Towards Bidirectional Ratcheted Key Exchange

Bertram Poettering[1] and Paul Rösler[2(✉)]

[1] Information Security Group, Royal Holloway, University of London, Egham, UK
bertram.poettering@rhul.ac.uk

[2] Horst-Görtz Institute for IT Security, Chair for Network and Data Security, Ruhr-University Bochum, Bochum, Germany
paul.roesler@rub.de

Abstract. Ratcheted key exchange (RKE) is a cryptographic technique used in instant messaging systems like Signal and the WhatsApp messenger for attaining strong security in the face of state exposure attacks. RKE received 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 (which might be weaker than the notion that it *should* achieve), the authors of the latter develop and instantiate a notion of security from scratch, independently of existing implementations. Unfortunately, however, their model is quite restricted, e.g. for considering only unidirectional communication and the exposure of only one of the two parties.

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 contribute. We follow a purist approach, aiming at finding strong yet convincing notions that cover a realistic communication model with 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 RKE require a stronger, HIBE-like component.

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,

The full version of this article is available in the IACR eprint archive as article 2018/296, at https://eprint.iacr.org/2018/296.

H. Shacham and A. Boldyreva (Eds.): CRYPTO 2018, LNCS 10991, pp. 3–32, 2018.
https://doi.org/10.1007/978-3-319-96884-1_1

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. 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 [18]. 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 in case of a different kind of attack: the skimming of the session state of an ongoing conversation [18].[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 in general long-lived, e.g., kept alive for weeks or months once established, so that state exposures are 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).

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

Ratcheting. Modern messaging protocols are designed with the goal of providing security even in the face of adversaries that perform the two types of attack discussed above (compromise of long-term secrets and/or session states) [18]. One technique used towards achieving this 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 [17] 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: As part of every outgoing message a fresh g^x value 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. This was introduced with the off-the-record (OTR) messaging protocol from [3,13] 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. 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 [9]) if it uses the described or similar techniques for achieving forward secrecy and security under state exposure attacks.

Ratcheting as a primitive. While many authors associate the word ratcheting with a set of techniques deployed with the aim of achieving certain (typically not formally defined) security goals, Bellare et al. recently pursued a different approach by proposing *ratcheted key exchange* (RKE) as a cryptographic primitive with clearly defined syntax, functionality, and security properties [1]. 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 [1] 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 [1]. 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 F, 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 H is 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 $(i+1, K', Y)$.

[2] Note that RKE, despite its name, is a tool to be used in the 'symmetric phase' that follows the preliminary key agreement. In [1], and also in this article, the latter is abstracted away into a dedicated state initialization algorithm (or: protocol).

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 actually not 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 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]

Contributions. We follow in the footsteps of [1] 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 of Bellare et al. towards bidirectional communication. In more detail, if we refer to the setting of [1] as URKE (unidirectional RKE), we introduce SRKE (sesquidirectional[4] RKE) and BRKE (bidirectional RKE; for space reasons only in the full version [14]). 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).

Secondly, we propose an improved security model for URKE, and introduce security models for SRKE and BRKE (the latter only in [14]). Our SRKE and BRKE 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, a ping-pong way). We develop our models following a purist approach: We start with giving the adversary the full set of options to undertake its attack (including 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

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

the (as we argued) insecure protocol from [1] is considered secure in the model of [1] because the latter was not designed with our strategy in mind, ultimately missing some attacks.

Thirdly, we give provably secure constructions of URKE and SRKE (and of BRKE in the full version [14]). While all prior RKE protocol proposals, including the one from [1], 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 exposures. The crucial difference to the protocol from [1] is that in our scheme the public key of Bob is changed after each use. Concretely, but omitting many details, the state information of Alice is (i, K, Y) as in [1] (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 [1] that we sketched above does not work against this protocol (the adversary would obtain a useless decryption 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 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 the algorithms 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 builds on 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.

Outlook on BRKE. We expose two BRKE constructions in the full version [14]. 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 (our solution does this with one-time signatures).

The second construction is less generic but slightly more efficient, namely by combining and interleaving the building blocks of our SRKE scheme in the right way.

Further related work. The idea of using 'hash chains' for achieving forward security of symmetric cryptographic primitives has been around for quite some time. For instance, [17] use this technique to protect the integrity of audit logs. The first formal treatment we are aware of is [2]. A messaging protocol that uses this technique is the (original) Silent Circle Instant Messaging Protocol [12].

