Determining the Core Primitive for Optimally Secure Ratcheting

Fatih Balli[1], Paul Rösler[2], Serge Vaudenay[1]

[1] LASEC, École polytechnique fédérale de Lausanne
`{firstname.lastname}@epfl.ch`
[2] Chair for Network and Data Security, Ruhr University Bochum
`paul.roesler@rub.de`

Abstract. After ratcheting attracted attention mostly due to practical real-world protocols, recently a line of work studied ratcheting as a primitive from a theoretic point of view. Literature in this line, pursuing the strongest security of ratcheting one can hope for, utilized for constructions strong, yet inefficient key-updatable primitives – based on hierarchical identity based encryption (HIBE). As none of these works formally justified utilizing these building blocks, we answer the yet open question under which conditions their use is actually *necessary.*

We revisit these strong notions of ratcheted key exchange (RKE), and propose a more realistic (and slightly stronger) security definition. In this security definition, both the exposure of the communicating parties' local states *and* the adversary's ability to attack the executions' randomness are considered. While these two attacks were partially considered in previous work, we are the first to unify them cleanly in a natural game based notion.

Our definitions are based on the systematic RKE notion by Poettering and Rösler (CRYPTO 2018). Due to slight (but meaningful) changes to regard attacks against randomness, we are ultimately able to show that, in order to fulfill strong security for RKE, public key cryptography with (independently) updatable key pairs is a necessary building block. Surprisingly, this implication already holds for the simplest RKE variant (which was previously instantiated with only standard public key cryptography).

Hence, (1) we model optimally secure RKE under randomness manipulation to cover realistic attacks, (2) we (provably) extract the core primitive that is necessary to realize strongly secure RKE, and (3) our results indicate under which conditions this primitive is necessary for strongly secure ratcheting and which relaxations in security allow for constructions that only rely on standard public key cryptography.

1 Introduction

The term "ratcheting" as well as the underlying concept of continuously updating session secrets for secure long-term communication settings originates from real-world messaging protocols [16,18,5,21,17]. In these protocols, first forward-secrecy [18,5,21] and later security after state exposures [17] (also known as future secrecy, backward secrecy, or post-compromise security) were aimed to be achieved as the exposure of the devices' local states was considered a practical threat. The main motivation behind this consideration is the typical lifetime of sessions in messaging apps. As messaging apps are nowadays usually run on smartphones, the lifetime of messaging sessions is proportional to the ownership duration of a smartphone (typically several years). Due to the long lifetime of sessions and the mobile use of smartphones, scenarios, in which the local storage – containing the messaging apps' secret state – can be exposed to an attacker, are extended in comparison to use cases of other cryptographic protocols.

Practical Relevance of Randomness Manipulation

In addition to exposures of locally stored state secrets, randomness for generating (new) secrets is often considered vulnerable. This is motivated by numerous attacks in practice against randomness sources (e.g., [11]), randomness generators (e.g., [23,7]), or exposures of random coins (e.g., [22]). Most theoretic approaches try to model this threat by allowing an adversary to *reveal* attacked random coins of a protocol execution (as it was also conducted in related work on ratcheting). This, however, assumes that the attacked protocol honestly and uniformly samples its random coins (either from a high-entropy source or using a random oracle) and that these coins are only afterwards leaked to the attacker. In contrast, practically relevant attacks against bad randomness generators or low-entropy sources (e.g., [11,23,7]) change the distribution from which random coins are sampled. Consequently, this threat is only covered by a security model if considered adversaries are also allowed to *influence* the execution's (distribution of) random coins. Thus, it is important to consider randomness *manipulation* (instead of reveal), if attacks against randomness are regarded practically relevant.

The overall goal of ratcheting protocols is to reduce the effect of any such non-permanent and/or non-fatal attack to a minimum. For example, an ongoing communication under a non-fatal attack should become secure as soon as the adversary ends this attack or countermeasures become effective. Examples for countermeasures are replacing bad randomness generators via software updates, eliminating state exposing viruses, etc. Motivated by this, most widely used messaging apps are equipped with mechanisms to regularly update the local secrets such that only a short time frame of communication is compromised if an adversary was successful due to obtaining local secrets and/or attacking random coins.

