$ 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).

Protocol description. Opp-BiKEM-CKA adapts the BiKEM-CKA protocol to the setting of SCKA using chunking and letting both parties opportunistically exchange encapsulation keys. A formal description is given in the full version.

The protocol is defined with respect to a (standard) KEM and erasure code $(\mathsf{Encode}, \mathsf{Decode})$. 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 $(\mathsf{ek}_A^{t+1}, \mathsf{dk}_A^{t+1}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ 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 $\mathsf{ch} \leftarrow \mathsf{Encode}(\mathsf{ek}_A^{t+1}, i_{\mathsf{ch}})$. 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 $(I_A^t, \mathsf{ct}_A^t) \leftarrow \mathsf{Enc}(\mathsf{ek}_B^t)$, outputs I_A^t, resets i_{ch}, and starts sending chunks $\mathsf{ch} \leftarrow \mathsf{Encode}(\mathsf{ct}_A^t, i_{\mathsf{ch}})$.

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 $\mathsf{ek}_B^t \leftarrow \mathsf{Decode}(L_{\mathsf{ch}})$. After acknowledging this to B and resetting L_{ch} it expects to receive ciphertext chunks and at some point recovers $\mathsf{ct}_B^{t+1} \leftarrow \mathsf{Decode}(L_{\mathsf{ch}})$. It outputs the epoch key $I_B^{t+1} \leftarrow \mathsf{Dec}(\mathsf{dk}_A^{t+1}, \mathsf{ct}_B^{t+1})$, 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 $(\mathsf{ek}_B^t, \mathsf{dk}_B^t)$ 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 $(I_B^{t+1}, \mathsf{ct}_B^{t+1}) \leftarrow \mathsf{Enc}(\mathsf{ek}_A^{t+1})$ and

starts sending chunks of $\mathsf{ct}_{\mathsf{B}}^{t+1}$. CKA-Rec-B processes the chunks sent by A to recover $\mathsf{ek}_{\mathsf{A}}^{t+1}$, $\mathsf{ct}_{\mathsf{A}}^{t}$, and in turn I_{A}^{t}.

Sending/receiving epoch and vulnerable epochs. The parties inform each other whether ciphertexts $\mathsf{ct}_{\mathsf{P}}^{t}$ with $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ were fully transmitted, and in turn whether the epoch key was received, by exchanging boolean acknowledgment $\mathsf{ack}^{t}.\mathsf{ct\text{-}rec}$. 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

$$\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}} = \max(\mathsf{t} : [\![\mathsf{ack}^{t}.\mathsf{ct\text{-}rec}]\!] \wedge [\![\mathsf{ack}^{t-1}.\mathsf{ct\text{-}rec}]\!]).$$

P includes a copy of $\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}}$ in protocol messages which is output as $\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{rcv}}$ by receiving party $\bar{\mathsf{P}}$.

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

$$\mathsf{st}_{\mathsf{P}}.\mathsf{vuln} = \begin{cases} \{\mathsf{t}\} & \text{if } [\![\mathsf{dk}_{\mathsf{P}}^{t} \neq \bot]\!] \\ \emptyset & \text{else.} \end{cases}$$

Instantiations. 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

$$(|\mathsf{ek}|, |\mathsf{ct}|) = (1184\mathrm{B}, 1088\mathrm{B}).$$

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.

4.3 Protocol Opp-RKEM-CKA

RKEM-CKA [14] at the basis of Opp-RKEM-CKA follows the structure of BiKEM-CKA. However, parties uses a *ratcheting* KEM to reuse the first part of an epoch's ciphertext as the subsequent epoch's encapsulation key. Loosely, it can be viewed as a lattice variant of Double Ratchet.

Protocol description. Opp-RKEM-CKA adapts Katana-CKA [14] to the setting of SCKA by using chunking and opportunistic sending. The formal description is in the full version.

