Ratcheted key exchange, revisited

Bertram Poettering[1], Paul Rösler[2]

[1] Information Security Group, Royal Holloway, University of London
bertram.poettering@rhul.ac.uk
[2] Horst-Görtz Institute for IT Security,
Chair for Network and Data Security, Ruhr-University Bochum
paul.roesler@rub.de

March 29, 2018

Abstract. Ratcheted key exchange (RKE) is a cryptographic technique used in instant messaging software like Signal and the WhatsApp messenger for attaining strong security in the face of state exposure attacks (including automatic healing from the latter). RKE received first academic attention in the recent works of Cohn-Gordon et al. (EuroS&P 2017) and Bellare et al. (CRYPTO 2017). While the former is analytical in the sense that it aims primarily at assessing the security that one particular protocol *does* achieve, rather than looking for a strong notion of security that it *could* achieve, the authors of the latter follow a different approach in that they first develop a notion of security they want to achieve, and then securely instantiate it. Unfortunately, however, their model is too restricted to serve for the analysis of real-world messenger apps, for considering exclusively unidirectional communication, and for considering only the exposure of the state of only one party.

In this article we resolve the limitations of prior work by developing alternative security definitions, for unidirectional RKE as well as for RKE where both parties can contribute. We follow a purist approach, aiming at finding strong yet convincing notions that cover a realistic communication model; in particular, and in contrast to prior work, our models support fully concurrent operation of both participants.

We further propose secure instantiations (as the protocols analyzed or proposed by Cohn-Gordon et al. and Bellare et al. turn out to be weak in our models). While our scheme for the unidirectional case builds on a generic KEM as the main building block (differently to prior work that requires explicitly Diffie–Hellman), our schemes for bidirectional communication require, perhaps surprisingly, considerably stronger tools.

Table of Contents

1 Introduction 1
2 Preliminaries 5
  2.1 Notation 5
  2.2 Cryptographic primitives 6
  2.3 Key-updatable key encapsulation mechanisms 7
3 Unidirectionally ratcheted key exchange 9
4 Constructing URKE 14
5 Sesquidirectionally ratcheted key exchange 15
6 Constructing SRKE 19
7 Bidirectionally ratcheted key exchange 22
8 Constructing BRKE 24
A Proof of URKE 28
B Rationales for SKRE design 33
  B.1 Signatures from A to B 33
  B.2 Key-updatable KEM for concurrent sending 34
  B.3 Encapsulation to all public keys 35
C Proof of SRKE 36
D Proof of BRKE 46
E Extended preliminaries 49
  E.1 Message authentication codes 49
  E.2 Key encapsulation mechanisms 50
  E.3 One-time signature schemes 50
F Modeling ratcheted key exchange 51
G Related schemes 52

1 Introduction

Asynchronous two-party communication. Assume an online chat situation where two parties, Alice and Bob, communicate by exchanging messages over the Internet (e.g., using a TCP/IP based protocol). Their communication shall follow the structure of a human conversation in the sense that participants send messages when they feel they want to contribute to the discussion, as opposed to in lockstep, i.e., when it is 'their turn'. In particular, in the considered asynchronous setting, Alice and Bob may send messages concurrently, and they also may receive them concurrently after a small delay introduced by the network. With other words, their messages may 'cross' on the wire.

As Alice and Bob are concerned with adversaries attacking their conversation, they deploy cryptographic methods. Standard security goals in this setting are the preservation of confidentiality and integrity of exchanged messages. These can be achieved, for instance, by combining an encryption primitive, a message authentication code, and transmission counters, where the latter serve for identifying replay and reordering attacks [1,16]. As the mentioned cryptographic primitives are based on symmetric keys, Alice and Bob typically engage in an interactive key agreement protocol prior to starting their conversation.

Forward secrecy. In this classic first-key-agreement-then-symmetric-protocol setup for two-party chats, the advantage of investing in an interactive key agreement session goes beyond fulfilling the basic need of the symmetric protocol (the allocation of shared key material): If the key agreement involves a Diffie–Hellman key exchange (DHKE), and this is nowadays the default, then the communication between Alice and Bob may be protected with forward secrecy. The latter means that even if the adversary finds a way, at a point in time after Alice and Bob finish their conversation, to obtain a copy of the long-term secrets they used during key establishment (signature keys, passwords, etc.), then this cannot be exploited to reveal their communication contents. Most current designs of cryptographic chat protocols consider forward secrecy an indispensable design goal [23]. The reason is that inadvertently disclosing long-term secrets is often more likely to happen than expected: system intruders might steal the keys, thieves might extract them from stolen Smartphones, law enforcement agencies might lawfully coerce users to reveal their keys, backup software might unmindfully upload a copy onto network storage, and so on.

Security with exposed state. Modern chat protocols also aim at protecting users from a different kind of attack: the skimming of the session state of an ongoing conversation [23].[1] Note that the session state information is orthogonal to the long-term secrets discussed above and, intuitively, an artifact of exclusively the second (symmetric) phase of communication. The necessity of being able to recover from session state leakage is usually motivated with two observations: messaging sessions are often long-lived, e.g., kept alive for weeks or months once established, so that state exposure is more damaging, more easily provoked, and more likely to happen by accident; and leaking state information is sometimes impossible to defend against (state information held in computer memory might eventually be swapped to disk and stolen from there, and in cloud computing it is standard to move virtual machine memory images around the world from one host to the other).

Ratcheting. Most modern messaging protocols are designed with the goal of providing security even in the face of adversaries that can perform the two types of attack discussed above (compromise of long-term secrets and/or session states) [23]. One technique used towards achieving this

[1] In this article, we consider the terms state reveal, state compromise, state corruption, and state exposure synonyms.

is via ‘hash chains’ where the symmetric key material contained in the session state is replaced, after each use, by a new value derived from the old value by applying some one-way function. This method mainly targets forward security and has a long tradition in cryptography (e.g., is used in [22] in the context of secure logging). A second technique is to let participants routinely redo a DHKE and mix the newly established keys into the session state. This was introduced with the off-the-record (OTR) messaging protocol from [18,4] and promises auto-healing after a state compromise, at least if the DHKE exponents are derived from fresh randomness gathered from an uncorrupted source after the state reveal took place. In brief, the OTR protocol sends, as part of every outgoing message, a fresh g^x value that is combined with prior and later values g^y contributed by the peer, with the goal of refreshing the session state as often as possible. Of course the two methods are not mutually exclusive but can be combined. We say that a messaging protocol employs a ‘key ratchet’ (this name can be traced back to [13]) if it uses the described or similar techniques for achieving forward secrecy and security under state exposure attacks.

Ratcheting as a primitive. According to what we described so far, the word ratcheting refers to a set of techniques deployed with the aim of achieving certain (typically not formally defined) security goals. Recently, Bellare et al. [2] pursued a different approach by proposing *ratcheted key exchange* (RKE) as a cryptographic primitive with clearly defined syntax, functionality, and security properties. This primitive establishes a sequence of session keys that allows for the construction of higher-level protocols, where instant messaging is just one example.[2] Building a messaging protocol on top of RKE offers clear advantages over using ad-hoc designs (as all messaging apps we are aware of do): the modularity allows for easier cryptanalysis, the substitution of constructions by alternatives, etc. We note, however, that the RKE formalization considered in [2] is too limited to serve directly as a building block for secure messaging. In particular, the syntactical framework requires all communication to be unidirectional (in the Alice-to-Bob direction), and the security model counterintuitively assumes that exclusively Alice’s state can be exposed.

We give more details on the results of [2]. In the proposed protocol, Alice’s state has the form (i, K, Y), where integer i counts her send operations, K is a key for a PRF, and $Y = g^y$ is a public key of Bob. Bob’s state has the form (i, K, y). When Alice performs a send operation, she samples a fresh randomness x, computes $\mu \leftarrow \mathrm{F}(K, g^x)$ and $(k, K') \leftarrow \mathrm{H}(i, \mu, g^x, Y^x)$ where F is a PRF and H a random oracle, and outputs k as the established session key and (g^x, μ) as a ciphertext that is sent to Bob. (Value μ serves as a message authentication code for g^x.) The next round’s PRF key is K', i.e., Alice’s new state is $(i + 1, K', Y)$. In this protocol, observe that F and H together implement a ‘hash chain’ and lead to forward secrecy, while the g^x, Y^x inputs to the random oracle can be seen as implementing one DHKE per transmission (where one exponent is static). Turning to the proposed RKE security model, while the corresponding game offers an oracle for compromising Alice’s state, there is no option for similarly exposing Bob. If the model had a corresponding oracle, the protocol would not anymore be secure. Indeed, the following (fully passive) attack exploits that Alice ‘encrypts’ to always the same key Y of Bob: The adversary first reveals Alice’s session state, learning (i, K, Y); it then makes Alice invoke her send routine a couple of times and delivers the respective ciphertexts to Bob’s receive routine in unmodified form; in the final step the adversary exposes Bob and recovers his past session keys using the revealed exponent y. Note that in a pure RKE sense these session keys

[2] Note that RKE, despite its name, is a tool to be used in the symmetric protocol following the preliminary key agreement phase. In [2], and also in this article, the latter is abstracted away into a dedicated state initialization algorithm (or: protocol).

should remain unknown to the adversary: Alice should have recovered from the state exposure, and forward secrecy should have made revealing Bob's state useless.[3]

An analysis of the Signal messaging protocol [15] was conducted by Cohn-Gordon et al. [6]. Concretely, the authors develop a "*model with adversarial queries and freshness conditions that capture the security properties intended by Signal*" [6]. While the work does propose a formal security model, for being geared towards confirming the security of one particular protocol, it may not necessarily serve as a good reference notion for RKE.[4]

Contributions. We follow in the footsteps of [2] and study RKE as a general cryptographic primitive. However, we significantly improve on their results, in three independent directions:

Firstly, we extend the strictly unidirectional RKE concept from [2] towards bidirectional communication. In more detail, if we refer to the setting of [2] as URKE (unidirectional RKE), we introduce SRKE (sesquidirectional[5] RKE) and BRKE (bidirectional RKE). In SRKE, while both Alice and Bob can send ciphertexts to the respective peer, only the ciphertexts sent from Alice to Bob establish session keys. Those sent by Bob have no direct functionality but may help him healing from state exposure. Also in BRKE both parties send ciphertexts, but here the situation is symmetric in that all ciphertexts establish keys (plus allow for healing from state exposure). As fully bidirectional RKE is the ultimate goal, URKE and SRKE introduce the necessary building blocks—both regarding the security model and the instantiation. Consequently we introduce them one after another.

Secondly, we propose an improved security model for URKE, and we introduce security models for SRKE and BRKE. Our bidirectional models assume the likely only practical communication setting for messaging protocols, namely the one in which the operations of both parties can happen concurrently (in contrast to, say, according to a ping-pong pattern). We develop our models following a purist approach: We start with giving the adversary the full set of options to undertake its attack (by allowing session key revelations and state exposures of *both* parties), and then exclude, one by one, those configurations that unavoidably lead to a 'trivial win' (an example for the latter is if the adversary first compromises Bob's state and then correctly 'guesses' the next session key he recovers from an incoming ciphertext). This approach leads to strong and convincing security models (and it becomes quite challenging to actually meet them). We note that the (as we argued) insecure protocol from [2] is considered secure in the model of [2] because the latter was not designed with our strategy in mind, ultimately missing some attacks.

Thirdly, we give provably secure constructions of URKE, SRKE, and BRKE. While all prior RKE protocol proposals, including the one from [2], are explicitly based on DHKE as a low-level tool, our constructions use generic primitives like KEMs, MACs, one-time signatures, and random oracles. The increased level of abstraction not only clarifies on the role that these components play in the constructions, it also increases the freedom when picking acceptable hardness assumptions.

Further details on our URKE construction. In brief, our (unidirectional) URKE scheme combines a hash chain and KEM encapsulations to achieve both forward secrecy and recoverability from state exposure. The crucial difference to the protocol from [2] is that in our scheme the public key of Bob is changed after each use. Concretely, but omitting many details, the

[3] A protocol that achieves security in the described setting is developed in this paper; the central idea behind our construction is that Bob's key pair (y, Y) does not stay fixed but is updated each time a ciphertext is processed.

[4] In fact it defines weaker security than would be natural for RKE. We elaborate on this in Appendix G where we explain why the Signal protocol is not secure in our model.

[5] Recall that 'sesqui' is Latin for one-and-a-half.

state information of Alice is (i, K, Y) as in [2] (but where Y is the *current* public key of Bob), for sending Alice freshly encapsulates a key k^* to Y, then computes $(k, K', k') \leftarrow \mathrm{H}(i, K, Y, k^*)$ using a random oracle H, and finally uses auxiliary key k' to update the old public key Y to a new public key Y that is to be used in her next sending operation. Bob does correspondingly, updating his secret key with each incoming ciphertext. Note that the attack against [2] that we sketched above does not work against this protocol (the adversary would obtain a useless secret key when revealing Bob's state).

Further details on our SRKE construction. Recall that, in SRKE, Bob can send update ciphertexts to Alice, with the idea that this will help him to recover from state exposures. Our protocol algorithms can handle fully concurrent operation of the two participants (in particular, ciphertexts may 'cross' on the wire). This unfortunately adds, as they need to handle multiple "epochs" at the same time, considerably to their complexity. Interestingly, the more involved communication setting is also reflected in stronger primitives that we require for our construction: Our SRKE construction uses a special KEM type that supports so-called key updates (also the latter primitive is constructed in this paper, from HIBE).

In a nutshell, in our SRKE construction, Bob heals from state exposures by generating a fresh (updatable) KEM key pair every now and then, and communicating the public key to Alice. Alice uses the key update functionality to 'fast-forward' these keys into a current state by making them aware of ciphertexts that were exchanged after the keys were sent (by Bob), but before they were received (by Alice). In her following sending operation, Alice encapsulates to a mix of old and new public keys.

Further details on our BRKE construction. We have two BRKE constructions. The first works via the amalgamation of two generic SRKE instances, deployed in reverse directions. To reach full security, the instances need to be carefully tied together, which we do via one-time signatures (akin to the CHK transform [14,5]). We give a proof for this generic construction. The second construction is less generic, namely by combining and interleaving the building blocks of our SRKE scheme in the right way. The advantage of the second scheme is that its ciphertexts are shorter (it saves precisely the one-time signatures).

Introducing SKRE as a natural building block for BRKE is consequently of particular value.

Related work. We already compared our work with the one of [2,6].

The idea of using 'hash chains' for achieving forward security of symmetric cryptographic primitives has been around for quite some time. For instance, [21,22] use this technique to protect the integrity of audit logs. See the overview paper [11] for further constructions and applications.The first formal treatment we are aware of is [3]. A messaging protocol that uses this technique is the (original) Silent Circle Instant Messaging Protocol [17].

The idea of mixing into the user state of messaging protocols additional key material that is continuously established with asymmetric techniques (in particular: DHKE) first appeared in the off-the-record (OTR) messaging protocol from [18,4]. Subsequently, the technique appeared in many communication protocols specifically designed to be privacy-friendly, including the ZRTP telephony protocol [24] and the messaging protocol *Double Ratchet Algorithm* [15] (formerly known as Axolotl). The latter, or close variants thereof, are used by the WhatsApp, Facebook Messenger, and Signal apps. In Appendix G we study these protocols more closely, proposing for each of them an attack that shows that it is not secure in our models.

Academic work in a related field was conducted by [7] who study post-compromise security in (classic) key exchange. Here, security shall be achieved even for sessions established after a full compromise of user secrets. This necessarily requires mixing user state information with key material that is newly established via asymmetric techniques, and is thus related to RKE.

However, we note the functionalities and models of (classic) key exchange and RKE are fundamentally different: The former generally considers multiple participants who have long-term keys and who can run multiple sessions, with the same or different peers, in parallel, while participants of the latter have no long-term keys at all, and thus any two sessions are completely independent. One can finally compare our work with that of [9] who consider (classic) key exchange that outputs not one but a series of keys. While also their model assumes users with long-term keys (and thus dependent sessions), a common denominator with our work is that keys established in a single session are required to be pairwise independent.

Organization. In Section 2 we fix notation and describe the building blocks of our RKE constructions: MACs, KEMs (but with a non-standard syntax), one-time signatures. In Section 3 we develop the URKE syntax and a suitable security model, and present a corresponding construction in Section 4. In Section 5 and 6 we do the same for SRKE and in Sections 7 and 8 for BRKE respectively.

2 Preliminaries

2.1 Notation

If A is a (deterministic or randomized) algorithm we write $A(x)$ for an invocation of A on input x. If A is randomized, we write $A(x) \Rightarrow y$ for the event that the invocation results in value y being the output. We further write $[A(x)] = \{y : \Pr[A(x) \Rightarrow y] > 0\}$ for the effective range of $A(x)$.

If $a \leq b$ are integers, we write $[a \mathinner{..} b]$ for the set $\{a, \ldots, b\}$ and we write $[\,a, \ldots]$ for the set $\{x \in \mathbb{N} : a \leq x\}$. We also give symbolic names to intervals and their boundaries (smallest and largest elements): For an interval $I = [a \mathinner{..} b]$ we write $I^{\vdash}$ for a and $I^{\dashv}$ for b. We denote the Boolean constants True and False with $\mathtt{T}$ and $\mathtt{F}$, respectively. We use Iverson brackets to convert Boolean values into bit values: $[\mathtt{T}] = 1$ and $[\mathtt{F}] = 0$. To compactly express if-then-else expressions we use the ternary operator known from the C programming language: If C is a Boolean condition and e_1, e_2 are arbitrary expressions, the composed expression "$C \mathbin{\mathtt{?}} e_1 : e_2$" evaluates to e_1 if $C = \mathtt{T}$ and to e_2 if $C = \mathtt{F}$.

When we refer to a *list* or *sequence* we mean a (row) vector that can hold arbitrary elements, where the empty list is denoted with ϵ and lists that hold precisely one element are notationally identified with the element itself. A list can be appended to another list with the (associative) concatenation operator $\|$, and we denote the is-prefix-of relation with $\preceq$. For instance, for lists $L_1 = \epsilon$ and $L_2 = a$ and $L_3 = b \,\|\, c$ we have $L_1 \,\|\, L_2 \,\|\, L_3 = a \,\|\, b \,\|\, c$ and $L_1 \preceq L_2 \not\preceq L_3$. Note that if the elements held by two lists are strings (over some alphabet) then the concatenation of the lists does not result in the strings being concatenated; in particular, $\texttt{"ab"} \,\|\, \texttt{"c"} \neq \texttt{"abc"}$. (We do not use string concatenation in this paper, so ambiguities are naturally avoided.)

Program code. We describe algorithms and security experiments using (pseudo-)code. In such code we distinguish the following operators for assigning values to variables: We use symbol '$\leftarrow$' when the assigned value results from a constant expression (including the output of a deterministic algorithm), and we write '$\leftarrow_{\$}$' when the value is either sampled uniformly at random from a finite set or is the output of a randomized algorithm. If we assign a value that is a tuple but we are actually not interested in some of its components, we use symbol '_' to mark positions that shall be ignored. For instance, $(_, b, _) \leftarrow (A, B, C)$ is equivalent to $b \leftarrow B$. If X, Y are sets we write $X \overset{\cup}{\leftarrow} Y$ shorthand for $X \leftarrow X \cup Y$, and if L_1, L_2 are lists we write $L_1 \overset{\|}{\leftarrow} L_2$ shorthand for $L_1 \leftarrow L_1 \,\|\, L_2$. We use bracket notation to denote associative arrays (a data structure that implements a dictionary). Associative arrays can be indexed with

elements from arbitrary sets. For instance, for an associative array A the instruction $A[7] \leftarrow 3$ assigns value 3 to index 7, and the expression $A[\mathtt{abc}] = 5$ tests whether the value at index abc is equal to 5. We write $A[\cdot] \leftarrow x$ to initialize the associative array A by assigning the default value x to all possible indices. For an integer a we write $A[..., a] \leftarrow x$ as a shortcut for 'For all $a' \leq a$: $A[a'] \leftarrow x$'.

GAMES. Our security definitions are based on games played between a challenger and an adversary. Such games are expressed using program code and terminate when the special 'Stop' instruction is executed; the argument of the latter is the outcome of the game. For instance, we write $\Pr[G \Rightarrow 1]$ for the probability that game G terminates by running into a 'Stop with 1' instruction. For a Boolean condition C, in games we write 'Require C' shorthand for 'If $\neg C$: Stop with 0' and we write 'Reward C' shorthand for 'If C: Stop with 1'. The two instructions are used for appraising the actions of the adversary: Intuitively, if the adversary behaves such that a required condition is violated then the adversary definitely 'loses' the game, and if it behaves such that a rewarded condition is met then it 'wins'.

We note there are different established ways to define security games related to key exchange. Some works give very compact definitions (in [2] a ratcheting security notion is compressed, without losing detail, into a single figure), while other works specify game families, parameterized for instance with separate freshnesh predicates (in [6], security notions for ratcheting are divided into the game description and a description of the freshness predicate). We follow the former approach and give a discussion on the modeling of ratcheting in Appendix F.

SCHEME SPECIFICATIONS. We also describe the algorithms of cryptographic schemes using program code. Some algorithms may abort or fail, indicating this by outputting the special symbol $\perp$. This is implicitly assumed to happen whenever an encoded data structure provided by the adversary is to be parsed into components but the encoding turns out to be invalid. A more explicit way of aborting is via the 'Require C' shortcut which, in algorithm specifications, stands for 'If $\neg C$: Return $\perp$'. Also this instruction is typically used to assert that certain conditions hold for user-provided input.

2.2 Cryptographic primitives

Our RKE constructions use MACs, one-time signature schemes, and KEMs as building blocks. As the requirements on the MACs and one-time signatures are absolutely standard, we provide only very reduced definitions here and defer the full specifications to Appendix E.1 and E.3. For KEMs, however, we use a specific syntax, assume a specific functionality, and require a non-standard notion of security.

MACS AND ONE-TIME SIGNATURES. For a MAC M we define that it is keyed from a key space $\mathcal{K}$, and that the tag and verification algorithms are called tag and $\mathrm{vfy}_{\mathsf{M}}$, respectively. Their syntax will always be clear from the context. As a security notion we define strong unforgeability, and the corresponding advantage of an adversary $\mathcal{A}$ we denote with $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{A})$. For a one-time signature scheme S we assume that the key generation algorithm, the signing algorithm, and the verification algorithm are called $\mathrm{gen}_{\mathsf{S}}$ and sgn and $\mathrm{vfy}_{\mathsf{S}}$, respectively. We note that $\mathrm{vfy}_{\mathsf{S}}$ outputs values T or F to indicate its decision, and that the remaining syntax will again be clear from the context. As a security notion we define strong unforgeability and the corresponding advantage of an adversary $\mathcal{A}$ we denote with $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{A})$.

KEY ENCAPSULATION MECHANISMS. We consider a type of KEM where key pairs are generated by first randomly sampling the secret key and then deterministically deriving the public key from it. While this syntax is non-standard, note that it can be assumed without loss of generality:

One can always understand the coins used for (randomized) key generation of a classic KEM as the secret key in our sense.

A *key encapsulation mechanism* (KEM) for a finite session-key space $\mathcal{K}$ is a triple $\mathsf{K} = (\text{gen}_\mathsf{K}, \text{enc}, \text{dec})$ of algorithms together with a samplable secret-key space $\mathcal{SK}$, a public-key space $\mathcal{PK}$, and a ciphertext space $\mathcal{C}$. In its regular form the public-key generation algorithm gen_K is deterministic, takes a secret key $sk \in \mathcal{SK}$, and outputs a public key $pk \in \mathcal{PK}$. We also use a shorthand form, writing gen_K for the randomized procedure of first picking $sk \leftarrow_\$ \mathcal{SK}$, then deriving $pk \leftarrow \text{gen}_\mathsf{K}(sk)$, and finally outputting the pair (sk, pk). Two shortcut notations for key generation are thus

$$\mathcal{SK} \to \text{gen}_\mathsf{K} \to \mathcal{PK} \qquad \text{gen}_\mathsf{K} \to \mathcal{SK} \times \mathcal{PK} \ .$$

The randomized encapsulation algorithm enc takes a public key $pk \in \mathcal{PK}$ and outputs a session key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$, and the deterministic decapsulation algorithm dec takes a secret key $sk \in \mathcal{SK}$ and a ciphertext $c \in \mathcal{C}$, and outputs either a session key $k \in \mathcal{K}$ or the special symbol $\perp \notin \mathcal{K}$ to indicate rejection. Shortcut notations for encapsulation and decapsulation are thus

$$\mathcal{PK} \to \text{enc} \to_\$ \mathcal{K} \times \mathcal{C} \qquad \mathcal{SK} \times \mathcal{C} \to \text{dec} \to \mathcal{K} \ / \perp \ .$$

For correctness we require that for all $(sk, pk) \in [\text{gen}_\mathsf{K}]$ and $(k, c) \in [\text{enc}(pk)]$ we have $\text{dec}(sk, c) = k$.

As a security property for KEMs we formalize a multi-receiver[6]/multi-challenge version of one-way security. In this notion, the adversary obtains challenge ciphertexts and has to recover any of the encapsulated keys. The adversary is supported by a key-checking oracle that, for a provided pair of ciphertext and (candidate) session key, tells whether the ciphertext decapsulates to the indicated key. The adversary is also allowed to establish new receivers, or to expose them, meaning to learn their secret keys. The details of this notion are in game OW in Figure 22 (in Appendix E.2, where we also give further details on the notion). For a KEM K, we associate with any adversary $\mathcal{A}$ its one-way advantage $\text{Adv}^{\text{ow}}_\mathsf{K}(\mathcal{A}) := \Pr[\text{OW}(\mathcal{A}) \Rightarrow 1]$. Intuitively, the KEM is secure if all practical adversaries have a negligible advantage.

2.3 Key-updatable key encapsulation mechanisms

We introduce a type of KEM that we refer to as key-updatable. Like a regular KEM the new primitive establishes secure session keys, but in addition a dedicated key-update algorithm derives new (updated) keys from old ones: Also taking an auxiliary input into account that we call the associated data, a secret key is updated to a new secret key, or a public key is updated to a new public key. A KEM key pair remains functional under such updates, meaning that session keys encapsulated for the public key can be recovered using the secret key, if both keys are updated compatibly, i.e., with matching associated data. Concerning security we require a kind of forward secrecy: Briefly, session keys encapsulated to a (potentially updated) public key shall remain secure even if the adversary gets hold of any incompatibly updated version of the secret key.

A *key-updatable key encapsulation mechanism* (kuKEM) for a finite session-key space $\mathcal{K}$ is a quadruple $\mathsf{K} = (\text{gen}_\mathsf{K}, \text{enc}, \text{dec}, \text{up})$ of algorithms together with a samplable secret-key space $\mathcal{SK}$, a public-key space $\mathcal{PK}$, a ciphertext space $\mathcal{C}$, and an associated-data space $\mathcal{AD}$. Algorithms $\text{gen}_\mathsf{K}, \text{enc}, \text{dec}$ are as for regular KEMs. The key-update algorithm up is deterministic and comes in two shapes: either it takes a secret key $sk \in \mathcal{SK}$ and associated data $ad \in \mathcal{AD}$ and outputs an updated secret key $sk' \in \mathcal{SK}$, or it takes a public key $pk \in \mathcal{PK}$ and associated data $ad \in \mathcal{AD}$ and

[6] Other works refer to the same entity as a user, i.e., consider multi-user security.

outputs an updated public key $pk' \in \mathcal{PK}$. Shortcut notations for the key update algorithm(s) are thus

$$\mathcal{SK} \times \mathcal{AD} \to \mathrm{up} \to \mathcal{SK} \qquad \mathcal{PK} \times \mathcal{AD} \to \mathrm{up} \to \mathcal{PK} \ .$$

For correctness we require that for all $(sk_0, pk_0) \in [\mathrm{gen}_{\mathsf{K}}]$ and $ad_1, \ldots, ad_n \in \mathcal{AD}$, if we let $sk_i = \mathrm{up}(sk_{i-1}, ad_i)$ and $pk_i = \mathrm{up}(pk_{i-1}, ad_i)$ for all i, then for all $(k, c) \in [\mathrm{enc}(pk_n)]$ we have $\mathrm{dec}(sk_n, c) = k$.

As a security property for kuKEMs we formalize a multi-receiver/multi-challenge version of one-way security that also reflects forward security in case of secret-key updates. The details of the notion are in game KUOW in Figure 1. For a key-updatable KEM K, we associate with any adversary $\mathcal{A}$ its one-way advantage $\mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{A}) := \Pr[\mathrm{KUOW}(\mathcal{A}) \Rightarrow 1]$. Intuitively, the kuKEM is secure if all practical adversaries have a negligible advantage.

We extend our definition for regular KEMs by allowing the adversary to also update the public keys held by senders (encryptors) and the secret keys held by receivers, such that if a sender performs s-many updates using associated data from the list $\mathrm{AS} = ad_1 \parallel \ldots \parallel ad_s$, a receiver performs r-many updates using associated data from the list $\mathrm{AR} = ad'_1 \parallel \ldots \parallel ad'_r$, and the receiver is then exposed, then session keys encapsulated by the sender for the receiver remain hidden from the adversary if keys were updated inconsistently (with conflicting associated data, or too often on the receiver side), i.e., technically, if $\mathrm{AR} \not\preceq \mathrm{AS}$ (AR is a not a prefix of AS).

Game KUOW($\mathcal{A}$)
00 $n \leftarrow 0$
01 Invoke $\mathcal{A}$
02 Stop with 0

Oracle Gen
03 $n \leftarrow n + 1$
04 $(sk_n, pk_n) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
05 $\mathrm{CK}_n[\cdot] \leftarrow \perp$; $\mathrm{XP}_n \leftarrow \emptyset$
06 $\mathrm{AS}_n \leftarrow \epsilon$; $\mathrm{AR}_n \leftarrow \epsilon$
07 $SK_n[\cdot] \leftarrow \perp$
08 $SK_n[\mathrm{AR}_n] \leftarrow sk_n$
09 Return pk_n

Oracle $\mathrm{Up}_S(i, ad)$
10 Require $1 \leq i \leq n$
11 $pk_i \leftarrow \mathrm{up}(pk_i, ad)$
12 $\mathrm{AS}_i \xleftarrow{\parallel} ad$
13 Return pk_i

Oracle Enc(i)
14 Require $1 \leq i \leq n$
15 $(k, c) \leftarrow_{\$} \mathrm{enc}(pk_i)$
16 $\mathrm{CK}_i[\mathrm{AS}_i, c] \leftarrow k$
17 Return c

Oracle Solve(i, A, c, k)
18 Require $1 \leq i \leq n$
19 Require $A \notin \mathrm{XP}_i$
20 Require $\mathrm{CK}_i[A, c] \neq \perp$
21 Reward $k = \mathrm{CK}_i[A, c]$
22 Return

Oracle $\mathrm{Up}_R(i, ad)$
23 Require $1 \leq i \leq n$
24 $sk_i \leftarrow \mathrm{up}(sk_i, ad)$
25 $\mathrm{AR}_i \xleftarrow{\parallel} ad$
26 $SK_i[\mathrm{AR}_i] \leftarrow sk_i$
27 Return

Oracle Check(i, A, c, k)
28 Require $1 \leq i \leq n$
29 Require $SK_i[A] \neq \perp$
30 $k' \leftarrow \mathrm{dec}(SK_i[A], c)$
31 Return $[k' = k]$

Oracle Expose(i)
32 Require $1 \leq i \leq n$
33 $\mathrm{XP}_i \xleftarrow{\cup} \{A \in \mathcal{AD}^* : \mathrm{AR}_i \preceq A\}$
34 Return sk_i

Fig. 1: Security experiment KUOW, modeling the one-way security of a key-updatable KEM in a multi-receiver/multi-challenge setting. Oracles Up_S and Up_R update senders and receivers, respectively, and for each receiver i the lists AS_i and AR_i record the associated data used for updating the corresponding sender and receiver keys, respectively. See Figure 22 for an explanation of the other game variables. The instruction in line 33 adjoins to set XP_i the set of all AR_i-prefixed sequences of associated data.