Real-World Protocols

The most prominent and most widely deployed real-world ratcheting protocol is the Signal protocol (used by the Signal Messenger, WhatsApp, Facebook Messenger, Skype, and others). The analysis of this protocol in a multi-stage key agreement model[1] [8] was the first theoretic treatment of ratcheting in the literature. Cohn-Gordon et al. [8], however, focus on grasping the precise security

[1] Please note our distinction between *key agreement* and *ratcheted key exchange* protocols. The former is run by parties to obtain a symmetric key for a subsequent session

that Signal offers rather than generically defining ratcheting as an independent primitive. While the security provided by Signal is sufficient in most real-world scenarios, we focus in this work on the theoretic analysis of the (optimally secure) primitive ratcheting with respect to its instantiability by smaller building blocks.

Generic Treatment of Ratcheting as a Primitive
In the following we shortly introduce and review previous modeling approaches for strongly secure ratcheting. We thereby abstractly highlight modeling choices that crucially affect the constructions, secure according to these models respectively. Specifically, we indicate why some models can be instantiated with only public key cryptography (PKC) – bypassing our implication result – and others cannot. In Table 1 we summarize this overview.

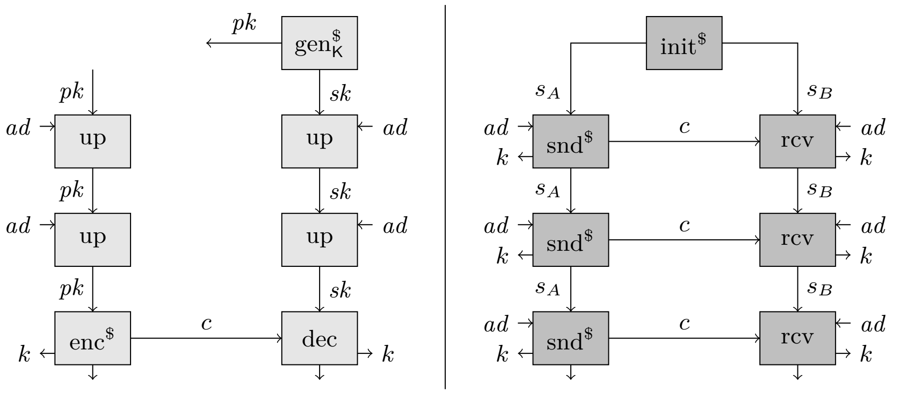


Fig. 1: Conceptual depiction of kuKEM* (on the left) and unidirectional RKE (on the right). '\$' in the upper index of an algorithm name denotes that the algorithm runs probabilistically and *ad* is associated data.

The initial generic work that considers ratcheted key exchange (RKE) as a primitive and defines its syntax, correctness, and security (in a yet impractical variant) is by Bellare et al. [4]. Abstractly, their concept of ratcheted key exchange, depicted in the right part of Figure 1, consist of an initialization that provides two session participants A and B with a state that can then be used by them to repeatedly compute new keys in this session (e.g., for use in higher level protocols). In their restricted communication model, A is allowed to compute new keys with her state and accordingly send ciphertexts to B who can then compute (the same) keys with his state. During these key computations, A's and B's states are updated respectively (to minimize the effect of state exposures). As B can only comprehend key computations from A (on receipt of a ciphertext) but cannot actively initiate the computation of new keys, this variant was

protocol. The latter is the session protocol that might utilize the initial key and that continuously outputs symmetric keys in the session independent of long-term keys.

later called unidirectional RKE [20]. Beyond this restriction of the communication model, the security definition by Bellare et al. only allows the adversary to expose A's temporary local state secrets, while B's state cannot be exposed (which in turn requires no forward-secrecy with respect to state updates by B). Following Bellare et al., Poettering and Rösler [20,19][2] propose a revised security definition of unidirectional RKE (URKE: allowing also the exposure of B's state) and extend the communication model to define syntax, correctness, and security of *sesqui*directional RKE (SRKE: additionally allows B to only send special update ciphertexts to A that do not trigger a new key computation but help him to recover from state exposures) and *bi*directional RKE (BRKE: defines A and B to participate equivalently in the communication). With a similar instantiation as Poettering and Rösler, Jaeger and Stepanovs [13] define security for bidirectional channels under state exposures and randomness reveal.

All of the above mentioned works define security *optimally* with respect to their syntax definition and the adversary's access to the primitive execution (modeled via oracles in the security game). This is reached by declaring secrets insecure *iff* the adversary conducted an unpreventable/trivial attack against them (i.e., a successful attack that no instantiation can prevent). Consequently, fixing syntax and oracle definitions, no stronger security definitions exist.