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 [3,13]. Subsequently, the technique appeared in many communication protocols specifically designed to be privacy-friendly, including the ZRTP telephony protocol [19] and the messaging protocol *Double Ratchet Algorithm* [10] (formerly known as Axolotl). The latter, or close variants thereof, are used by WhatsApp, the Facebook Messenger, and Signal app. In the full version [14] we study these protocols more closely, proposing for each of them an attack that shows that it is not secure in our models.

Widely used messaging protocols were recently analyzed by Cohn-Gordon et al. [4] and Rösler et al. [16]. In particular, [4] contributes an analysis of the Signal messaging protocol [10] by developing a "*model with adversarial queries and freshness conditions that capture the security properties intended by Signal*". 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 reference notion for RKE.[5]

Academic work in a related field was conducted by [5] 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.

Organization. In Sect. 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 Sect. 3 we develop the URKE syntax and a suitable security model, and present a corresponding construction in Sect. 4. In Sects. 5 and 6 we do the same for SRKE. In Sect. 7 we give an intuition of how SRKE can be extended to BRKE.

[5] In fact it defines weaker security than would be natural for RKE. We elaborate on this in the full version [14] where we explain why the Signal protocol is not secure in our model.

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 an 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 $[a\,..\,b]$ for the set $\{a, \ldots, b\}$ and we write $[\,a, ...\,]$ 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 $I = [a\,..\,b]$ 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 write 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 "$C\ \mathtt{?}\ e_1\ \mathtt{:}\ e_2$" 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. A list can be appended to another list with the 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 $L_1 \preceq L_2 \not\preceq L_3$.

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 $X \xleftarrow{\cup} Y$ shorthand for $X \leftarrow X \cup Y$, and if L_1, L_2 are lists we write $L_1 \xleftarrow{\|} L_2$ 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 $\mathtt{abc}$ 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 definitely 'wins'.

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 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 $\perp$'. This instruction is typically used to assert that certain conditions hold for user-provided input.

2.2 Classic Cryptographic Building Blocks

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 standard, we provide only very reduced definitions here and defer the full specifications to [14]. For KEMs, however, we assume a specific non-standard syntax, functionality, and notion of security; the details can be found below.

MACs and One-Time Signatures. We denote the key space of a MAC M with $\mathcal{K}$, and assume 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 assume that $\mathrm{vfy}_{\mathsf{S}}$ outputs values $\mathtt{T}$ or $\mathtt{F}$ 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 $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{enc}, \mathrm{dec})$ 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 $\mathrm{gen}_{\mathsf{K}}$ 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 $\mathrm{gen}_{\mathsf{K}}$ 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

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

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

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

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

We formalize a multi-receiver/multi-challenge version of one-way security as a security property for KEMs. 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 expose receivers, learning their secret keys. The details of this notion are in game OW in the full version [14]. 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: 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 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. It should be hard for an adversary to recover encapsulated keys even if it obtained secret keys that are further or differently updated than the challenge secret key(s). The details of the notion are in game KUOW in the full version [14]. 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.

Observe that kuKEMs are related to hierarchical identity-based encryption (HIBE, [7]): 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 expressing the algorithms; we give the details and a specific construction in the full version [14].

3 Unidirectionally Ratcheted Key Exchange (URKE)

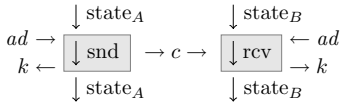
We give a definition of unidirectional RKE and its security. While, in principle, our syntactical definition is in line with the one from [1], our naming convention deviates significantly from the latter for the sake of a more clear distinction between (session) keys, (session) states, and ciphertexts[6]. Further, looking ahead, our security notion for URKE is stronger than the one of [1]. A speciality of our formalization is that we let the sending and receiving algorithms of Alice and Bob accept and process an associated data string [15] 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

[6] The mapping between our names (on the left of the equality sign) and the ones of [1] (on the right) is as follows: '(session) key' = 'output key', '(session) state' = 'session key plus sender/receiver key', 'ciphertext' = 'update information'.

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

$$\begin{array}{rcl} & \text{init} & \rightarrow \ \mathcal{S}_A \times \mathcal{S}_B \\ \mathcal{S}_A \times \mathcal{AD} & \rightarrow \ \text{snd} & \rightarrow \ \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\ \mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} & \rightarrow \ \text{rcv} & \rightarrow \ \mathcal{S}_B \times \mathcal{K} \ / \ \perp \end{array}$$



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 Fig. 1.[7] Concretely, we say scheme R is *correct* if $\Pr[\text{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ 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 considered 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 13, 14), 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 17) if the sender and the receiver compute different keys while still being in-sync. Note finally that lines 12, 16 ensure that once the rcv algorithm rejects, the adversary is notified of this and further queries to the RcvB oracle are not accepted.