Constructing key-updatable KEMs. Observe that kuKEMs are related to hierarchical identity-based encryption (HIBE, [10]): Intuitively, updating a secret key using associated

data ad in the kuKEM world corresponds in the HIBE world with extracting the decryption/delegation key for the next-lower hierarchy level, using partial identity ad. Indeed, a kuKEM scheme is immediately constructed from a generic HIBE, with only cosmetic changes necessary when annotating the algorithms; a construction is provided in Figure 2. Also the security reduction from kuKEM to HIBE is immediate. For concreteness, in Figure 3 we further detail the algorithm specifications for the case when our generic kuKEM construction is instantiated with the (pairing-based) Gentry–Silverberg HIBE [10]. Other HIBE designs, e.g., lattice-based ones, can be used to instantiate our construction as easily.

Proc gen_K
00 $(sk, pk) \leftarrow_\$ \text{ID.gen}$
01 $ad_0 \leftarrow$ ""
02 $sk \leftarrow_\$ \text{ID.extract}(sk, ad_0)$
03 $A \leftarrow ad_0$; $pk \leftarrow (pk, A)$
04 Return (sk, pk)

Proc $\text{up}(pk, ad)$
05 $(pk', A) \leftarrow pk$
06 $pk \leftarrow (pk', A \parallel ad)$
07 Return pk

Proc $\text{enc}(pk)$
08 $(pk', A) \leftarrow pk$
09 $(k, c) \leftarrow_\$ \text{ID.enc}(pk', A)$
10 Return (k, c)

Proc $\text{dec}(sk, c)$
11 $k \leftarrow \text{ID.dec}(sk, c)$
12 Return k

Proc $\text{up}(sk, ad)$
13 $sk \leftarrow \text{ID.extract}(sk, ad)$
14 Return sk

Fig. 2: Construction of a key-updatable KEM from a generic HIBE. The extraction operation during key generation in line 02 is necessary as some HIBEs (for instance the Gentry–Silverberg scheme [10]) do not support encapsulating to the root node of the hierarchy; our construction thus does the first descent during key setup.

While HIBE schemes generically imply kuKEMs, it is unclear whether the same holds in the reverse direction, i.e., whether HIBEs can be constructed from kuKEMs. The crucial observation is that kuKEMs support only one strand of secret-key updates (recall our KUOW notion says nothing about what happens when a receiver duplicates its secret key and updates it twice, with different associated data), while HIBE schemes support at least two subidentities per node. Indeed, a (separating) example of a secure kuKEM that results in a weak HIBE when converted in the intuitive way is easily found.[7] Our conclusion is that while all kuKEM constructions we are aware of require HIBE and thus practically undesirable building blocks like pairings or lattices, kuKEMs seem to be a strictly weaker primitive than HIBE, so it is more likely to find constructions in the bare DLP setting. We leave it as an open problem to find such a construction.

3 Unidirectionally ratcheted key exchange

We give a definition of unidirectional RKE and its security. While, in principle, our syntactical definition is in line with the one from [2], our naming convention deviates significantly from the latter for the sake of a more clear distinction between (session) keys, (session) states, and ciphertexts[8] and we stress that, looking ahead, our security notion for URKE is strictly stronger than the one of [2]. A speciality of our formalization is that we let the sending and

[7] For instance, conceptually, when updating a secret key to a new one, the kuKEM could secret-share the old key using a two-out-of-two threshold scheme, randomly pick one of the shares and include it in the new key, discarding the other share. While this would not hurt kuKEM security, a naively derived HIBE would be trivial to break.

[8] The mapping between our names (on the left of the equality sign) and the ones of [2] (on the right) is as follows: '(session) key' = 'output key', '(session) state' = 'session key plus sender/receiver key', 'ciphertext' = 'update information'.

```
Proc gen_K                          Proc enc(pk)
00 P_0, P_1 ←$ G_1                  20 (Q_0, _, l, P) ← pk
01 P ← (P_0, P_1)                   21 (P_0, ..., P_l) ← P
02 s_0 ←$ Z_q                       22 r ←$ Z_q
03 S_1 ← s_0 P_1                    23 K ← r⟨Q_0, P_1⟩
04 Q_0 ← s_0 P_0                    24 For i ∈ {0, 2, ..., l}:
05 sk ← (P_0, ε, ε, 1, S_1)         25    C_i ← r P_i
06 pk ← (Q_0, ε, 1, P)              26 C ← (C_0, C_2, ..., C_l)
07 Return (sk, pk)                  27 Return (K, C)

Proc up(pk, ad)                     Proc dec(sk, C)
08 (Q_0, A, l, P) ← pk              28 (_, Q, _, l, S_l) ← sk
09 A ←" ad; P_{l+1} ← H(A)          29 (Q_1, ..., Q_{l−1}) ← Q
10 P ←" P_{l+1}                     30 (C_0, C_2, ..., C_l) ← C
11 pk ← (Q_0, A, l+1, P)            31 K ← ⟨C_0, S⟩
12 Return pk                        32 For i ∈ {2, ..., l}:
                                    33    K ← K − ⟨Q_{i−1}, C_i⟩
Proc up(sk, ad)                     34 Return K
13 (P_0, Q, A, l, S_l) ← sk
14 A ←" ad; P_{l+1} ← H(A)
15 s_l ←$ Z_q
16 Q_l ← s_l P_0; Q ←" Q_l
17 S_{l+1} ← S_l + s_l P_{l+1}
18 sk ← (P_0, Q, A, l+1, S_{l+1})
19 Return sk
```

Fig. 3: Direct construction of a key-updatable KEM, inspired by the Gentry–Silverberg HIBE [10]. We assume (additively written) groups G_1, G_2 of prime order q, a pairing $\langle\cdot,\cdot\rangle\colon G_1 \times G_1 \to G_2$, and a random oracle $H\colon \{0,1\}^* \to G_1$. To see the scheme's correctness, observe for instance that after two updates we have $l = 3$ and in line 31 obtain $K = \langle C_0, S\rangle = rs_0\langle P_0, P_1\rangle + rs_1\langle P_0, P_2\rangle + rs_2\langle P_0, P_3\rangle$, from which in line 33 we substract $s_1 r\langle P_0, P_2\rangle$ and $s_2 r\langle P_0, P_3\rangle$, so that the overall result is $rs_0\langle P_0, P_1\rangle$, which matches the assignment in line 23. Note that the update procedure for secret keys is randomized, conflicting with our syntactical definition of a kuKEM; however, derandomizing it using a PRF, with the corresponding key stored as a further element of *sk*, is straight-forward.

receiving algorithms of Alice and Bob accept and process an associated data string [20] that, for functionality, has to match on both sides.

A *unidirectionally ratcheted key exchange* (URKE) for a finite key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$ is a triple $\mathsf{R} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$ of algorithms together with a sender state space $\mathcal{S}_A$, a receiver state space $\mathcal{S}_B$, and a ciphertext space $\mathcal{C}$. The randomized initialization algorithm init returns a sender state $S_A \in \mathcal{S}_A$ and a receiver state $S_B \in \mathcal{S}_B$. The randomized sending algorithm snd takes a state $S_A \in \mathcal{S}_A$ and an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $S'_A \in \mathcal{S}_A$, a key $k \in \mathcal{K}$, and a ciphertext $c \in \mathcal{C}$. Finally, the deterministic receiving algorithm rcv takes a state $S_B \in \mathcal{S}_B$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and either outputs an updated state $S'_B \in \mathcal{S}_B$ and a key $k \in \mathcal{K}$, or the special symbol $\perp$ to indicate rejection. A shortcut notation for these syntactical definitions and a visual illustration of the URKE communication setup is

$$\begin{array}{rcccl} & & \mathrm{init} & \to & \mathcal{S}_A \times \mathcal{S}_B \\ \mathcal{S}_A \times \mathcal{AD} & \to & \mathrm{snd} & \to & \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\ \mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} & \to & \mathrm{rcv} & \to & \mathcal{S}_B \times \mathcal{K} \ / \ \perp \end{array}$$

$$\begin{array}{ccccc} & \downarrow \mathrm{state}_A & & \downarrow \mathrm{state}_B & \\ ad \to & \boxed{\downarrow \mathrm{snd}} & \to c \to & \boxed{\downarrow \mathrm{rcv}} & \leftarrow ad \\ k \leftarrow & & & & \to k \\ & \downarrow \mathrm{state}_A & & \downarrow \mathrm{state}_B & \end{array}$$

Correctness of URKE. Assume a sender and a receiver that were jointly initialized with init. Then, intuitively, the URKE scheme is correct if for all sequences (ad_i) of associated-data strings, if (k_i) and (c_i) are sequences of keys and ciphertexts successively produced by the sender on

input the strings in (ad_i), and if (k'_i) is the sequence of keys output by the receiver on input the (same) strings in (ad_i) and the ciphertexts in (c_i), then the keys of the sender and the receiver match, i.e., it holds that $k_i = k'_i$ for all i.

We formalize this requirement via the FUNC game in Figure 4.[9] Concretely, we say scheme R is *correct* if $\Pr[\mathrm{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ for all adversaries $\mathcal{A}$. In the game, the adversary lets the sender and the receiver process associated-data strings and ciphertexts of its choosing, and its goal is to let the two parties compute keys that do not match when they should. Variables s_A and r_B count the send and receive operations, associative array adc_A jointly records the associated-data strings respected by and the ciphertexts produced by the sender, flag is_B is an indicator that tracks whether the receiver is still 'in-sync' (in contrast to: was exposed to non-matching associated-data strings or ciphertexts; note how the transition between in-sync and out-of-sync is detected and recorded in lines 12,13), and associative array key_A records the keys established by the sender to allow for a comparison with the keys recovered (or not) by the receiver. The correctness requirement boils down to declaring the adversary successful (in line 16) if the sender and the receiver compute different keys while still being in-sync. Note finally that lines 11,15 ensure that once the rcv algorithm rejects, the adversary is notified of this and further queries to the RcvB oracle are not accepted.

Game $\mathrm{FUNC}_{\mathsf{R}}(\mathcal{A})$
00 $s_A \leftarrow 0;\ r_B \leftarrow 0$
01 $adc_A[\cdot] \leftarrow \bot;\ is_B \leftarrow \mathtt{T}$
02 $key_A[\cdot] \leftarrow \bot$
03 $(S_A, S_B) \leftarrow_\$ \mathrm{init}$
04 Invoke $\mathcal{A}$
05 Stop with 0

Oracle $\mathrm{SndA}(ad)$
06 $(S_A, k, c) \leftarrow_\$ \mathrm{snd}(S_A, ad)$
07 $adc_A[s_A] \leftarrow (ad, c)$
08 $key_A[s_A] \leftarrow k$
09 $s_A \leftarrow s_A + 1$
10 Return c

Oracle $\mathrm{RcvB}(ad, c)$
11 Require $S_B \neq \bot$
12 If $is_B \wedge adc_A[r_B] \neq (ad, c)$:
13 $\quad is_B \leftarrow \mathtt{F}$
14 $(S_B, k) \leftarrow \mathrm{rcv}(S_B, ad, c)$
15 If $S_B = \bot$: Return $\bot$
16 Reward $is_B \wedge k \neq key_A[r_B]$
17 $r_B \leftarrow r_B + 1$
18 Return

Fig. 4: Game FUNC for URKE scheme R.

Security of URKE. We formalize a key indistinguishability notion for URKE. In a nutshell, from the point of view of the adversary, keys established by the sender and recovered by the receiver shall look uniformly distributed in the key space. In our model, the adversary, in addition to scheduling the regular URKE operations via the SndA and RcvB oracles, has to its disposal the four oracles ExposeA, ExposeB, Reveal, and Challenge, used for exposing users by obtaining copies of their current state, for learning established keys, and for requesting real-or-random challenges on established keys, respectively. For an URKE scheme R, in Figure 5 we specify corresponding key indistinguishability games $\mathrm{KIND}^b_{\mathsf{R}}$, where $b \in \{0,1\}$ is the challenge bit, and we associate with any adversary $\mathcal{A}$ its key distinguishing advantage $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) := |\Pr[\mathrm{KIND}^1_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathrm{KIND}^0_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1]|$. Intuitively, R offers key indistinguishability if all practical adversaries have a negligible key distinguishing advantage.

Most lines of code in the KIND^b games are tagged with a '$\cdot$' right after the line number; to the subset of lines marked in this way we refer to as the games' *core*. Conceptually, the

[9] Formalizing correctness of URKE via a game might at first seem overkill. However, for SRKE and BRKE, which allow for interleaved interaction in two directions, game-based definitions seem to be natural and notationally superior to any other approach. For consistency we use a game-based definition also for URKE.

```
Game KIND^b_R(A)
00 · s_A ← 0; r_B ← 0
01 · adc_A[·] ← ⊥; is_B ← T
02 · key_A[·] ← ⊥; key_B[·] ← ⊥
03   XP_A ← ∅
04   TR_A ← ∅; TR_B ← ∅
05   CH_A ← ∅; CH_B ← ∅
06 · (S_A, S_B) ←$ init
07 · b' ←$ A
08   Require TR_A ∩ CH_A = ∅
09   Require TR_B ∩ CH_B = ∅
10 · Stop with b'

Oracle SndA(ad)
11 · (S_A, k, c) ←$ snd(S_A, ad)
12 · adc_A[s_A] ← (ad, c)
13 · key_A[s_A] ← k
14 · s_A ← s_A + 1
15 · Return c

Oracle ExposeA
16   XP_A ←∪ {s_A}
17 · Return S_A

Oracle Reveal(u, i)
18 · Require key_u[i] ∈ K
19 · k ← key_u[i]
20   key_u[i] ← ⋄
21 · Return k

Oracle RcvB(ad, c)
22 · Require S_B ≠ ⊥
23 · If is_B ∧ adc_A[r_B] ≠ (ad, c):
24 ·    is_B ← F
25      If r_B ∈ XP_A:
26         TR_B ←∪ [r_B, ...]
27 · (S_B, k) ← rcv(S_B, ad, c)
28 · If S_B = ⊥: Return ⊥
29   If is_B: k ← ⋄
30 · key_B[r_B] ← k
31 · r_B ← r_B + 1
32 · Return

Oracle ExposeB
33   TR_B ←∪ [r_B, ...]
34   If is_B:
35      TR_A ←∪ [r_B, ...]
36 · Return S_B

Oracle Challenge(u, i)
37 · Require key_u[i] ∈ K
38 · k ← b ? key_u[i] : $(K)
39   key_u[i] ← ⋄
40   CH_u ←∪ {i}
41 · Return k
```

Fig. 5: Games KIND^b, $b \in \{0,1\}$, for URKE scheme R. We require $\diamond \notin \mathcal{K}$, and in Reveal and Challenge queries we require $u \in \{A, B\}$. If the notation in lines 26 or 38 is unclear, please consult Section 2.

cores contain all relevant game logic (participant initialization, specifications of how queries are answered, etc.); the code lines available only in the full game, i.e., the untagged ones, introduce certain restrictions on the adversary that are necessary to exclude trivial attacks (see below). The games' cores should be self-explanatory, in particular when comparing them to the FUNC game, with the understanding that lines 18,37 (in Figure 5) ensure that only keys can be revealed or challenged that actually have been established before, and that line 38 assigns to variable k, depending on bit b, either the real key or a freshly sampled element from the key space.

Note that, in the pure core code, the adversary can use the four new oracles to bring itself into the position to distinguish real and random keys in a trivial way. In the following we discuss five different strategies to do so. We illustrate each strategy by specifying an example adversary in pseudocode and we explain what measures the full games take for disregarding the respective class of attack. (That is, the example adversaries would gain high advantage if the games consisted of just their cores, but in the full games their advantage is zero.)

The first two strategies leverage on the interplay of Reveal and Challenge queries; they do not involve exposing participants.

(a) The adversary requests a challenge on a key that it also reveals, it requests two challenges on the same key, or similar. Example: fix some ad; $c \leftarrow \text{SndA}(ad)$; $k \leftarrow \text{Reveal}(A, 0)$; $k' \leftarrow \text{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; output b'. The full games, in lines 20,39, overwrite keys that are revealed or challenged with the special symbol $\diamond \notin \mathcal{K}$. Because of lines 18,37, this prevents any second Reveal or Challenge query involving the same key.

(b) The adversary combines an attack from (a) with the correctness guarantee, i.e., that in-sync receivers recover the keys established by senders. For instance, the adversary reveals a

sender key and requests a challenge on the corresponding receiver key. Example: fix some ad; $c \leftarrow \text{SndA}(ad)$; $k \leftarrow \text{Reveal}(A, 0)$; $\text{RcvB}(ad, c)$; $k' \leftarrow \text{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; output b'. The full games, in line 29, overwrite in-sync receiver keys, as they are known (by correctness) to be the same on the sender side, with the special symbol $\diamond \notin \mathcal{K}$. By lines 18,37, this rules out the attack.

The remaining three strategies involve exposing participants and using their state to either trace their computations or impersonate them to their peer. In the full games, the set variables $\text{XP}_A, \text{TR}_A, \text{TR}_B, \text{CH}_A, \text{CH}_B$ (lines 03–05) help identifying when such attacks occur. Concretely, set XP_A tracks the points in time the sender is exposed (the unit of time being the number of past sending operations; see line 16), sets TR_A, TR_B track the indices of keys that are 'traceable' (in particular: recoverable by the adversary) using an exposed state (see below), and sets CH_A, CH_B record the indices of keys for which a challenge was requested (see line 40). Lines 08,09 ensure that any adversary that requests to be challenged on a traceable key has advantage zero. Strategies (c) and (d) are state tracing attacks, while strategy (e) is based on impersonation.

(c) The adversary exposes the receiver and uses the obtained state to trace its computations: By iteratively applying the rcv algorithm to all later inputs of the receiver, and updating the exposed state correspondingly, the adversary implicitly obtains a copy of all later receiver keys. Example: fix some ad; $c \leftarrow \text{SndA}(ad)$; $S_B^* \leftarrow \text{ExposeB}()$; $(S_B^*, k) \leftarrow \text{rcv}(S_B^*, ad, c)$; $\text{RcvB}(ad, c)$; $k' \leftarrow \text{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; output b'. When an exposure of the receiver happens, the full games, in line 33, mark all future receiver keys as traceable.

(d) The adversary combines the attack from (c) with the correctness guarantee, i.e., that in-sync receivers recover the keys established by senders: After exposing an in-sync receiver, by iteratively applying the rcv algorithm to all later outputs of the sender, the adversary implicitly obtains a copy of all later sender keys. Example: fix some ad; $c \leftarrow \text{SndA}(ad)$; $S_B^* \leftarrow \text{ExposeB}()$; $(S_B^*, k) \leftarrow \text{rcv}(S_B^*, ad, c)$; $k' \leftarrow \text{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; output b'. When an exposure of an in-sync receiver happens, the full games, in lines 34,35, mark all future sender keys as traceable.

(e) Exposing the sender allows for impersonating it: The adversary obtains a copy of the sender's state and invokes the snd algorithm with it, obtaining a key and a ciphertext. The latter is provided to an in-sync receiver (rendering the latter out-of-sync), who recovers a key that is already known to the adversary. Example: fix some ad; $S_A^* \leftarrow \text{ExposeA}()$; $(S_A^*, k, c) \leftarrow_{\$} \text{snd}(S_A^*, ad)$; $\text{RcvB}(ad, c)$; $k' \leftarrow \text{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; output b'. The full games, in lines 25,26, detect the described type of impersonation and mark all future receiver keys as traceable.

We conclude with some notes on our URKE model. First, the model excludes the (anyway unavoidable) trivial attack conditions we identified, but nothing else. This establishes confidence in the model, as no attacks can be missed. Further, observe that it is not possible to recover from an attack based on state exposure (i.e., of the (c)–(e) types): If *one* key of a participant becomes weak as a consequence of a state exposure, then necessarily *all* later keys of that participant become weak as well. On the other hand, exposing the sender and *not* bringing the receiver out-of-sync does not affect security at all.[10] Finally, exposing an out-of-sync receiver does not harm later sender keys. In later sections we consider ratcheting primitives (SRKE, BRKE) that resume safe operation after state exposure attacks.

[10] This is precisely the distinguishing auto-recovery property of ratcheted key exchange.

4 Constructing URKE

We construct an URKE scheme that is provably secure in the model presented in the previous section. The ingredients are a KEM (with deterministic public-key generation, see Section 2), a SUF-CMA secure MAC, and a random oracle H. The algorithms of our scheme are specified in Figure 7.

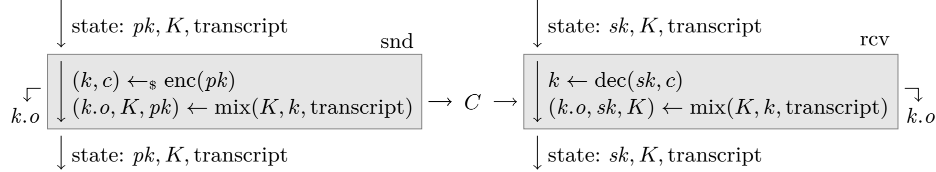


Fig. 6: Principle of our URKE construction (simplified). To obtain the full construction, MAC computations have to be added, likewise associated-data support, and the mix function has to be instantiated. (Observe that the mixing function in snd outputs a public key while the mixing function in rcv outputs a secret key.)

We describe protocol states and algorithms in more detail. The state of Alice consists of (Bob's) KEM public key pk, a chaining key K, a MAC key $k.m$, and a transcript variable t that accumulates the associated data strings and ciphertexts that Alice processed so far. The state of Bob is almost the same, but instead of the KEM public key he holds the corresponding secret key sk. Initially, sk and pk are freshly generated, random values are assigned to K and $k.m$, and the transcript accumulator t is set to the empty string. A sending operation of Alice consists of invoking the KEM encapsulation routine with Bob's current public key, computing a MAC tag over the ciphertext and the associated data, updating the transcript accumulator, and jointly processing the session key established by the KEM, the chaining key, and the current transcript with the random oracle H. The output of H is split into the URKE session key $k.o$, an updated chaining key, an updated MAC key, and, indirectly, the updated public key (of Bob) to which Alice encapsulates in the next round. The receiving operation of Bob is analogue to these instructions. While our scheme has some similarity with the one of [2], a considerable difference is that the public and secret keys held by Alice and Bob, respectively, are constantly changed. This rules out the attack described in the introduction.

Proc init	**Proc** snd(S_A, ad)	**Proc** rcv(S_B, ad, C)
00 $(sk, pk) \leftarrow_\$ \text{gen}_\mathsf{K}$	06 $(pk, K, k.m, t) \leftarrow S_A$	16 $(sk, K, k.m, t) \leftarrow S_B$
01 $K \leftarrow_\$ \mathcal{K}$; $k.m \leftarrow_\$ \mathcal{K}$	07 $(k, c) \leftarrow_\$ \text{enc}(pk)$	17 $c \parallel \tau \leftarrow C$
02 $t \leftarrow \epsilon$	08 $\tau \leftarrow_\$ \text{tag}(k.m, ad \parallel c)$	18 Require $\text{vfy}_\mathsf{M}(k.m, ad \parallel c, \tau)$
03 $S_A \leftarrow (pk, K, k.m, t)$	09 $C \leftarrow c \parallel \tau$	19 $k \leftarrow \text{dec}(sk, c)$
04 $S_B \leftarrow (sk, K, k.m, t)$	10 $t \overset{\shortparallel}{\leftarrow} ad \parallel C$	20 Require $k \neq \bot$
05 Return (S_A, S_B)	11 $k.o \parallel K \parallel k.m \parallel sk \leftarrow$	21 $t \overset{\shortparallel}{\leftarrow} ad \parallel C$
	12 $\text{H}(K, k, t)$	22 $k.o \parallel K \parallel k.m \parallel sk \leftarrow$
	13 $pk \leftarrow \text{gen}_\mathsf{K}(sk)$	23 $\text{H}(K, k, t)$
	14 $S_A \leftarrow (pk, K, k.m, t)$	24 $S_B \leftarrow (sk, K, k.m, t)$
	15 Return $(S_A, k.o, C)$	25 Return $(S_B, k.o)$

Fig. 7: Construction of an URKE scheme from a key-encapsulation mechanism $\mathsf{K} = (\text{gen}_\mathsf{K}, \text{enc}, \text{dec})$, a message authentication code $\mathsf{M} = (\text{tag}, \text{vfy}_\mathsf{M})$, and a random oracle H. For simplicity we denote the key space of the MAC and the space of chaining keys with the same symbol $\mathcal{K}$.

Note that our scheme is specified such that participants accumulate in their state the full past communication history. While this eases the security analysis (random oracle evaluations

of Alice and Bob are guaranteed to be on different inputs once the in-sync bit is cleared), it also seems to impose a severe implementation obstacle. However, as current hash functions like SHA2 and SHA3 process inputs in an online fashion (i.e., left-to-right with a small state overhead), they can process append-only inputs like transcripts such that computations are efficiently shared with prior invocations. In particular, with such a hash function our URKE scheme can be implemented with constant-size state. (This requires, though, rearranging the input of H such that t comes first).[11]

Theorem 1. *The URKE protocol* R *from Figure 7 offers key indistinguishability. More precisely, if function* H *is modeled as a random oracle, for every adversary* $\mathcal{A}$ *for games* $\mathrm{KIND}^b_{\mathsf{R}}$ *from Figure 5 there exists an adversary* $\mathcal{B}$ *for game* OW *from Figure 22 and an adversary* $\mathcal{C}$ *for game* SUF *from Figure 21 such that* $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{ow}}_{\mathsf{K}}(\mathcal{B}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{C}) + \frac{q_S q_F}{|\mathcal{K}_{\mathsf{K}}|}$*, where* $\mathcal{K}_{\mathsf{K}}$ *is the session-key space of the KEM, the running time of* $\mathcal{B}$ *is about that of* $\mathcal{A}$ *plus* q_F *key checking and solve operations, the running time of* $\mathcal{C}$ *is about that of* $\mathcal{A}$*, and* q_F, q_S *are the numbers of* $\mathcal{A}$*'s random oracle and* SndA *queries, respectively.*

The proof of Theorem 1 is in Appendix A. Briefly, it first shows that none of Alice's established session keys can be derived by the adversary without breaking the security of the KEM as long as no previous secret key of Alice's public keys was exposed. Then we show that Bob will only establish session keys out of sync if Alice was impersonated towards him, his state was exposed before, or a MAC forgery was conducted by the adversary. The latter will be reduced to the security of the MAC. Consequently the adversary either breaks one of the employed primitives' security or has information-theoretically small advantage in winning the KIND game.

5 Sesquidirectionally ratcheted key exchange

We introduce *sesquidirectionally ratcheted key exchange* (SRKE)[5] as a generalization of URKE. The basic functionality of the two primitives is the same: Sessions involve two parties, A and B, where A can establish keys and safely share them with B by providing the latter with ciphertexts. In contrast to the URKE case, in SRKE also party B can generate and send ciphertexts (to A); however, B's invocations of the sending routine do not establish keys. Rather, the idea behind B communicating ciphertexts to A is that this may increase the security of the keys established by A. Indeed, as we will see, in SRKE it is possible for B to recover from attacks involving state exposure. We proceed with formalizing syntax and correctness of SRKE.

Formally, a SRKE scheme for a finite key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$ is a tuple $\mathsf{R} = (\mathrm{init}, \mathrm{snd}_A, \mathrm{rcv}_B, \mathrm{snd}_B, \mathrm{rcv}_A)$ of algorithms together with a state space $\mathcal{S}_A$, a state space $\mathcal{S}_B$, and a ciphertext space $\mathcal{C}$. The randomized initialization algorithm init returns a state $S_A \in \mathcal{S}_A$ and a state $S_B \in \mathcal{S}_B$. The randomized sending algorithm snd_A takes a state $S_A \in \mathcal{S}_A$ and an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $S'_A \in \mathcal{S}_A$, a key $k \in \mathcal{K}$, and a ciphertext $c \in \mathcal{C}$. The deterministic receiving algorithm rcv_B takes a state $S_B \in \mathcal{S}_B$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and outputs either an updated state $S'_B \in \mathcal{S}_B$ and a key $k \in \mathcal{K}$, or the special symbol $\perp$ to indicate rejection. The randomized sending algorithm snd_B takes a state $S_B \in \mathcal{S}_B$ and an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $S'_B \in \mathcal{S}_B$ and a ciphertext $c \in \mathcal{C}$. Finally, the deterministic receiving algorithm rcv_A takes a state $S_A \in \mathcal{S}_A$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext

[11] A different approach to achieve a constant-size state is to replace lines 10 and 21 by the (non-accumulating) assignments $t \leftarrow (ad, C)$. We believe our scheme would also be secure in this case as, intuitively, chaining key K reflects the full past communication.