The protocol is defined with respect to ratcheting KEM RKEM and erasure code $(\mathsf{Encode}, \mathsf{Decode})$. For odd epochs $\mathsf{t} = 2k+1$ party A generates epoch key I_{A}^{t} and communicates it to B by transmitting $\mathsf{ct}_{\mathsf{A}}^{t}$ and $\mathsf{ek}_{\mathsf{A}}^{t+1}$. Here, the latter intuitively serves as the ciphertext's first, $\mathsf{ek}_{\mathsf{B}}^{t}$-independent part. The ciphertext is generated as $(\mathsf{ct}_{\mathsf{A}}^{t}, I_{\mathsf{A}}^{t}, \mathsf{dk}_{\mathsf{A}}^{t+1}) \leftarrow \mathsf{REnc\text{-}A}(\mathsf{ek}_{\mathsf{B}}^{t}, \mathsf{dk}_{\mathsf{A}}^{t+1})$, 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 $\mathsf{t} = 2k$ generates epoch keys as $(\mathsf{ct}_{\mathsf{B}}^{t}, I_{\mathsf{B}}^{t}, \mathsf{dk}_{\mathsf{B}}^{t+1}) \leftarrow \mathsf{REnc\text{-}B}(\mathsf{ek}_{\mathsf{A}}^{t}, \mathsf{dk}_{\mathsf{B}}^{t+1})$ and transmits the ciphertext as well as $\mathsf{ek}_{\mathsf{B}}^{t+1}$ to A.

Assume A received an acknowledgment that B recovered both $\mathsf{ct}_{\mathsf{A}}^{t-2}$ and $\mathsf{ek}_{\mathsf{A}}^{t-1}$, the values A generated for epoch $\mathsf{t}-2$. Now, the first call to CKA-Send-A generates a new RKEM key pair $(\mathsf{dk}_{\mathsf{A}}^{t+1}, \mathsf{ek}_{\mathsf{A}}^{t+1})$, 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 $\mathsf{ch} \leftarrow \mathsf{Encode}(\mathsf{ek}_{\mathsf{A}}^{t+1}, i_{\mathsf{ch}})$ until A receives an acknowledgment that the encapsulation key was received by B. In order to proceed by generating ciphertext $\mathsf{ct}_{\mathsf{A}}^{t}$ party A must already have recovered and updated the corresponding encapsulation key $\mathsf{ek}_{\mathsf{B}}^{t}$. The latter is done by calling $(I_{\mathsf{B}}^{t-1}, \mathsf{ek}_{\mathsf{B}}^{t}) \leftarrow \mathsf{RDec\text{-}A}(\mathsf{ct}_{\mathsf{B}}^{t-1}, \mathsf{dk}_{\mathsf{A}}^{t-1}, \mathsf{ek}_{\mathsf{B}}^{t})$. Thus, after B acknowledged receiving $\mathsf{ek}_{\mathsf{A}}^{t+1}$, sending algorithm CKA-Send-A generates $(\mathsf{ct}_{\mathsf{A}}^{t}, I_{\mathsf{A}}^{t}, \mathsf{dk}_{\mathsf{A}}^{t+1}) \leftarrow \mathsf{REnc\text{-}A}(\mathsf{ek}_{\mathsf{B}}^{t}, \mathsf{dk}_{\mathsf{A}}^{t+1})$ in the first call to the algorithm after receiving both $\mathsf{ek}_{\mathsf{B}}^{t}$ and $\mathsf{ct}_{\mathsf{B}}^{t-1}$. After resetting i_{ch}, CKA-Send-A from this point on generates protocol messages containing chunks $\mathsf{ch} \leftarrow \mathsf{Encode}(\mathsf{ct}_{\mathsf{A}}^{t}, i_{\mathsf{ch}})$. After receiving an acknowledgment of B recovering $\mathsf{ct}_{\mathsf{A}}^{t}$ party A is able to restart the procedure, generating a key pair for epoch $\mathsf{t}+3$.

Receiving algorithm CKA-Rec-B stores chunks contained in protocol messages in a list L_{ch} until it is able to first recover $\mathsf{ek}_{\mathsf{A}}^{t+1}$ then, after resetting the list, $\mathsf{ct}_{\mathsf{A}}^{t}$. When it has done so, it recovers epoch key $(I_{\mathsf{A}}^{t}, \mathsf{ek}_{\mathsf{A}}^{t+1}) \leftarrow \mathsf{RDec\text{-}B}(\mathsf{ct}_{\mathsf{A}}^{t}, \mathsf{dk}_{\mathsf{B}}^{t}, \mathsf{ek}_{\mathsf{A}}^{t+1})$ also updating A's encapsulation key in the process and deletes $\mathsf{dk}_{\mathsf{B}}^{t}$. 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