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,

[7] 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 $\mathrm{FUNC}_{\mathsf{R}}(\mathcal{A})$	**Oracle** $\mathrm{SndA}(ad)$	**Oracle** $\mathrm{RcvB}(ad, c)$
00 $s_A \leftarrow 0$; $r_B \leftarrow 0$	07 $(S_A, k, c) \leftarrow_{\$} \mathrm{snd}(S_A, ad)$	12 Require $S_B \neq \bot$
01 $adc_A[\cdot] \leftarrow \bot$	08 $adc_A[s_A] \leftarrow (ad, c)$	13 If $is_B \wedge adc_A[r_B] \neq (ad, c)$:
02 $is_B \leftarrow \mathtt{T}$	09 $key_A[s_A] \leftarrow k$	14 $\quad is_B \leftarrow \mathtt{F}$
03 $key_A[\cdot] \leftarrow \bot$	10 $s_A \leftarrow s_A + 1$	15 $(S_B, k) \leftarrow \mathrm{rcv}(S_B, ad, c)$
04 $(S_A, S_B) \leftarrow_{\$} \mathrm{init}$	11 Return c	16 If $S_B = \bot$: Return $\bot$
05 Invoke $\mathcal{A}$		17 Reward $is_B \wedge k \neq key_A[r_B]$
06 Stop with 0		18 $r_B \leftarrow r_B + 1$
		19 Return

Fig. 1. Game FUNC for URKE scheme R.

in Fig. 2 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 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 Fig. 2) 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; $c \leftarrow \mathrm{SndA}(ad)$; $k \leftarrow \mathrm{Reveal}(A, 0)$; $k' \leftarrow \mathrm{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; 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.

Game $\mathrm{KIND}^b_{\mathsf{R}}(\mathcal{A})$
00 · $s_A \leftarrow 0$; $r_B \leftarrow 0$
01 · $adc_A[\cdot] \leftarrow \perp$; $is_B \leftarrow \mathtt{T}$
02 · $key_A[\cdot] \leftarrow \perp$; $key_B[\cdot] \leftarrow \perp$
03 $\mathrm{XP}_A \leftarrow \emptyset$
04 $\mathrm{TR}_A \leftarrow \emptyset$; $\mathrm{TR}_B \leftarrow \emptyset$
05 $\mathrm{CH}_A \leftarrow \emptyset$; $\mathrm{CH}_B \leftarrow \emptyset$
06 · $(S_A, S_B) \leftarrow_\$ \mathrm{init}$
07 · $b' \leftarrow_\$ \mathcal{A}$
08 Require $\mathrm{TR}_A \cap \mathrm{CH}_A = \emptyset$
09 Require $\mathrm{TR}_B \cap \mathrm{CH}_B = \emptyset$
10 · Stop with b'

Oracle $\mathrm{SndA}(ad)$
11 · $(S_A, k, c) \leftarrow_\$ \mathrm{snd}(S_A, ad)$
12 · $adc_A[s_A] \leftarrow (ad, c)$
13 · $key_A[s_A] \leftarrow k$
14 · $s_A \leftarrow s_A + 1$
15 · Return c

Oracle ExposeA
16 $\mathrm{XP}_A \xleftarrow{\cup} \{s_A\}$
17 · Return S_A

Oracle $\mathrm{Reveal}(u, i)$
18 · Require $key_u[i] \in \mathcal{K}$
19 · $k \leftarrow key_u[i]$
20 $key_u[i] \leftarrow \diamond$
21 · Return k

Oracle $\mathrm{RcvB}(ad, c)$
22 · Require $S_B \neq \perp$
23 · If $is_B \wedge adc_A[r_B] \neq (ad, c)$:
24 · $is_B \leftarrow \mathtt{F}$
25 If $r_B \in \mathrm{XP}_A$:
26 $\mathrm{TR}_B \xleftarrow{\cup} [r_B, \ldots]$
27 · $(S_B, k) \leftarrow \mathrm{rcv}(S_B, ad, c)$
28 · If $S_B = \perp$: Return $\perp$
29 If is_B: $k \leftarrow \diamond$
30 · $key_B[r_B] \leftarrow k$
31 · $r_B \leftarrow r_B + 1$
32 · Return

Oracle ExposeB
33 $\mathrm{TR}_B \xleftarrow{\cup} [r_B, \ldots]$
34 If is_B:
35 $\mathrm{TR}_A \xleftarrow{\cup} [r_B, \ldots]$
36 · Return S_B

Oracle $\mathrm{Challenge}(u, i)$
37 · Require $key_u[i] \in \mathcal{K}$
38 · $k \leftarrow b \,?\, key_u[i] : \$(\mathcal{K})$
39 $key_u[i] \leftarrow \diamond$
40 $\mathrm{CH}_u \xleftarrow{\cup} \{i\}$
41 · Return k

Fig. 2. 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 Sect. 2.1.