$c \in \mathcal{C}$, and outputs either an updated state $S'_A \in \mathcal{S}_A$ or the special symbol $\perp$ to indicate rejection. A shortcut notation for these syntactical definitions is

$$\begin{array}{rcccl} & & \text{init} & \rightarrow_\$ & \mathcal{S}_A \times \mathcal{S}_B \\ \mathcal{S}_A \times \mathcal{AD} & \rightarrow & \text{snd}_A & \rightarrow_\$ & \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\ \mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} & \rightarrow & \text{rcv}_B & \rightarrow & \mathcal{S}_B \times \mathcal{K} \quad / \quad \perp \\ \mathcal{S}_B \times \mathcal{AD} & \rightarrow & \text{snd}_B & \rightarrow_\$ & \mathcal{S}_B \times \mathcal{C} \\ \mathcal{S}_A \times \mathcal{AD} \times \mathcal{C} & \rightarrow & \text{rcv}_A & \rightarrow & \mathcal{S}_A \qquad / \quad \perp \quad . \end{array}$$

$$\begin{array}{ccccc} & \downarrow \text{state}_A & & \downarrow \text{state}_B & \\ ad \rightarrow & \boxed{\downarrow \text{snd}_A} & \rightarrow c \rightarrow & \boxed{\downarrow \text{rcv}_B} & \leftarrow ad \\ k \leftarrow & & & & \rightarrow k \\ & \downarrow \text{state}_A & & \downarrow \text{state}_B & \\ & \ldots & & \ldots & \\ & \downarrow \text{state}_A & & \downarrow \text{state}_B & \\ ad \rightarrow & \boxed{\downarrow \text{rcv}_A} & \leftarrow c \leftarrow & \boxed{\downarrow \text{snd}_B} & \leftarrow ad \\ & \downarrow \text{state}_A & & \downarrow \text{state}_B & \end{array}$$

Correctness of SRKE. Our definition of SRKE functionality is via game FUNC in Figure 8. (The lines of code not tagged with a '$\cdot$' right after the line number are redundant, i.e., play no role in the game outcome; we ignore these lines in this paragraph.) We say scheme R is *correct* if $\Pr[\text{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ for all adversaries $\mathcal{A}$. In comparison with the corresponding URKE game in Figure 4, there are two more oracles, SndB and RcvA, and four new game variables, s_B, r_A, adc_B, is_A, that control and monitor the communication in the B-to-A direction akin to how $\text{SndA}, \text{RcvB}, s_A, r_B, adc_A, is_B$ do (like in the URKE case) for the A-to-B direction. In particular, the is_A flag is the in-sync indicator of party A that tracks whether the latter was exposed to non-matching associated-data strings or ciphertexts (the transition between in-sync and out-of-sync is detected and recorded in lines 19,20). Given that the specifications of oracles SndA and RcvB of Figures 4 and 8 coincide (with one exception: lines 13,38 are guarded by in-sync checks (in lines 12,37) so that parties go out-of-sync not only when processing unauthentic associated data or ciphertexts, but also when they process ciphertexts that were generated by an out-of-sync peer[12]), and that also the specifications of oracles SndB and RcvA of Figures 8 are quite similar to them (besides the reversion of the direction of communication, the difference is that all key-related components were stripped off), the logics of the FUNC game in Figure 8 should be clear. Overall, like in the URKE case, the correctness requirement boils down to declaring the adversary successful, in line 32, if A and B compute different keys while still being in-sync.

Epochs of SRKE. The intuition behind having the B-to-A direction of communication in SRKE is that it allows B to refresh his state every now and then, and to inform A about this. The goal is to let B recover from state exposure.

Imagine a SRKE session where B has the following view on the communication: first he sends four refresh ciphertexts (to A) in a row; then he receives a key-establishing ciphertext (from A). As we assume a fully concurrent setting and do not impose timing constraints on the network delivery, the incoming ciphertext can have been crafted by A after her having received (from B) between zero and four ciphertexts. That is, even though B refreshed his state a couple of times, to achieve correctness he has to remain prepared for recovering keys from ciphertexts that were generated by A before she recognized any of the refreshs. However, after processing A's ciphertext, if A created it after receiving some of B's ciphertexts (say, the first three), then the situation changes in that B is no longer required to process ciphertexts that refer to refreshs older than the one to which A's current answer is responding to (in the example: the first two).

These ideas turn out to be pivotal in the definition of SRKE security. We formalize them by introducing the notion of an *epoch*. Epochs start when the snd_B algorithm is invoked (each invocation starts one epoch), they are sequentially numbered, and the first epoch (with number zero) is implicitly started by the init algorithm. Each rcv_A invocation makes A aware of one new epoch, and subsequent snd_A invocations can be seen as occurring in its context. Finally, on

[12] This approach is borrowed from [16].

Game $\text{FUNC}_{\mathsf{R}}(\mathcal{A})$
00 · $s_A \leftarrow 0$; $r_B \leftarrow 0$
01 · $s_B \leftarrow 0$; $r_A \leftarrow 0$
02 $e_A \leftarrow 0$; $\text{EP}_A[\cdot] \leftarrow \bot$
03 $E_B^{\vdash} \leftarrow 0$; $E_B^{\dashv} \leftarrow 0$
04 · $adc_A[\cdot] \leftarrow \bot$; $is_B \leftarrow \texttt{T}$
05 · $adc_B[\cdot] \leftarrow \bot$; $is_A \leftarrow \texttt{T}$
06 · $key_A[\cdot] \leftarrow \bot$
07 · $(S_A, S_B) \leftarrow_{\$}$ init
08 · Invoke $\mathcal{A}$
09 · Stop with 0

Oracle $\text{SndA}(ad)$
10 · Require $S_A \neq \bot$
11 · $(S_A, k, c) \leftarrow_{\$} \text{snd}_A(S_A, ad)$
12 · If is_A:
13 · $\quad adc_A[s_A] \leftarrow (ad, c)$
14 $\quad \text{EP}_A[s_A] \leftarrow e_A$
15 · $key_A[s_A] \leftarrow k$
16 · $s_A \leftarrow s_A + 1$
17 · Return c

Oracle $\text{RcvA}(ad, c)$
18 · Require $S_A \neq \bot$
19 · If $is_A \wedge adc_B[r_A] \neq (ad, c)$:
20 · $\quad is_A \leftarrow \texttt{F}$
21 If is_A: $e_A \leftarrow e_A + 1$
22 · $S_A \leftarrow \text{rcv}_A(S_A, ad, c)$
23 · If $S_A = \bot$: Return $\bot$
24 · $r_A \leftarrow r_A + 1$
25 · Return

Oracle $\text{RcvB}(ad, c)$
26 · Require $S_B \neq \bot$
27 · If $is_B \wedge adc_A[r_B] \neq (ad, c)$:
28 · $\quad is_B \leftarrow \texttt{F}$
29 If is_B: $E_B^{\vdash} \leftarrow \text{EP}_A[r_B]$
30 · $(S_B, k) \leftarrow \text{rcv}_B(S_B, ad, c)$
31 · If $S_B = \bot$: Return $\bot$
32 · Reward $is_B \wedge k \neq key_A[r_B]$
33 · $r_B \leftarrow r_B + 1$
34 · Return

Oracle $\text{SndB}(ad)$
35 · Require $S_B \neq \bot$
36 · $(S_B, c) \leftarrow_{\$} \text{snd}_B(S_B, ad)$
37 · If is_B:
38 · $\quad adc_B[s_B] \leftarrow (ad, c)$
39 $\quad E_B^{\dashv} \leftarrow E_B^{\dashv} + 1$
40 · $s_B \leftarrow s_B + 1$
41 · Return c

Fig. 8: Game FUNC for SRKE scheme R. Note the lines of code not tagged with a '·' do not influence the game outcome.

B's side multiple epochs may be active at the same time, reflecting that B has to be ready to process ciphertexts that were generated by A in the context of one of potentially many possible epochs. Intuitively, epochs end (on B's side) if a ciphertext is received (from A) that was sent in the context of a newer epoch.

The lines of code in the FUNC game (in Figure 8) that are not tagged with a '·' keep track of epochs. We represent the span of epochs supported by B with the interval variable E_B: its boundaries $E_B^{\vdash}$ and $E_B^{\dashv}$ reflect at any time the earliest and the latest such epoch. Further, we use variable e_A to track the latest epoch started by B that party A is aware of, and associative array EP_A to register for each of A's sending operations the context, i.e., the epoch number that A is (implicitly) refering to. In more detail, the invocation of init is accompanied by setting $E_B^{\vdash}, E_B^{\dashv}, e_A$ to zero (in lines 02,03), each sending operation of B introduces one more supported epoch (line 39), each receiving operation of A increases the latter's awareness of epochs supported by B (line 21), the context of each sending operation of A is recorded in EP_A (line 14), and each receiving operation of B potentially reduces the number of supported epochs by dropping absolete ones (line 29). Observe that tracking epochs is not meaningful after participants get out-of-sync; we thus guard lines 21,29 with corresponding tests.

Security of SRKE. Our SRKE security model lifts the one for URKE to the bidirectional (more precisely: sesquidirectional) setting. The goal of the adversary is again to distinguish established keys from random. For a SRKE scheme R, the corresponding key indistinguishability games $\text{KIND}_{\mathsf{R}}^{b}$, for challenge bit $b \in \{0, 1\}$, are specified in Figure 9. With any adversary $\mathcal{A}$ we associate

Game $\text{KIND}^b_{\mathsf{R}}(\mathcal{A})$
00 $s_A \leftarrow 0$; $r_B \leftarrow 0$
01 $s_B \leftarrow 0$; $r_A \leftarrow 0$
02 $e_A \leftarrow 0$; $\text{EP}_A[\cdot] \leftarrow \bot$
03 $E^{\vdash}_B \leftarrow 0$; $E^{\dashv}_B \leftarrow 0$
04 $adc_A[\cdot] \leftarrow \bot$; $is_B \leftarrow \mathtt{T}$
05 $adc_B[\cdot] \leftarrow \bot$; $is_A \leftarrow \mathtt{T}$
06 $key_A[\cdot] \leftarrow \bot$; $key_B[\cdot] \leftarrow \bot$
07 $\text{XP}_A \leftarrow \emptyset$; $\text{XP}_B \leftarrow \emptyset$
08 $\text{TR}_A \leftarrow \emptyset$; $\text{TR}_B \leftarrow \emptyset$
09 $\text{CH}_A \leftarrow \emptyset$; $\text{CH}_B \leftarrow \emptyset$
10 $(S_A, S_B) \leftarrow_{\$} \text{init}$
11 $b' \leftarrow_{\$} \mathcal{A}$
12 Require $\text{TR}_A \cap \text{CH}_A = \emptyset$
13 Require $\text{TR}_B \cap \text{CH}_B = \emptyset$
14 Stop with b'

Oracle $\text{SndA}(ad)$
15 Require $S_A \neq \bot$
16 $(S_A, k, c) \leftarrow_{\$} \text{snd}_A(S_A, ad)$
17 If is_A:
18 $\quad adc_A[s_A] \leftarrow (ad, c)$
19 $\quad \text{EP}_A[s_A] \leftarrow e_A$
20 · $key_A[e_A, s_A] \leftarrow k$
21 $s_A \leftarrow s_A + 1$
22 Return c

Oracle $\text{RcvA}(ad, c)$
23 Require $S_A \neq \bot$
24 If $is_A \wedge adc_B[r_A] \neq (ad, c)$:
25 $\quad is_A \leftarrow \mathtt{F}$
26 $\quad$ If $r_A \in \text{XP}_B$:
27 · $\qquad \text{TR}_A \xleftarrow{\cup} \mathbb{N} \times [s_A, \ldots]$
28 If is_A: $e_A \leftarrow e_A + 1$
29 $S_A \leftarrow \text{rcv}_A(S_A, ad, c)$
30 If $S_A = \bot$: Return $\bot$
31 $r_A \leftarrow r_A + 1$
32 Return

Oracle $\text{RcvB}(ad, c)$
33 Require $S_B \neq \bot$
34 If $is_B \wedge adc_A[r_B] \neq (ad, c)$:
35 $\quad is_B \leftarrow \mathtt{F}$
36 $\quad$ If $r_B \in \text{XP}_A$:
37 · $\qquad \text{TR}_B \xleftarrow{\cup} \mathbb{N} \times [r_B, \ldots]$
38 If is_B: $E^{\vdash}_B \leftarrow \text{EP}_A[r_B]$
39 $(S_B, k) \leftarrow \text{rcv}_B(S_B, ad, c)$
40 If $S_B = \bot$: Return $\bot$
41 If is_B: $k \leftarrow \diamond$
42 · $key_B[E^{\vdash}_B, r_B] \leftarrow k$
43 $r_B \leftarrow r_B + 1$
44 Return

Oracle $\text{SndB}(ad)$
45 Require $S_B \neq \bot$
46 $(S_B, c) \leftarrow_{\$} \text{snd}_B(S_B, ad)$
47 If is_B:
48 $\quad adc_B[s_B] \leftarrow (ad, c)$
49 $\quad E^{\dashv}_B \leftarrow E^{\dashv}_B + 1$
50 $s_B \leftarrow s_B + 1$
51 Return c

Oracle ExposeA
52 If is_A: $\text{XP}_A \xleftarrow{\cup} \{s_A\}$
53 Return S_A

Oracle ExposeB
54 · $\text{TR}_B \xleftarrow{\cup} [E^{\vdash}_B \,..\, E^{\dashv}_B] \times [r_B, \ldots]$
55 If is_B:
56 $\quad \text{XP}_B \xleftarrow{\cup} \{s_B\}$
57 · $\quad \text{TR}_A \xleftarrow{\cup} [E^{\vdash}_B \,..\, E^{\dashv}_B] \times [r_B, \ldots]$
58 Return S_B

Oracle $\text{Reveal}(u, i)$
as in URKE (Fig. 5)

Oracle $\text{Challenge}(u, i)$
as in URKE (Fig. 5)

Fig. 9: Games KIND^b, $b \in \{0, 1\}$, for SRKE scheme R.

its key distinguishing advantage $\text{Adv}^{\text{kind}}_{\mathsf{R}}(\mathcal{A}) := |\Pr[\text{KIND}^1_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] - \Pr[\text{KIND}^0_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1]|$. Intuitively, R offers key indistinguishability if all practical adversaries have a negligible key distinguishing advantage.

The new KIND games are the natural amalgamation of the (URKE) KIND games of Figure 5 with the (SRKE) FUNC game of Figure 8 (with the exceptions discussed below). Concerning the trivial attack conditions on URKE that we identified in Section 3, we note that conditions (a) and (b) remain valid for SRKE without modification, conditions (c) and (d) (that consider attacks on participants by tracing their computations) need a slight adaptation to reflect that updating epochs repairs the damage of state exposures, and condition (e) (that considers impersonation of exposed A to B), besides needing a slight adaptation, requires to be complemented by a new condition that considers that exposing B allows for impersonating him to A.

When comparing the KIND games from Figures 5 and 9, note that a crucial difference is that the key_A, key_B arrays in the URKE model are indexed with simple counters, while in the SRKE model they are indexed with pairs where the one element is the same counter as in the URKE case and the other element indicates the epoch for which the corresponding key

was established[13]. The lines in Figure 9 that are tagged with '$\cdot$' are the ones affected by this modification. The new indexing mechanism allows, when B is exposed, for marking as traceable only those future keys of A and B that belong to the epochs managed by B at the time of exposure (lines 54,57). This already implements the necessary adaptation of conditions (c) and (d) to the SRKE setting. The announced adaptation of condition (e) is executing line 52 only if $is_A = \mathtt{T}$; the change is due as the motivation given on p. 13 is valid only if A is in-sync (which is always the case in URKE, but not in SRKE). Finally, complementing condition (e), we identify the following new trivial attack condition:

(f) Exposing party B allows for impersonating it: Assume parties A and B are in-sync. The adversary obtains a copy of B's state and invokes the snd_B algorithm with it, obtaining a ciphertext which it provides to party A (rendering the latter out-of-sync). If then A generates a new key using the snd_A algorithm, the adversary can feed the resulting ciphertext into the rcv_B algorithm, recovering the key. Example: fix some ad, ad'; $S_B^* \leftarrow \text{ExposeB}()$; $(S_B^*, c) \leftarrow_\$ \text{snd}_B(S_B^*, ad)$; $\text{RcvA}(ad, c)$; $c' \leftarrow \text{SndA}(ad')$; $(S_B^*, k) \leftarrow \text{rcv}_B(S_B^*, ad', c')$; $k' \leftarrow \text{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; output b'. Lines 26,27 (in conjunction with lines 07,56) detect the described type of impersonation and mark all future keys of A as traceable.

This completes the description of our SRKE security model. As in URKE, it excludes the minimal set of attacks, indicating that it gives strong security guarantees.

6 Constructing SRKE

We present a SRKE construction that generalizes our URKE scheme to the sesquidirectional setting. The core intuition is as follows: Like in the URKE scheme, A-to-B ciphertexts correspond with KEM ciphertexts where the corresponding public and secret keys are held by A and B, respectively, and the two keys are evolved to new keys after each use. In addition to this, with the goal of letting B heal from state exposures, our SRKE construction gives him the option to sanitize his state by generating a fresh KEM key pair and communicating the corresponding public key to A (using the B-to-A link specific to SRKE). The algorithms of our protocol are specified in Figure 10. Although the sketched approach might sound simple and natural, the algorithms, quite surprisingly, are involved and require strong cryptographic building blocks (a key-updatable KEM and a one-time signature scheme, see Section 2). Their complexity is a result of SRKE protocols having to simultaneously offer solutions to multiple inherent challenges. We discuss these in the following.

Epoch management. Recall that we assume a concurrent setting for SRKE and that, thus, if B refreshes his state via the snd_B algorithm, then he still has to keep copies of the secret keys maintained for prior epochs (so that the rcv_B algorithm can properly process incoming ciphertexts created for them). Our protocol algorithms implement this by including in B's state the array $SK[\cdot]$ in which snd_B stores all keys it generates (line 58; obsolete keys of expired epochs are deleted by rcv_B in line 49). Beyond that, both A and B maintain an interval variable E in their state: its boundaries $E^{\vdash}$ and $E^{\dashv}$ are used by B to reflect the earliest and latest supported epoch, and by A to keep track of epoch updates that occur in direct succession (i.e., that are still waiting for their 'activation' by snd_A). Note finally that the snd_A algorithm communicates to rcv_B in every outgoing ciphertext the epoch in which A is operating (line 12).

Secure state update. Assume A executes one time the snd_A algorithm, then twice the rcv_A algorithm, and then again the snd_A algorithm. That is, following the above sketch of

[13] The adversary always knows the epoch numbers associated with keys, so it can pose meaningful Reveal and Challenge queries just as before.

Proc init
00 $(sgk, vfk) \leftarrow_{\$} \text{gen}_{\mathsf{S}}$
01 · $(sk, pk) \leftarrow_{\$} \text{gen}_{\mathsf{K}}$
02 · $K \leftarrow_{\$} \mathcal{K}$; $k.m \leftarrow_{\$} \mathcal{K}$; $t \leftarrow \epsilon$
03 $E^{\vdash} \leftarrow 0$; $E^{\dashv} \leftarrow 0$
04 $s \leftarrow 0$; $r \leftarrow 0$
05 $PK[\cdot] \leftarrow \bot$; $PK[0] \leftarrow pk$
06 $SK[\cdot] \leftarrow \bot$; $SK[0] \leftarrow sk$
07 $L_A[\cdot] \leftarrow \bot$; $L_B[\cdot] \leftarrow \bot$; $L_A[0] \leftarrow \diamond$
08 $S_A \leftarrow (PK, E, s, L_A, vfk, K, k.m, t)$
09 $S_B \leftarrow (SK, E, r, L_B, sgk, K, k.m, t)$
10 Return (S_A, S_B)

Proc $\text{snd}_A(S_A, ad)$
11 $(PK, E, s, L, vfk, K, k.m, t) \leftarrow S_A$
12 $k^* \leftarrow \epsilon$; $C \leftarrow E^{\dashv}$
13 For $e' \leftarrow E^{\vdash}$ to $E^{\dashv}$:
14 · $(k, c) \leftarrow_{\$} \text{enc}(PK[e'])$
15 $k^* \xleftarrow{\shortparallel} k$; $C \xleftarrow{\shortparallel} c$
16 · $\tau \leftarrow_{\$} \text{tag}(k.m, ad \parallel C)$
17 · $C \xleftarrow{\shortparallel} \tau$; $t \xleftarrow{\shortparallel} \triangleright \parallel ad \parallel C$
18 · $k.o \parallel K \parallel k.m \parallel sk \leftarrow \text{H}(K, k^*, t)$
19 · $pk \leftarrow \text{gen}_{\mathsf{K}}(sk)$
20 $PK[..., (E^{\dashv} - 1)] \leftarrow \bot$; $PK[E^{\dashv}] \leftarrow pk$
21 $E^{\vdash} \leftarrow E^{\dashv}$; $s \leftarrow s + 1$; $L[s] \leftarrow ad \parallel C$
22 $S_A \leftarrow (PK, E, s, L, vfk, K, k.m, t)$
23 Return $(S, k.o, C)$

Proc $\text{rcv}_A(S_A, ad, C)$
24 $(PK, E, s, L, vfk, K, k.m, t) \leftarrow S_A$
25 $t \xleftarrow{\shortparallel} \triangleleft \parallel ad \parallel C$; $C \parallel \sigma \leftarrow C$
26 Require $\text{vfy}_{\mathsf{S}}(vfk, ad \parallel C, \sigma)$
27 $r \parallel pk^* \parallel vfk \leftarrow C$
28 Require $L[r] \neq \bot$
29 $L[..., (r - 1)] \leftarrow \bot$; $L[r] \leftarrow \diamond$
30 For $s' \leftarrow r + 1$ to s:
31 $pk^* \leftarrow \text{up}(pk^*, L[s'])$
32 $E^{\dashv} \leftarrow E^{\dashv} + 1$; $PK[E^{\dashv}] \leftarrow pk^*$
33 $S_A \leftarrow (PK, E, s, L, vfk, K, k.m, t)$
34 Return S_A

Proc $\text{rcv}_B(S_B, ad, C)$
35 $(SK, E, r, L, sgk, K, k.m, t) \leftarrow S_B$
36 $t^* \leftarrow ad \parallel C$; $C \parallel \tau \leftarrow C$
37 · Require $\text{vfy}_{\mathsf{M}}(k.m, ad \parallel C, \tau)$
38 $k^* \leftarrow \epsilon$; $e \parallel C \leftarrow C$
39 Require $E^{\vdash} \leq e \leq E^{\dashv}$
40 $t \xleftarrow{\shortparallel} L[E^{\vdash} + 1] \parallel \ldots \parallel L[e]$
41 $L[..., e] \leftarrow \bot$
42 For $e' \leftarrow E^{\vdash}$ to e:
43 $c \parallel C \leftarrow C$
44 · $k \leftarrow \text{dec}(SK[e'], c)$
45 · Require $k \neq \bot$
46 $k^* \xleftarrow{\shortparallel} k$
47 $t \xleftarrow{\shortparallel} \triangleright \parallel t^*$
48 · $k.o \parallel K \parallel k.m \parallel sk \leftarrow \text{H}(K, k^*, t)$
49 $SK[..., (e - 1)] \leftarrow \bot$; $SK[e] \leftarrow sk$
50 For $e' \leftarrow e + 1$ to $E^{\dashv}$:
51 $SK[e'] \leftarrow \text{up}(SK[e'], t^*)$
52 $E^{\vdash} \leftarrow e$; $r \leftarrow r + 1$
53 $S_B \leftarrow (SK, E, r, L, sgk, K, k.m, t)$
54 Return $(S_B, k.o)$

Proc $\text{snd}_B(S_B, ad)$
55 $(SK, E, r, L, sgk, K, k.m, t) \leftarrow S_B$
56 $(sk^*, pk^*) \leftarrow_{\$} \text{gen}_{\mathsf{K}}$
57 $(sgk^*, vfk^*) \leftarrow_{\$} \text{gen}_{\mathsf{S}}$
58 $E^{\dashv} \leftarrow E^{\dashv} + 1$; $SK[E^{\dashv}] \leftarrow sk^*$
59 $C \leftarrow r \parallel pk^* \parallel vfk^*$
60 $\sigma \leftarrow_{\$} \text{sgn}(sgk, ad \parallel C)$
61 $C \xleftarrow{\shortparallel} \sigma$; $L[E^{\dashv}] \leftarrow \triangleleft \parallel ad \parallel C$
62 $S_B \leftarrow (SK, E, r, L, sgk^*, K, k.m, t)$
63 Return (S_B, C)

Fig. 10: Construction of a SRKE scheme from a key-updatable KEM $\mathsf{K} = (\text{gen}_{\mathsf{K}}, \text{enc}, \text{dec})$, a message authentication code $\mathsf{M} = (\text{tag}, \text{vfy}_{\mathsf{M}})$, a one-time signature scheme $\mathsf{S} = (\text{gen}_{\mathsf{S}}, \text{sgn}, \text{vfy}_{\mathsf{S}})$, and a random oracle H. For simplicity we denote the key space of the MAC and the space of chaining keys with the same symbol $\mathcal{K}$.
Notation: Lines 07,29: If an entry of an array is expected to contain a ciphertext, but clearly the value of the ciphertext will not any more matter, we instead store the placeholder symbol $\diamond$. Line 40: If $E^{\vdash} = e$ then no value shall be concatenated to t. Line 43: The last iteration of the loop is meant to clear C; a more precise version of the line would say "If $e' < e$ then $c \parallel C \leftarrow C$ else $c \leftarrow C$". Lines 17,25,47,61: We use labels $\triangleright$ and $\triangleleft$ in transcript fragments to distinguish whether they emerged in the A-to-B or B-to-A direction.

our protocol, as part of her first snd_A invocation she will encapsulate to a public key that she subsequently updates (akin to how she would do in our URKE solution, see lines 07,13 of Figure 7), then she will receive two fresh public keys from B, and finally she will again encapsulate to a public key that she subsequently updates. The question is: Which public key shall she use in the last step? The one resulting from the update during her first snd_A invocation, the one obtained in her first rcv_A invocation, or the one obtained in her second rcv_A invocation? We found that only one configuration is safe against key distinguishing attacks: Our SRKE protocol is such that she encapsulates to all three, combining the established session

keys into one via concatenation.[14] The algorithms implement this by including in A's state the array $PK[\cdot]$ in which rcv_A stores incoming public keys (line 32) and which snd_A consults when establishing outgoing ciphertexts (lines 13–15; the counterpart on B's side is in lines 42–46). Once the switch to the new epoch is completed, the obsolete public keys are removed from A's state (line 20). If A executes snd_A many times in succession, then all but the first invocation will, akin to the URKE case, just encapsulate to the (one) evolved public key from the preceding invocation.

We discuss a second issue related to state updates. Assume B executes three times the snd_B algorithm and then once the rcv_B algorithm, the latter on input a well-formed but non-authentic ciphertext (e.g., the adversary could have created the ciphertext, after exposing A's state, using the snd_A algorithm). In the terms of our security model the latter action brings B out-of-sync, which means that if he is subsequently exposed then this should not affect the security of further session keys established by A. On the other hand, according to the description provided so far, exposing B's state means obtaining a copy of array $SK[\cdot]$, i.e., of the decapsulation keys of all epochs still supported by B. We found that this easily leads to key distinguishing attacks,[15] so in order to protect the elements of $SK[\cdot]$ they are evolved by the rcv_B algorithm whenever an incoming ciphertext is processed. We implement the latter via the dedicated update procedure up provided by the key-updatable KEM. The corresponding lines are 50–51 (note that t^* is the current transcript fragment, see line 36). Of course A has to synchronize on B's key updates, which she does in lines 30–31, where array $L[\cdot]$ is the state variable that keeps track of the corresponding past A-to-B transcript fragments. (Outgoing ciphertexts are stored in $L[\cdot]$ in line 21, and obsolete ones are removed from it in line 29.) Note that A, for staying synchronized with B, also needs to keep track of the ciphertexts that he received (from her) so far; for this reason, B indicates in every outgoing ciphertext the number r of incoming ciphertexts he has been exposed to (lines 27,59).

Transcript management. Recall that one element of the participants' state in our URKE scheme (in Figure 7) is the variable t that accumulates transcript information (associated data and ciphertexts) of prior communication so that it can be input to key derivation. This is a common technique to ensure that the keys established on the two sides diverge when an active attack occurs. Also our SRKE construction follows this approach, but accumulating transcripts is more involved if communication is concurrent: If both A and B would add outgoing ciphertexts to their transcript accumulator directly after creating them, then concurrent sending would immediately desynchronize the two parties. This issue is resolved in our construction as follows: In the B-to-A direction, while A appends incoming ciphertexts (from B) to her transcript variable in the moment she receives them (line 25), when creating the ciphertexts, B will just record them in his state variable $L[\cdot]$ (line 61), and postpone adding them to his transcript variable to the point when he is able to deduce (from A's ciphertexts) the position of when she did (line 40; obsolete entries are removed in line 41). The A-to-B direction is simpler[16] and handled as in our URKE protocol: A updates her transcript when sending a ciphertext (line 17), and B updates his transcript when receiving it (lines 36,47). Note we tag transcript fragments with labels $\triangleright$ or $\triangleleft$ to indicate whether they emerged in the A-to-B or B-to-A direction of communication (e.g., in lines 17,61).

Authentication. To reach security against active adversaries we protect the SRKE ciphertexts against manipulation. Recall that in our URKE scheme a MAC was sufficient for this. In SRKE,

[14] We discuss why it is unsafe to encapsulate to only a subset of the keys in Appendix B.3.
[15] We discuss this further in Appendix B.2.
[16] Intuitively the disbalance comes from the fact that keys are only established by A-to-B ciphertexts and that transcripts are only used for key derivation.

a MAC is still sufficient for the A-to-B direction (lines 16,37), but for the B-to-A direction, to defend against attacks where the adversary first exposes A's state and then uses the obtained MAC key to impersonate B to her,[17] we need to employ a one-time signature scheme: Each ciphertext created by B includes a freshly generated verification key that is used to authenticate the next B-to-A ciphertext (lines 57,59,60,26,27; note how this rules out the described attack).

The only lines we did not comment on are 18,19,48,56—those that also form the core of our URKE protocol and which are discussed extensively in Section 4.

Theorem 2. *The SRKE protocol* R *from Figure 10 offers key indistinguishability. More precisely, if function* H *is modeled as a random oracle, for every adversary* $\mathcal{A}$ *for games* $\mathrm{KIND}^b_{\mathsf{R}}$ *from Figure 9 there exists an adversary* $\mathcal{B}$ *for game* KUOW *from Figure 1, an adversary* $\mathcal{C}_{\mathsf{S}}$ *for game* SUF *from Figure 23, and an adversary* $\mathcal{C}_{\mathsf{M}}$ *for game* SUF *from Figure 21 such that* $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq 3\mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B}) + q_R\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{C}_{\mathsf{S}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{C}_{\mathsf{M}}) + \frac{q_S q_F}{|\mathcal{K}_{\mathsf{K}}|}$*, where* $\mathcal{K}_{\mathsf{K}}$ *is the session-key space of the kuKEM, the running time of* $\mathcal{B}$ *is about that of* $\mathcal{A}$ *plus* q_F *key checking and solve operations, the running time of* $\mathcal{C}_{\mathsf{S}}$ *and* $\mathcal{C}_{\mathsf{M}}$ *is about that of* $\mathcal{A}$*, and* q_F, q_S, q_R *are the numbers of* $\mathcal{A}$*'s random oracle,* SndA*, and* RcvA *queries, respectively.*

The proof of Theorem 2 is in Appendix C. The approach is the same as in our URKE proof but with small yet important differences: 1) the proof reduces signature forgeries to the SUF security of the signature scheme to show that communication from B to A is authentic, 2) the security of session keys established by A is reduced to the KUOW security of the kuKEM. The reduction to the KUOW game is split into three cases: a) session keys established by A in sync, b) the first session key established by A out of sync, and c) all remaining session keys established by A out of sync. This distinction is made as in each of these cases a different encapsulated key—as part of the random oracle input—is assumed to be unknown to the adversary. Finally the SRKE proof—as in the URKE proof—makes use of the MAC's SUF security to show that B will never establish challengeable keys out of sync.