Relaxed Security Notions

Subsequent to these strongly secure ratcheting notions, multiple weaker formal definitions for ratcheting were proposed that consider special properties such as strong explicit authentication [10], out of order receipt of ciphertexts [2], or primarily target on allowing efficient instantiations [15,6].

While these works are syntactically similar, we shortly sketch their different relaxations regarding security – making their security notions sub-optimal. Durak and Vaudenay [10] and Caforio et al. [6] forbid the adversary to perform impersonation attacks against the communication between A and B during the establishment of a *secure* key. Thus, they do not require recovery from state exposures – which are a part of impersonation attacks – in all possible cases, which we denote as "partial recovery" (see Table 1). Furthermore, both works neglect bad randomness as an attack vector. In the security experiments by Jost et al. [15] and Alwen et al. [2] constructions can delay the recovery from attacks longer than necessary (Jost et al. therefore temporarily forbid the exposure of the local state). Additionally, they do not require the participants' states to become incompatible (immediately) on active attacks against the communication.

Instantiations of Ratcheting

Interestingly, both mentioned *uni*directional RKE instantiations that were defined to depict optimal security [4,20] as well as bidirectional real-world exam-

[2] We explicitly cite the extended version [19] for results that are not captured in the CRYPTO 2018 proceedings [20].

[1] '*Unnecessary*' refers to restrictions beyond those that are immediately implied by optimal security definitions (that only restrict the adversary with respect to unpreventable/trivial attacks).

	(a) Interaction	**(b) State Exposure**	**(c) Bad Randomness**	**(d) Recovery**
C+ [8]	$\leftrightarrow$	Always allowed	Reveal	Delayed
B+ [4]	$\rightarrow$	Only allowed for A	Reveal	Immediate
PR [20]	$\rightarrow$	Always allowed	Not considered	Immediate
	$\mapsto$	Always allowed	Not considered	Immediate
	$\leftrightarrow$	Always allowed	Not considered	Immediate
JS [13]	$\leftrightarrow$	Always allowed	Reveal	Immediate
DV [10]	$\leftrightarrow$	Always allowed	Not considered	Partial
JMM [15]	$\rightarrow$	Partially restricted	Reveal	(Immediate)
	$\mapsto$	Partially restricted	Reveal	(Immediate)
	$\leftrightarrow$	Partially restricted	Reveal	(Immediate)
ACD [2]	$\leftrightarrow$	Always allowed	Manipulation	Delayed
CDV [6]	$\leftrightarrow$	Always allowed	Not considered	Delayed
This work	$\rightarrow$	Always allowed	Manipulation	Immediate

Table 1: Differences in security notions of ratcheting regarding (a) uni- ($\rightarrow$), sesqui- ($\mapsto$), and bidirectional ($\leftrightarrow$) interaction between A and B, (b) when the adversary is allowed to expose A's and B's state (or when this is *unnecessarily* restricted), (c) the adversary's ability to reveal or manipulate algorithm invocations' random coins, and (d) how soon and how complete recovery from these two attacks into a secure state is required of *secure* constructions (or if *unnecessary* delays or exceptions for recovery are permitted).[1] Recovery from attacks required by Jost et al. [15] is *immediate* in so far as their restrictions of state exposures introduce delays implicitly. Gray marked cells indicate the reason (i.e., relaxations in security) why respective instantiations can rely on standard PKC only (circumventing our implication result). Rows without gray marked cells have no construction based on pure PKC.

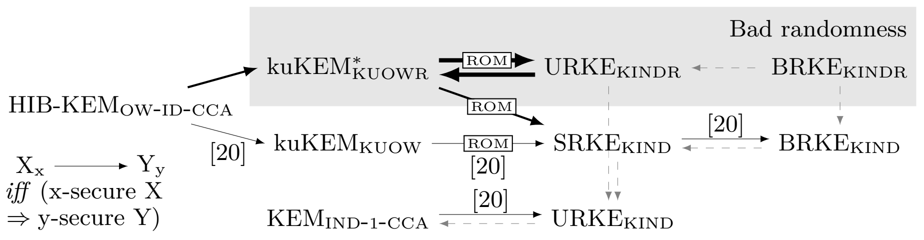
ples such as the Signal protocol (analyzed in [8]), and instantiations of the above named relaxed security notions [10,15,2,6] only rely on standard PKC (cf. rows in Table 1 with gray cells).