$$\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}} = \max(\mathsf{t} : [\![\mathsf{ack}^{t}.\mathsf{ct\text{-}rec}]\!] \wedge [\![\mathsf{ack}^{t-1}.\mathsf{ct\text{-}rec}]\!])$$

for user $\mathsf{P} \in \{\mathsf{A},\mathsf{B}\}$ where acknowledgment $\mathsf{ack}^{t}.\mathsf{ct\text{-}rec}$ indicates whether the ciphertext for epoch t was fully transmitted. P includes a copy of $\mathsf{t}_{\mathsf{P}}^{\mathsf{snd}}$ in protocol messages which is output as $\mathsf{t}_{\bar{\mathsf{P}}}^{\mathsf{rcv}}$ by receiving party $\bar{\mathsf{P}}$.

Regarding the vulnerable epoch set, note that the only secret key material contained in users' states are the decapsulation keys. If a key $\mathsf{dk}_{\mathsf{P}}^{t}$ 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 $\mathsf{t}-1$ 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 $\mathsf{dk}_{\mathsf{A}}^{t+1}$ before receiving and processing ciphertext $\mathsf{ct}_{\mathsf{B}}^{t-1}$ and deleting $\mathsf{dk}_{\mathsf{A}}^{t-1}$. Accordingly, the vulnerable epoch set is given by

$$\mathsf{st}_{\mathsf{P}}.\mathsf{vuln} = \{\mathsf{t} : [\![\mathsf{dk}_{\mathsf{P}}^{t} \neq \bot]\!]\} \cup \{\mathsf{t}-1 : [\![\bot \neq \mathsf{dk}_{\mathsf{P}}^{t} \text{ not updated}]\!]\}.$$

Instantiations. We instantiate Opp-RKEM-CKA with the lattice-based RKEM Katana [14]. It exhibits encapsulation

key and ciphertext sizes of

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

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.

For additional details on Katana see [14].

5 Experimental Result

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. 4 or their non-opportunistic counterparts (see App. B).

5.1 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*, $\iota_{\mathsf{P}}^{\mathsf{on}} : \mathbb{Z}_N \to \{0,1\}$ 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 $p_{\mathsf{P}}^{\mathsf{send}}$ 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 $\iota_{\mathsf{P}}^{\mathsf{on}}(t \bmod N)$. If it returns 0 (i.e., party is offline), no action is taken for that party.

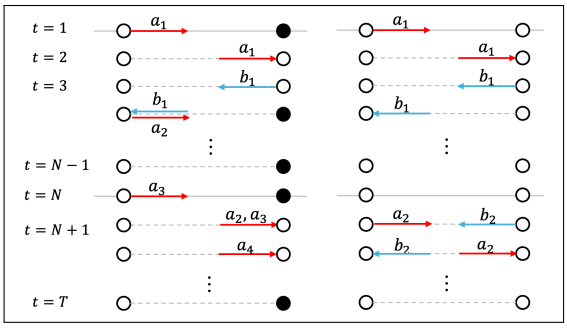

Figure 4: Messaging behavior of a primary-desktop model (left) and a primary-primary model (right). The white (resp. black) circles indicate that a device is online (resp. offline), i.e., $\iota_{\mathsf{P}}^{\mathsf{on}}(t \bmod N) = 1$ ($\iota_{\mathsf{P}}^{\mathsf{on}}(t \bmod N) = 0$). Arrows coming out (resp. in) a circle indicate the message that is being sent (resp. received) at a given time t.

Otherwise, the online party receives all incoming messages and then sends a single message with probability $p_{\mathsf{P}}^{\mathsf{send}}$.

For all experiments reported, we choose $N = 240$, translating to one timestep being 6 minutes long. For most experiments, we then take $p_{\mathsf{A}}^{\mathsf{send}} = p_{\mathsf{B}}^{\mathsf{send}} = 0.4$ to model balanced communications, where parties send about 4 messages per hour. We also experiment with unbalanced communications with $(p_{\mathsf{A}}^{\mathsf{send}}, p_{\mathsf{B}}^{\mathsf{send}}) = (0.04, 0.36)$ and $(0.36, 0.04)$.[5] 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:[6]

- *primary-primary* models interaction between two primary devices that are always online to receive messages. For all $t \in \mathbb{Z}_N$, $\iota_{\mathsf{A}}^{\mathsf{on}}(t) = \iota_{\mathsf{B}}^{\mathsf{on}}(t) = 1$.
- *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. 4.