(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; $c \leftarrow \mathrm{SndA}(ad)$; $k \leftarrow \mathrm{Reveal}(A, 0)$; $\mathrm{RcvB}(ad, c)$; $k' \leftarrow \mathrm{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; 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 $\mathrm{XP}_A, \mathrm{TR}_A, \mathrm{TR}_B, \mathrm{CH}_A, \mathrm{CH}_B$ (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 $\mathrm{TR}_A, \mathrm{TR}_B$ track the indices of keys that are

'traceable' (in particular: recoverable by the adversary) using an exposed state (see below), and sets $\mathrm{CH}_A, \mathrm{CH}_B$ 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; $c \leftarrow \mathrm{SndA}(ad)$; $S_B^* \leftarrow \mathrm{ExposeB}()$; $(S_B^*, k) \leftarrow \mathrm{rcv}(S_B^*, ad, c)$; $\mathrm{RcvB}(ad, c)$; $k' \leftarrow \mathrm{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; 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; $c \leftarrow \mathrm{SndA}(ad)$; $S_B^* \leftarrow \mathrm{ExposeB}()$; $(S_B^*, k) \leftarrow \mathrm{rcv}(S_B^*, ad, c)$; $k' \leftarrow \mathrm{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; 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; $S_A^* \leftarrow \mathrm{ExposeA}()$; $(S_A^*, k, c) \leftarrow_{\$} \mathrm{snd}(S_A^*, ad)$; $\mathrm{RcvB}(ad, c)$; $k' \leftarrow \mathrm{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; 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.[8] 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.

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

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

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

Fig. 3. Construction of an URKE scheme from a key-encapsulation mechanism $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{enc}, \mathrm{dec})$, a message authentication code $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_{\mathsf{M}})$, 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 $\mathcal{K}$.

generation, see Sect. 2.2), a strongly unforgeable MAC, and a random oracle H. The algorithms of our scheme are specified in Fig. 3.

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 [1], 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.

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 (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).[9]

Theorem 1 (informal). *The URKE protocol* R *from Fig. 3 offers key indistinguishability if function* H *is modeled as a random oracle, the KEM provides* OW *security, the MAC provides* SUF *security, and the session-key space of the KEM is sufficiently large.*

The exact theorem statement and the respective proof are in the full version [14]. Briefly, the proof 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. 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 (SRKE)

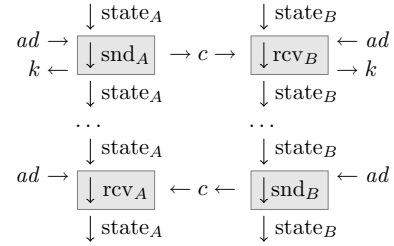
We introduce *sesquidirectionally ratcheted key exchange* (see Footnote 4) as a generalization of URKE. The basic functionality of the two primitives is the same: Sessions involve two parties, A and B, where A can establish keys and safely share them with B by providing the latter with ciphertexts. In 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

[9] A different approach to achieve a constant-size state is to replace lines 10 and 20 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.

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 $c \in \mathcal{C}$, 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

$$
\begin{array}{rcl}
 & \mathrm{init} \rightarrow_{\$} & \mathcal{S}_A \times \mathcal{S}_B \\
\mathcal{S}_A \times \mathcal{AD} \rightarrow & \mathrm{snd}_A \rightarrow_{\$} & \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\
\mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} \rightarrow & \mathrm{rcv}_B \rightarrow & \mathcal{S}_B \times \mathcal{K} \;/\; \perp \\
\mathcal{S}_B \times \mathcal{AD} \rightarrow & \mathrm{snd}_B \rightarrow_{\$} & \mathcal{S}_B \times \mathcal{C} \\
\mathcal{S}_A \times \mathcal{AD} \times \mathcal{C} \rightarrow & \mathrm{rcv}_A \rightarrow & \mathcal{S}_A \;/\; \perp
\end{array}
$$