In contrast, both mentioned optimally secure bidirectional ratcheting variants (i.e., sesquidirectional and bidirectional RKE [20], and bidirectional strongly secure channel [13]) are based on a strong cryptographic building block, called *key-updatable public key encryption*, which can be built from hierarchical identity based encryption (HIBE). Intuitively, key-updatable public key encryption is standard public key encryption that additionally allows to update public key and secret key independently with respect to some associated data (a conceptual depiction of this is on the left side of Figure 1). Thereby an updated secret key cannot be used to decrypt ciphertexts that were encrypted to previous (or different) versions of this secret key (where versions are defined over the associated data used for updates).

We emphasize a significant difference between key-updatable public key encryption and HkuPke (introduced by Jost et al. [15] and recently used by Alwen et al. [3,1]): in HkuPke key updates rely on interactive communication between holders of public key and secret key, and associated data for key updates is not fully adversary-controlled. These two differences make it a strictly weaker primitive, insufficient for optimal security of RKE (on which we further elaborate in Section 3).

Necessity for Strong Building Blocks

Natural questions that arise from this line of work are, whether and under which conditions such strong (HIBE-like) building blocks are not only sufficient but also necessary to instantiate the strong security of (bidirectional) RKE. In order to answer these questions, we build key-updatable public key cryptography from ratcheted key exchange. Consequently we affirm the necessity and provide (sufficient) conditions for relying on these strong building blocks. We therefore minimally adjust the syntax of key-updatable key encapsulation mechanism (kuKEM) [20] and consider the manipulation of algorithm invocations' random coins in our security definitions of kuKEM and RKE.[2]



Bad randomness
$\text{kuKEM}^*_{\text{KUOWR}}$
ROM
$\text{URKE}_{\text{KINDR}}$
$\text{BRKE}_{\text{KINDR}}$
$\text{HIB-KEM}_{\text{OW-ID-CCA}}$
ROM
[20]
$\text{kuKEM}_{\text{KUOW}}$
ROM
[20]
$\text{SRKE}_{\text{KIND}}$
[20]
$\text{BRKE}_{\text{KIND}}$
$X_x \longrightarrow Y_y$
iff (x-secure X
⇒ y-secure Y)
[20]
$\text{KEM}_{\text{IND-1-CCA}}$
$\text{URKE}_{\text{KIND}}$

Fig. 2: The contributions of this paper (bold arrows) and their connection to previous work (thin arrows) involving RKE (uni-, sesqui-, and bidirectional) and KEM (standard, hierarchical-identity-based, and key-updatable) primitives. ROM indicates that the proof holds in the random oracle model. $\text{kuKEM}^*_{\text{KUOWR}} \Rightarrow_{\text{ROM}} \text{SRKE}_{\text{KIND}}$ is not formally proven in this paper, but we point out that the proof of $\text{kuKEM}_{\text{KUOW}} \Rightarrow_{\text{ROM}}$ $\text{SRKE}_{\text{KIND}}$ from [20] can be rewound. Gray dashed connections indicate trivial implications (due to strictly weaker syntax or security definitions).

While, despite these changes of syntax and security towards prior definitions, we prove that RKE can still be built from kuKEM, we also prove that kuKEM can be built from RKE (see Figure 2). As a result we show that:

- kuKEM* (with one-way security under manipulation of randomness)[3] $\Rightarrow_{\text{ROM}}$ Unidirectional RKE (with key indistinguishability under manipulation of randomness),
- Unidirectional RKE (with key indistinguishability under manipulation of randomness) ⇒ kuKEM* (with one-way security under manipulation of randomness).

[2] Recall that randomness manipulation was not considered in a security definition that aimed for optimal security in the literature of ratcheting yet (cf. Table 1).

[3] The asterisk at kuKEM* indicates the minimal adjustment to the kuKEM syntax definition from [20]. For the kuKEM* we consider one-way security as it suffices to achieve strong security for RKE. It is obvious that the same results hold for key indistinguishability.