[5] The concrete values of $p_{\mathsf{A}}^{\mathsf{send}}$ and $p_{\mathsf{B}}^{\mathsf{send}}$ 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.

[6] We experimented with other settings, e.g., varying the overlap time when two devices are online. However, we did not include them as the experimental results did not add much to the representative 4 presented messaging behaviors.

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 $\mathsf{st}_\mathsf{P}.\mathsf{vuln}$. 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 [1] to implement the KEM using MLKEM [21]. 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 [14]. All protocols are subjected to a test suite that uses different messaging behaviors, drops messages, and delivers messages out of order. For completeness, all experimental results are given in the full version

5.2 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 Def. 3.2. As discussed in Sec. 3.3, we take a pragmatic approach to assessing the protocols' quality.

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 |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.
- |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.

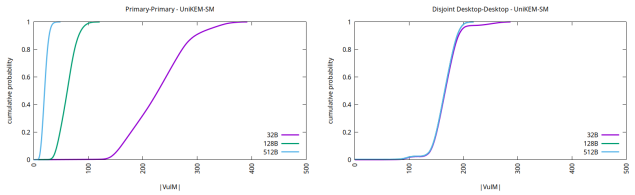

Figure 5: Cumulative probability of |VulM| for UniKEM-SM with chunk sizes 32, 128, 512 bytes. Left (resp. right) is primary-primary (resp. disjoint desktop-desktop) model.

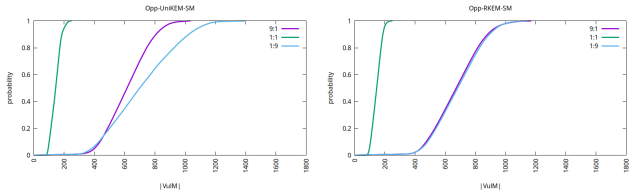

Figure 6: Cumulative probability of |VulM| for Opp-UniKEM-SM (left) and Opp-RKEM-SM (right) with chunk size 32 bytes. $(p_\mathsf{A}^{\mathsf{send}}, p_\mathsf{B}^{\mathsf{send}})$ is set to $(0.36, 0.04)$ (ratio $9:1$), $(0.4, 0.4)$ (ratio $1:1$), and $(0.04, 0.36)$ (ratio $1:9$).

5.3 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, |VulM| 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. 5 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 |VulM| larger. Fig. 6 illustrates this.

Offline vs online parties. Similarly to above, a compromised party that is offline cannot recover, and thus, the average and variance of |VulM| increase. This is illustrated in Fig. 7 (left). Note that the primary-primary model and the overlapping desktop-desktop model produce similar cumulative probability of |VulM| as both send a similar number of messages and are online at the same time enough to avoid

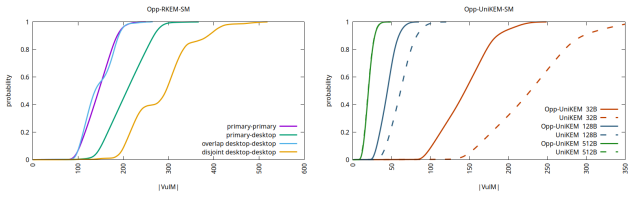

Figure 7: Cumulative probability of |VulM| (Left), and (Right) all protocols with chunk size 32 bytes in primary-primary model.

blocking; For primary-desktop we see a higher average of |VulM| 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 |VulM|) 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. 7 (right).

5.4 Comparing Opportunistic Protocols

We now compare Opp-UniKEM-SM, Opp-BiKEM-SM, and Opp-RKEM-SM in Fig. 8 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 |VulM| 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.