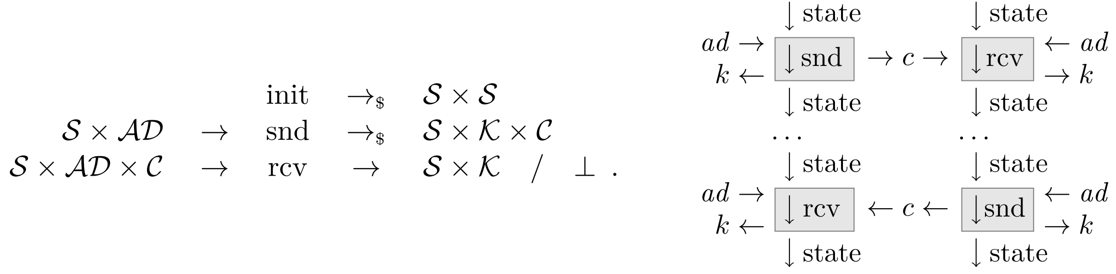
7 Bidirectionally ratcheted key exchange

The URKE and SRKE primitives are unbalanced in that they allow only one of the two participants to actively establish new keys. As the ratcheting notion first appeared in the context of (bidirectional) instant messaging [13,4,15] it is natural to ask for a fully balanced primitive where both participants have the capability of establishing fresh keys independently of each other. In this section we correspondingly study *bidirectional ratcheted key exchange* (BRKE) by first defining its syntax, functionality, and security, and then proposing two constructions. We note that the BRKE notions are natural extensions of those of URKE and SRKE, effectively duplicating specific parts of the security model and constructions so that they are available in both directions of communication. The main challenge is to properly interweave the communication in the two directions.

Formally, a BRKE scheme for a finite key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$ is a triple $\mathsf{R} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$ of algorithms together with a state space $\mathcal{S}$ and a ciphertext space $\mathcal{C}$. The randomized initialization algorithm init returns a pair of states $(S_A, S_B) \in \mathcal{S} \times \mathcal{S}$. The randomized sending algorithm snd takes a state $S \in \mathcal{S}$ and an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $S' \in \mathcal{S}$, a key $k \in \mathcal{K}$, and a ciphertext $c \in \mathcal{C}$. Finally, the deterministic receiving algorithm rcv takes a state $S \in \mathcal{S}$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and *either* outputs an updated state $S' \in \mathcal{S}$ and a key $k \in \mathcal{K}$ *or* outputs

[17] Note this is not an issue in the other direction: Exposing B and impersonating A to him leads to marking all future keys of B as traceable anyway, without any option to recover. We expand on this in Appendix B.1.

the special symbol $\perp$ to indicate rejection. A shortcut notation for these syntactical definitions is



Correctness of BRKE. We formalize the correctness of BRKE via the FUNC game in Figure 11. Concretely, we say scheme R is *correct* if $\Pr[\mathrm{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ for all adversaries $\mathcal{A}$. The game is best understood by comparing it with the functionality game of SRKE (in Figure 8): As in BRKE the roles of the participants are symmetric, the Snd and Rcv oracles in Figure 11 are effectively the amalgamation of the SndA and SndB oracles, respectively, the RcvA and RcvB oracles, from Figure 8. Observe that, as in BRKE the snd invocations of both participants create fresh keys and start new epochs, in the FUNC game each participant has its individual copy of the game variables $\mathit{key}, \mathrm{EP}, e, E^{\vdash}, E^{\dashv}$; this is in contrast with the SRKE case where variables $\mathit{key}, \mathrm{EP}, e$ were specific to one party, and variables $E^{\vdash}, E^{\dashv}$ were specific to the other.

```
Game FUNC_R(A)
00 For u ∈ {A, B}:
01    s_u ← 0; r_u ← 0
02    e_u ← 0; EP_u[·] ← ⊥
03    E_u^⊢ ← 0; E_u^⊣ ← 0
04    adc_u[·] ← ⊥; is_u ← T
05    key_u[·] ← ⊥
06 (S_A, S_B) ←$ init
07 Invoke A
08 Stop with 0

Oracle Snd(u, ad)
09 Require S_u ≠ ⊥
10 (S_u, k, c) ←$ snd(S_u, ad)
11 If is_u:
12    adc_u[s_u] ← (ad, c)
13    EP_u[s_u] ← e_u
14    E_u^⊣ ← E_u^⊣ + 1
15 key_u[s_u] ← k
16 s_u ← s_u + 1
17 Return c

Oracle Rcv(u, ad, c)
18 Require S_u ≠ ⊥
19 If is_u ∧ adc_ū[r_u] ≠ (ad, c):
20    is_u ← F
21 If is_u:
22    E_u^⊢ ← EP_ū[r_u]
23    e_u ← e_u + 1
24 (S_u, k) ← rcv(S_u, ad, c)
25 If S_u = ⊥: Return ⊥
26 Reward is_u ∧ k ≠ key_ū[r_u]
27 r_u ← r_u + 1
28 Return
```

Fig. 11: Game FUNC for BRKE scheme R. For a user $u \in \{A, B\}$ we write $\bar{u}$ for its peer; that is, we always have $\{u, \bar{u}\} = \{A, B\}$.

Security of BRKE. Our BRKE security model is derived by lifting the indistinguishability notion from SRKE from Figure 9 to the fully bidirectional case, again amalgamating SndA and SndB oracles and RcvA and RcvB oracles to single Snd and Rcv oracles, respectively, and using the notation of the BRKE functionality game from Figure 11. The result are the key indistinguishability games $\mathrm{KIND}^b_{\mathsf{R}}$, for challenge bit $b \in \{0, 1\}$, specified in Figure 12. The only noteable novelty, required as in BRKE keys can be established by both participants, is that the

game manages two copies of the *key* array per user: We represent keys that user u establishes with the role of a sender as $key_u[\mathtt{S},\ldots]$, and we represent keys that u recovers as a receiver as $key_u[\mathtt{R},\ldots]$. For a BRKE scheme R, with any adversary $\mathcal{A}$ we associate its key distinguishing advantage $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) := |\Pr[\mathrm{KIND}^1_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathrm{KIND}^0_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1]|$. Intuitively, R offers key indistinguishability if all practical adversaries have a negligible key distinguishing advantage.

Game $\mathrm{KIND}^b_{\mathsf{R}}(\mathcal{A})$
00 For $u \in \{A, B\}$:
01 $\quad s_u \leftarrow 0;\ r_u \leftarrow 0$
02 $\quad e_u \leftarrow 0;\ \mathrm{EP}_u[\cdot] \leftarrow \bot$
03 $\quad E_u^{\vdash} \leftarrow 0;\ E_u^{\dashv} \leftarrow 0$
04 $\quad adc_u[\cdot] \leftarrow \bot;\ is_u \leftarrow \mathtt{T}$
05 $\quad key_u[\cdot] \leftarrow \bot;\ \mathrm{XP}_u \leftarrow \emptyset$
06 $\quad \mathrm{TR}_u \leftarrow \emptyset;\ \mathrm{CH}_u \leftarrow \emptyset$
07 $(S_A, S_B) \leftarrow_\$ \mathrm{init}$
08 $b' \leftarrow_\$ \mathcal{A}$
09 For $u \in \{A, B\}$:
10 $\quad$ Require $\mathrm{TR}_u \cap \mathrm{CH}_u = \emptyset$
11 Stop with b'

Oracle $\mathrm{Snd}(u, ad)$
12 Require $S_u \neq \bot$
13 $(S_u, k, c) \leftarrow_\$ \mathrm{snd}(S_u, ad)$
14 If is_u:
15 $\quad adc_u[s_u] \leftarrow (ad, c)$
16 $\quad \mathrm{EP}_u[s_u] \leftarrow e_u$
17 $\quad E_u^{\dashv} \leftarrow E_u^{\dashv} + 1$
18 $key_u[\mathtt{S}, e_u, s_u] \leftarrow k$
19 $s_u \leftarrow s_u + 1$
20 Return c

Oracle $\mathrm{Reveal}(u, i)$
as in URKE/SRKE (Fig. 5)

Oracle $\mathrm{Rcv}(u, ad, c)$
21 Require $S_u \neq \bot$
22 If $is_u \wedge adc_{\bar{u}}[r_u] \neq (ad, c)$:
23 $\quad is_u \leftarrow \mathtt{F}$
24 $\quad$ If $r_u \in \mathrm{XP}_{\bar{u}}$:
25 $\quad\quad \mathrm{TR}_u \xleftarrow{\cup} \{\mathtt{S}\} \times \mathbb{N} \times [s_u, \ldots]$
26 $\quad\quad \mathrm{TR}_u \xleftarrow{\cup} \{\mathtt{R}\} \times \mathbb{N} \times [r_u, \ldots]$
27 If is_u:
28 $\quad E_u^{\vdash} \leftarrow \mathrm{EP}_{\bar{u}}[r_u]$
29 $\quad e_u \leftarrow e_u + 1$
30 $(S_u, k) \leftarrow \mathrm{rcv}(S_u, ad, c)$
31 If $S_u = \bot$: Return $\bot$
32 If is_u: $k \leftarrow \diamond$
33 $key_u[\mathtt{R}, E_u^{\vdash}, r_u] \leftarrow k$
34 $r_u \leftarrow r_u + 1$
35 Return

Oracle $\mathrm{Expose}(u)$
36 $\mathrm{TR}_u \xleftarrow{\cup} \{\mathtt{R}\} \times [E_u^{\vdash} .. E_u^{\dashv}] \times [r_u, \ldots]$
37 If is_u:
38 $\quad \mathrm{XP}_u \xleftarrow{\cup} \{s_u\}$
39 $\quad \mathrm{TR}_{\bar{u}} \xleftarrow{\cup} \{\mathtt{S}\} \times [E_u^{\vdash} .. E_u^{\dashv}] \times [r_u, \ldots]$
40 Return S_u

Oracle $\mathrm{Challenge}(u, i)$
as in URKE/SRKE (Fig. 5)

Fig. 12: Games KIND^b, $b \in \{0, 1\}$, for BRKE scheme R. Symbols $\mathtt{S}$ and $\mathtt{R}$ are labels that distinguish whether keys were established in a sending or a receiving operation.

8 Constructing BRKE

We propose two constructions of the BRKE primitive. Their common denominator is that they internally use two instances of a SRKE protocol—one in the Alice-to-Bob direction and one in the Bob-to-Alice direction. The challenge is to properly interveawe their operations such that, overall, BRKE security is reached. (For instance, attacking one of the instances needs to automatically escalate to an attack on the second as otherwise attacks on KIND security become feasible). Our first solution achieves this via strongly unforgeable one-time signatures. Our second solution is slightly more efficient but ad-hoc; it is derived from the specific SRKE protocol from Figure 10 and carefully interleaves the use of its inner building blocks.

Generic construction with one-time signatures. Let SR denote a SRKE protocol, and assume a strongly unforgeable one-time signature scheme as an auxiliary building block. The snd and rcv algorithms of our first BRKE construction are in Figure 13. (The init algorithm is not in the figure; it just performs two invocations of $\mathrm{init}_{\mathsf{SR}}$, and the initial states of users consist of one sending and one receiving state.) Concretely, our snd algorithm performs internally two snd invocations of the underlying SRKE scheme (in lines 02,03), which results in a key $k.o$ and

two ciphertexts c_1, c_2. These ciphertexts are protected by a one-time signature before being sent to the peer: a fresh signature key pair is generated per snd invocation (in line 01), and the pair c_1, c_2 signed with it (in line 04). Note that the signature verification key is included in the associated-data field of both internal snd invocations (see line 01). To allow for signature verification on the side of the peer, the verification key is sent along with the ciphertexts. The peer processes the ciphertext in the obvious way.

We describe the rationale behind our construction. The goal is to bind the two ciphertext components c_1, c_2 together such that any manipulation of the pair will be detected by both underlying SRKE instances. One could try to implement this directly via the associated-data fields on the SRKE, that is, by including c_1 in ad when producing c_2 or by including c_2 in ad when producing c_1. It turns out that both these options are too weak and allow for attacks on key indistinguishability of the composed BRKE scheme. By using the one-time signature scheme we side-step this one-before-the-other dependency. The security argument for our construction in Figure 13 is as follows: Note first that each verification key recovered in line 09 is either authentic or not. If it is, then also c_1, c_2, σ are authentic (otherwise the adversary would have broken the strong unforgeability of the one-time signature scheme). If it is not, then this will be reflected in the changed associated-data field ad line 09, i.e., both SRKE instances will be notified of this.

Proc $\mathrm{snd}(S, ad)$	**Proc** $\mathrm{rcv}(S, ad, c)$
00 $(S_1, S_2) \leftarrow S$	08 $(S_1, S_2) \leftarrow S$
01 $(sgk, vfk) \leftarrow_\$ \mathrm{gen}_\mathsf{S}$; $ad \xleftarrow{\shortparallel} vfk$	09 $vfk \parallel c_1 \parallel c_2 \parallel \sigma \leftarrow c$; $ad \xleftarrow{\shortparallel} vfk$
02 $(S_1, k.o, c_1) \leftarrow_\$ \mathrm{snd}_A(S_1, ad)$	10 Require $\mathrm{vfy}_\mathsf{S}(vfk, c_1 \parallel c_2, \sigma)$
03 $(S_2, c_2) \leftarrow_\$ \mathrm{snd}_B(S_2, ad)$	11 $S_1 \leftarrow \mathrm{rcv}_A(S_1, ad, c_2)$
04 $\sigma \leftarrow_\$ \mathrm{sgn}(sgk, c_1 \parallel c_2)$	12 Require $S_1 \neq \perp$
05 $c \leftarrow vfk \parallel c_1 \parallel c_2 \parallel \sigma$	13 $(S_2, k.o) \leftarrow \mathrm{rcv}_B(S_2, ad, c_1)$
06 $S \leftarrow (S_1, S_2)$	14 Require $S_2 \neq \perp$
07 Return $(S, k.o, c)$	15 $S \leftarrow (S_1, S_2)$
	16 Return $(S, k.o)$

Fig. 13: Generic construction of BRKE scheme BR from a SRKE scheme $\mathsf{SR} = (\mathrm{init}_\mathsf{SR}, \mathrm{snd}_A, \mathrm{snd}_B, \mathrm{rcv}_A, \mathrm{rcv}_B)$ and a one-time signature scheme $\mathsf{S} = (\mathrm{gen}_\mathsf{S}, \mathrm{sgn}, \mathrm{vfy}_\mathsf{S})$.

Theorem 3. *The BRKE protocol* BR *from Figure 13 offers key indistinguishability. More precisely, for every adversary* $\mathcal{A}$ *for games* $\mathrm{KIND}^b_\mathsf{BR}$ *from Figure 12 (BRKE) there exists an adversary* $\mathcal{B}$ *for game* $\mathrm{KIND}_\mathsf{SR}$ *from Figure 9 (SRKE) and an adversary* $\mathcal{C}$ *for game* SUF *from Figure 23 such that* $\mathrm{Adv}^{\mathrm{kind}}_\mathsf{BR}(\mathcal{A}) \leq 2\mathrm{Adv}^{\mathrm{kind}}_\mathsf{SR}(\mathcal{B}) + q_R \mathrm{Adv}^{\mathrm{suf}}_\mathsf{S}(\mathcal{C})$*, where the running times of* $\mathcal{B}$ *and* $\mathcal{C}$ *are about that of* $\mathcal{A}$*, and* q_R *is the number of* $\mathcal{A}$*'s* RcvA *queries.*

We prove Theorem 3 in Appendix D.

Ad-hoc construction. Our ad-hoc construction in Figure 14 directly adopts the SRKE construction and combines both sending algorithms and both receiving algorithms. To derive the necessary binding that was described in the previous paragraph, the sending algorithms intertwine by signing both ciphertext parts together and then feeding the whole ciphertext—including the signature—into the random oracle. As such, a manipulation of parts of the ciphertext directly affects both SRKE states.

We split the blocks taken from a different algorithm of the SRKE construction respectively by leaving blank lines in Figure 14. All lines not marked with a '·' at the left margin are directly copied from the SRKE construction. In line 20 instead of setting the ciphertext to the newest

Proc init

00 For $u \in \{A, B\}$:

01 $\quad (sgk_u, vfk_u) \leftarrow_\$ \text{gen}_\mathsf{S}$

02 $\quad (sk_u, pk_u) \leftarrow_\$ \text{gen}_\mathsf{K}$

03 $\quad K_u \leftarrow_\$ \mathcal{K};\ t \leftarrow \epsilon$

04 $\quad E^{\vdash} \leftarrow 0;\ E^{\dashv} \leftarrow 0$

05 $\quad s \leftarrow 0;\ r \leftarrow 0$

06 $\quad PK_u[\cdot] \leftarrow \perp;\ PK_u[0] \leftarrow pk$

07 $\quad SK_u[\cdot] \leftarrow \perp;\ SK_u[0] \leftarrow sk$

08 $\quad L_S[\cdot] \leftarrow \perp;\ L_R[\cdot] \leftarrow \perp;\ L_S[0] \leftarrow \diamond$

09 $\quad S_u \leftarrow (PK_{\bar{u}}, E, s, L_S, vfk_{\bar{u}}, K_{\bar{u}}, t)$

10 $\quad R_u \leftarrow (SK_u, E, r, L_R, sgk_u, K_u, t)$

11 $\quad ST_u \leftarrow (R_u, S_u)$

12 Return (ST_A, ST_B)

Proc $\text{snd}(ST, ad)$

13 $(R, S) \leftarrow ST$

14 $(SK, E_R, r, L_R, sgk, K_R, t_R) \leftarrow R$

15 $(sk^*, pk^*) \leftarrow_\$ \text{gen}_\mathsf{K}$

16 $(sgk^*, vfk^*) \leftarrow_\$ \text{gen}_\mathsf{S}$

17 $E_R^{\dashv} \leftarrow E_R^{\dashv} + 1;\ SK[E_R^{\dashv}] \leftarrow sk^*$

18 $C \leftarrow r \,\|\, pk^* \,\|\, vfk^*$

19 $(PK, E_S, s, L_S, vfk, K_S, t_S) \leftarrow S$

20 · $k^* \leftarrow \epsilon;\ C \stackrel{\shortparallel}{\leftarrow} E_S^{\dashv}$

21 For $e' \leftarrow E_S^{\vdash}$ to $E_S^{\dashv}$:

22 $\quad (k, c) \leftarrow_\$ \text{enc}(PK[e'])$

23 $\quad k^* \stackrel{\shortparallel}{\leftarrow} k;\ C \stackrel{\shortparallel}{\leftarrow} c$

24 $\sigma \leftarrow_\$ \text{sgn}(sgk, ad \,\|\, C)$

25 · $C \stackrel{\shortparallel}{\leftarrow} \sigma;\ L_R[E_R^{\dashv}] \leftarrow \triangleright_u \,\|\, ad \,\|\, C$

26 $R \leftarrow (SK, E_R, r, L_R, sgk^*, K_R, t_R)$

27 $t_S \stackrel{\shortparallel}{\leftarrow} \triangleright_u \,\|\, ad \,\|\, C$

28 $k.o \,\|\, K_S \,\|\, k.m \,\|\, sk \leftarrow \text{H}(K_S, k^*, t_S)$

29 $pk \leftarrow \text{gen}_\mathsf{K}(sk)$

30 $PK[..., (E_S^{\dashv} - 1)] \leftarrow \perp;\ PK[E_S^{\dashv}] \leftarrow pk$

31 $E_S^{\vdash} \leftarrow E_S^{\dashv};\ s \leftarrow s + 1;\ L_S[s] \leftarrow ad \,\|\, C$

32 $S \leftarrow (PK, E_S, s, L_S, vfk, K_S, t_S)$

33 $ST \leftarrow (R, S)$

34 Return $(ST, k.o, C)$

Proc $\text{rcv}(ST, ad, C)$

35 $(R, S) \leftarrow ST$

36 $(PK, E_S, s, L_S, vfk, K_S, t_S) \leftarrow S$

37 · $t^* \leftarrow ad \,\|\, C;\ t_S \stackrel{\shortparallel}{\leftarrow} \triangleright_{\bar{u}} \,\|\, t^*;\ C \,\|\, \sigma \leftarrow C$

38 Require $\text{vfy}_\mathsf{S}(vfk, ad \,\|\, C, \sigma)$

39 · $r \,\|\, pk^* \,\|\, vfk \,\|\, C \leftarrow C$

40 Require $L_S[r] \neq \perp$

41 $L_S[..., (r-1)] \leftarrow \perp;\ L_S[r] \leftarrow \diamond$

42 For $s' \leftarrow r + 1$ to s:

43 $\quad pk^* \leftarrow \text{up}(pk^*, L_S[s'])$

44 $E_S^{\dashv} \leftarrow E_S^{\dashv} + 1;\ PK[E_S^{\dashv}] \leftarrow pk^*$

45 $S \leftarrow (PK, E_S, s, L_S, vfk, K_S, t_S)$

46 $(SK, E_R, r, L_R, sgk, K_R, t_R) \leftarrow R$

47 $k^* \leftarrow \epsilon;\ e \,\|\, C \leftarrow C$

48 Require $E_R^{\vdash} \leq e \leq E_R^{\dashv}$

49 $t_R \stackrel{\shortparallel}{\leftarrow} L_R[E_R^{\vdash} + 1] \,\|\, \ldots \,\|\, L_R[e]$

50 $L_R[..., e] \leftarrow \perp$

51 For $e' \leftarrow E_R^{\vdash}$ to e:

52 $\quad c \,\|\, C \leftarrow C$

53 $\quad k \leftarrow \text{dec}(SK[e'], c)$

54 $\quad$ Require $k \neq \perp$

55 $\quad k^* \stackrel{\shortparallel}{\leftarrow} k$

56 · $t_R \stackrel{\shortparallel}{\leftarrow} \triangleright_{\bar{u}} \,\|\, t^*$

57 $k.o \,\|\, K_R \,\|\, k.m \,\|\, sk \leftarrow \text{H}(K_R, k^*, t_R)$

58 $SK[..., (e-1)] \leftarrow \perp;\ SK[e] \leftarrow sk$

59 For $e' \leftarrow e + 1$ to $E_R^{\dashv}$:

60 $\quad SK[e'] \leftarrow \text{up}(SK[e'], t^*)$

61 $E_R^{\vdash} \leftarrow e;\ r \leftarrow r + 1$

62 $R \leftarrow (SK, E_R, r, L_R, sgk, K_R, t_R)$

63 $ST \leftarrow (R, S)$

64 Return $(ST, k.o)$

Fig. 14: Construction of BRKE from our SRKE construction in Figure 10 by intertwining the respective algorithms.

epoch number, the ciphertext is appended by this number. As a result, both ciphertexts of the sending algorithms of SRKE are concatenated. In line 25 we index the encoding by the user identifier of the sending party. While in SRKE, each algorithm can only be used by one of the two communicating parties, in BRKE a unique encoding for each party becomes necessary to separate the parts of the transcript with respect to their origin. Similarly the encoding in the receive algorithm is equipped with user indexed encoding (see lines 37,56). In order to input the whole just received ciphertext and associated data string to the random oracle, in line 37 a copy of it is stored in t^* (in line 56 this string is appended to the current transcript). Finally in line 39 the ciphertexts of both SRKE instantiations are split again to process them at receipt.

Please note that the whole ciphertext is thereby signed at sending and fed into the random oracle. As such, the ciphertexts of SRKE are authenticated in this ad-hoc BRKE construction without employing an additional one-time signature.

References

1. Bellare, M., Kohno, T., Namprempre, C.: Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. ACM Trans. Inf. Syst. Secur. 7(2), 206–241 (2004), http://doi.acm.org/10.1145/996943.996945
2. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 619–650. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2017)
3. Bellare, M., Yee, B.S.: Forward-security in private-key cryptography. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 1–18. Springer, Heidelberg, Germany, San Francisco, CA, USA (Apr 13–17, 2003)
4. Borisov, N., Goldberg, I., Brewer, E.A.: Off-the-record communication, or, why not to use PGP. In: Atluri, V., Syverson, P.F., di Vimercati, S.D.C. (eds.) Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004. pp. 77–84. ACM (2004), http://doi.acm.org/10.1145/1029179.1029200
5. Canetti, R., Halevi, S., Katz, J.: Chosen-ciphertext security from identity-based encryption. In: Cachin, C., Camenisch, J. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 207–222. Springer, Heidelberg, Germany, Interlaken, Switzerland (May 2–6, 2004)
6. Cohn-Gordon, K., Cremers, C.J.F., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017. pp. 451–466. IEEE (2017), https://doi.org/10.1109/EuroSP.2017.27
7. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 164–178. IEEE Computer Society (2016), https://doi.org/10.1109/CSF.2016.19
8. Cremers, C.J.F., Feltz, M.: Beyond eCK: Perfect forward secrecy under actor compromise and ephemeral-key reveal. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 734–751. Springer, Heidelberg, Germany, Pisa, Italy (Sep 10–12, 2012)
9. Fischlin, M., Günther, F.: Multi-stage key exchange and the case of Google's QUIC protocol. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 14. pp. 1193–1204. ACM Press, Scottsdale, AZ, USA (Nov 3–7, 2014)
10. Gentry, C., Silverberg, A.: Hierarchical ID-based cryptography. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 548–566. Springer, Heidelberg, Germany, Queenstown, New Zealand (Dec 1–5, 2002)
11. Itkis, G.: Forward security, adaptive cryptography: Time evolution (2004), www.cs.bu.edu/~itkis/pap/forward-secure-survey.pdf
12. LaMacchia, B.A., Lauter, K., Mityagin, A.: Stronger security of authenticated key exchange. In: Susilo, W., Liu, J.K., Mu, Y. (eds.) ProvSec 2007. LNCS, vol. 4784, pp. 1–16. Springer, Heidelberg, Germany, Wollongong, Australia (Nov 1–2, 2007)
13. Langley, A.: Source code of Pond (05 2016), https://github.com/agl/pond
14. MacKenzie, P.D., Reiter, M.K., Yang, K.: Alternatives to non-malleability: Definitions, constructions, and applications (extended abstract). In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 171–190. Springer, Heidelberg, Germany, Cambridge, MA, USA (Feb 19–21, 2004)
15. Marlinspike, M., Perrin, T.: The double ratchet algorithm (11 2016), https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf
16. Marson, G.A., Poettering, B.: Security notions for bidirectional channels. IACR Trans. Symm. Cryptol. 2017(1), 405–426 (2017)
17. Moscaritolo, V., Belvin, G., Zimmermann, P.: Silent Circle Instant Messaging Protocol protocol specification (2012), https://silentcircle.com/sites/default/themes/silentcircle/assets/downloads/SCIMP_paper.pdf
18. Off-the-Record Messaging. http://otr.cypherpunks.ca (2016)
19. Raimondo, M.D., Gennaro, R., Krawczyk, H.: Secure off-the-record messaging. In: Atluri, V., di Vimercati, S.D.C., Dingledine, R. (eds.) Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, WPES 2005, Alexandria, VA, USA, November 7, 2005. pp. 81–89. ACM (2005), http://doi.acm.org/10.1145/1102199.1102216
20. Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) ACM CCS 02. pp. 98–107. ACM Press, Washington D.C., USA (Nov 18–22, 2002)
21. Schneier, B., Kelsey, J.: Cryptographic support for secure logs on untrusted machines. In: USENIX Security Symposium. USENIX Association (1998)

22. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Trans. Inf. Syst. Secur. 2(2), 159–176 (1999), http://doi.acm.org/10.1145/317087.317089
23. Unger, N., Dechand, S., Bonneau, J., Fahl, S., Perl, H., Goldberg, I., Smith, M.: SoK: Secure messaging. In: 2015 IEEE Symposium on Security and Privacy. pp. 232–249. IEEE Computer Society Press, San Jose, CA, USA (May 17–21, 2015)
24. Zimmermann, P., Johnston, A., Callas, J.: ZRTP: Media path key agreement for unicast secure RTP. RFC 6189, RFC Editor (April 2011), http://www.rfc-editor.org/rfc/rfc6189.txt

A Proof of URKE

Overview In order to prove the construction's security, we proceed in a sequence of games that pursues two targets: on the one hand the simulation is to be conducted without the usage of secret keys that belong to A's public keys, and on the other hand the adversary's random oracle requests and its forged MAC tags are used to solve the underlying hardness assumptions. While games G_2, G_3 and G_4 answer the former purpose, G_3 and G_5 (and in SRKE G_1) entail abortions of the game for events that are assumed to occur with negligible probability.

To unify the proof description for URKE and SRKE, we harmonize the numbering of the games. Thereby the abortion of signature forgeries in SRKE is conducted in game G_1. Since the URKE construction makes no use of signatures, G_1 in URKE equals the original game and is thereby a placeholder.

The description of the game hops for the SRKE proof consequently mainly focuses on the differences and peculiarities.

Notation Figures 15 and 16 depict the proof, split into the KIND_{R} game containing the URKE construction and the random oracle. The modifications by the game hops are included into the figures and denoted as follows:

The symbol at the right margin of a line annotates for which games a manipulation due to the game hop is valid. A line marked with a symbol $G_{\geq 3}$ is valid for game G_3 and all subsequent games. If a line is not valid for the final game, this line is struck through. Thereby either only the game is denoted in which a line becomes invalid by $G_{<y}$ or an interval of games for which a line is valid is marked by G_{x-y} where G_x is the game in which a line is introduced and G_y is the first game in which the line is disregarded.

Procedures introduced by a game hop are denoted by the symbol described above only in the first line of the procedure to reduce redundancy (e.g., see the procedures in Figure 16).

Game 2 – Synchronous simulation of B From correctness, we can conclude that B will compute the same values as A after processing the ciphertext from A under the condition that the ciphertext was not manipulated in transmission (A further generates the public key from the secret key and removes the secret key afterwards). Therefore we do not simulate the receiving of B if synchronicity was not disrupted. In order to simulate the exposure of B correctly, we introduce two arrays $SK_\star, KT_\star$ that track A's internal outputs (secret key and symmetric keys with transcript) after sending.

As soon as B receives a manipulated ciphertext, his state is established from these arrays and then the simulation is further computed according to the construction. The index of the arrays $SK_\star, KT_\star$ ends with [R] for stored values of only B and for the last common values among A and B. We denote incrementing of index $i+1 = (s, \mathtt{U})+1 = (s+1, \mathtt{U})$ such that the counter s is incremented. To shorten and clarify the description, we merge the index parameters in variable i which is thereby different from the usage of j in the Reveal and Challenge oracles.