Given the security notions established in honest randomness setting and their connections to each other, one would also expect

- Group RKE $\Rightarrow$ Bidirectional RKE $\Rightarrow$ Sesquidirectional RKE $\Rightarrow$ Unidirectional RKE

to follow. Hence, our results indicate that stronger RKE variants also likely require building blocks as hard as kuKEM*. Furthermore, due to our results, it becomes clear that: One-way security under manipulation of randomness of kuKEM* $\Rightarrow_{\text{ROM}}$ Key indistinguishability of *sesquidirectional* RKE. Interestingly, these results induce that (when considering strong security) ratcheted key exchange requires these strong (HIBE-like) building blocks not only for bidirectional communication settings, but already for the unidirectional case. Both mentioned previous unidirectional RKE schemes can bypass our implication because they forbid exposures of B's state [4] or assume secure randomness [20] (see Table 1). We describe attacks against each of both constructions in our security definition in Appendix C. Since the mentioned relaxed security definitions of ratcheting [8,10,15,2,6] restrict the adversary more than necessary in exposing states, solving (potentially valid) embedded game challenges, manipulating the communication between the session participants, or attacking invocations' random coins (and thus violate either of our security definition's conditions), it remains feasible to instantiate them with standard public key primitives as well (see Table 1). Although our analysis was partially motivated by the use of kuKEM in [20,13], we do not ultimately answer whether these particular constructions necessarily relied on it. Rather we provide a clean set of conditions under which RKE and kuKEM clearly imply each other as we do not consider the justification of previous constructions but a clear relation for future work important.

Thus, we show that sufficient conditions for necessarily relying on kuKEM as a building block of RKE are: (a) unrestricted exposure of both parties' local states, (b) consideration of attacks against algorithm invocations' random coins, and (c) required immediate recovery from these two attacks into a secure state by the security definition (i.e., the adversary is only restricted with respect to unpreventable/trivial attacks).[4]

Contributions

The contributions of our work can be summarized as follows:

- We are the first who systematically define optimal security of key-updatable KEM and unidirectional RKE under randomness manipulation (in sections 3 and 4) and thereby consider this practical threat in addition to state exposures in an instantiation-independent notion of RKE. Thereby we substantially enhance the respective models by Poettering and Rösler [20].

[4] Note that there may exist further sets of sufficient conditions for relying on kuKEMs since, for example, sesqui- and bidirectional RKE by Poettering and Rösler [20,19] violate condition (b) but base on kuKEMs as well. We refer the reader to Appendix B.2 in [19] for a detailed explanation of why their scheme presumably also must rely on a kuKEM. We leave the identification of further sets of conditions as future work.

- In Section 5, we construct unidirectional RKE generically from a kuKEM* to show that the latter suffices as a building block for the former under manipulation of randomness.
- To show that kuKEM* is not only sufficient but also necessary to build unidirectional RKE (under randomness manipulation), we provide a construction of kuKEM* from a generic unidirectional RKE scheme in Section 6.

With our results we distill the core building block of strongly secure ratcheted key exchange down to its syntax and security definition. This allows further research to be directed towards instantiating kuKEM* schemes that are more familiar and easier in terms of security requirements, rather than attempting to construct seemingly more complex RKE primitives.[5] Simultaneously, our results indicate the cryptographic hardness of ratcheted key exchange and thereby help to systematize and comprehend the security definitions and different dimensions of ratcheting in the literature. As a consequence, our results contribute to a fact-based trade-off between security and efficiency for RKE by providing requirements for relying on heavy building blocks and thereby revealing respective bypasses.

2 Preliminaries

2.1 Notation

By $x \leftarrow y$ we define the assignment of the value of variable y to variable x and thus for a function X, $x \leftarrow \mathrm{X}(y)$ means that x is assigned with the evaluation output of X on input y. We define $\mathtt{T}, \mathtt{F}$ as Boolean values for true and false. The shortcut notion $w \leftarrow x \mathbin{\texttt{?}} y \mathbin{\texttt{:}} z$ means that 'if $x = \mathtt{T}$, then $w \leftarrow y$, otherwise $w \leftarrow z$'. For a probabilistic algorithm Y, $x \leftarrow_\$ \mathrm{Y}(y)$ denotes the probabilistic evaluation of Y on input y with output x and $x \leftarrow \mathrm{Y}(y; r)$ denotes the deterministic evaluation of Y on y with output x where the evaluation's randomness is fixed to r. For a set $\mathcal{X}$, $x \leftarrow_\$ \mathcal{X}$ is the uniform random sampling of value x from $\mathcal{X}$. We use the shortcut notion $\mathcal{X} \overset{\cup}{\leftarrow} \mathcal{Y}$ to denote the union $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{Y}$ of sets $\mathcal{X}$ and $\mathcal{Y}$.