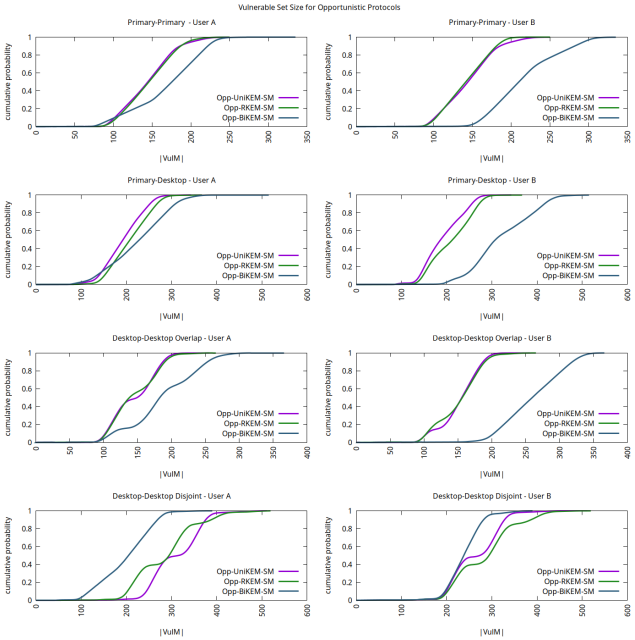

Figure 8: Cumulative probability of |VulM| for the opportunistic protocol variants using 32 byte chunks under all four message behavior models.

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

Ethical Considerations

Transitioning existing classical two-party secure messaging protocols to post-quantum protocols have been an active movement in practice in recent years: Apples's PQ3 protocol and the Triple Ratchet protocol currently being investigated by the Signal team [14]. Due to the large communication overhead of post-quantum primitives, particular design choices non-existing in the classical setting have to be made, rendering comparison of secure messaging protocols difficult, if not impossible. This makes it relevant to introduce a methodology to compare two-party secure messaging protocols. In this

work, we further construct several secure messaging protocols and compare them.

Our work follows the ethical guidelines of the conference. All of our comparison is made either on new protocols we design or on the (post-quantum component of the) Triple Ratchet protocol, currently being investigated by the Signal team. As [14] have thoroughly investigated the security of the Triple Ratchet protocol, we deem any risk of finding previously unknown vulnerabilities unlikely. If any significant issues had been found, we would have shared our findings to the Signal team; however, as this is still an ongoing investigation, it is unlikely to have any negative results for already used or implemented systems.

Open Science Policy

All source code for simulations, along with scripts to reproduce all plots presented in the paper and appendix, are available at https://zenodo.org/records/15571276.

References

[1] libcrux - the formally verified crypto library.

[2] 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.

[3] 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/.

[4] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Cham, Switzerland.

[5] 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.

[6] Olivier Blazy, Angèle Bossuat, Xavier Bultel, Pierre-Alain Fouque, Cristina Onete, and Elena Pagnin. SAID: Reshaping signal into an identity-based asynchronous messaging protocol with authenticated ratcheting. In *2019 IEEE European Symposium on Security and Privacy*, pages 294–309, Stockholm, Sweden, June 17–19, 2019. IEEE Computer Society Press.

[7] Olivier Blazy, Ioana Boureanu, Pascal Lafourcade, Cristina Onete, and Léo Robert. How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 5917–5934, Anaheim, CA, USA, August 9–11, 2023. USENIX Association.

[8] Andrea Caforio, F. Betül Durak, and Serge Vaudenay. Beyond security and efficiency: On-demand ratcheting with security awareness. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 649–677, Virtual Event, May 10–13, 2021. Springer, Cham, Switzerland.

[9] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 3–33, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.

[10] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy*, pages 451–466, Paris, France, April 26–28, 2017. IEEE Computer Society Press.

[11] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone detection in secure messaging: Improving post-compromise security in practice. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1481–1495, Virtual Event, USA, November 9–13, 2020. ACM Press.

[12] Cas Cremers, Charlie Jacomme, and Aurora Naska. Formal analysis of session-handling in secure messaging: Lifting security from sessions to conversations. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1235–1252, Anaheim, CA, USA, August 9–11, 2023. USENIX Association.

[13] Cas Cremers, Niklas Medinger, and Aurora Naska. Impossibility results for post-compromise security in real-world communication systems. Cryptology ePrint Archive, Paper 2024/1886, 2024.

[14] 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.

[15] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

[16] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188, Darmstadt, Germany, May 19–23, 2019. Springer, Cham, Switzerland.