Correctness of SRKE. Our definition of SRKE functionality is via game FUNC in Fig. 4. We say scheme R is *correct* if $\Pr[\mathrm{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ for all adversaries $\mathcal{A}$. In the figure, the lines of code tagged with a '$\cdot$' right after the line number also appear in the URKE FUNC game (Fig. 1). In comparison with that game, 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 SndA, 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 35, 36). Given that the specifications of oracles SndA and RcvB of Figs. 1 and 4 coincide (with one exception: lines 13, 21 are guarded by in-sync checks (in lines 12, 20) 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[10]), and that also the specifications of oracles SndB and RcvA of Figs. 4 are quite similar to them (besides the reversion of the direction of communication, the difference is that all session-key related components were stripped off), the logics of the FUNC game in Fig. 4 should be clear. Overall, like in the URKE case, the correctness requirement boils down to declaring the adversary successful, in line 31, if A and B compute different keys while still being in-sync.

Epochs. 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, for example, 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

[10] This approach is borrowed from [6,11].

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

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

Oracle $\mathrm{SndB}(ad)$
18 Require $S_B \neq \bot$
19 $(S_B, c) \leftarrow_{\$} \mathrm{snd}_B(S_B, ad)$
20 If is_B:
21 $adc_B[s_B] \leftarrow (ad, c)$
22 $E_B^{\dashv} \leftarrow E_B^{\dashv} + 1$
23 $s_B \leftarrow s_B + 1$
24 Return c

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

Oracle $\mathrm{RcvA}(ad, c)$
34 Require $S_A \neq \bot$
35 If $is_A \wedge adc_B[r_A] \neq (ad, c)$:
36 $is_A \leftarrow \mathtt{F}$
37 If is_A: $e_A \leftarrow e_A + 1$
38 $S_A \leftarrow \mathrm{rcv}_A(S_A, ad, c)$
39 If $S_A = \bot$: Return $\bot$
40 $r_A \leftarrow r_A + 1$
41 Return

Fig. 4. Game FUNC for SRKE scheme R. The lines of code tagged with a '·' also appear in the URKE FUNC game. Note that the variables $e_A, \mathrm{EP}_A, E_B^{\vdash}, E_B^{\dashv}$ do not influence the the game outcome.

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 refreshes. 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 refreshes 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

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 later epoch.

We represent the span of epochs supported by B with the interval variable E_B (see Sect. 2.1): 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) referring 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 22), each receiving operation of A increases the latter's awareness of epochs supported by B (line 37), 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 obsolete ones (line 28). Observe that tracking epochs is not meaningful after participants get out-of-sync; we thus guard lines 28, 37 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 $\mathrm{KIND}^b_{\mathsf{R}}$, for challenge bit $b \in \{0,1\}$, are specified in Fig. 5. 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.

The new KIND games are the natural amalgamation of the (URKE) KIND games of Fig. 2 with the (SRKE) FUNC game of Fig. 4 (with the exceptions discussed below). Concerning the trivial attack conditions on URKE that we identified in Sect. 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 Figs. 2 and 5, 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[11]. 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

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

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

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

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

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

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

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

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

Oracle $\mathrm{Reveal}(u, i)$
as in URKE (Fig. 2)

Oracle $\mathrm{Challenge}(u, i)$
as in URKE (Fig. 2)

Fig. 5. Games KIND^b, $b \in \{0, 1\}$, for SRKE scheme R. Lines of code tagged with a '·' similarly appear in the SRKE FUNC game in Fig. 4.

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 in Sect. 3 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'; $S_B^* \leftarrow \mathrm{ExposeB}()$; $(S_B^*, c) \leftarrow_{\$} \mathrm{snd}_B(S_B^*, ad)$; $\mathrm{RcvA}(ad, c)$; $c' \leftarrow \mathrm{SndA}(ad')$; $(S_B^*, k) \leftarrow \mathrm{rcv}_B(S_B^*, ad', c')$; $k' \leftarrow \mathrm{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; 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 Fig. 6. 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 Sect. 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 27; obsolete keys of expired epochs are deleted by rcv_B in line 47). 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).