Symbol 'ϵ' denotes an empty string and symbol '$\perp$' denotes an undefined element or an output that indicates rejections (thus it is not an element of explicitly defined sets).

By $\mathcal{X}^*$, we denote the set of all lists of arbitrary size whose elements belong to $\mathcal{X}$. We abuse the notation of empty string 'ϵ' by writing $L = \epsilon$ for an empty

[5] For example, the bidirectional channel construction in the proceedings version of [13] is not secure according to the security definition (but a corrected version is published as [14]), in the acknowledgments of [19] it is mentioned that an early submitted version of their construction was also flawed, and for an earlier version of [10] we detected during our work (and informed the authors) that the construction was insecure under bad randomness such that the updated proceedings version (also available as [9]) disregards attacks against randomness entirely. Finally, we detected and reported that the construction of HkuPke in [15] is not even correct.

list L. If an element $x \in \mathcal{X}$ is appended to list L then we denote this by $L \leftarrow L \| x$ (or simply $L \overset{\shortparallel}{\leftarrow} x$). Thus, '$\|$' denotes a special concatenation symbol that is not an element of any of the explicitly defined sets. We define relations prefix-or-equal $\preceq$ and strictly-prefix $\prec$ over two lists. For instance, for lists $L, L_0 = L \| x, L_1 = L \| y$ where $x, y \in \mathcal{X}, x \neq y$ we have that $L \preceq L, L \not\prec L, L \prec L_0, L \prec L_1, L_0 \not\preceq L_1, L_1 \not\preceq L_0$ meaning that L is a prefix of L_0 and L_1 but neither of L_0, L_1 is a prefix of the other. By $X[\cdot]$ we denote an associative array.

In our security experiments, that we denote with **Game**, we invoke adversaries via instruction 'Invoke'. These adversaries are denoted by $\mathcal{A}, \mathcal{B}$. Adversaries have access to the security experiment's interface, which is defined by oracles that are denoted by the term **Oracle**. Games are terminated via instructions 'Stop with x' (meaning that x is returned by the game) or 'Reward b' (meaning that the game terminates and returns 1 if $b = \mathtt{T}$). In procedures that we denote by **Proc** and in oracles, we use the shortcut notion 'Require x'. Depending on the procedure's or oracle's number of return values n, that means 'If $x = \mathtt{F}$, then return $\perp^n$'.

2.2 Message Authentication Code

We define a message authentication code to be a set of algorithms $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_{\mathsf{M}})$ over a set of symmetric keys $\mathcal{K}$, a message space $\mathcal{M}$, and a tag space $\mathcal{T}$. The syntax is defined as:

$$\mathcal{K} \times \mathcal{M} \to \mathrm{tag} \to \mathcal{T}$$
$$\mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \mathrm{vfy}_{\mathsf{M}} \to \{\mathtt{T}, \perp\}$$

Please note that we define the tag algorithm explicitly deterministic.

For correctness of a MAC we define that for all $k \in \mathcal{K}$ and all $m \in \mathcal{M}$ it is required that $\mathrm{vfy}_{\mathsf{M}}(k, m, \mathrm{tag}(k, m)) = \mathtt{T}$.

We define a one-time multi-instance strong unforgeability notion SUF for MAC security – that is equivalent with standard strong unforgeability – for which the formal security game is depicted in Appendix E Figure 14. That is, for a game in which an adversary can generate instances i (with independent uniformly random keys $k_i \leftarrow_{\$} \mathcal{K}$) via an oracle Gen, the adversary can query a Tag oracle on a message m from message space $\mathcal{M}$ for each instance at most once to obtain the respective MAC tag. Additionally, the adversary can verify MAC tags for specified messages and instances via oracle Vfy and obtain an instance's key by querying an Expose oracle for this instance. The adversary wins by providing a forgery (m, τ) for an instance i to the Vfy oracle if there was no Tag(i, m) query before with output τ and if i's key was not exposed via oracle Expose. We define the advantage of winning the SUF game against a MAC scheme M as $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{A}) = \Pr[\mathrm{SUF}_{\mathsf{M}}(\mathcal{A}) \to 1]$.