[17] 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.

[18] Ehren Kret and Rolfe Schmidt. The pqxdh key agreement protocol, 2023. Available at https://signal.org/docs/specifications/pqxdh/.

[19] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm, 2016. Available at https://signal.org/docs/specifications/doubleratchet/.

[20] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

[21] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède 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.

[22] Douglas Stebila. Security analysis of the iMessage PQ3 protocol. Cryptology ePrint Archive, Report 2024/357, 2024.

A Example: Incomparability of Messaging Protocols

Consider protocols Opp-UniKEM-SM and Opp-BiKEM-SM, instantiations of our secure messaging protocol with the SCKAs of Secs. 4.1 and 4.2. We provide an example showing that the protocols are generally incomparable. To this end, we consider two adversaries that after corrupting both parties execute the protocols for (artificial) communication patterns. With respect to the first Opp-UniKEM-SM regains security but Opp-BiKEM-SM does not, while for the second the opposite holds.

We consider the protocols using chunks of size 32B. Recall that in Opp-UniKEM-SM 37 chunks have to be received to recover ek, 30 to recover ct_0, and 4 to recover ct_1. For Opp-BiKEM-SM 37 chunks have to be received to recover ek and 34 to recover ct. For both adversaries all generated protocol messages are immediately delivered to the other party. More formally, calls to $(\mathsf{ct}, \mathsf{rand}) \leftarrow \mathsf{Send\text{-}P}$ do not make use randomness leakage and are followed up by a call to $\mathsf{Receive\text{-}A}(\mathsf{ct})$.

At a high level, our example uses that:

1. In Opp-UniKEM-SM only A needs to send an encapsulation key at the beginning of an epoch. However, the protocol requires then B to send several additional chunks after receiving A's encapsulation key.

2. In protocol Opp-BiKEM-SM, on the other hand, after both parties have exchanged encapsulation keys either of them is able to establish an epoch key by transmitting a short ciphertext.

We can thus see that Opp-UniKEM-SM requires less upfront communication from B to A but more communication in that direction once the keys have been exchanged.

Opp-UniKEM-SM **outperforming** Opp-BiKEM-SM**.** Assume both parties are compromised immediately after initialization. This implies that until the underlying SCKA returns a new epoch key that is incorporated in the key schedule all messages are compromised. The parties take turns on sending messages as follows.

- A sends 37 messages
- B sends 30 messages
- A sends 1 message
- B sends 4 messages
- A sends 1 message

Consider Opp-UniKEM-SM processing the sequence of messages. After A sent 37 messages, B has fully recovered A's new encapsulation key ek_A, the corresponding dk_A being unknown to the adversary. After B sending 30 messages A recovered the first part ct_0 of the ciphertext, which is acknowledged in A's following message. The following 4 messages by user B suffice to transmit ct_1. Thus A recovers the epoch key and incorporates it in the key schedule before sending the

final message. As a consequence, the final sent message is secure.

Now consider Opp-BiKEM-SM handling the same sequence of messages. After A sent 37 messages, B has fully recovered A's new encapsulation key ek_A. B acknowledges this and sends 30 chunks of their own new encapsulation key ek_B which is not sufficient for A to recover it. Thus A's message contains a chunk of ciphertext ct_A. B sends 4 additional chunks of ek_B, still not sufficient for A to recover ek_B. The final message sent by A contains another chunk of ciphertext ct_A. This is, however, not enough for B to recover ct_A and in turn the encapsulated epoch key. Since no new epoch key has been established after the users' compromise the final message is vulnerable.

Opp-BiKEM-SM **outperforming** Opp-UniKEM-SM**.** Again, assume both parties are compromised immediately after initialization. The parties take turns on sending messages as follows.

- B sends 37 messages
- A sends 37 messages
- B sends 1 message
- A sends 34 messages
- B sends 1 message

Consider Opp-UniKEM-SM processing the sequence of messages. After B sent 37 messages containing chunks of ct_0 (note that A could not acknowledge receiving ct_0) party A has received ct_0. After A sending the following 37 messages B has recovered A's new encapsulation key ek_A. The following message sends one chunk of ct_1 which is not sufficient to recover the ciphertext. Party A's following 34 messages contain no chunks. Since no new epoch key has been established, B's final message (containing the second chunk of ct_1) is vulnerable.