Proc init
00 $(sgk, vfk) \leftarrow_\$ \mathrm{gen}_\mathsf{S}$
01 · $(sk, pk) \leftarrow_\$ \mathrm{gen}_\mathsf{K}$
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 $PK[\cdot] \leftarrow \perp$; $PK[0] \leftarrow pk$
06 $SK[\cdot] \leftarrow \perp$; $SK[0] \leftarrow sk$
07 $L_A[\cdot] \leftarrow \perp$; $L_B[\cdot] \leftarrow \perp$; $L_A[0] \leftarrow \diamond$
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 $\mathrm{snd}_A(S_A, ad)$
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 · $(k, c) \leftarrow_\$ \mathrm{enc}(PK[e'])$
15 $k^* \xleftarrow{\shortparallel} k$; $C \xleftarrow{\shortparallel} c$
16 · $\tau \leftarrow_\$ \mathrm{tag}(k.m, ad \,\|\, C)$
17 · $C \xleftarrow{\shortparallel} \tau$; $t \xleftarrow{\shortparallel} \triangleright \,\|\, ad \,\|\, C$
18 · $k.o \,\|\, K \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(K, k^*, t)$
19 · $pk \leftarrow \mathrm{gen}_\mathsf{K}(sk)$
20 $PK[..., (E^\dashv - 1)] \leftarrow \perp$; $PK[E^\dashv] \leftarrow pk$
21 $E^\vdash \leftarrow E^\dashv$; $s \leftarrow s + 1$; $L[s] \leftarrow ad \,\|\, C$
22 $S_A \leftarrow (PK, E, s, L, vfk, K, k.m, t)$
23 Return $(S, k.o, C)$

Proc $\mathrm{snd}_B(S_B, ad)$
24 $(SK, E, r, L, sgk, K, k.m, t) \leftarrow S_B$
25 $(sk^*, pk^*) \leftarrow_\$ \mathrm{gen}_\mathsf{K}$
26 $(sgk^*, vfk^*) \leftarrow_\$ \mathrm{gen}_\mathsf{S}$
27 $E^\dashv \leftarrow E^\dashv + 1$; $SK[E^\dashv] \leftarrow sk^*$
28 $C \leftarrow r \,\|\, pk^* \,\|\, vfk^*$
29 $\sigma \leftarrow_\$ \mathrm{sgn}(sgk, ad \,\|\, C)$
30 $C \xleftarrow{\shortparallel} \sigma$; $L[E^\dashv] \leftarrow \triangleleft \,\|\, ad \,\|\, C$
31 $S_B \leftarrow (SK, E, r, L, sgk^*, K, k.m, t)$
32 Return (S_B, C)

Proc $\mathrm{rcv}_B(S_B, ad, C)$
33 $(SK, E, r, L, sgk, K, k.m, t) \leftarrow S_B$
34 $t^* \leftarrow ad \,\|\, C$; $C \,\|\, \tau \leftarrow C$
35 · Require $\mathrm{vfy}_\mathsf{M}(k.m, ad \,\|\, C, \tau)$
36 $k^* \leftarrow \epsilon$; $e \,\|\, C \leftarrow C$
37 Require $E^\vdash \le e \le E^\dashv$
38 $t \xleftarrow{\shortparallel} L[E^\vdash + 1] \,\|\, \ldots \,\|\, L[e]$
39 $L[..., e] \leftarrow \perp$
40 For $e' \leftarrow E^\vdash$ to e:
41 $c \,\|\, C \leftarrow C$
42 · $k \leftarrow \mathrm{dec}(SK[e'], c)$
43 · Require $k \neq \perp$
44 $k^* \xleftarrow{\shortparallel} k$
45 $t \xleftarrow{\shortparallel} \triangleright \,\|\, t^*$
46 · $k.o \,\|\, K \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(K, k^*, t)$
47 $SK[..., (e-1)] \leftarrow \perp$; $SK[e] \leftarrow sk$
48 For $e' \leftarrow e + 1$ to $E^\dashv$:
49 $SK[e'] \leftarrow \mathrm{up}(SK[e'], t^*)$
50 $E^\vdash \leftarrow e$; $r \leftarrow r + 1$
51 $S_B \leftarrow (SK, E, r, L, sgk, K, k.m, t)$
52 Return $(S_B, k.o)$

Proc $\mathrm{rcv}_A(S_A, ad, C)$
53 $(PK, E, s, L, vfk, K, k.m, t) \leftarrow S_A$
54 $t \xleftarrow{\shortparallel} \triangleleft \,\|\, ad \,\|\, C$; $C \,\|\, \sigma \leftarrow C$
55 Require $\mathrm{vfy}_\mathsf{S}(vfk, ad \,\|\, C, \sigma)$
56 $r \,\|\, pk^* \,\|\, vfk \leftarrow C$
57 Require $L[r] \neq \perp$
58 $L[..., (r-1)] \leftarrow \perp$; $L[r] \leftarrow \diamond$
59 For $s' \leftarrow r + 1$ to s:
60 $pk^* \leftarrow \mathrm{up}(pk^*, L[s'])$
61 $E^\dashv \leftarrow E^\dashv + 1$; $PK[E^\dashv] \leftarrow pk^*$
62 $S_A \leftarrow (PK, E, s, L, vfk, K, k.m, t)$
63 Return S_A