3 Sufficient Security for Key-Updatable KEM

A key-updatable key encapsulation mechanism (kuKEM) is a key encapsulation mechanism that provides update algorithms for public key and secret key with respect to some associated data respectively. Prior to our work, this primitive was used to instantiate sesquidirectional RKE. In order to allow for our equivalence result, we minimally adjust the original kuKEM notion by Poettering and Rösler [20] and call it kuKEM*. The small, yet crucial changes comprise allowed updates of public and secret key during encapsulation and decapsulation (in our syntax definition) as well as the adversary's ability to manipulate utilized randomness of encapsulations (in our security definition). In Section 6 the rationales behind these changes are clarified. In order to provide a coherent definition, we not only describe alterations towards previous work but define kuKEM* entirely (as we consider our changes to be a significant contribution and believe that this strengthens comprehensibility).

Syntax A kuKEM* is a set of algorithms $\mathsf{K} = (\mathrm{gen}_\mathsf{K}, \mathrm{up}, \mathrm{enc}, \mathrm{dec})$ with sets of public keys $\mathcal{PK}$ and secret keys $\mathcal{SK}$, a set of associated data $\mathcal{AD}$ for updating the keys, a set of ciphertexts $\mathcal{C}$ (with $\mathcal{AD} \cap \mathcal{C} = \emptyset$), and a set of encapsulated keys $\mathcal{K}$. Furthermore we define $\mathcal{R}$ as the set of random coins used during the encapsulation:

$$\begin{aligned}
&\mathrm{gen}_\mathsf{K} \rightarrow_\$ \mathcal{PK} \times \mathcal{SK}\\
&\mathcal{PK} \times \mathcal{AD} \rightarrow \mathrm{up} \rightarrow \mathcal{PK}\\
&\mathcal{SK} \times \mathcal{AD} \rightarrow \mathrm{up} \rightarrow \mathcal{SK}\\
&\mathcal{PK} \times \mathcal{R} \rightarrow \mathrm{enc} \rightarrow \mathcal{PK} \times \mathcal{K} \times \mathcal{C} \text{ or } \mathcal{PK} \rightarrow \mathrm{enc} \rightarrow_\$ \mathcal{PK} \times \mathcal{K} \times \mathcal{C}\\
&\mathcal{SK} \times \mathcal{C} \rightarrow \mathrm{dec} \rightarrow (\mathcal{SK} \times \mathcal{K}) \cup \{(\perp, \perp)\}
\end{aligned}$$

Please note that the encapsulation and decapsulation may modify the public key and the secret key respectively – as a result, the kuKEM* is stateful (where the public key is a public state).[6]

Correctness The correctness for kuKEM* is (for simplicity) defined through game CORR_K (see Figure 3), in which an adversary $\mathcal{A}$ can query encapsulation, decapsulation, and update oracles. The adversary (against correctness) wins if different keys are computed during decapsulation and the corresponding encapsulation even though compatible key updates were conducted and ciphertexts from encapsulations were directly forwarded to the decapsulation oracle.

Definition 1 (kuKEM* **correctness).** *A* kuKEM* *scheme* K *is correct if for every* $\mathcal{A}$*, the probability of winning game* CORR_K *from Figure 3 is* $\Pr[\mathrm{CORR}_\mathsf{K}(\mathcal{A}) \rightarrow 1] = 0$.

Security Here we describe KUOWR security of kuKEM* as formally depicted in Figure 4. KUOWR defines <u>o</u>ne-<u>w</u>ay security of <u>ku</u>KEM* under <u>r</u>andomness manipulation in a multi-instance/multi-challenge setting.

[6] As kuKEM* naturally provides no security for encapsulated keys if the adversary can manipulate the randomness for gen_K already, we only consider the manipulation of random coins for enc.

```
Game CORR_K(A)
00 $(pk, sk) \leftarrow_\$ \mathrm{gen}_\mathsf{K}$
01 $key[\cdot] \leftarrow \bot$
02 $trs \leftarrow \epsilon$; $trr \leftarrow \epsilon$
03 Invoke $\mathcal{A}$
04 Stop with 0

Oracle Up_S(ad)
05 Require $ad \in \mathcal{AD}$
06 $pk \leftarrow \mathrm{up}(pk, ad)$
07 $trs \overset{\shortparallel}{\leftarrow} ad$
08 Return

Oracle Up_R(ad)
09 Require $ad \in \mathcal{AD}$
10 $sk \leftarrow \mathrm{up}(sk, ad)$
11 $trr \overset{\shortparallel}{\leftarrow} ad$
12 Return

Oracle Enc()
13 $(pk, k, c) \leftarrow_\$ \mathrm{enc}(pk)$
14 $trs \overset{\shortparallel}{\leftarrow} c$
15 $key[trs] \leftarrow k$
16 Return $(pk, c)$

Oracle Dec(c)
17 Require $c \in \mathcal{C}$
18 $(sk, k) \leftarrow \mathrm{dec}(sk, c)$
19 $trr \overset{\shortparallel}{\leftarrow} c$
20 If $trr \preceq trs$:
21     Reward $k \neq key[trr]$
22 Return
```

Fig. 3: The correctness notion of kuKEM* captured through game CORR.