Additionally we introduce the array CK and the set XSK. XSK is used to track which secret keys of A's public keys are (potentially) exposed. Thereby the indexes of both, the secret key

Game $\text{KIND}^b_{\mathsf{R}}(\mathcal{A})$

00 $R[\cdot] \leftarrow \bot$ — $\mathbf{G}_{\geq 0}$
01 $s_A \leftarrow 0;\ r_B \leftarrow 0$
02 $adc_A[\cdot] \leftarrow \bot;\ is_B \leftarrow \mathtt{T}$
03 $key_A[\cdot] \leftarrow \bot;\ key_B[\cdot] \leftarrow \bot$
04 $\text{XP}_A \leftarrow \emptyset$
05 $\text{TR}_A \leftarrow \emptyset;\ \text{TR}_B \leftarrow \emptyset$
06 $\text{CH}_A \leftarrow \emptyset;\ \text{CH}_B \leftarrow \emptyset$
07 $SK_\star[\cdot] \leftarrow \bot;\ KT_\star[\cdot] \leftarrow \bot$ — $\mathbf{G}_{\geq 2}$
08 $CK[\cdot] \leftarrow \bot;\ XSK \leftarrow \emptyset$ — $\mathbf{G}_{\geq 2}$
09 $sk \leftarrow_\$ \mathcal{SK}$
10 $pk \leftarrow \text{gen}_{\mathsf{K}}(sk)$
11 $(K, k.m) \leftarrow_\$ \mathcal{K}^2;\ t \leftarrow \epsilon$
12 $S_A \leftarrow (pk, K, k.m, t)$
13 ~~$S_B \leftarrow (sk, K, k.m, t)$~~ — $\mathbf{G}_{<2}$
14 $KT_\star[s_A, \mathtt{S}] \leftarrow (K, k.m, t)$ — $\mathbf{G}_{\geq 2}$
15 $SK_\star[s_A, \mathtt{S}] \leftarrow sk$ — $\mathbf{G}_{\geq 2}$
16 $b' \leftarrow_\$ \mathcal{A}$
17 Require $\text{TR}_A \cap \text{CH}_A = \emptyset$
18 Require $\text{TR}_B \cap \text{CH}_B = \emptyset$
19 Stop with b'

Oracle $\text{SndA}(ad)$

20 $i \leftarrow (s_A, \mathtt{S})$ — $\mathbf{G}_{\geq 2}$
21 $(pk, K, k.m, t) \leftarrow S_A$
22 $(k, c) \leftarrow_\$ \text{enc}(pk)$
23 $CK[c, i] \leftarrow k$ — $\mathbf{G}_{\geq 2}$
24 $\tau \leftarrow \text{tag}(k.m, ad \,\|\, c)$
25 $C \leftarrow c \,\|\, \tau$
26 $t \xleftarrow{\shortparallel} ad \,\|\, C$
27 ~~$k.o \,\|\, K^* \,\|\, k.m \,\|\, sk \leftarrow \text{H}(K, k, t)$~~ — $\mathbf{G}_{<3}$
28 $k.o \,\|\, K^* \,\|\, _ \leftarrow \text{G}(K, t, i, \mathtt{T})$ — $\mathbf{G}_{\geq 3}$
29 $k.m \leftarrow_\$ \mathcal{K};\ sk \leftarrow_\$ \mathcal{SK}$ — $\mathbf{G}_{\geq 3}$
30 ~~$\text{SetO}(K, t, i, k.m, sk)$~~ — $\mathbf{G}_{3-4}$
31 $pk \leftarrow \text{gen}_{\mathsf{K}}(sk)$
32 $S_A \leftarrow (pk, K^*, k.m, t)$
33 $adc_A[s_A] \leftarrow (ad, C)$
34 $key_A[s_A] \leftarrow k.o$
35 $s_A \leftarrow s_A + 1$
36 $i \leftarrow (s_A, \mathtt{S})$ — $\mathbf{G}_{\geq 2}$
37 $KT_\star[i] \leftarrow (K^*, k.m, t)$ — $\mathbf{G}_{\geq 2}$
38 $SK_\star[i] \leftarrow sk$ — $\mathbf{G}_{\geq 2}$
39 Return C

Oracle ExposeA
As in KIND

Oracle $\text{Reveal}(u, j)$
As in KIND

Oracle $\text{RcvB}(ad, C)$

40 Require $S_B \neq \bot$
41 If $is_B \wedge adc_A[r_B] \neq (ad, C)$:
42 $\quad SK_\star[r_B, \mathtt{R}] \leftarrow SK_\star[r_B, \mathtt{S}]$ — $\mathbf{G}_{\geq 2}$
43 $\quad KT_\star[r_B, \mathtt{R}] \leftarrow KT_\star[r_B, \mathtt{S}]$ — $\mathbf{G}_{\geq 2}$
44 $\quad is_B \leftarrow \mathtt{F}$
45 $\quad$ If $r_B \in \text{XP}_A$:
46 $\quad\quad \text{TR}_B \xleftarrow{\cup} [r_B, \ldots]$
47 $\quad$ Else if $(r_B, \mathtt{S}) \notin XSK$: — $\mathbf{G}_{\geq 5}$
48 $\quad\quad forge \leftarrow \mathtt{T}$ — $\mathbf{G}_{\geq 5}$
49 If not is_B: — $\mathbf{G}_{\geq 2}$
50 $\quad i \leftarrow (r_B, \mathtt{R})$ — $\mathbf{G}_{\geq 2}$
51 $\quad$ ~~$(sk, K, k.m, t) \leftarrow S_B$~~ — $\mathbf{G}_{<2}$
52 $\quad (K, k.m, t) \leftarrow KT_\star[i]$ — $\mathbf{G}_{\geq 2}$
53 $\quad$ ~~$sk \leftarrow SK_\star[i]$~~ — $\mathbf{G}_{2-4}$
54 $\quad c \,\|\, \tau \leftarrow C$
55 $\quad$ Require $\text{vfy}_{\mathsf{M}}(k.m, ad \,\|\, c, \tau)$
56 $\quad$ If $forge$: Abort — $\mathbf{G}_{\geq 5.2}$
57 $\quad$ ~~$k \leftarrow \text{dec}(sk, c)$~~ — $\mathbf{G}_{<3}$
58 $\quad$ ~~Require $k \neq \bot$~~ — $\mathbf{G}_{<3}$
59 $\quad$ Require $\text{dec}(SK_\star[i], c) \neq \bot$ — $\mathbf{G}_{\geq 3}$
60 $\quad t \xleftarrow{\shortparallel} ad \,\|\, C$
61 $\quad$ ~~$k.o \,\|\, K^* \,\|\, k.m \,\|\, sk \leftarrow \text{H}(K, k, t)$~~ — $\mathbf{G}_{<3}$
62 $\quad k.o \,\|\, K^* \,\|\, k.m \,\|\, sk \leftarrow \text{G}(K, t, i, \mathtt{F})$ — $\mathbf{G}_{\geq 3}$
63 $\quad$ ~~$S_B \leftarrow (sk, K^*, k.m, t)$~~ — $\mathbf{G}_{<2}$
64 If is_B: $k \leftarrow \diamond$
65 $key_B[r_B] \leftarrow k$
66 $r_B \leftarrow r_B + 1$
67 If not is_B: — $\mathbf{G}_{\geq 2}$
68 $\quad i \leftarrow (r_B, \mathtt{R})$ — $\mathbf{G}_{\geq 2}$
69 $\quad KT_\star[i] \leftarrow (K^*, k.m, t)$ — $\mathbf{G}_{\geq 2}$
70 $\quad SK_\star[i] \leftarrow sk$ — $\mathbf{G}_{\geq 2}$
71 Return

Oracle ExposeB

72 $\text{TR}_B \xleftarrow{\cup} [r_B, \ldots]$
73 If is_B:
74 $\quad \text{TR}_A \xleftarrow{\cup} [r_B, \ldots]$
75 $\quad \mathtt{U} \leftarrow \mathtt{S}$ — $\mathbf{G}_{\geq 2}$
76 $\quad XSK \xleftarrow{\cup} [r_B, \ldots] \times [\mathtt{S}]$ — $\mathbf{G}_{\geq 2}$
77 Else: $\mathtt{U} \leftarrow \mathtt{R}$ — $\mathbf{G}_{\geq 2}$
78 $(K, k.m, t) \leftarrow KT_\star[r_B, \mathtt{U}]$ — $\mathbf{G}_{\geq 2}$
79 $sk \leftarrow SK_\star[r_B, \mathtt{U}]$ — $\mathbf{G}_{\geq 2}$
80 $S_B \leftarrow (sk, K, k.m, t)$ — $\mathbf{G}_{\geq 2}$
81 Return S_B

Oracle $\text{Challenge}(u, j)$
As in KIND

Fig. 15: Proof of URKE.

that is actually exposed and all subsequent secret keys that are derived from it are unified in XSK. Deriving one secret key from another one means that the earlier secret key (in combination with the simultaneously exposed chaining key K) can be used to obtain all information (the encapsulated key k) to request the random oracle outputting the next secret key. Array CK stores the encapsulated KEM keys k of A for ciphertexts c under the public key with secret key i. The proof makes thereby use of this array as CK is used in the OW game (see Figure 22).

Random oracle We construct our random oracle by defining several procedures that program the simulated function and that can request the output on a provided input. Procedures G and SetO are defined only for the simulation's internal requests and oracle H is provided to the adversary.

The public oracle is initially defined as a function that randomly samples an output value on the first request of the input value and outputs it on all further requests of this input. In addition to the output, consisting of three symmetric keys and the secret key, a flag *sen* to trace the requests' origin is stored for every entry (i.e., whether A's simulation—the sender—initially requested the random oracle for that entry or not).

Figure 16 as the description of the random oracle follows the same principle as the one for the proof of SRKE (see Figure 19).

Oracle $\mathrm{H}(K, k, t)$ — $\mathbf{G}_{\geq 0}$

- 00 Require $k \in \mathcal{K}$
- 01 $t' \parallel ad \parallel c \parallel \tau \leftarrow t$
- 02 $(k.o, K^*, k.m) \leftarrow_\$ \mathcal{K}^3$; $sk \leftarrow_\$ \mathcal{SK}$
- 03 $i \leftarrow \epsilon$; $sen \leftarrow \mathtt{F}$
- 04 If $\exists i : R[K, t, k, i] \neq \bot$:
- 05 $\quad (k.o, K^*, k.m, sk, sen) \leftarrow R[K, t, k, i]$
- 06 Else if $\exists i : R[K, t, \epsilon, i] \neq \bot$ — $\mathbf{G}_{\geq 3}$
  - $\wedge i \in \mathbb{N} \times [\mathtt{R}]$ — $\mathbf{G}_{\geq 3}$
  - $\wedge \mathrm{dec}(sk, c) = k, sk \leftarrow SK_\star[i]$: — $\mathbf{G}_{\geq 3}$
- 07 $\quad (k.o, K^*, k.m, sk, sen) \leftarrow R[K, t, \epsilon, i]$ — $\mathbf{G}_{\geq 3}$
- 08 Else if $\exists i : R[K, t, \epsilon, i] \neq \bot$ — $\mathbf{G}_{\geq 3}$
  - $\wedge i \in \mathbb{N} \times [\mathtt{S}] \wedge CK[c, i] = k$: — $\mathbf{G}_{\geq 3}$
- 09 $\quad (k.o, K^*, k.m, sk, sen) \leftarrow R[K, t, \epsilon, i]$ — $\mathbf{G}_{\geq 3}$
- 10 $\quad$ If $i \notin XSK$: Abort — $\mathbf{G}_{\geq 5.1}$
- 11 $\quad sk \leftarrow SK_\star[i+1]$ — $\mathbf{G}_{\geq 4}$
- 12 $\quad (_, k.m, _) \leftarrow KT_\star[i+1]$ — $\mathbf{G}_{\geq 4}$
- 13 $R[K, t, k, i] \leftarrow (k.o, K^*, k.m, sk, sen)$
- 14 Return $k.o \parallel K^* \parallel k.m \parallel sk$

Proc $\mathrm{G}(K, t, i, sen)$ — $\mathbf{G}_{\geq 3}$

- 15 $t' \parallel ad \parallel c \parallel \tau \leftarrow t$
- 16 $k \leftarrow \epsilon$
- 17 If $sen \wedge \exists k : R[K, t, k, \epsilon] \neq \bot$:
- 18 $\quad$ Abort
- 19 Else if $\neg sen \wedge \exists k : R[K, t, k, \epsilon] \neq \bot$:
  - $\mathrm{dec}(sk, c) = k, sk \leftarrow SK_\star[i]$:
- 20 $\quad (k.o, K^*, k.m, sk, sen') \leftarrow R[K, t, k, \epsilon]$
- 21 Else:
- 22 $\quad (k.o, K^*) \leftarrow_\$ \mathcal{K}^2$; $k.m \leftarrow \epsilon$; $sk \leftarrow \epsilon$
- 23 $\quad$ If $\neg sen$: $k.m \leftarrow_\$ \mathcal{K}$; $sk \leftarrow_\$ \mathcal{SK}$
- 24 $\quad R[K, t, k, i] \leftarrow (k.o, K^*, k.m, sk, sen)$
- 25 Return $k.o \parallel K^* \parallel k.m \parallel sk$

Proc $\mathrm{SetO}(K, t, i, k.m, sk)$ — $\mathbf{G}_{\geq 3}$

- 26 $(k.o, K^*, k.m', sk', sen) \leftarrow R[K, t, \epsilon, i]$
- 27 $R[K, t, \epsilon, i] \leftarrow (k.o, K^*, k.m, sk, sen)$
- 28 Return

Fig. 16: Random oracle description for proof of URKE.

Game 3 – Internal access to random oracle We introduce the internal procedures to request the random oracle. These procedures provide the simulation the opportunity to request the random oracle output on an undefined input key k. The index for the array is thereby defined by chaining key K, transcript t—hence implicitly by the last ciphertext c—, and the index of the key pair for which the ciphertext is encapsulated. For internally requested ciphertexts c, the correct but unknown key k is implicitly programmed as input into the random oracle.

By requesting the random oracle without providing the decapsulated key, the decapsulation and hence the previous secret key of B are not necessary for the simulation of receiving anymore and thereby are removed. In order to correctly simulate, the output of the decapsulation still needs to be compared to the $\bot$ symbol. When reducing the security to the OW game, this comparison can be conducted by requesting the game's Check oracle without using the respective secret key explicitly (see Figure 22).

External queries to the random oracle H are stored in entries of the simulation's array R, indexed by input tuples (K, k, t) and ϵ implying that the entries are not associated to a secret key that is used by the simulation. The output values are sampled uniformly at random for an initial request and afterwards reaccessed from the described array entry. Initial internal queries

are stored with an empty string ϵ at the index parameter position of the input KEM key k. Additionally the index of the secret key i of the previous KEM operation is set as array index parameter (for A's simulation: index of secret key for public key with which encapsulation of the last c in t was executed; for B's simulation: index of secret key with which decapsulation would have been called). For initial internal queries of A's simulation, only the output session key $k.o$ and the output chaining key K^* are sampled and set within G. The remaining outputs are sampled independently and set by the procedure SetO. This makes no difference regarding the simulation but will be important for lazy sampling in game G_4. For the simulation of B's internal queries, all outputs values are sampled at once.

In order to show how correctness and output distribution of the random oracle are preserved, we first describe how the sequence of 1) an external request before 2) an internal request to the same entry is processed and then vice versa. Note that equality of the queries cannot be verified by input values (K, k, t) since k is not specified for internal requests directly.

Two successive external requests are still processed as before: if the respective entry does not exist yet, it is generated, otherwise it is accessed and returned. Two successive internal requests cannot occur and therefore do not need to be considered. Since the inputs to the random oracle include the transcript, two successive requests with the same transcript cannot be made by A nor B respectively nor first by one of them and then by the other one (note that B's simulation only request the random oracle out of sync).

An internal random oracle request of A's simulation is marked by the last parameter of G set true. For requests of this form, an abortion is conducted if there already exists any random oracle entry with the same transcript and chaining key (independent of the KEM key input k). Since the last ciphertext c in the transcript t is the result of an encapsulation, performed right before the internal random oracle request—and thereby not known to the adversary—, the probability of the abortion can be bounded by the birthday bound. We can use the key space as set over which the birthday bound is drawn since we can assume that encapsulated keys are sampled uniformly at random (and thereby use this distribution instead of the resulting ciphertexts' distribution).

For internal requests of B's simulation, previous random oracle entries with the same transcript and chaining key are validated with respect to the tuple (c, i, k) where c is the last ciphertext in t, k is the input KEM key of an existing random oracle entry queried externally, and i is the index of the secret key with which G is invoked. If k can be decapsulated from c with $SK_\star[i]$, then the existing random oracle entry equals the one that was intended to be queried by the simulation of B. Consequently all output values are taken from the found external entry and returned by G.

If there exists an entry made internally (and hence without specified k) for an external random oracle request, a validation with respect to the tuple (c, i, k) is conducted as well. Thereby external request and internal entry are equal if the ciphertext of both inputs can be decapsulated by secret key with index i from the internal entry's index parameters to the input key k of the external request. This validation is split into entries made by B and entries made by the A with secret key i (tracked by array CK).

The array CK represents the Solve oracle of the OW game (see Figure 22). As such, it may only be queried for unexposed secret keys. However, for the reduction we assume that the CK array is manually filled for exposed secret keys and the Solve oracle is requested for unexposed secret keys respectively. Therefore we can assume that for all ciphertext-secret key pairs of A an entry exists in CK.

Due to the abortion in the random oracle in line 18, a random oracle entry that is internally requested by A was not requested by the adversary before. As argued before, the probability of an abortion in game G_3 can be upper bounded by the birthday bound, where q_S is the number

of invocations of the SndA oracle and $\mathcal{K}_{\mathsf{K}}$ is the key space of the KEM[18].

$$\mathrm{Adv}^{G_2,G_3}(\mathcal{D}) \leq \frac{q_F q_S}{|\mathcal{K}_{\mathsf{K}}|}$$

Game 4 – Random oracle with lazy sampling In game G_4 we stop to set secret key and MAC key as part of the output values of the internal random oracle requests of A's sending. Instead the actual output of the external random oracle is instantly set as soon as the adversary requests it.

Random oracle entries, defined by the internal requests of B, are still equipped with all output values as before. As we will show in the next game, these outputs will not be challengeable anymore.

Game 5 – Abortion In game G_5 we finally abort if the adversary queries the random oracle for entries that reveal a challengeable key.

At first game $G_{5.1}$ is aborted if the adversary externally queries an entry that was made by A with a ciphertext (encoded in t) that is designated for an unexposed secret key (see Figure 16 line 10). Since only entries made by A result in this abortion, it only occurs if the adversary was able to derive a key k from a ciphertext c sent by A to correctly request the random oracle.

To reduce the probability of this abortion event to the OW security of the KEM, the secret key with which the key k can be decapsulated from the ciphertext c must not have been used by the simulation. This holds because on the one hand, the secret key was not exposed (by the abortion condition) and on the other hand, the random oracle entry that would have contained this secret key as the output was not requested before, because otherwise the simulation would have been aborted before.

It is now important to observe that the exposed keys in XSK correspond to the traceable established keys in TR_A (i.e., $\mathrm{TR}_A = XSK$). Hence for the game to be aborted, the random oracle must be requested for any challengeable established key of A. Consequently either the adversary did not request the random oracle for an entry with which it would know an established key of A or the game aborted and thereby the OW game can be won.

$$\mathrm{Adv}^{G_5,G_{5.1}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{ow}}_{\mathsf{K}}(\mathcal{B})$$

After game $G_{5.1}$, for gaining an advantage in winning the game, the adversary can only gather information via requesting the random oracle for asynchronously established keys of B.

Therefore consider that at receiving the first manipulated ciphertext, the following conditions need to hold for $\mathit{forge} = \mathtt{T}$ (see lines 45–48): 1) A was not exposed to be impersonated towards B and 2) B's current secret key was not marked to be exposed. In addition, these conditions imply that 3) the last *common* random oracle query was not requested by the adversary. Condition 3 holds because condition 2 implies that A is challengeable and an external query to a random oracle entry outputting a challengeable key would have resulted in an abortion of game $G_{5.1}$.

As a result of these conditions, the simulation did not leak the MAC key in B's state at receiving the manipulated ciphertext—except that it was used to generate the MAC tag of a potentially sent ciphertext of A—and the MAC key was sampled uniformly at random. Now distinguishing between games $G_{5.1}$ and $G_{5.2}$ can be reduced to the SUF security of the MAC M because $G_{5.2}$ only aborts if the MAC tag was valid but the ciphertext was manipulated and, as shown above, the MAC key was not provided to the adversary. Hence in order to distinguish

[18] For simplification, we treat the KEM and MAC key spaces equally in the construction description. For accuracy, we separate them in the advantage consideration.

between these two games, a distinguisher needs to provide a forged MAC tag. Therefore a distinguisher's advantage can be bounded by:

$$\mathrm{Adv}^{G_{5.1},G_{5.2}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{C})$$

If $G_{5.2}$ did not abort, B cannot be challenged because either one of conditions 1 or 2 were violated and therefore the remaining keys of B are set to be traceable (see lines 46 and 74), or the state was erased due to an invalid MAC tag.

Finally the adversary can win game $G_{5.2}$ with probability 1/2 because no information on a challengeable key of either A or B can be gained by the adversary.

$$\mathrm{Adv}^{G_{5.2}}(\mathcal{A}) = 0$$

Proof result Taking the bounds drawn in the game hops above provides us the upper bound of the advantage that an adversary has in the URKE KIND$_{\mathsf{R}}$ game depending on the advantage of the adversaries $\mathcal{B}$ and $\mathcal{C}$:

$$\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{ow}}_{\mathsf{K}}(\mathcal{B}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{C}) + \frac{q_S q_F}{|\mathcal{K}_{\mathsf{K}}|}$$

Please note that the URKE proof makes no use of the Up$_R$ oracle of the KUOW game and therefore a generic CCA secure KEM suffices to reach URKE (with a loss factor of q_S). However, the update algorithm could be used instead of generating new key pairs after each sending and receiving since it provides the same functionality and security but A would never learn the secret key to her public key.

B Rationales for SKRE design

We sketched the reasons for employing sophisticated primitives as basic blocks for our design of SRKE in the main body. In this section we develop more detailed arguments for our design choices by providing attacks on constructions different from our design. At first it is described why SRKE requires signatures for protecting the communication from B to A—in contrast to employing a MAC from A to B. Then we will evaluate the requirements for the KEM key pair update in the setting of concurrent sending of A and B.

B.1 Signatures from A to B

While a MAC suffices to protect authenticity for ciphertexts sent from A to B it does not suffice to protect the authenticity in the counter direction. The reason for this lies within the conditions with which future session keys of A and B are marked traceable in the KIND$_{\mathsf{R}}$ game of SRKE. An impersonation of A towards B has the same effect on the traceability of B's future session keys as if the adversary exposes B's state and then brings B out of sync. Either way all future session keys of B are marked traceable (see Figure 9 lines 37 and 54,38). In the first scenario, the adversary can compute the same session keys as B because the adversary initiates the key establishment impersonating A. In the second scenario, the adversary can comprehend B's computations during the receipt of ciphertexts because it possesses the same state information as B.

For computations of A, however, only the former scenario is applicable: if the adversary impersonated B towards A, then again the adversary is in the position to trace the establishment of session keys of A because it can simulate the respective counterpart's receiver computations. In contrast to this, when exposing A and bringing her out of sync, according to the KIND$_{\mathsf{R}}$

game, the adversary must not obtain information on her future session keys (see Figure 9 lines 52 et seqq.). As a result, the exposure of A's state should not enable the adversary to impersonate B towards A. Consequently the authentication of the communication from B to A cannot be reached by a primitive with a symmetric secret but rather the protocol needs to ensure that B needs to be exposed in order to impersonate him towards A.

The non-trivial attack that is defended by employing signatures consists of the following adversary behavior: $S_A \leftarrow \text{ExposeA}$; *Extract authentication secrets from S_A to derive S'_B*; $(C', S''_B) \leftarrow_\$ \text{snd}_B(S'_B, \epsilon)$; $\text{RcvA}(C', \epsilon)$; $C_{A1} \leftarrow_\$ \text{SndA}(\epsilon)$; $k_b \leftarrow_\$ \text{Challenge}(A, 1)$. Thereby the adversary must not be able to decide whether it obtained the real or random key for ciphertext C_{A1} from the challenge oracle. Please note that this is related to *key-compromise impersonation* resilience (while in this case ephemeral signing keys are compromised).

B.2 Key-updatable KEM for concurrent sending

There exist two crucial properties that are required from the key pair update of the KEM in the setting in which A and B send concurrently. Firstly, the key update needs to be forward secure which means that an updated secret key does not reveal information on encapsulations to previous secret keys or to differently updated secret keys. Secondly, the update of the public key must not reveal information on keys that will be encapsulated to its respective secret key. We will explain the necessity of these requirements one after another.

The key pair update for concurrently sending only affects epochs that have been proposed by B, but that have not been processed by A yet. These updates have to consider ciphertexts that A sent during the transmission of the public key for a new epoch from B to A. Subsequently we describe an example scenario in which these updates are necessary for defending a non-trivial attack: In the worst case, all secrets among A and B have been exposed to the adversary before B proposes a new epoch ($S_A \leftarrow \text{ExposeA}$; $S_B \leftarrow \text{ExposeB}$). Thereby only a public key sent by B after the exposure will provide security for future session key establishments initiated by A. Now consider a scenario in which B proposes this new public key to A ($C_{B1} \leftarrow_\$ \text{SndB}(\epsilon)$; $\text{RcvA}(C_{B1}, \epsilon)$) and A is simultaneously impersonated towards B ($(S'_A, k', C') \leftarrow_\$ \text{snd}_A(S_A, \epsilon)$; $\text{RcvB}(C', \epsilon)$). Since B proposed the new public key within C_{B1} in sync and A received it in sync respectively—and B was not exposed under the new state—, future established session keys of A are considered to be indistinguishable from random key space elements again ($C_{A1} \leftarrow_\$ \text{SndA}(\epsilon)$; $k_b \leftarrow_\$ \text{Challenge}(A, 1)$). Due to the impersonation of A towards B, however, B became out of sync. Becoming out of sync cannot be detected by B because the adversary can send a valid ciphertext C' under the exposed state of A S_A. Exposing B out of sync afterwards ($S'_B \leftarrow \text{ExposeB}$), by definition, must not have an impact on the security of session keys established by A (see Figure 9 line 55). As a result, after the adversary performed these steps, the challenged session key is required to be indistinguishable from a random element from the key space. Consequently B must perform an update of the secret key for the newest epoch when receiving C' such that the public key transmitted in C_{B1} still provides its security guarantees when using it in A's final send operation (remember that all previous secrets among A and B were exposed before).

When accepting that an update of B's future epoch's secret keys is required at the receipt of ciphertexts, another condition arises for the respective update of A's public keys. For maintaining correctness, A of course needs to compute updates of a received new public key with respect to all previously sent ciphertexts that B was not aware of when sending the public key. Suppose A's and B's secrets have all been exposed towards the adversary again ($S_A \leftarrow \text{ExposeA}$; $S_B \leftarrow \text{ExposeB}$). Now A sends a new key establishing ciphertext and B proposes a new epoch public key ($C_{A1} \leftarrow_\$ \text{SndA}(\epsilon)$; $C_{B1} \leftarrow_\$ \text{SndB}(\epsilon)$). According to the previous paragraph, A needs to update the received public key in C_{B1} with respect to C_{A1} after receiving C_{B1} ($\text{RcvA}(C_{B1}, \epsilon)$).

Since C_{B1} introduces a new epoch, the next send operation of A needs to establish a secure session key again ($C_{A2} \leftarrow_{\$} \text{SndA}(\epsilon)$; $k_b \leftarrow_{\$} \text{Challenge}(A, 2)$). Now observe that in order to update the received public key, A can only use information from her state S_A—which is known by the adversary—, public information like the transmitted ciphertexts, and randomness. Essentially, the update can hence only depend on information that the adversary knows plus random coins which cannot be transmitted confidentially to B before performing the update (because there exist no secrets apart from the key pair that first needs to be updated). Since B probably received C_{A1} before A received C_{B1}, A cannot influence the update performed by B on his secret key. This means that the updates of A and B need to be conducted independently. As such, the adversary is able to perform the update on the same information that A has (only randomness of A and the adversary can differ). Nevertheless, both updates—the one performed by the adversary and the one performed by A—need to be compatible to the secret key that B derives from his update. As a result, the update of the public key must not reveal the respective secret key (or any other information that can be used to obtain information on keys encapsulated to this updated public key). Otherwise, the adversary would obtain this information as well (and thereby the security of key ($A, 2$) would not be preserved).

Both requirements are reflected in the security game of the kuKEM (see Figure 1).

B.3 Encapsulation to all public keys

Subsequently we describe a scenario in which A only maintains one public key in her state to which she can securely encapsulate keys (while the state contains multiple *useless* public keys). This scenario is crucial because A does not know, which of her public keys provides security, and the SRKE protocol is required to output secure session keys in this scenario. Consequently only encapsulating to all public keys in A's state solves the underlying issue. The reasons for encapsulating to all public keys in A's state is closely related to the reasons for employing a kuKEM in SRKE (see the previous subsection).

Assume the adversary exposes the states of both parties ($S_A \leftarrow \text{ExposeA}$; $S_B \leftarrow \text{ExposeB}$). Consequently none of A's public keys provides any security guarantees for the encapsulation towards the adversary anymore. If the adversary lets B send a ciphertext and thereby propose a new public key to A, A's future session keys are required to be secure again ($C_{B1} \leftarrow_{\$} \text{SndB}(\epsilon)$; $\text{RcvA}(C_{B1}, \epsilon)$). Impersonating A towards B and then exposing B to obtain his state has—according to the KIND_R game—no influence on the traceability of A's future session keys ($(S'_A, k', C') \leftarrow_{\$} \text{snd}_A(S_A, \epsilon)$; $\text{RcvB}(C', \epsilon)$; $S'_B \leftarrow \text{ExposeB}$). However, our construction allows the adversary to impersonate B towards A afterwards: the impersonation of A towards B only *invalidates* the kuKEM secret key in B's state via the key update in B's receive algorithm. The signing key in B's state is still valid for the communication to A since it was not modified at the receipt of the impersonating ciphertext. As such, the adversary may use the signing key and then implant further public keys in A's state by sending these public keys to A ($(S''_B, C'') \leftarrow_{\$} \text{snd}_B(S'_B, \epsilon)$; $\text{RcvA}(C'', \epsilon)$). These public keys do not provide security with respect to A's session keys since the adversary can freely choose them. As a result, only the public key that B sent in sync before A was impersonated towards B belongs to a secret key that the adversary does not know (public key in C_{B1}). Since A has no indication which public key's secret key is not known by the adversary (note that A and B were exposed at the beginning of the presented scenario and the adversary planted own public keys in A's state at the end of the scenario by sending valid ciphertexts), A needs to encapsulate to all public keys in order to obtain at least one encapsulated key as secret input to the random oracle such that the session key also remains secure ($C_{A1} \leftarrow_{\$} \text{SndA}(\epsilon)$; $k_b \leftarrow_{\$} \text{Challenge}(A, 1)$).