Fig. 6. Construction of a SRKE scheme from a key-updatable KEM $\mathsf{K} = (\mathrm{gen}_\mathsf{K}, \mathrm{enc}, \mathrm{dec})$, a message authentication code $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_\mathsf{M})$, a one-time signature scheme $\mathsf{S} = (\mathrm{gen}_\mathsf{S}, \mathrm{sgn}, \mathrm{vfy}_\mathsf{S})$, 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, 58: 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 38: If $E^\vdash = e$ then no value shall be concatenated to t. Line 41: The last iteration of the loop is meant to clear C; a more precise version of the line would say "If $e' < e$ then $c \,\|\, C \leftarrow C$ else $c \leftarrow C$". Lines 17, 45, 54, 30: We use labels $\triangleright$ and $\triangleleft$ in transcript fragments to distinguish whether they emerged in the A-to-B or B-to-A direction. Lines of code tagged with a '$\cdot$' depict the URKE construction's core.

Secure state update. Assume A executes once the snd_A algorithm, then twice the rcv_A algorithm, and then again once the snd_A algorithm. That is, following the above sketch of 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, 12 of Fig. 3), 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.[12,13] The algorithms implement this by including in A's state the array $PK[\cdot]$ in which rcv_A stores incoming public keys (line 61) and which snd_A consults when establishing outgoing ciphertexts (lines 13–15; the counterpart on B's side is in lines 40–44). 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,[14] 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 48–49 (note that t^* is the current transcript fragment, see line 34). Of course A has to synchronize on B's key updates, which she does in lines 59–60, 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 58.) 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 56, 28).

[12] We discuss why it is unsafe to encapsulate to only a subset of the keys in Appendix A.3.

[13] The concatenation of keys of an OW secure KEM can be seen as the implementation of a secure combiner in the spirit of [8].

[14] We discuss this further in Appendix A.2.

Transcript management. Recall that one element of the participants' state in our URKE scheme (in Fig. 3) 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 start diverging in the moment 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 54), when creating the ciphertexts, B will just record them in his state variable $L[\cdot]$ (line 30), 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 38; obsolete entries are removed in line 39). The A-to-B direction is simpler[15] 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 34, 45). 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, 30).

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, a MAC is still sufficient for the A-to-B direction (lines 16, 35), 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,[16] 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 26, 28, 29, 55, 56).

The only lines we did not comment on are 18, 19, 25, 46 — those that also form the core of our URKE protocol (which are discussed in Sect. 4).

Practicality of our construction. We remark that the number of updates per kuKEM key pair is bounded by the number of ciphertexts sent by A during one round-trip time (RTT) on the network between A and B (intuitively by the number of ciphertexts sent by A that *cross* the wire with one epoch update ciphertext from B). Ciphertexts that B did not know of when proposing an epoch (1/2 RTT) and ciphertexts A sent until she received the epoch proposal (1/2 RTT) are regarded for an update of a key pair. As a result, the hierarchy of an HIBE can be bounded by this number of ciphertexts when used for building a kuKEM for SRKE.

[15] 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.

[16] Note this is not an issue in the A-to-B 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 A.1.

Theorem 2 (informal). *The SRKE protocol* R *from Fig. 6 offers key indistinguishability if function* H *is modeled as a random oracle, the kuKEM provides* KUOW *security, the one-time signature scheme provides* SUF *security, the MAC provides* SUF *security, and the session-key space of the kuKEM is sufficiently large.*

The exact theorem statement and the respective proof are in the full version [14]. The approach of the proof 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 From URKE and SRKE to BRKE

In Sects. 3–6 we proposed security models and constructions for URKE and SRKE. For space reasons we defer the corresponding formalizations for BRKE (bidirectional RKE) to the full version [14]. Here we quickly sketch how one can obtain notions and constructions for the latter from the former.

The syntax, correctness, and security definitions for BRKE can be seen as an amalgamation of two copies of the corresponding definitions for SRKE, one in each direction of communication. Fortunately, several of the game variables can be unified so that the games remain relatively compact.

The same type of amalgamation can be applied to obtain a BRKE construction: While just running two generic SRKE instances side by side (in reverse directions) is not sufficient to obtain a secure solution, carefully binding them together, in our case with one-time signatures as an auxiliary tool, is. More precisely, each BRKE send operation results in (1) the creation of a fresh one-time signature key pair, (2) the invocation of the two SRKE send routines (the one in the A-to-B and the other in the B-to-A direction) where the signature verification key is provided as associated data, (3) encoding the verification key and the two SRKE ciphertexts into a single ciphertext and securing the latter with a signature. See [14] for the details.