Now consider Opp-BiKEM-SM handling the same sequence of messages. After B and A each sending 37 messages each party recovered the other party's new encapsulation key with the corresponding decapsulation key being unknown to the adversary. The following message by B contains an acknowledgment of B receiving ek_A. Thus, the subsequent 34 messages contain chunks of ciphertext ct_A created by A and encapsulating new epoch key I_A. Party B recovers the epoch key from the 34 chunks and incorporates it in the key schedule before sending the final message. As a consequence, the final sent message is secure.

B Description of Non-opportunistic Protocol Variants

As discussed in Sec. 4, 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.

B.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 $(\mathsf{Encode}, \mathsf{Decode})$. For epoch t user CKA-Send-A generates key-pair $(\mathsf{ek}_\mathsf{A}^\mathsf{t}, \mathsf{dk}_\mathsf{A}^\mathsf{t}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and starts sending chunks of $\mathsf{ek}_\mathsf{A}^\mathsf{t}$. 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 $(\mathsf{ct}_\mathsf{B}^\mathsf{t}, I_\mathsf{B}^\mathsf{t})$ and starts sending chunks of $\mathsf{ct}_\mathsf{B}^\mathsf{t}$, 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 $\mathsf{dk}_\mathsf{A}^\mathsf{t}$, 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

$$\mathsf{t}_\mathsf{A}^\mathsf{snd} = \mathsf{t} - 1 = \mathsf{t}_\mathsf{B}^\mathsf{snd}.$$

Protocol messages include the epoch t of the sent cryptographic material allowing the processing party to compute the matching $\mathsf{t}_\mathsf{A}^\mathsf{rcv}$ or $\mathsf{t}_\mathsf{B}^\mathsf{rcv}$ respectively.

Regarding the vulnerable epoch set note that the only secret stored in the users' state is $\mathsf{dk}_\mathsf{A}^\mathsf{t}$. We thus have $\mathsf{st}_\mathsf{B}.\mathsf{vuln} = \emptyset$ and

$$\mathsf{st}_\mathsf{A}.\mathsf{vuln} = \begin{cases} \{\mathsf{t}\} & \text{if } [\![\mathsf{dk}_\mathsf{A}^\mathsf{t} \neq \bot]\!] \\ \emptyset & \text{else} \end{cases}.$$

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

$$(|\mathsf{ek}|, |\mathsf{ct}|) = (1184\mathrm{B}, 1088\mathrm{B}).$$

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.

B.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 $(ct_A^t, I_A^t) \leftarrow \mathsf{Enc}(ek_B^t)$. 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

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

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

$$\mathsf{st.vuln}_P = \begin{cases} \{t\} & \text{if } [\![dk_P^t \neq \bot]\!] \\ \emptyset & \text{else} \end{cases}.$$

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

$$(|\mathsf{ek}|, |\mathsf{ct}|) = (1184\mathrm{B}, 1088\mathrm{B}).$$

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.

B.3 RKEM-CKA

We provide an overview on RKEM-CKA [14].

Protocol description. The protocol is defined with respect to ratcheting KEM RKEM and erasure code $(\mathsf{Encode}, \mathsf{Decode})$. 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 $(dk_A^{t+1}, ek_A^{t+1}) \leftarrow \mathsf{RKeyGen\text{-}A}(\mathsf{par})$, generating $(ct_A^t, I_A^t, dk_A^{t+1}) \leftarrow \mathsf{REnc\text{-}A}(ek_B^t, dk_A^{t+1})$, 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 $(I_A^t, ek_A^{t+1}) \leftarrow \mathsf{RDec\text{-}B}(ct_A^t, dk_B^t, ek_A^{t+1})$, 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

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

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

$$\mathsf{st}_P.\mathsf{vuln} = \begin{cases} \{t-1, t\} & \text{if } [\![\bot \neq dk_P^t \text{ not updated}]\!] \\ \{t\} & \text{if } [\![\bot \neq dk_P^t \text{ updated}]\!] \\ \emptyset & \text{else} \end{cases}$$

Instantiations. We instantiate RKEM-CKA with the lattice based ratcheting KEM Katana [14]. It exhibits encapsulation key and ciphertext sizes of

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

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.

Contents