Observe that the scenario, described above, lacks an argument why also the first public key in A's state needs to be used for the encapsulation if A received further public keys from

B afterwards. The reason for also using the first public key, that is always derived from the previous random oracle output, lies within A's sending after becoming out of sync. A became out of sync by receiving C'' (see above). When sending C_{A1}, A derived a new public key for her state. The secret key to this public key was part of the same random oracle output as the session key that is challenged afterwards $(A, 1)$. As argued before, this session key is secure (for all details we refer the reader to the proof in Appendix C). Consequently the public key in A's state after sending C_{A1} provides security against the adversary regrading encapsulations. However, the adversary can still plant new public keys to A's state $\big((S_B''', C''') \leftarrow_\$ \mathrm{snd}_B(S_B'', \epsilon);$ $\mathrm{RcvA}(C''', \epsilon)\big)$. As such, only the first public key in A's state provides security after A became out of sync (and sent once afterwards). All remaining public keys may belong to secret keys chosen by the adversary. Since A will not notice when she became out of sync, she also needs to include the first public key in her state for encapsulating within her send algorithm in order to compute secure session keys $\big(C_{A2} \leftarrow_\$ \mathrm{SndA}(\epsilon);\ k_{b2} \leftarrow_\$ \mathrm{Challenge}(A, 2)\big)$.

As a result, A always needs to encapsulate to all public keys in her state such that at least one encapsulated key is a secret input to the random oracle (in case her future session keys were not marked traceable by the $\mathrm{KIND}_{\mathsf{R}}$ game).

C Proof of SRKE

Model and construction of SRKE are very similar to the model and construction of URKE respectively. Therefore the proof approaches by the same idea. The exact game hops are however more complex because of the more sophisticated employed primitives in the SRKE construction and the management of variables due to the extended communication setup (both parties can send and receive anytime).

Even though the security of URKE is implied by the security of SRKE, for didactic reasons we provide both proofs.

Overview In addition to the proof of URKE, SRKE employs an abortion rule in game G_1 in case of a signature forgery. All remaining game hops follow the same idea as the URKE proof: G_2 limits B's simulation to sending and *out of sync* receiving. Receiving *in sync* of B is simulated by the computations of A's sending—which by correctness of the construction has the same result. G_3 programs the random oracle with internal procedures for the simulation without the knowledge of encapsulated kuKEM keys such that sent ciphertexts include embedded kuKEM challenges. To reduce the adversary's random oracle queries to the solution of the employed hardness assumptions, G_4 starts to lazy sample the random oracle outputs—including the next secret key and MAC key. Finally G_5 aborts on bad events which are proven to only occur if the adversary broke one of the hardness assumptions.

Figure 17 and Figure 18 depict the SRKE $\mathrm{KIND}_{\mathsf{R}}$ game (see Figure 9) including our SRKE construction from Figure 10 and the game hops described below. Figure 17 includes the initialization of game and scheme, and the communication from B to A with a helper procedure GetSK. The opposite communication direction is depicted in Figure 18 with the oracles for the adversary to expose, reveal, and challenge. The description of the random oracle can be found in Figure 19.

Game 1 – Excluding signature forgeries Game G_1 aborts if the adversary successfully forges a signature to drift the parties' states out of sync. Thereby distinguishing between the original game $\mathrm{KIND}_{\mathsf{R}}$ and G_1 can be reduced to the SUF security of the one-time signature scheme. In order to do so, the reduction guesses, when the adversary plants a signature forgery (i.e., for which signature key pair a forgery is received by A). For this key pair the generation,

Game $\mathrm{KIND}^b_{\mathsf{R}}(\mathcal{A})$

00 $R[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq 0}$
01 $s_A \leftarrow 0$; $r_B \leftarrow 0$; $is_A \leftarrow \mathtt{T}$
02 $s_B \leftarrow 0$; $r_A \leftarrow 0$; $is_B \leftarrow \mathtt{T}$
03 $e_A \leftarrow 0$; $\mathrm{EP}_A[\cdot] \leftarrow \bot$
04 $\mathrm{E}_B^{\vdash} \leftarrow 0$; $\mathrm{E}_B^{\dashv} \leftarrow 0$
05 $adc_A[\cdot] \leftarrow \bot$; $key_B[\cdot] \leftarrow \bot$
06 $adc_B[\cdot] \leftarrow \bot$; $key_A[\cdot] \leftarrow \bot$
07 $\mathrm{XP}_A \leftarrow \emptyset$; $\mathrm{TR}_A \leftarrow \emptyset$; $\mathrm{CH}_A \leftarrow \emptyset$
08 $\mathrm{XP}_B \leftarrow \emptyset$; $\mathrm{TR}_B \leftarrow \emptyset$; $\mathrm{CH}_B \leftarrow \emptyset$
09 $oos_B \leftarrow \infty$; $forge_B \leftarrow \mathtt{F}$ $\quad \mathbf{G}_{\geq 1}$
10 $SK_\star[\cdot] \leftarrow \bot$; $KT_\star[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq 2}$
11 $es_A \leftarrow 0$; $er_B \leftarrow 0$; $ims \leftarrow \mathtt{F}$ $\quad \mathbf{G}_{\geq 2}$
12 $XSK \leftarrow \emptyset$; $\Gamma[\cdot] \leftarrow 0$; $CK[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq 2}$
13 $oos_A \leftarrow \infty$; $forge_A \leftarrow \mathtt{F}$ $\quad \mathbf{G}_{\geq 5}$
14 $(sgk_\star, vfk) \leftarrow_\$ \mathrm{gen}_{\mathsf{S}}$ $\quad \mathbf{G}_{\geq 2}\ \star$
15 $(sk, pk) \leftarrow_\$ \mathrm{gen}_{\mathsf{K}}$
16 $(K, k.m) \leftarrow_\$ \mathcal{K}^2$; $t \leftarrow \epsilon$
17 $E^{\vdash} \leftarrow 0$; $E^{\dashv} \leftarrow 0$
18 $s \leftarrow 0$; ~~$r \leftarrow 0$~~ $\quad \mathbf{G}_{<2}$ ~~r~~
19 $E^{\vdash}_\star \leftarrow 0$; $E^{\dashv}_\star \leftarrow 0$ $\quad \mathbf{G}_{\geq 2}\ \star$
20 $PK[\cdot] \leftarrow \bot$; $PK[0] \leftarrow pk$
21 ~~$SK[\cdot] \leftarrow \bot$; $SK[0] \leftarrow sk$~~ $\quad \mathbf{G}_{<2}$
22 $L_A[\cdot] \leftarrow \bot$; $L_A[0] \leftarrow \diamond$
23 $L_{B\star}[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq 2}\ \star$
24 $S_A \leftarrow (PK, E, s, L_A, vfk, K, k.m, t)$
25 ~~$S_B \leftarrow (SK, E, r, L_B, sgk, K, k.m, t)$~~ $\quad \mathbf{G}_{<2}$
26 $KT_\star[0,0,0,\mathtt{S}] \leftarrow (K, k.m, \epsilon)$ $\quad \mathbf{G}_{\geq 2}$
27 $SK_\star[0,0,0,\mathtt{S}] \leftarrow (sk, 0)$ $\quad \mathbf{G}_{\geq 2}$
28 $b' \leftarrow_\$ \mathcal{A}$
29 Require $\mathrm{TR}_A \cap \mathrm{CH}_A = \emptyset$
30 Require $\mathrm{TR}_B \cap \mathrm{CH}_B = \emptyset$
31 Stop with b'

Oracle $\mathrm{SndB}(ad)$

32 Require $S_B \neq \bot$
33 ~~$(SK, E, r, L, sgk, K, k.m, t) \leftarrow S_B$~~ $\quad \mathbf{G}_{<2}$
34 $(sk^*, pk^*) \leftarrow_\$ \mathrm{gen}_{\mathsf{K}}$
35 $(sgk^*, vfk^*) \leftarrow_\$ \mathrm{gen}_{\mathsf{S}}$
36 $E^{\dashv}_\star \leftarrow E^{\dashv}_\star + 1$; ~~$SK[E^{\dashv}] \leftarrow sk^*$~~ $\quad \mathbf{G}_{<2}\ \star$
37 $C \leftarrow r_B \,\|\, pk^* \,\|\, vfk^*$ $\quad \mathbf{G}_{\geq 2}\ \star$
38 $\sigma \leftarrow_\$ \mathrm{sgn}(sgk_\star, ad \,\|\, C)$ $\quad \mathbf{G}_{\geq 2}\ \star$
39 $sgk_\star \leftarrow sgk^*$ $\quad \mathbf{G}_{\geq 2}\ \star$
40 $C \leftarrow C \,\|\, \sigma$; $L_{B\star}[E^{\dashv}_\star] \leftarrow \triangleleft \,\|\, ad \,\|\, C$ $\quad \mathbf{G}_{\geq 2}\ \star$
41 ~~$S_B \leftarrow (SK, E, r, L, sgk, K, k.m, t)$~~ $\quad \mathbf{G}_{<2}$
42 If is_B:
43 $\quad adc_B[s_B] \leftarrow (ad, c)$
44 $\quad \mathrm{E}_B^{\dashv} \leftarrow \mathrm{E}_B^{\dashv} + 1$
45 $s_B \leftarrow s_B + 1$
46 If is_B: $SK_\star[E^{\dashv}_\star, 0, 0, \mathtt{S}] \leftarrow (sk^*, s_B)$ $\quad \mathbf{G}_{\geq 2}$
47 Else: $SK_\star[E^{\dashv}_\star, 0, 0, \mathtt{R}] \leftarrow (sk^*, s_B)$ $\quad \mathbf{G}_{\geq 2}$
48 Return c

Oracle $\mathrm{RcvA}(ad, c)$

49 Require $S_A \neq \bot$
50 If $is_A \wedge adc_B[r_A] \neq (ad, c)$:
51 $\quad is_A \leftarrow \mathtt{F}$
52 $\quad oos_A \leftarrow r_A$ $\quad \mathbf{G}_{\geq 5}$
53 $\quad$ If $r_A \in \mathrm{XP}_B$:
54 $\quad\quad \mathrm{TR}_A \xleftarrow{\cup} \mathbb{N} \times [s_A, \ldots]$
55 $\quad\quad ims \leftarrow \mathtt{T}$ $\quad \mathbf{G}_{\geq 2}$
56 $\quad$ Else if $oos_B > r_A$: $forge_B \leftarrow \mathtt{T}$ $\quad \mathbf{G}_{\geq 1}$
57 If is_A: $e_A \leftarrow e_A + 1$
58 $(PK, E, s, L, vfk, K, k.m, t) \leftarrow S_A$
59 $t \xleftarrow{\shortparallel} \triangleleft \,\|\, ad \,\|\, C$; $C \,\|\, \sigma \leftarrow C$
60 Require $\mathrm{vfy}_{\mathsf{S}}(vfk, ad \,\|\, C, \sigma)$
61 If $forge_B$: Abort $\quad \mathbf{G}_{\geq 1}$
62 $r \,\|\, pk^* \,\|\, vfk \leftarrow C$
63 Require $L[r] \neq \bot$
64 $L[\ldots, (r-1)] \leftarrow \bot$; $L[r] \leftarrow \diamond$
65 For $s' \leftarrow r+1$ to s:
66 $\quad pk^* \leftarrow \mathrm{up}(pk^*, L[s'])$
67 $\quad \Gamma[E^{\dashv}+1, \mathtt{S}] \leftarrow \Gamma[E^{\dashv}+1, \mathtt{S}] + 1$ $\quad \mathbf{G}_{\geq 2}$
68 $E^{\dashv} \leftarrow E^{\dashv} + 1$; $PK[E^{\dashv}] \leftarrow pk^*$
69 $S_A \leftarrow (PK, E, s, L, vfk, K, k.m, t)$
70 If $S_A = \bot$: Return $\bot$
71 $r_A \leftarrow r_A + 1$
72 Return

Proc $\mathrm{GetSK}(\mathtt{U}_1, \mathtt{U}_2)$ $\quad \mathbf{G}_{\geq 2}$

73 If $\mathtt{U}_1 = \mathtt{S}$: $(__, L_A, __) \leftarrow S_A$; $P \leftarrow L_A$
74 $SK[\cdot] \leftarrow \bot$
75 $(sk, s) \leftarrow SK_\star[E^{\vdash}_\star, er_B, 0, \mathtt{U}_1]$
76 $SK[E^{\vdash}_\star] \leftarrow sk$; $SK_\star[E^{\vdash}_\star, er_B, 0, \mathtt{U}_2] \leftarrow (sk, s)$
77 For ε from $E^{\vdash}_\star + 1$ to $E^{\dashv}_\star$:
78 $\quad l \leftarrow \max(\ell : SK_\star[\varepsilon, 0, \ell, \mathtt{U}_1] \neq \bot)$
79 $\quad (sk, s) \leftarrow SK_\star[\varepsilon, 0, l, \mathtt{U}_1]$
80 $\quad SK_\star[\varepsilon, 0, l, \mathtt{U}_2] \leftarrow (sk, s)$
81 $\quad$ For $\ell \leftarrow l+1$ to $r_B - s$:
82 $\quad\quad p \leftarrow P[r_B - s + \ell]$
83 $\quad\quad sk \leftarrow_\$ \mathrm{up}(sk, p)$
84 $\quad\quad SK_\star[\varepsilon, 0, \ell, \mathtt{U}_2] \leftarrow (sk, s)$
85 $\quad SK[\varepsilon] \leftarrow sk$
86 $\quad \Gamma[\varepsilon, \mathtt{R}] \leftarrow r_B - s$
87 Return SK

Fig. 17: Proof of SRKE containing initialization, communication from B to A, and two helper procedures. Note that $\mathrm{E}_B, E, E_\star$ refer to different epoch intervals: E_B is the $\mathrm{KIND}_{\mathsf{R}}$ game's interval for B which stops increasing *out of sync*, E denotes the intervals in A's and B's states *locally*, and $E_\star$ is the global substitution for the interval in B's state (for $G_{\geq 2}$).

signing, and verification algorithms are replaced by the SUF game's oracles. All other signature

Oracle SndA(ad)

00 Require $S_A \neq \bot$
01 $(PK, E, s, L, vfk, K, k.m, t) \leftarrow S_A$
02 $i \leftarrow (E^{\vdash}, es_A, 0, \mathtt{S})$ — $\mathbf{G}_{\geq 2}$
03 If $E^{\vdash} \neq E^{\dashv}$: $es_A \leftarrow 0$ — $\mathbf{G}_{\geq 2}$
04 $k^* \leftarrow \epsilon$; $C \leftarrow E^{\dashv}$
05 For $e' \leftarrow E^{\vdash}$ to $E^{\dashv}$:
06 $\quad (k, c) \leftarrow_{\$} \text{enc}(PK[e'])$
07 $\quad CK[c, i] \leftarrow k$ — $\mathbf{G}_{\geq 2}$
08 $\quad$ ~~$k^* \xleftarrow{\shortparallel} k$~~; $C \xleftarrow{\shortparallel} c$ — $\mathbf{G}_{<3}$ ~~$k^* \xleftarrow{\shortparallel} k$~~
09 $\quad I \xleftarrow{\shortparallel} i$; $i \leftarrow (e', 0, \Gamma[e', \mathtt{S}], \mathtt{S})$ — $\mathbf{G}_{\geq 2}$
10 $\tau \leftarrow \text{tag}(k.m, ad \,\|\, C)$
11 $C \xleftarrow{\shortparallel} \tau$; $t \xleftarrow{\shortparallel} \triangleright \,\|\, ad \,\|\, C$
12 If ims: $y \,\|\, k.m \,\|\, sk \leftarrow \text{H}(K, k^*, t)$ — $\mathbf{G}_{\geq 3}$ If ims
13 Else: — $\mathbf{G}_{\geq 3}$
14 $\quad y \,\|\, __ \leftarrow \text{G}(K, t, I, \mathtt{T})$ — $\mathbf{G}_{\geq 3}$
15 $\quad k.m \leftarrow_{\$} \mathcal{K}$; $sk \leftarrow_{\$} \mathcal{SK}$ — $\mathbf{G}_{\geq 3}$
16 $\quad$ ~~$\text{SetO}(K, t, I, k.m, sk)$~~ — $\mathbf{G}_{3-4}$
17 $k.o \,\|\, K \leftarrow y$
18 $pk \leftarrow \text{gen}_{\mathsf{K}}(sk)$
19 $PK[..., (E^{\dashv} - 1)] \leftarrow \bot$; $PK[E^{\dashv}] \leftarrow pk$
20 $E^{\vdash} \leftarrow E^{\dashv}$; $s \leftarrow s + 1$; $L[s] \leftarrow ad \,\|\, C$
21 $S_A \leftarrow (PK, E, s, L, vfk, K, k.m, t)$
22 If is_A:
23 $\quad adc_A[s_A] \leftarrow (ad, C)$
24 $\quad \text{EP}_A[s_A] \leftarrow e_A$
25 $\quad key_A[e_A, s_A] \leftarrow k.o$
26 $s_A \leftarrow s_A + 1$
27 $es_A \leftarrow es_A + 1$; $i \leftarrow (E^{\dashv}, es_A, 0, \mathtt{S})$ — $\mathbf{G}_{\geq 2}$
28 If not ims: — $\mathbf{G}_{\geq 2}$
29 $\quad KT_\star[i] \leftarrow (K, k.m, t)$ — $\mathbf{G}_{\geq 2}$
30 $\quad SK_\star[i] \leftarrow (sk, \bot)$ — $\mathbf{G}_{\geq 2}$
31 Return C

Oracle ExposeA

32 If is_A: $\text{XP}_A \xleftarrow{\cup} \{s_A\}$
33 Return S_A

Oracle ExposeB

34 $\text{TR}_B \xleftarrow{\cup} [\text{E}_B^{\vdash} .. \text{E}_B^{\dashv}] \times [r_B, ...]$
35 If is_B:
36 $\quad \text{XP}_B \xleftarrow{\cup} \{s_B\}$
37 $\quad \text{TR}_A \xleftarrow{\cup} [\text{E}_B^{\vdash} .. \text{E}_B^{\dashv}] \times [r_B, ...]$
38 $\quad \mathtt{U} \leftarrow \mathtt{S}$ — $\mathbf{G}_{\geq 2}$
39 $\quad XSK \xleftarrow{\cup} [E_\star^{\vdash}] \times [er_B, ...] \times \mathbb{N} \times [\mathtt{S}]$ — $\mathbf{G}_{\geq 2}$
40 $\quad XSK \xleftarrow{\cup} [E_\star^{\vdash} + 1 .. E_\star^{\dashv}] \times \mathbb{N}^2 \times [\mathtt{S}]$ — $\mathbf{G}_{\geq 2}$
41 Else: $\mathtt{U} \leftarrow \mathtt{R}$ — $\mathbf{G}_{\geq 2}$
42 $(K, k.m, t) \leftarrow KT_\star[E_\star^{\vdash}, er_B, 0, \mathtt{U}]$ — $\mathbf{G}_{\geq 2}$
43 $SK \leftarrow \text{GetSK}(\mathtt{U}, \mathtt{U})$ — $\mathbf{G}_{\geq 2}$
44 $S_B \leftarrow (SK, E_\star, r_B, L_{B\star}, sgk_\star, K, k.m, t)$ — $\mathbf{G}_{\geq 2}$
45 Return S_B

Oracle Reveal(u, j)
as in URKE (Fig. 5)

Oracle Challenge(u, j)
as in URKE (Fig. 5)

Oracle RcvB(ad, C)

46 Require $S_B \neq \bot$
47 If $is_B \wedge adc_A[r_B] \neq (ad, C)$:
48 $\quad is_B \leftarrow \mathtt{F}$
49 $\quad oos_B \leftarrow s_B$ — $\mathbf{G}_{\geq 1}$
50 $\quad \text{GetSK}(\mathtt{S}, \mathtt{R})$ — $\mathbf{G}_{\geq 2}$
51 $\quad KT_\star[E_\star^{\vdash}, er_B, 0, \mathtt{R}] \leftarrow KT_\star[E_\star^{\vdash}, er_B, 0, \mathtt{S}]$ — $\mathbf{G}_{\geq 2}$
52 $\quad$ If $r_B \in \text{XP}_A$:
53 $\quad\quad \text{TR}_B \xleftarrow{\cup} \mathbb{N} \times [r_B, ...]$
54 $\quad$ Else if $(E_\star^{\vdash}, er_B, 0, \mathtt{S}) \notin XSK$: — $\mathbf{G}_{\geq 5}$
55 $\quad\quad forge_A \leftarrow \mathtt{T}$ — $\mathbf{G}_{\geq 5}$
56 If $is_B \wedge \text{E}_B^{\vdash} \neq \text{EP}_A[r_B]$:
57 $\quad \text{E}_B^{\vdash} \leftarrow \text{EP}_A[r_B]$
58 $\quad E_\star^{\vdash} \leftarrow \text{EP}_A[r_B]$; $er_B \leftarrow 0$ — $\mathbf{G}_{\geq 2}$
59 If not is_B: — $\mathbf{G}_{\geq 2}$
60 $\quad i \leftarrow (E_\star^{\vdash}, er_B, 0, \mathtt{R})$ — $\mathbf{G}_{\geq 2}$
61 $\quad$ ~~$(SK, E, r, L, sgk, K, k.m, t) \leftarrow S_B$~~ — $\mathbf{G}_{<2}$
62 $\quad (K, k.m, t) \leftarrow KT_\star[i]$ — $\mathbf{G}_{\geq 2}$
63 $\quad SK \leftarrow \text{GetSK}(\mathtt{R}, \mathtt{R})$ — $\mathbf{G}_{\geq 2}$
64 $\quad t^* \leftarrow ad \,\|\, C$; $C \,\|\, \tau \leftarrow C$
65 $\quad$ Require $\text{vfy}_{\mathsf{M}}(k.m, ad \,\|\, C, \tau)$
66 $\quad$ If $forge_A$: Abort — $\mathbf{G}_{\geq 5.4}$
67 $\quad k^* \leftarrow \epsilon$; $e \,\|\, C \leftarrow C$
68 $\quad$ Require $E_\star^{\vdash} \leq e \leq E_\star^{\dashv}$ — $\mathbf{G}_{\geq 2} \star$
69 $\quad$ If $E_\star^{\vdash} \neq e$: $er_B \leftarrow 0$ — $\mathbf{G}_{\geq 2}$
70 $\quad t \xleftarrow{\shortparallel} L_{B\star}[E_\star^{\vdash} + 1] \,\|\, \ldots \,\|\, L[e]$ — $\mathbf{G}_{\geq 2} \star$
71 $\quad L_{B\star}[..., e] \leftarrow \bot$ — $\mathbf{G}_{\geq 2} \star$
72 $\quad$ For $e' \leftarrow E_\star^{\vdash}$ to e: — $\mathbf{G}_{\geq 2} \star$
73 $\quad\quad c \,\|\, C \leftarrow C$
74 $\quad\quad$ ~~$k \leftarrow \text{dec}(SK[e'], c)$~~ — $\mathbf{G}_{<3}$
75 $\quad\quad$ ~~Require $k \neq \bot$~~ — $\mathbf{G}_{<3}$
76 $\quad\quad$ Require $\text{dec}(SK_\star[i], c) \neq \bot$ — $\mathbf{G}_{\geq 3}$
77 $\quad\quad$ ~~$k^* \xleftarrow{\shortparallel} k$~~ — $\mathbf{G}_{<3}$
78 $\quad\quad I \xleftarrow{\shortparallel} i$; $i \leftarrow (e', 0, \Gamma[e', \mathtt{R}], \mathtt{R})$ — $\mathbf{G}_{\geq 2}$
79 $\quad t \xleftarrow{\shortparallel} \triangleright \,\|\, t^*$
80 $\quad$ ~~$k.o \,\|\, K \,\|\, k.m \,\|\, sk \leftarrow \text{H}(K, k^*, t)$~~ — $\mathbf{G}_{<3}$
81 $\quad k.o \,\|\, K \,\|\, k.m \,\|\, sk \leftarrow \text{G}(K, t, I, \mathtt{F})$ — $\mathbf{G}_{\geq 3}$
82 $\quad$ ~~$SK[..., (e-1)] \leftarrow \bot$; $SK[e] \leftarrow sk$~~ — $\mathbf{G}_{<2}$
83 $\quad$ ~~For $e' \leftarrow e + 1$ to $E^{\dashv}$:~~ — $\mathbf{G}_{<2}$
84 $\quad\quad$ ~~$SK[e'] \leftarrow \text{up}(SK[e'], t^*)$~~ — $\mathbf{G}_{<2}$
85 $\quad E^{\vdash} \leftarrow e$; ~~$r \leftarrow r + 1$~~ — $\mathbf{G}_{<2}$ ~~r~~
86 $\quad$ ~~$S_B \leftarrow (SK, E, r, L, sgk, K, k.m, t)$~~ — $\mathbf{G}_{<2}$
87 If $S_B = \bot$: Return $\bot$
88 If is_B: $k.o \leftarrow \diamond$
89 $key_B[\text{E}_B^{\vdash}, r_B] \leftarrow k.o$
90 $r_B \leftarrow r_B + 1$
91 $er_B \leftarrow er_B + 1$ — $\mathbf{G}_{\geq 2}$
92 If not is_B: — $\mathbf{G}_{\geq 2}$
93 $\quad i \leftarrow (E_\star^{\vdash}, er_B, 0, \mathtt{R})$ — $\mathbf{G}_{\geq 2}$
94 $\quad KT_\star[i] \leftarrow (K, k.m, t)$ — $\mathbf{G}_{\geq 2}$
95 $\quad SK_\star[i] \leftarrow (sk, \bot)$ — $\mathbf{G}_{\geq 2}$
96 $\quad P[r_B] \leftarrow t^*$ — $\mathbf{G}_{\geq 2}$
97 $\quad \text{GetSK}(\mathtt{R}, \mathtt{R})$ — $\mathbf{G}_{\geq 3}$
98 Return

Fig. 18: Proof of SRKE considering communication from A to B and the remaining oracles of KIND_{R}.

operations are simulated by the reduction directly. The signer key sgk for which the adversary provides a forgery—to distinguish within the first game hop—was not exposed because otherwise either the receive counter r_A is in the set of exposed counters XP_B or the A was out of sync earlier such that $oos_B \leq r_A$ holds and therefore $forge_B$ would not be set (see lines 53–56).

Let q_F be the number of RcvA queries by the adversary, then the advantage of an adversary $\mathcal{D}$ to distinguish between the original game and G_1 can be upper bounded by:

$$\mathrm{Adv}_{\mathsf{R}}^{\mathrm{kind},G_1}(\mathcal{D}) \leq q_R \mathrm{Adv}_{\mathsf{S}}^{\mathrm{suf}}(\mathcal{C}_{\mathsf{S}})$$

Game 2 – Synchronous simulation of *B* Using the correctness of our construction, we can again simulate the receiving of unmodified ciphertexts by B with the simulation of A. Therefore we remove B's state in game G_2 and trace the respective variables with global arrays and counters.

To fully understand this game hop, we divide its description on the basis of the introduced and used variables. At first the counters ($E^{\vdash}_{\star}, E^{\dashv}_{\star}, es_A, er_B$)—and thereby the indexing scheme of the introduced arrays—are explained. Then the usage of the arrays, tracing secret keys ($SK_{\star}$) and symmetric keys ($KT_{\star}$), are presented. The array SK in B's state is not simulated directly with one array but instead compiled from the array of *all* secret keys $SK_{\star}$ as soon as it is needed. Due to the complexity of the compilation, the simulation of array SK (under usage of arrays $SK_{\star}, \Gamma$ and procedure GetSK) is explained separately. Conclusively this game introduces an array (XSK) and a flag (ims) to comprehensively indicate which secret keys are exposed and how this influences an impersonation towards the sender.

Defining global variables and counters The interval of epochs E, the array of sent ciphertext-associated-data pairs L_B, and the signature signer state sgk in B's state are declared as global variables to simulate exposures without maintaining B's state as a whole. To highlight this modification, the variable symbols are indexed with the $\star$ symbol ($E_{\star}, L_{B\star}, sgk_{\star}$). The variable r in B's state is replaced by the already existing variable r_B. These replacements are denoted by a "$\star$" at the right margin of the corresponding lines. We can use r_B instead of r because both variables are incremented equally. Additionally the counters es_A, er_B are introduced for counting the send and receive operations within an epoch. Hence both counters are reset for every new epoch.

Variable of A	Explanation	Corresponding Variable of B	Explanation
$E^{\vdash}$ in S_A	Epoch to which A sent last ciphertext	$E^{\vdash}_{\star}$	Epoch for which B received last ciphertext
$E^{\dashv}$ in S_A	Newest epoch that was proposed to A	$E^{\dashv}_{\star}$	Newest epoch that B proposed to A
es_A	Number of sent ciphertexts to the current epoch $E^{\vdash}$	er_B	Number of received ciphertexts for the oldest cached epoch $E^{\vdash}_{\star}$
s_A	Total number of sent ciphertexts	r_B	Total number of received ciphertexts

Table 1: Variables for indexing arrays in the simulation of games $G_{\geq 2}$. The following conditions hold in presence of a passive adversary: $E^{\vdash}_{\star} \leq E^{\vdash} \leq E^{\dashv} \leq E^{\dashv}_{\star}$, $E^{\vdash} = E^{\vdash}_{\star} \Rightarrow es_A \geq er_B$.

Indexing arrays The index of the arrays $SK_{\star}, KT_{\star}$, and the set XSK consists of four parameters: 1) the epoch counter, 2) *within this epoch*, the number of send or receive operations (specified by es_A or er_B), 3) the number of updates (also called *level*) of the secret key via the kuKEM

up algorithm (set to 0 for array $KT_\star$), and 4) a tag that indicates whether the entry was used by B's simulation. In comparison to the former two index parameters of this scheme, the index scheme of the key array *key* consists of the epoch counter and the number of *total* send or receive operations (specified by s_A or r_B). It is easy to see that there exists a function for transferring the indexing schemes among each other when disregarding the level and tag parameter. While the indexing via total operation counters simplifies the phrasing of the adversary's winning conditions, indexing via counters within epochs is more natural for the simulation. Table 1 provides an intuitive explanation of the index parameters for the simulation.