Acknowledgments. We thank Fabian Weißberg for very inspiring discussions at the time we first explored the topic of ratcheted key exchange. We further thank Giorgia Azzurra Marson and anonymous reviewers for comments and feedback on the article. (This holds especially for a EUROCRYPT 2018 reviewer who identified an issue in a prior version of our URKE construction.) Bertram Poettering conducted part of the work at Ruhr University Bochum supported by ERC Project ERCC (FP7/615074). Paul Rösler received support by SyncEnc, funded by the German Federal Ministry of Education and Research (BMBF, FKZ: 16KIS0412K).

References

1. 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, Cham (2017). https://doi.org/10.1007/978-3-319-63697-9_21
2. Bellare, M., Yee, B.: Forward-security in private-key cryptography. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 1–18. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36563-X_1
3. 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 WPES 2004, Washington, DC, USA, 28 October 2004, pp. 77–84. ACM (2004)
4. 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 EuroS&P 2017, Paris, France, 26–28 April 2017, pp. 451–466. IEEE (2017)
5. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: IEEE CSF 2016, Lisbon, Portugal, 27 June–1 July 2016, pp. 164–178. IEEE Computer Society (2016)
6. Eugster, P.T., Marson, G.A., Poettering, B.: A cryptographic look at multi-party channels. In: 31st IEEE Computer Security Foundations Symposium (2018, to appear)
7. Gentry, C., Silverberg, A.: Hierarchical ID-based cryptography. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 548–566. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36178-2_34
8. Giacon, F., Heuer, F., Poettering, B.: KEM combiners. In: Abdalla, M., Dahab, R. (eds.) PKC 2018, Part I. LNCS, vol. 10769, pp. 190–218. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-76578-5_7
9. Langley, A.: Source code of Pond, May 2016. https://github.com/agl/pond
10. Marlinspike, M., Perrin, T.: The double Ratchet algorithm, November 2016. https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf
11. Marson, G.A., Poettering, B.: Security notions for bidirectional channels. IACR Trans. Symm. Cryptol. **2017**(1), 405–426 (2017)
12. 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
13. Off-the-record messaging (2016). http://otr.cypherpunks.ca
14. Poettering, B., Rösler, P.: Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296 (2018). https://eprint.iacr.org/2018/296

15. Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) ACM CCS 2002, Washington D.C., USA, 18–22 November 2002, pp. 98–107. ACM Press (2002)
16. Rösler, P., Mainka, C., Schwenk, J.: More is less: on the end-to-end security of group chats in Signal, WhatsApp, and Threema. In: IEEE EuroS&P 2018 (2018)
17. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Trans. Inf. Syst. Secur. **2**(2), 159–176 (1999)
18. 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, San Jose, CA, USA, 17–21 May 2015, pp. 232–249. IEEE Computer Society Press (2015)
19. 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 Rationale for SRKE 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.

A.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 $\mathrm{KIND}_{\mathsf{R}}$ 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 Fig. 5 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 $\mathrm{KIND}_{\mathsf{R}}$ game, the adversary must not obtain information on her future session keys (see Fig. 5 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).

A.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 Fig. 5 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 full version [14]).

A.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 ($C_{B1} \leftarrow_\$ \mathrm{SndB}(\epsilon)$; $\mathrm{RcvA}(C_{B1}, \epsilon)$). 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 ($(S'_A, k', C') \leftarrow_\$ \mathrm{snd}_A(S_A, \epsilon)$; $\mathrm{RcvB}(C', \epsilon)$; $S'_B \leftarrow \mathrm{ExposeB}$). 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 ($(S''_B, C'') \leftarrow_\$ \mathrm{snd}_B(S'_B, \epsilon)$; $\mathrm{RcvA}(C'', \epsilon)$). 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 ($C_{A1} \leftarrow_\$ \mathrm{SndA}(\epsilon)$; $k_b \leftarrow_\$ \mathrm{Challenge}(A, 1)$).

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 the full version [14]). 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 ($(S'''_B, C''') \leftarrow_\$ \mathrm{snd}_B(S''_B, \epsilon)$; $\mathrm{RcvA}(C''', \epsilon)$). 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 ($C_{A2} \leftarrow_\$ \mathrm{SndA}(\epsilon)$; $k_{b2} \leftarrow_\$ \mathrm{Challenge}(A, 2)$).

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 KIND_R game).