The indexing scheme for the arrays is constructed to model the usage of secret keys in SRKE—thereby it can also be used to index the symmetric key array $KT_\star$. The index for a freshly generated secret key for a new epoch is obtained by increasing the epoch counter and resetting all remaining index parameters (see Figure 17 lines 36, 46, 47 and Figure 18 lines 03, 58, 69). When deriving a secret key from the random oracle, the counter within the epoch is increased and the level parameter to be explained hereafter is reset (see Figure 18 lines 27, 91, 93). Finally secret keys can be updated. Therefore the index includes the level parameter that is increased with every update operation (see Figure 17 lines 81–84). In addition to the epoch number, the counter within the epoch, and the level of a secret key, a tag $\mathtt{U} \in \{\mathtt{S}, \mathtt{R}\}$ indexes $SK_\star, KT_\star, XSK$. As in the URKE proof, $\mathtt{S}$ denotes values that are used for simulation of A. Entries marked with $\mathtt{R}$ are used by the explicit simulation of B—consequently these entries are only used from the moment of receiving the ciphertext that causes B to become out of sync onwards (see Figure 18 lines 50–51).

Usage and computation of arrays As in the URKE proof, the arrays $SK_\star, KT_\star$ store the secret keys and tuples of chaining key, MAC key, and transcript after each sending and receiving operation respectively. Writing and reading the array $KT_\star$ is conducted straight forward as it is in the URKE proof.

Freshly generated secret keys and secret keys as output of the random oracle are directly stored in $SK_\star$. To reassemble the construction's array SK at exposures or for simulating the receiving out of sync, we use the array of secret keys $SK_\star$ in procedure GetSK (see Figure 17). As for the original array SK, this procedure sets the current epoch's secret key in the entry for the current epoch, and all subsequent entries are filled with the first secret key within the respective epoch. To derive the correct updated secret keys for cached epochs in SK (epoch index greater than $E_\star^{\vdash}$), updates in sync use the L_A array of A's state to obtain the correct ciphertexts and associated data as update parameters. Out of sync the array P is first filled with the entries of L_A and then further filled by the simulation of B (see Figure 17 line 73, Figure 18 line 96). To correctly simulate the updates, already defined secret keys are taken and further updates are based on the latest existing updated secret key[19] (see Figure 17 lines 78 ff.). To adopt secret keys marked with $\mathtt{S}$ when B becomes out of sync, GetSK copies the most often updated entry (highest level) of each secret key and potentially further updates these keys accordingly. To track when a secret key was initially generated, the counter s is stored for each secret key sent by B (see Figure 17 lines 46,47). The number of necessary updates can then be derived by the difference between the number of received ciphertexts by Bob and the value of the send counter attached to the secret key in $SK_\star$.

For the next game hop, the index of each key pair, used for the next encapsulation (when A sends) or decapsulation (when B receives), needs to be known (see Figure 18 lines 09, 78). In array I all key pair indexes i are concatenated for which a tuple (c, k) is input to the random oracle query within the same send or receive operation. As described above, the third index

[19] Please note that the update algorithm is a one way function and regarding the reduction, the simulation has to comply with the KUOW game's update oracle which can only be called sequentially.

parameter indicates the level of the respective secret key (number of updates). The array Γ stores the number of updates for each secret key. While for the simulation of A Γ is filled at receiving a new public key (see Figure 17 line 67), the simulation of B can compute Γ along the computation of the secret key in GetSK (see Figure 17 line 86). Note that only the first secret key within an epoch needs to be tracked because later keys will not be updated in our SRKE construction.

Exposure and impersonation For tracing exposed secret keys, we introduce the set XSK that is filled with the indexes of A's exposed secret keys (secret keys that belong to A's public keys). Secret keys are marked to be exposed if they are stored in the exposed array SK. Additionally all subsequent secret keys that are directly derived from these secret keys (i.e., generated within the same epoch) are marked to be exposed, since they can be obtained from the random oracle with the exposed secret keys and the chaining key that is also exposed, or from the publicly known update parameter (ciphertext and associated data). If A and B were out of sync, the key update via kuKEM update or random oracle was computed with different values ($ad \parallel C$ of A and B were per definition different). Hence none of A's public keys' secret keys are exposed by B out of sync.

If B is impersonated towards A—i.e., the in-sync state of B for computing the ciphertext to A was exposed and the ciphertext-associated-data pair received by A was not sent by B—, then ims is set (see Figure 17 line 55). Thereby all future computations by A are traceable by the adversary. After the flag ims is set, we stop to store secret keys from the random oracle outputs in A's simulation. Secret keys generated by A after the impersonation are never used by B (because from this moment on the parties are out of sync) and A only needs the respective public keys anyway.

As in URKE, the array CK is introduced in game G_2 to track the triple (c, k, i) after each encapsulation.

Apart from preparations for subsequent games (e.g., setting up indexes, XSK, and ims), this game hop substitutes B's state by introducing global variables and counters. Thereby B is only explicitly simulated for sending and for computations out of sync. All remaining computations are either conducted by A's simulation or by the helper procedure GetSK.

Game 3 – Internal access to random oracle Game G_3 is also directly adapted from the URKE proof: the simulation requests the random oracle without providing the input key k^*. There are only two minor differences that do not affect the underlying principle. Firstly, the input to the random oracle is a vector of keys $k^* = k_1 \parallel .. \parallel k_n$. The input transcript t is accordingly also composed of the concatenation of ciphertext vectors $c_1 \parallel .. \parallel c_n$ and the index provided to the internal procedure G consists of multiple secret key indexes $I = i_1 \parallel .. \parallel i_n$. Secondly, not all internal random oracle requests of A are issued via the internal procedure G.

In case an impersonation of B towards A was performed by the adversary such that ims is set, the simulation of A strictly follows the construction description and no (kuKEM) challenges are embedded by the simulation. This is valid since no future established session key will be challengeable. Consequently the respective random oracle requests do not need to be issued without the knowledge of the kuKEM keys in k^* and, as such, can be computed by using H instead of G.

As described in the previous game, the cumulative index I contains a vector of all indexes to which the send or receive operations encapsulated or decapsulated right before the random oracle invocation respectively. Thereby the tuple $(C, I, k^*) = (c_1 \parallel .. \parallel c_n, i_1 \parallel .. \parallel i_n, k_1 \parallel .. \parallel k_n)$ is processed in SRKE instead of a tuple consisting of one value each in URKE. The validation of the inputs to the random oracle for finding existing equal entries now works accordingly:

Oracle $\mathrm{H}(K, k^*, t)$ $\mathbf{G}_{\geq 0}$

00 $k_1 \parallel .. \parallel k_n \leftarrow k^*$
01 $t' \parallel ad \parallel e \parallel c_1 \parallel .. \parallel c_n \leftarrow t$
02 $(k.o, K^*, k.m) \leftarrow_\$ \mathcal{K}^3$; $sk \leftarrow_\$ \mathcal{SK}$
03 $I \leftarrow \epsilon$; $sen \leftarrow \mathtt{F}$
04 If $\exists I : R[K, t, k, I] \neq \bot$:
05 $\quad (k.o, K^*, k.m, sk, sen) \leftarrow R[K, t, k, I]$
06 Else if $\exists I = i_1 \parallel .. \parallel i_n : R[K, t, \epsilon, I] \neq \bot$ $\mathbf{G}_{\geq 3}$
$\quad \wedge \forall j, 1 \leq j \leq n : i_j \in \mathbb{N}^3 \times [\mathtt{R}]$ $\mathbf{G}_{\geq 3}$
$\quad \wedge \mathrm{dec}(sk_j, c_j) = k_j, (sk_j, s) \leftarrow SK_\star[i_j]$: $\mathbf{G}_{\geq 3}$
07 $\quad (k.o, K^*, k.m, sk, sen) \leftarrow R[K, t, \epsilon, I]$ $\mathbf{G}_{\geq 3}$
08 Else if $\exists I = i_1 \parallel .. \parallel i_n : R[K, t, \epsilon, I] \neq \bot$ $\mathbf{G}_{\geq 3}$
$\quad \wedge \forall j, 1 \leq j \leq n : i_j \in \mathbb{N}^3 \times [\mathtt{S}]$
$\quad \wedge CK[c_j, i_j] = k_j$: $\mathbf{G}_{\geq 3}$
09 $\quad (k.o, K^*, k.m, sk, sen) \leftarrow R[K, t, \epsilon, I]$ $\mathbf{G}_{\geq 3}$
10 $\quad (e_j, __) \leftarrow i_j$ $\mathbf{G}_{\geq 5}$
11 $\quad$ If $e_n < oos_A \wedge i_n \notin XSK$: Abort $\mathbf{G}_{\geq 5.1}$
12 $\quad$ If $e_1 < oos_A \wedge e_n \geq oos_A$: Abort $\mathbf{G}_{\geq 5.2}$
13 $\quad$ If $e_1 \geq oos_A$: Abort $\mathbf{G}_{\geq 5.3}$
14 $\quad (sk, _) \leftarrow SK_\star[i_n + 1]$ $\mathbf{G}_{\geq 4}$
15 $\quad (_, k.m, _) \leftarrow KT_\star[i_n + 1]$ $\mathbf{G}_{\geq 4}$
16 $R[K, t, k, I] \leftarrow (k.o, K^*, k.m, sk, sen)$
17 Return $k.o \parallel K^* \parallel k.m \parallel sk$

Proc $\mathrm{G}(K, t, I, sen)$ $\mathbf{G}_{\geq 3}$

18 $t' \parallel ad \parallel e \parallel c_1 \parallel .. \parallel c_n \leftarrow t$
19 $i_1 \parallel .. \parallel i_n \leftarrow I$
20 $k^* \leftarrow \epsilon$
21 If $sen \wedge \exists k^* : R[K, t, k^*, \epsilon] \neq \bot$:
22 $\quad$ Abort
23 Else if $\neg sen \wedge \exists k^* = k_1 \parallel .. \parallel k_n$:
$\quad R[K, t, k^*, \epsilon] \neq \bot \wedge \forall j, 1 \leq j \leq n$:
$\quad \mathrm{dec}(sk_j, c_j) = k_j, (sk_j, s) \leftarrow SK_\star[i_j]$:
24 $\quad (k.o, K^*, k.m, sk, sen') \leftarrow R[K, t, k^*, \epsilon]$
25 Else:
26 $\quad (k.o, K^*) \leftarrow_\$ \mathcal{K}^2$; $k.m \leftarrow \epsilon$; $sk \leftarrow \epsilon$
27 $\quad$ If $\neg sen$: $k.m \leftarrow_\$ \mathcal{K}$; $sk \leftarrow_\$ \mathcal{SK}$
28 $\quad R[K, t, k^*, I] \leftarrow (k.o, K^*, k.m, sk, sen)$
29 Return $k.o \parallel K^* \parallel k.m \parallel sk$

Proc $\mathrm{SetO}(K, t, I, k.m, sk)$ $\mathbf{G}_{\geq 3}$

30 $(k.o, K^*, k.m', sk', sen) \leftarrow R[K, t, \epsilon, I]$
31 $R[K, t, \epsilon, I] \leftarrow (k.o, K^*, k.m, sk, sen)$
32 Return

Fig. 19: Random oracle description for proof of SRKE.

If there exist an entry created by an external query for the internal query by B's simulation with the same transcript t and each ciphertexts c_j at the end of t can be decapsulated with the respective secret key with index i_j in I to the key k_j of the external entry's input k^*, then the queries were equal and the output of the internal query is copied from the external one. For queries of A's simulation that collide with an entry made externally, the game aborts as in the URKE proof.

If the adversary externally requests the random oracle, the validation is again split (in order to comply with the KUOW game's oracles). The vector of ciphertexts in the transcript and the vector of keys in the input key k^* are validated with respect to the existing internally made entries and their vectors of secret key indexes. If there exists an entry made internally (marked with ϵ at the index position of k^*) such that 1) either each secret key i_j from the cumulative index I of this entry can be used to decapsulate the respective ciphertext to the respective key in k^* (see line 06) or 2) all tuples (c_j, i_j, k_j) were stored in the array CK (see line 08), then the external query and internal entry are equal. In this case the output of the external random oracle query is copied from this internal entry. To ensure that all secret keys are updated according to the construction of the receive algorithm, such that validation in the random oracle is correct, GetSK is invoked a second time at the end of B's receive operation.

Splitting the validation is done to prepare the reduction to the KUOW game. The decryption for entries made by B's simulation can be conducted by using the Check oracle. The Solve oracle is modeled by array CK. As in URKE, the reduction will fill CK for exposed secret keys explicitly because the Solve oracle can only be used for unexposed secret keys. In addition to that, the reduction will fill CK also directly for public keys for which the simulation never has access to the respective secret keys. This is the case for all public keys received by A after becoming out of sync without an impersonation (*ims* was not set). The adversary can instead impersonate B *delayed*: if first A is impersonated towards B and then B is exposed, B's signing key is still valid such that the adversary can use it to send valid own ciphertexts to A. Thereby the adversary

can send public keys to A for which the adversary (but not the simulation) knows the secret keys. In this case, CK is filled instantly after the encapsulation.

As in the URKE proof, the output MAC key and secret key are sampled independently of the random oracle for internal queries by A.

The probability of the abortion for predefined random oracle entries colliding with internal queries by A can again be bounded by the birthday bound. Let q_F be the number of random oracle requests and q_S be the number of queries to the SndA oracle then distinguishing between games G_2 and G_3 can be bounded as:

$$\text{Adv}^{G_2,G_3}(\mathcal{D}) \leq \frac{q_F q_S}{|\mathcal{K}_{\mathsf{K}}|}$$

Note that this is an upper bound for the advantage. In case the ciphertext consists of a vector of kuKEM ciphertexts—and thereby the key, as input to the random oracle, is a vector of kuKEM keys—the probability for a collision is lower (because the size of the key space in which a collision occurs is larger respectively).

Requesting the random oracle without providing the cumulated key k^* allows to disregard the decapsulation and hence the usage of secret keys for the simulation of B. For comparing the decapsulation output with the $\perp$ symbol, the KUOW's Check oracle will be used such that the secret key of B does not need to be used explicitly. To emphasize this, the input to the decapsulation in line 76 is the respective element from the global secret key array $SK_\star$, disregarding its second value s.

Game 4 – Random oracle with lazy sampling In game G_4 we stop to set the output MAC keys and secret keys for random oracle queries by A's simulation if ims is not set. Thereby the explicit usage of these keys is shifted to the validation in the random oracle, providing the output for external random oracle requests, and to exposures of B. We will show that the former and letter use cases can be simulated by the reduction to the KUOW or SUF game respectively. The second use case will either not occur without breaking the underlying hardness assumption, or it can also be simulated by using the oracles of the respective games in the reduction.

Please note that we denote incrementing of index i such that the counter es within the epoch is incremented and the level parameter is reset: $i + 1 = (e, es, l, \mathtt{U}) + 1 = (e, es + 1, 0, \mathtt{U})$.

Game 5 – Abortions Our abortion conditions for SRKE split the key establishment in four cases: 1) keys established by A in sync, 2) the first key established by A out of sync, 3) all remaining keys established by A out of sync, and 4) keys established out of sync by B.

We first want to provide an intuition for the regarded cases: At the beginning of the communication, the parties are in sync. Thereby only an abortion according to case 1) can occur. Manipulating a ciphertext from A to B now introduces cases 4) and, with a delay until the next ciphertext in the opposite direction, also cases 2) and 3). It is necessary to understand that for provoking an abortion in games $G_{5.2}$ and $G_{5.3}$, a ciphertext to A can only be manipulated after B was already out of sync. The reason for this lies in the abortion of game G_1. If G_1 does not abort, for ciphertexts to A either an impersonation occurred, or an invalid ciphertext was received by A, or the ciphertext of B was correctly delivered to A. The former case prevents the simulation from embedding challenges to the random oracle, the latter case does not drift A's state out of sync, and the second case provokes the state of A to be erased because the signature verification will fail.

We split the first three abortion rules for didactic reasons. Therefore the reduction loses the factor 3 with respect to the advantage in winning the KUOW game. It will become obvious that all three abortions can be summarized to one condition that can be reduced to one instance

of the KUOW game such that the reduction would be tight with respect to the employed assumptions.

In the subsequent paragraphs it is called *explicit use* of a value, if the simulation provides this value to the adversary. If the value was only used for a computation—which can be simulated by the underlying game's oracles in the reduction—it was *not explicitly used*.

Game 5.1 – Keys established by A in sync Aborting in game $G_{5.1}$ depends on two conditions for the external random oracle query for which an internally defined entry of A exists: A was in sync when requesting the random oracle for the entry that is externally requested and the secret key sk_n with index i_n, that can be used to decapsulate the last key k_n from the last c_n in t, as input to the random oracle, was not exposed. The public key to the initial secret key of the same epoch as sk_n—secret key with the same value in the epoch index parameter but send and level parameter set to 0—was originally sent by B in sync and correctly delivered to A because otherwise A would have been out of sync already (which would violate the first condition). By condition two, the secret key sk_n was not exposed. This condition can be fulfilled in two ways: either B derived the same secret key sk_n as A, or B became out of sync before an exposure and thereby derived different secret keys in this epoch. In the first case, the condition simply holds because otherwise the secret key would be marked to be exposed. In the second case, the secret keys of B are differently updated or freshly generated in the random oracle such that the exposure of B's secret keys has no influence on A's established keys. Thereby the reduction can make use of the KUOW game's oracles Up_R and Gen—and if needed Expose—for simulating B's updates and outputs of the random oracle for the differently derived secret keys.

Apart from obtaining the secret key via an exposure of B, the adversary can obtain it from the output of the random oracle if $n = 1$ holds (i.e., $sk_n = sk_1$ was derived from the random oracle). This, however, can be excluded for the following reason: for this previous internal random oracle entry, outputting secret key with index i_1, there exists a secret key with index i'_n. Both indexes have the same epoch parameters since i'_n was the last secret key index of the random oracle entry outputting secret key with index i_1. As a consequence, the same conditions regarding an abortion in this game for an external query of the random oracle hold. If the adversary requested the random oracle for this entry, outputting the secret key with index i_1, the game would have been aborted before.

Since with providing the correct k_n for c_n and secret key with index i_n, the adversary solves the challenge of the KUOW game, and, as described above, the secret key with i_n was not explicitly used by the simulation, the advantage in distinguishing between G_5 and $G_{5.1}$ can be bounded by the advantage of winning the KUOW game:

$$\mathrm{Adv}^{G_5,G_{5.1}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B})$$

Game 5.2 – First key established by A out of sync Game $G_{5.2}$ aborts if the adversary queries the random oracle for the entry that was created by the internal request of A directly after becoming out of sync (with G). Thereby at most three types of public keys are used by A for the encapsulations before the random oracle request: 1) The public key derived from the last random oracle query's output secret key sk_1, 2) public keys that were sent by B in sync and correctly delivered to A, and 3) public keys received by A that caused her to become out of sync or that were received thereafter. While there exist at least one public key of type 1 and one of type 3, it is not necessary that A also encapsulated to a new received public key that was sent in sync by B.

As described earlier, A only becomes out of sync because B drifted out of sync before. Letting A drift out of sync solely would cause the *ims* flag to be set, or G_1 to abort, or the state of A being erased. According to this, B must not have been exposed before drifting out

of sync because otherwise the ciphertext, that causes A to become out of sync, is considered as an impersonation such that *ims* is set. As a consequence, the newest public key used by A before querying the random oracle, that results from a public key sent by B in sync, was not exposed[20]. This is either the public key to the secret key sk_1 with index i_1 and hence of type 1, or the public key to a secret key sk_j with index $i_j, 1 < j < n$ of type 2. Either way the respective secret key (sk_1 or sk_j) was not exposed before B became out of sync and was updated or derived differently from A when B became out of sync afterwards. If this public key is of type 2, its secret key sk_j can only be obtained by an exposure of B because it was freshly generated. Secret key sk_1 of public key of type 1 can also be obtained by the random oracle. The random oracle query that outputs this secret key sk_1 is associated to the previous secret key sk'_n in the same epoch since i'_n was the last secret key index for this previous random oracle entry. If no public key of type 2 exists, this previous secret sk'_n must not have been exposed because otherwise the whole epoch would be marked as exposed which would cause the ciphertext drifting A out of sync to be considered as an impersonation. As such, the random oracle entry outputting the secret key sk_1 fulfills the conditions for an abortion of game $G_{5.1}$ and can thereby be excluded for game $G_{5.2}$.

Conclusively, if there exists a public key of type 2 with secret key index i_j, then this secret key was not explicitly used by the simulation and thereby the adversary's external random oracle with input k_j and c_j can be used to solve the KUOW challenge. Similarly, if only two kuKEM keys are fed into the internal random oracle query of A, then the secret key with index i_1 was not explicitly used by the simulation and (k_1, c_1) solve the KUOW challenge. Therefore distinguishing between $G_{5.1}$ and $G_{5.2}$ can be reduced to the KUOW security of the kuKEM:

$$\mathrm{Adv}^{G_{5.1},G_{5.2}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B})$$

Game 5.3 – Further keys established by A out of sync In game $G_{5.3}$ for an internally created random oracle entry that causes an abortion, A and B were out of sync when A requested the random oracle for this entry. The first public key, used for the encapsulation before the request, is the result of A's distinct previous random oracle request. The previous random oracle request was distinct because by the condition $(e_1 \geq oos_A)$ A requested the random oracle out of sync at least once before since i_1 has always the same epoch as i'_n from the last random oracle request. Since A and B are out of sync—and thereby their random oracle requests are independent—, the secret key to each first public key for the encapsulation can only be obtained by the random oracle. Requesting the random oracle for one of these entries externally would cause the game to abort—either according to game $G_{5.2}$ for the first random oracle query out of sync, or according to this game for all subsequent queries of A. Therefore by the conditions of the abortion, the secret key to the first public key of each sending operation of A out of sync was not explicitly used and hence not known to the adversary such that the tuple (k_1, c_1) solves the KUOW challenge of the kuKEM. Please note that impersonations of B towards A are implicitly excluded since A's simulation invokes G only if *ims* was not set. Hence we can bound the probability of an abortion in $G_{5.3}$ by the advantage of winning the KUOW game[21]:

$$\mathrm{Adv}^{G_{5.2},G_{5.3}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B})$$

[20] Results from means that the public key sent in sync by B and received in sync by A was possibly updated and possibly used to feed the random oracle to obtain the public key that was actually used by A in the considered abortion—hence both keys (*pk* used by A and *pk* sent by B) are in the same epoch.

[21] Please note that the abortion event of this game hop can be reduced to the OW security of the KEM solely because the first public key in A's state is only freshly generated as output of the random oracle but never updated.

Game 5.4 – Keys established by B out of sync The abortion rule of game $G_{5.4}$ of SRKE can be reduced to the one of game $G_{5.2}$ of URKE. For aborting the game, A must not be exposed for the last state in sync between A and B, and B's oldest cached secret key at receiving the ciphertext drifting him out of sync must not have been exposed. These conditions imply that an external random oracle query to the entry, that defines the last common state values (e.g., $k.m$), causes an abortion of $G_{5.1}$. Consequently the MAC key was neither exposed by A, nor by B, nor provided to the adversary via an external random oracle query. Manipulating a ciphertext and delivering it to B without erasing B's state implies that the adversary forged the MAC tag. Consequently the abortion of $G_{5.4}$ is equivalent to winning the SUF game with respect to the MAC:

$$\mathrm{Adv}^{G_{5.3},G_{5.4}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{C}_{\mathsf{M}})$$

To conclude the proof, it is necessary to show that the abortions reduce all random oracle queries by the adversary that output challengeable keys to one of the employed hardness assumptions. Equivalently we show that the game aborts for the queries that do not output traceable keys.

According to the transformability between the index scheme of the $\mathrm{KIND}_{\mathsf{R}}$ game and the index scheme for the simulation, the union of A's traceable keys TR_A and the union of exposed secret key indexes XSK at an exposure of B are equivalent (see Figure 18 lines 37, 39, 40). Since $G_{5.1}$ aborts for all keys established by A in sync for which i_n is not exposed, no challengeable key of A in sync can be obtained from the random oracle. Please note that if $i_n \in XSK$ holds, it is implied that $\forall 1 \leq j \leq n : i_j \in XSK$ because all secret keys are stored in the same state of B.

When drifting out of sync, the $\mathrm{KIND}_{\mathsf{R}}$ game does not increase epochs of the respective party anymore. Consequently impersonations cause all future keys to be traceable. Equivalently the proof does not embed challenges into the random oracle if B was impersonated towards A and does not consider a valid MAC for a manipulated ciphertext to B as a forgery if A was exposed right before. A forgery is not regarded as such if the oldest epoch in B's state was exposed, either—which is equivalent to the absence of epoch increasing when drifting out of sync.

Since the abortions of games $G_{5.1} - G_{5.3}$ cover all possible internal random oracle queries of A and $G_{5.4}$ excludes the establishment of challengeable keys by B, the adversary cannot derive a challengeable key without letting the game abort. Hence the advantage in winning the game $G_{5.4}$ is 0.

$$\mathrm{Adv}^{G_{5.4}}(\mathcal{A}) = 0$$

Proof result Summing up the loss due to the game hops described above, provides us with the advantage of an adversary in winning the SRKE $\mathrm{KIND}_{\mathsf{R}}$ game depending on the advantages of adversaries $\mathcal{B}$, $\mathcal{C}_{\mathsf{S}}$, and $\mathcal{C}_{\mathsf{M}}$:

$$\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq 3\mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B}) + q_R \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{C}_{\mathsf{S}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{C}_{\mathsf{M}}) + \frac{q_S q_F}{|\mathcal{K}_{\mathsf{K}}|}$$

D Proof of BRKE

We give proof for the security of our generic BRKE construction in Figure 13. This proof is described below and depicted in Figure 20. Our proof first reduces the creation of signature forgeries to the security of the used signature scheme. Then we show that the adversary cannot win the BRKE $\mathrm{KIND}_{\mathsf{BR}}$ game without breaking the underlying SRKE scheme's security.

Game 1 – Excluding signature forgeries Similarly to the SRKE proof, game G_1 aborts on signature forgeries that drift the parties' states out of sync. The probability of this event can be reduced to the advantage in winning the SUF game against the one-time signature scheme.

The major difference between game G_1 in SRKE and BRKE lies within the abortion conditions: In SRKE, impersonations cannot entail forgeries by definition because the respective signing key is leaked to the adversary during the exposure of the impersonated party. The signing key in BRKE can never be exposed to the adversary because it is only used temporarily during the computations of the send algorithm. As such, in BRKE also a ciphertext received as impersonation can contain a signature forgery.

If the adversary injects a manipulated ciphertext that causes the receiving party to drift out of sync, that contains the original verification key, and that is valid with respect to the signature verification, game G_1 aborts. If the respective sender was out of sync before (i.e., if the adversary already injected a manipulated ciphertext in the other communication direction before), then this ciphertext does not *cause* the parties to drift out of sync, but only propagates the asynchrony. This latter case is not considered in the detection of signature forgeries because the underlying SRKE schemes already handle the receipt of manipulated SRKE ciphertexts and the propagation of out-of-sync states.

An adversary distinguishing between the original $\text{KIND}_{\mathsf{BR}}$ game and game G_1 can be used to win the SUF game against the signature scheme. The reduction replaces the singing and verification algorithms with the SUF game's oracles for the ciphertext that entails the first forgery. Thereby the reduction needs to guess, which ciphertext is the first one that contains the forgery, such that the advantage in distinguishing between $\text{KIND}_{\mathsf{BR}}$ and G_1 can be bounded by:

$$\text{Adv}_{\mathsf{BR}}^{\text{kind},G_1}(\mathcal{D}) \leq q_R \text{Adv}_{\mathsf{S}}^{\text{suf}}(\mathcal{C})$$

where q_R is the number of invocations of the Rcv oracle.

After game G_1 the following invariant holds: If a manipulated ciphertext is received by one of the parties, then either this party's state is erased (due to an invalid signature), or both SRKE receiver states are affected by the manipulation (due to an impersonation). The latter case is true because the adversary must use an own signing key pair for the ciphertext's signature of which the verification key is used as associated data for both SRKE receive algorithms.

Game 2,3 – Key indistinguishability of SRKE In games G_2, G_3 we use the key indistinguishability of the underlying SRKE schemes to show that the adversary, breaking BRKE, can be used to break one of the employed SRKE instances.

We replace the BRKE challenge keys by random elements from the key space. In game G_2 we do this for keys established from A to B, then in game G_3 for the counter direction respectively. By matching the winning conditions of an adversary in game G_2 (and G_3) with the winning conditions in the SRKE $\text{KIND}_{\mathsf{SR}}$ game, one can see that these conditions are identical. Please note that, due to the first game hop to game G_1, manipulations of ciphertexts always affect both SRKE ciphertexts.

As a result, the advantage of an adversary, distinguishing between G_1 and G_2 and between G_2 and G_3 respectively, can be bounded by:

$$\text{Adv}_{\mathsf{BR}}^{G_1,G_2}(\mathcal{D}) \leq \text{Adv}_{\mathsf{SR}}^{\text{kind}}(\mathcal{B}), \quad \text{Adv}_{\mathsf{BR}}^{G_2,G_3}(\mathcal{D}) \leq \text{Adv}_{\mathsf{SR}}^{\text{kind}}(\mathcal{B})$$

Since all challenged keys are sampled uniformly at random from the key space in game G_3, the adversary cannot derive information on bit b and consequently the advantage of an adversary is 0:

$$\text{Adv}_{\mathsf{BR}}^{G_3}(\mathcal{A}) = 0$$

Proof result Summing up the loss due to the game hops described above, provides us with the advantage of an adversary in winning the BRKE $\text{KIND}_{\mathsf{BR}}$ game depending on the advantages of adversaries $\mathcal{B}$ and $\mathcal{C}$:

$$\text{Adv}^{\text{kind}}_{\mathsf{BR}}(\mathcal{A}) \leq 2\text{Adv}^{\text{kind}}_{\mathsf{SR}}(\mathcal{B}) + q_R \text{Adv}^{\text{suf}}_{\mathsf{S}}(\mathcal{C})$$

Game $\text{KIND}^b_{\mathsf{BR}}(\mathcal{A})$

00 For $u \in \{A, B\}$:
01 $\quad s_u \leftarrow 0;\ r_u \leftarrow 0$
02 $\quad e_u \leftarrow 0;\ \text{EP}_u[\cdot] \leftarrow \bot$
03 $\quad E^{\vdash}_u \leftarrow 0;\ E^{\dashv}_u \leftarrow 0$
04 $\quad adc_u[\cdot] \leftarrow \bot;\ is_u \leftarrow \mathtt{T}$
05 $\quad key_u[\cdot] \leftarrow \bot;\ \text{XP}_u \leftarrow \emptyset$
06 $\quad \text{TR}_u \leftarrow \emptyset;\ \text{CH}_u \leftarrow \emptyset$
07 $\quad oos_u \leftarrow \infty;\ forge_u \leftarrow \mathtt{F}$ $\quad\mathbf{G_1}$
08 $\quad VK_u[\cdot] \leftarrow \bot$ $\quad\mathbf{G_1}$
09 $(S_{A,S}, S_{B,R}) \leftarrow_\$ \text{init}_{\mathsf{SR}}$
10 $(S_{B,S}, S_{A,R}) \leftarrow_\$ \text{init}_{\mathsf{SR}}$
11 $S_A \leftarrow (S_{A,S}, S_{A,R})$
12 $S_B \leftarrow (S_{B,S}, S_{B,R})$
13 $b' \leftarrow_\$ \mathcal{A}$
14 For $u \in \{A, B\}$:
15 $\quad$ Require $\text{TR}_u \cap \text{CH}_u = \emptyset$
16 Stop with b'

Oracle $\text{Snd}(u, ad)$

17 Require $S_u \neq \bot$
18 $(S_1, S_2) \leftarrow S_u$
19 $(sgk, vfk) \leftarrow_\$ \text{gen}_{\mathsf{S}};\ ad \xleftarrow{\shortparallel} vfk$
20 $VK_u[s_u] \leftarrow vfk$ $\quad\mathbf{G_1}$
21 $(S_1, k.o, c_1) \leftarrow_\$ \text{snd}_A(S_1, ad)$
22 $(S_2, c_2) \leftarrow_\$ \text{snd}_B(S_2, ad)$
23 $\sigma \leftarrow_\$ \text{sgn}(sgk, c_1 \parallel c_2)$
24 $c \leftarrow vfk \parallel c_1 \parallel c_2 \parallel \sigma$
25 $S_u \leftarrow (S_1, S_2)$
26 If is_u:
27 $\quad adc_u[s_u] \leftarrow (ad, c)$
28 $\quad \text{EP}_u[s_u] \leftarrow e_u$
29 $\quad E^{\dashv}_u \leftarrow E^{\dashv}_u + 1$
30 $key_u[\mathtt{S}, e_u, s_u] \leftarrow k.o$
31 $s_u \leftarrow s_u + 1$
32 Return c

Oracle $\text{Reveal}(u, i)$

as in URKE/SRKE (Fig. 5)

Oracle $\text{Rcv}(u, ad, c)$

33 Require $S_u \neq \bot$
34 If $is_u \wedge adc_{\bar{u}}[r_u] \neq (ad, c)$:
35 $\quad is_u \leftarrow \mathtt{F}$
36 $\quad oos_u \leftarrow s_u$ $\quad\mathbf{G_1}$
37 $\quad$ If $oos_{\bar{u}} > r_u$: $forge_u \leftarrow \mathtt{T}$ $\quad\mathbf{G_1}$
38 $\quad$ If $r_u \in \text{XP}_{\bar{u}}$:
39 $\quad\quad \text{TR}_u \xleftarrow{\cup} \{\mathtt{S}\} \times \mathbb{N} \times [s_u, \ldots]$
40 $\quad\quad \text{TR}_u \xleftarrow{\cup} \{\mathtt{R}\} \times \mathbb{N} \times [r_u, \ldots]$
41 If is_u:
42 $\quad E^{\vdash}_u \leftarrow \text{EP}_{\bar{u}}[r_u]$
43 $\quad e_u \leftarrow e_u + 1$
44 $(S_1, S_2) \leftarrow S_u$
45 $vfk \parallel c_1 \parallel c_2 \parallel \sigma \leftarrow c;\ ad \xleftarrow{\shortparallel} vfk$
46 Require $\text{vfy}_{\mathsf{S}}(vfk, c_1 \parallel c_2, \sigma)$
47 If $forge_u \wedge vfk = VK_{\bar{u}}[r_u]$: Abort $\quad\mathbf{G_1}$
48 $S_1 \leftarrow \text{rcv}_A(S_1, ad, c_2)$
49 Require $S_1 \neq \bot$
50 $(S_2, k.o) \leftarrow \text{rcv}_B(S_2, ad, c_1)$
51 Require $S_2 \neq \bot$
52 $S_u \leftarrow (S_1, S_2)$
53 If $S_u = \bot$: Return $\bot$
54 If is_u: $k.o \leftarrow \diamond$
55 $key_u[\mathtt{R}, E^{\vdash}_u, r_u] \leftarrow k.o$
56 $r_u \leftarrow r_u + 1$
57 Return

Oracle $\text{Expose}(u)$

58 $\text{TR}_u \xleftarrow{\cup} \{\mathtt{R}\} \times [E^{\vdash}_u \mathinner{..} E^{\dashv}_u] \times [r_u, \ldots]$
59 If is_u:
60 $\quad \text{XP}_u \xleftarrow{\cup} \{s_u\}$
61 $\quad \text{TR}_{\bar{u}} \xleftarrow{\cup} \{\mathtt{S}\} \times [E^{\vdash}_u \mathinner{..} E^{\dashv}_u] \times [r_u, \ldots]$
62 Return S_u

Oracle $\text{Challenge}(u, i)$

63 Require $key_u[i] \in \mathcal{K}$
64 $k \leftarrow b\ ?\ key_u[i] : \$(\mathcal{K})$
65 If $u = A \wedge i \in \{\mathtt{S}\} \times \mathbb{N}^2$ $\quad\mathbf{G_2}$
$\quad \vee u = B \wedge i \in \{\mathtt{R}\} \times \mathbb{N}^2$: $\quad\mathbf{G_2}$
66 $\quad k \leftarrow_\$ \mathcal{K}$ $\quad\mathbf{G_2}$
67 If $u = B \wedge i \in \{\mathtt{S}\} \times \mathbb{N}^2$ $\quad\mathbf{G_3}$
$\quad \vee u = A \wedge i \in \{\mathtt{R}\} \times \mathbb{N}^2$: $\quad\mathbf{G_3}$
68 $\quad k \leftarrow_\$ \mathcal{K}$ $\quad\mathbf{G_3}$
69 $key_u[i] \leftarrow \diamond$
70 $\text{CH}_u \xleftarrow{\cup} \{i\}$
71 Return k

Fig. 20: Proof of BRKE scheme from Figure 13 in BRKE $\text{KIND}_{\mathsf{BR}}$ game from Figure 12.

E Extended preliminaries

We give the exact security games of the primitives presented in Section 2.

E.1 Message authentication codes

A *message authentication code* (MAC) for a message space $\mathcal{M}$ is a pair $\mathsf{M} = (\text{tag}, \text{vfy}_\mathsf{M})$ of algorithms together with a samplable key space $\mathcal{K}$ and a tag space $\mathcal{T}$. The tag-generation algorithm tag may be randomized and takes a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, and outputs a tag $\tau \in \mathcal{T}$. The deterministic tag-verification algorithm vfy_M takes a key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$, and a tag $\tau \in \mathcal{T}$, and outputs a Boolean value: either $\texttt{T}$ (for accept) or $\texttt{F}$ (for reject). Shortcut notations for tag generation and verification are thus

$$\mathcal{K} \times \mathcal{M} \to \text{tag} \to \mathcal{T} \qquad \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \text{vfy}_\mathsf{M} \to \{\texttt{T}, \texttt{F}\} \enspace .$$

For correctness we require that for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$ and $\tau \in [\text{tag}(k, m)]$ we have $\text{vfy}_\mathsf{M}(k, m, \tau) = \texttt{T}$.

As a security property for MACs we formalize a multi-instance version of (strong) unforgeability. In this notion the adversary has to produce, for a message of its choosing, a fresh but valid tag (for any out of a set of independent instances, each initialized with a uniformly picked key). The adversary is supported by tag generation and verification oracles. The adversary is also allowed to create new instances, or to expose them, meaning to learn their keys. The details of this notion are in game SUF in Figure 21. For a MAC M, we associate with any adversary $\mathcal{A}$ its strong unforgeability advantage $\text{Adv}^{\text{suf}}_\mathsf{M}(\mathcal{A}) := \Pr[\text{SUF}(\mathcal{A}) \Rightarrow 1]$. Intuitively, the MAC is secure if all practical adversaries have a negligible advantage.

Game SUF($\mathcal{A}$)	**Oracle** Mac(i, m)
00 $n \leftarrow 0$; $\text{XP} \leftarrow \emptyset$	10 Require $1 \le i \le n$
01 Invoke $\mathcal{A}$	11 $\tau \leftarrow_\$ \text{tag}(k_i, m)$
02 Stop with 0	12 $\text{MT}_i[m] \xleftarrow{\cup} \{\tau\}$
Oracle Gen	13 Return τ
03 $n \leftarrow n + 1$	**Oracle** Vfy(i, m, τ)
04 $k_n \leftarrow_\$ \mathcal{K}$	14 Require $1 \le i \le n$
05 $\text{MT}_n[\cdot] \leftarrow \emptyset$	15 $v \leftarrow \text{vfy}_\mathsf{M}(k_i, m, \tau)$
06 Return	16 If $i \notin \text{XP} \wedge \tau \notin \text{MT}_i[m]$:
Oracle Expose(i)	17 Reward v
07 Require $1 \le i \le n$	18 Return v
08 $\text{XP} \xleftarrow{\cup} \{i\}$	
09 Return k_i	

Fig. 21: Security experiment SUF, modeling the (strong) unforgeability of a MAC in a multi-instance setting. Variable n indicates the number of established instances, set XP keeps track of the instances that are exposed, and for each instance i the associative array MT_i keeps track of the messages and corresponding tags that are processed in Mac queries.

Our variant of unforgeability is equivalent to the standard notion with only one instance and no exposure. However, the corresponding reduction loses a factor of n, where n is the total number of instances: The instance for which the adversary successfully forges a tag is guessed, and the $n - 1$ remaining instances are simulated with knowledge of the corresponding key.

E.2 Key encapsulation mechanisms

Figure 22 formalizes the security game OW for one-way security of a KEM $\mathsf{K} = (\text{gen}_\mathsf{K}, \text{enc}, \text{dec})$ in a multi-receiver/multi-challenge setting allowing exposures.

Game $\text{OW}(\mathcal{A})$
00 $n \leftarrow 0$; $\text{XP} \leftarrow \emptyset$
01 Invoke $\mathcal{A}$
02 Stop with 0

Oracle Gen
03 $n \leftarrow n + 1$
04 $(sk_n, pk_n) \leftarrow_\$ \text{gen}_\mathsf{K}$
05 $CK_n[\cdot] \leftarrow \bot$
06 Return pk_n

Oracle $\text{Enc}(i)$
07 Require $1 \leq i \leq n$
08 $(k, c) \leftarrow_\$ \text{enc}(pk_i)$
09 $CK_i[c] \leftarrow k$
10 Return c

Oracle $\text{Solve}(i, c, k)$
11 Require $1 \leq i \leq n$
12 Require $i \notin \text{XP}$
13 Require $CK_i[c] \neq \bot$
14 Reward $k = CK_i[c]$
15 Return

Oracle $\text{Check}(i, c, k)$
16 Require $1 \leq i \leq n$
17 $k' \leftarrow \text{dec}(sk_i, c)$
18 Return $[k' = k]$

Oracle $\text{Expose}(i)$
19 Require $1 \leq i \leq n$
20 $\text{XP} \xleftarrow{\cup} \{i\}$
21 Return sk_i

Fig. 22: Security experiment OW, modeling the one-way security of a KEM in a multi-receiver/multi-challenge setting. Variable n indicates the number of established receivers, set XP keeps track of the receivers that are exposed, and for each receiver i the associative array CK_i keeps track of the ciphertexts and session keys that are processed in Enc queries.

Our variant of one-wayness is equivalent to the standard notion with only one receiver, one challenge encapsulation, and no exposure. However, the corresponding reduction loses a factor of nm,[22] where n is the total number of receivers and m is the total number of challenge encapsulations per receiver: The receiver for which the adversary successfully recovers the session key is guessed, and the $n - 1$ remaining receivers are simulated with knowledge of their secret key; further, the later-broken encapsulation query of the identified user is guessed, and the remaining encapsulation queries simulated using the regular encapsulation algorithm.

E.3 One-time signature schemes

A *one-time signature scheme* for a message space $\mathcal{M}$ is a triple $\mathsf{S} = (\text{gen}_\mathsf{S}, \text{sgn}, \text{vfy}_\mathsf{S})$ of algorithms together with a signer key-space $\mathcal{SK}$, a verifier key-space $\mathcal{VK}$, and a signature space Σ. The randomized key-generation algorithm gen_S outputs a signer key $sgk \in \mathcal{SK}$ and a verifier key $vfk \in \mathcal{VK}$. The signing algorithm sgn may be randomized and takes a signer key $sgk \in \mathcal{SK}$ and a message $m \in \mathcal{M}$, and outputs a signature $\sigma \in \Sigma$. The deterministic verification algorithm vfy_S takes a verifier key $vfk \in \mathcal{VK}$, a message $m \in \mathcal{M}$, and a (candidate) signature $\sigma \in \Sigma$, and outputs a bit $b \in \{\mathtt{T}, \mathtt{F}\}$, indicating acceptance and rejection, respectively. Shortcut notations for the three algorithms are thus

$$\text{gen}_\mathsf{S} \to \mathcal{SK} \times \mathcal{VK} \qquad \mathcal{SK} \times \mathcal{M} \to \text{sgn} \to \Sigma \qquad \mathcal{VK} \times \mathcal{M} \times \Sigma \to \text{vfy}_\mathsf{S} \to \{\mathtt{T}, \mathtt{F}\} \ .$$

For correctness we require that for all $(sgk, vfk) \in [\text{gen}_\mathsf{S}]$ and $m \in \mathcal{M}$ and $\sigma \in [\text{sgn}(sgk, m)]$ we have $\mathtt{T} = \text{vfy}_\mathsf{S}(vfk, m, \sigma)$.

We formalize a security notion of (strong) unforgeability for one-time signatures. Concretely, the adversary controls the message processed by the signer, it sees the resulting signature, and

[22] For all KEM applications in this paper we actually have $m = 1$, i.e., the security loss is effectively only by a factor of n.

its goal is to make the verifier accept a message-signature pair that was not processed by the signer. The details of this notion are in game SUF in Figure 23. For a one-time signature scheme S, we associate with any adversary $\mathcal{A}$ its strong unforgeability advantage $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{A}) := \Pr[\mathrm{SUF}(\mathcal{A}) \Rightarrow 1]$. Intuitively, the one-time signature scheme is secure if all practical adversaries have a negligible advantage.

Game SUF($\mathcal{A}$)	**Oracle** Sgn(m)	**Oracle** Vfy(m, σ)
00 $ms \leftarrow \bot$; $s \leftarrow \mathtt{F}$	04 Require $s = \mathtt{F}$	08 If not $\mathrm{vfy}_{\mathsf{S}}(vfk, m, \sigma)$:
01 $(vfk, sgk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$	05 $\sigma \leftarrow_{\$} \mathrm{sgn}(sgk, m)$	09 Return $\mathtt{F}$
02 Invoke $\mathcal{A}(vfk)$	06 $ms \leftarrow (m, \sigma)$; $s \leftarrow \mathtt{T}$	10 Reward $ms \neq (m, \sigma)$:
03 Stop with 0	07 Return σ	11 Return $\mathtt{T}$

Fig. 23: Security experiment SUF, modeling the (strong) unforgeability of a one-time signature scheme. Variable s counts the signing operations and variable ms records the message-signature combination processed by the signer.

F Modeling ratcheted key exchange

A common criticism in the key exchange community is that many constructions are proposed with an own model for defining their security. Different models for schemes of a similar nature hamper comparability and comprehensibility. We anticipate this by comparing our approach to model security of ratcheted key exchange with prior work on ratcheting and on key agreement in general. Essentially we conclude that our model is in line with prior strategies and with prior notation. However, ratcheted key exchange is only loosely related to classic key agreement.

A security model (for key exchange) mainly consists of three components: 1) communication model with partnering definition, 2) the adversary's ability to obtain information on the communicating parties' secrets, and 3) a winning condition for the security game defined by excluding trivial attacks.

In our definitions (see Figures 5,9,12) we depict all three parts of the model in one figure respectively. The communication model is implicitly given by the oracles Snd, Rcv. The partnering is defined via the *is* bit (please note that the definition is related to *matching conversations*). The remaining oracles (Reveal, Expose) define the adversary's ability to obtain secrets from the communicating parties. Finally, the challenge oracle together with the described excluded trivial attacks define the winning conditions for the adversary. Note that excluding trivial attacks within the oracles is comparable to defining a freshness condition separately. By combining all components of the model in a single compact game definition, the dependencies among them become visible. This especially plays a role in our model since, in contrast to classic key agreement models, our model allows and is based on concurrency in the communication (which e.g., influences the trivial attacks).

Please note that there is an important difference between ratcheted key exchange and classic key agreement: while key agreement protocols aim to provide the initialization of a communication, ratcheted key exchange serves as a primitive that provides an already initialized session with continuously updated session keys. One can imagine that both worlds (classic key agreement for initialization and ratcheted key exchange for protecting an initialized session) can be composed by using the key, derived from the classic key agreement, to initialize the states for the ratcheted key exchange. As such, our model does not need to consider an environment with

multiple users (and multiple sessions each). Consequently users and *long-term keys* do not play a role in ratcheted key exchange[23].

Both, by defining the security model within one compact game definition, and by disregarding the explicit communication initialization, we are in line with the approach of Bellare et al. [2]. In contrast, for example Cohn-Gordon et al. [6] provide a model that presents the three previously named components one after another. We believe that the choice of notation is a matter of taste and in our case, one compact game description is more appropriate.

As described before, our technique for deriving a model for ratcheted key exchange differs from previous work on ratcheting significantly, resulting in a comprehensible and naturally strong security definition. One could derive weaker notions of security by restricting the communication model or the adversary's access to the exposure oracles. These weaker notions could be comparable to earlier modeling approaches and would allow for more efficient protocols. However, they would not comply with our idea of ratcheted key exchange.

G Related schemes

In this section we introduce schemes from the related literature that reach certain types of security for keys in the presence of exposing adversaries. For clarity, we focus on schemes that also reach security after an exposure and leave out schemes that only provide *forward secrecy.*

We briefly summarize the schemes' approaches to satisfy the defined requirements for a URKE, SRKE, or a BRKE respectively. Thereby we classify the schemes in one of these three categories. For schemes that employ the BRKE functionality, we consider only one communication direction and thereby describe their guarantees in the SRKE setting to reduce complexity.

ZRTP. ZRTP is a protocol for "establishing unicast Secure Real-time Transport Protocol (SRTP) sessions for Voice over IP (VoIP) applications" [24]. Thereby it holds a state between establishing two sessions with the same partner. Symmetric information of the previous state and a fresh Diffie-Hellman key exchange are mixed to derive a session secret and to update the symmetric state. Consequently an exposure can only be used to attack the next session because the Diffie-Hellman key exchange refreshes the state on both sides untraceably for the attacker.

Epoch update Since ZRTP is synchronous, it does not match our use-case. However, one can classify each session between two communicating users as an epoch. Thereby the initiator is the receiver of a SRKE scheme and the responder is sender.

Authenticated key exchange. Cohn-Gordon et al. provide the first notable scientific work on security of authenticated key exchange protocols considering the security guarantees after a compromise of one communicating party [7]. They first define security guarantees for this purpose based on existing models. In order to provide a strong security notion that covers security guarantees after a compromise, they use the security model eCK^w [8] that is related to the extended Canetti-Krawczyk security model [12] and modify it accordingly to derive a security model named eCK^w-PCS.

Security model This defined model focuses on sessions that are executed successively between communicating parties. Thereby an adversary can be successful in the eCK^w model if it can challenge a key but:

[23] Please note that the construction of Bellare et al. [2] does not suffice our model *because* it employs a long-term key for Bob.

1. none of the communicating parties revealed the challenged key,
2. none of the communicating parties exposed both the whole state and the randomness that was used to generate the challenged session key, and
3. if the challenged user initiated the challenged session, her partner did not expose her state.

Instead of the last requirement, the eCK^w-PCS model requires:

3. if the challenged user initiated the challenged session and her partner exposed her state, then there was a session executed between the communicating parties after the partner's exposure but before the challenged session.

Contrary to our model, their paper deals with the robustness of the communication. Thereby they require explicit authentication in order to prevent denial-of-service attacks.

Epoch update Subsequently they propose a generic transformation that uses an authenticated key exchange (AKE) secure according to eCK^w, a MAC, and a KDF to derive a protocol that is eCK^w-PCS secure.

This resulting protocol is a three-way protocol that establishes a key after the three messages and updates the parties' state after every received message. Consequently it is necessary to interactively communicate in order to derive a key. Thereby the protocol can be seen as a protocol for epoch updates while there is no key establishment within an epoch.

The model covers whole sessions instead of single ciphertexts. Consequently a whole session needs to be executed after an exposure to recover from it, while our model requires that the states are recovered as soon as a single ciphertext is sent and received after an exposure.

Their properties result from the fact that they consider a different environment and investigate a different primitive. One may derive a comparable model by restricting the communication model of our BRKE definition respectively.

OTR, Signal, Pond. OTR [4,19], Signal [15,6], and Pond [13] are implementations of key exchange protocols that recover after an exposure. Their main use case is instant messaging. Pond implements the Signal protocol now and therefore is not further considered. OTR, which is mainly used in a synchronous environment, can be seen as Signal's predecessor, that is designed for an asynchronous setting.

OTR, like the previously described synchronous schemes, updates only epochs. Keys are static within an epoch. Signal in addition also updates the keys within every epoch. Since the epoch update of OTR and Signal are slightly different, we describe them separately.

We omit the protocols' initial key exchange from our considerations since our syntax summarizes it in a non-interactive algorithm.

Notation The summarized protocol descriptions below follow this notation:

$$A(\text{initial state of } A); B(\text{initial state of } B)$$

i. Message direction(Random value) : State update at sending   Transmitted ciphertext

By writing $(\mathbf{k_{3..i-1}}, ck) \leftarrow_{2..i-2} H(ck)$, we denote multiple invocations of the same procedure with multiple assignments such that each key $\mathbf{k_x}$ is set once. Thereby the value of ck is updated at every call and used for the next call respectively.

Epoch update: OTR With every ciphertext that is sent in response to an epoch update *from* the partner, an epoch update *to* the partner is performed. Thereby a new Diffie-Hellman share is generated and sent. The receiving partner uses this Diffie-Hellman share and computes a key with its own established private Diffie-Hellman exponent for sending new ciphertext afterwards. The authenticity of each ciphertext is preserved by attaching a MAC tag of the Diffie-Hellman share under the previous hashed key:

$$A(g^{a_0}, a_0, g^{b_0}); B(g^{b_0}, b_0, g^{a_0})$$

$$\begin{aligned}
1.\ A(a_1) \rightarrow B &: \mathbf{k_1} \leftarrow g^{a_0 b_0} && C_1 \leftarrow g^{a_1} \parallel F(H(k_1), g^{a_1}) \\
2, \ldots, i-1.\ A() \rightarrow B &: \mathbf{k_{2..i-1}} \leftarrow k_1 \\
i.\ B(b_1) \rightarrow A &: \mathbf{k_i} \leftarrow g^{a_1 b_0} && C_i \leftarrow g^{b_1} \parallel F(H(k_i), g^{b_1}) \\
i+1, \ldots, j-1.\ B() \rightarrow A &: \mathbf{k_{i+1..j-1}} \leftarrow k_i \\
j.\ A(a_2) \rightarrow B &: \mathbf{k_j} \leftarrow g^{a_1 b_1} && C_j \leftarrow g^{a_2} \parallel F(H(k_j), g^{a_2})
\end{aligned}$$

The transmitted ciphertext of the payload data, that is encrypted under the respective key k_x, is always MACed under the hashed key $H(k_x)$; so in case of an epoch update, the new Diffie-Hellman share and the ciphertext are concatenated.

Epoch update: Signal Signal also only updates an epoch if the previously received ciphertexts was an epoch update. However, their key computation is more complex. To maintain clarity, we only describe the SRKE setting and indicate how the BRKE scheme is derived from it.

$$A(rk, g^{a_0}, a_0, g^{b_0}); B(rk, g^{b_0}, b_0, g^{a_0})$$

$$\begin{aligned}
1.\ B(b_1) \rightarrow A :& (rk, ck) \leftarrow H(rk, g^{a_0 b_1});\ (ck, \mathbf{k_1}) \leftarrow H(ck) \\
& C_1 \leftarrow g^{b_1} \parallel F(H(k_1), g^{b_1}) \\
2.\ A(a_1) \rightarrow B :& (rk, ck) \leftarrow H(rk, g^{a_1 b_1});\ (ck, \mathbf{k_2}) \leftarrow H(ck) \\
& C_2 \leftarrow g^{a_1} \parallel F(H(k_2), g^{a_1}) \\
3, \ldots, i-1.\ A() \rightarrow B :&\ (\mathbf{k_{3..i-1}}, ck) \leftarrow_{2..i-2} H(ck) \\
i.\ B(b_2) \rightarrow A :& (rk, ck) \leftarrow H(rk, g^{a_1 b_2});\ (ck, \mathbf{k_i}) \leftarrow H(ck) \\
& C_i \leftarrow g^{b_2} \parallel F(H(k_i), g^{b_2}) \\
i+1.\ A(a_2) \rightarrow B :& (rk, ck) \leftarrow H(rk, g^{a_2 b_2});\ (ck, \mathbf{k_{i+1}}) \leftarrow H(ck) \\
& C_{i+1} \leftarrow g^{a_2} \parallel F(H(ck), g^{a_2}) \\
i+2, \ldots, j-1.\ A() \rightarrow B :&\ (\mathbf{k_{i+2..j-1}}, ck) \leftarrow_{i+1..j-2} H(ck)
\end{aligned}$$

The epoch update only takes place in ciphertexts $1, 2, i, i+1$. Thereby the root key rk is updated with the new Diffie-Hellman key exchange, and a new chaining key ck is computed. This chaining key is used to derive the output key k_x. The remaining two steps build the key derivation for the communication within the epochs.

Communication within epoch The design of the key establishment in an epoch consist of a KDF that uses the chaining key ck from the previous ciphertext and produces a new chaining key ck and the output session key k_x. Since this computation is deterministic, no information needs to be transmitted to the partner.

Shortcomings Since OTR provides no security measures within an epoch (the key stays static therein), previous keys can be challenged by a later exposure during the same epoch. Example: 1. $c \leftarrow$ SndA; $(\dots, k, \dots) \leftarrow$ ExposeA; $k' \leftarrow$ Challenge$(A, 1)$; $b' \leftarrow [k = k']$; output b'.

The key derivation of Signal within an epoch prevents this attack. Exposing the sender before sending and thereby deriving a key – which is allowed in our model – however reveals all secrets, since the key derivation is deterministic. Exposing the sender, sending and receiving and then challenging the receiver within an epoch is possible the same way. Example: $(\dots, ck, \dots) \leftarrow$ ExposeA; 1. SndA; $k' \leftarrow$ Challenge$(A, 1)$; $(k, ck^*) \leftarrow H(ck)$; $b' \leftarrow [k = k']$; output b'.

Due to the delay of an epoch update until the partner sent an epoch update, another attack which is allowed in our model, is possible against OTR and Signal. Sending twice and exposing after the first sent ciphertext, receiving both ciphertexts at the partner's side, and then sending at the partner's side is challengeable because the sent ciphertext after the exposure does not propose a new epoch. Example: 1. $c_1 \leftarrow$ SndB; $(\dots, ck_1, \dots) \leftarrow$ ExposeB; 2. $c_2 \leftarrow$ SndB; 1. RcvA(c_1); 2. RcvA(c_2); 3. SndA; $k' \leftarrow$ Challenge$(A, 3)$; $(k_2, ck_2) \leftarrow H(ck_1)$; $(k, ck_3) \leftarrow H(ck_2)$; $b' \leftarrow [k = k']$; output b'.

Ratcheted key exchange. Bellare et al. define the first URKE model [2]. They additionally provide a scheme that they prove secure in their model. Their model requires a scheme to be less resilient to attacks and their scheme provides less security accordingly in comparison to our model and scheme.

Security model Similar to our URKE model, their model provides the adversary with oracles SndA, RcvB, Challenge, and Expose. While the adversary in their model is able to challenge both parties as well, they only permit to expose the sender (*Alice*). As a consequence their model requires no forward secrecy at the receiver side.

As a side effect, also implicit authentication is not required because if sender and receiver are out of sync, the receiver's state is not exposable. Consequently the receiver's state does not need to change on a manipulated ciphertext.

Communication within epoch Basically their URKE scheme consists of an ElGamal KEM, a KDF (modeled by a random oracle), and a PRF (used as a MAC).

$$A(0, Y, K); B(0, K, y)$$

$$\begin{array}{ll} 1.\ A(x_1) \rightarrow B : \mu \leftarrow F(K, g^{x_1});\ (\mathbf{k_1}, K) \leftarrow H(1, \mu, g^{x_1}, Y^{x_1}) & C_1 \leftarrow g^{x_1} \parallel \mu \\ 2.\ A(x_2) \rightarrow B : \mu \leftarrow F(K, g^{x_2});\ (\mathbf{k_2}, K) \leftarrow H(2, \mu, g^{x_2}, Y^{x_2}) & C_2 \leftarrow g^{x_2} \parallel \mu \end{array}$$

The receiver's secret key y is static while the symmetric secret K is updated with every ciphertext and used for the authenticity of the next ciphertext. This symmetric secret is derived by the key derivation of the last symmetric secret and the Diffie-Hellman key exchange between the receiver's static value and the sender's fresh Diffie-Hellman share. The output key is derived from the same values. For simplicity, we restrict the description to the most relevant values.

Shortcomings In contrast to epochs in Signal, that are deterministically computed, the URKE scheme of Bellare et al. includes randomness in the computation of every key. However, the receiver's secret key is static and thereby every key can be computed by exposing the receiver after the key establishment. While their model does not enable the adversary to perform this attack, our model permits it and our scheme prevents the attack to be successful. Example:

1. $X \parallel \mu \leftarrow$ SndA; 1. RcvB($X \parallel \mu$); $(\dots, y, \dots) \leftarrow$ ExposeB; $k' \leftarrow$ Challenge$(A, 1)$; $(k, K) \leftarrow H(1, \mu, X, X^y)$; $b' \leftarrow [k = k']$; output b'.

In addition to that, a more sophisticated attack, similar to the previous one, is possible. To achieve the security required by our model, the scheme must diverge the parties' states after the adversary actively attacked the communication such that the receiver's state cannot be used to derive the same keys as the sender anymore. This is not provided by the described URKE scheme as well. Example: $(\dots, K, \dots) \leftarrow$ ExposeA; $x \leftarrow_{\$}$; 1. RcvB($g^x \parallel F(K, g^x)$); $(\dots, y, \dots) \leftarrow$ ExposeB; 1. $X' \parallel \mu \leftarrow$ SndA; $k' \leftarrow$ Challenge$(A, 1)$; $(k, K) \leftarrow H(1, \mu, X', X'^y)$; $b' \leftarrow [k = k']$; output b'.