Intuitively, the KUOWR game requires that a secret key can only be used for decapsulation of a ciphertext if prior to this decapsulation all updates of this secret key and all decapsulations with this secret key were consistent with the updates of and encapsulations with the respective public key. This is reflected by using the transcript (of public key updates and encapsulations or secret key updates and decapsulations) as a reference to encapsulated "challenge keys" and secret keys.

In order to let the adversary play with the kuKEM*'s algorithms, the game provides oracles Gen, Up_S, Up_R, Enc, and Dec. Thereby instances (i.e., key pairs) can be generated via oracle Gen and are referenced in the remaining oracles by a counter that refers to when the respective instance was generated.

For encapsulation via oracle Enc, the adversary can either choose the invocation's random coins by setting rc to some value that is not the empty string ϵ or let the encapsulation be called on fresh randomness by setting $rc = \epsilon$ (line 16). In the former case, the adversary trivially knows the encapsulated key. Thus, only when called with fresh randomness, the encapsulated key is marked as a challenge key in array CK (line 20).

The variables CK, SK, and XP (the latter two are explained below) are indexed via the transcript of operations on the respective key pair part. As public keys and secret keys can uniquely be referenced via the associated data under which they are updated and via ciphertexts that have been encapsulated or decapsulated by them, the concatenation of these values (i.e., sent or received transcripts trs, trr) are used as references to them in the KUOWR game.

On decapsulation of a key that is not marked as a challenge, the respective key is output to the adversary. Challenge keys are of course not provided to the adversary as thereby the challenge would be trivially solved (line 36).

Via oracle Expose, the adversary can obtain a secret key of specified instance i that results from an operation referenced by transcript tr. As described above, the transcript, to which a secret key refers, is built from the associated data of updates to this secret key (via oracle Up_R) and the ciphertexts of decapsulations with this secret key (via oracle Dec) as these two operations may modify the

```
Game KUOWR$_\mathsf{K}(\mathcal{A})$
00   $n \leftarrow 0$
01   Invoke $\mathcal{A}$
02   Stop with 0

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

Oracle Up$_S(i, ad)$
10   Require $1 \leq i \leq n \wedge ad \in \mathcal{AD}$
11   $pk_i \leftarrow \text{up}(pk_i, ad)$
12   $trs_i \overset{\shortparallel}{\leftarrow} ad$
13   Return $pk_i$

Oracle Enc$(i, rc)$
14   Require $1 \leq i \leq n$
15 · Require $rc \in \mathcal{R} \cup \{\epsilon\}$
16 · If $rc = \epsilon$: $mr \leftarrow \mathtt{F}$; $rc \leftarrow_\$ \mathcal{R}$
17 · Else: $mr \leftarrow \mathtt{T}$
18 · $(pk_i, k, c) \leftarrow \text{enc}(pk_i; rc)$
19 · $trs_i \overset{\shortparallel}{\leftarrow} c$
20 · If $mr = \mathtt{F}$: $\text{CK}_i[trs_i] \leftarrow k$
21 · Return $(pk_i, c)$

Oracle Solve$(i, tr, k)$
22   Require $1 \leq i \leq n$
23   Require $tr \notin \text{XP}_i$
24   Require $\text{CK}_i[tr] \neq \bot$
25   Reward $k = \text{CK}_i[tr]$
26   Return

Oracle Up$_R(i, ad)$
27   Require $1 \leq i \leq n \wedge ad \in \mathcal{AD}$
28   $sk_i \leftarrow \text{up}(sk_i, ad)$
29   $trr_i \overset{\shortparallel}{\leftarrow} ad$
30   $SK_i[trr_i] \leftarrow sk_i$
31   Return

Oracle Dec$(i, c)$
32   Require $1 \leq i \leq n \wedge c \in \mathcal{C}$
33 · $(sk_i, k) \leftarrow \text{dec}(sk_i, c)$
34 · $trr_i \overset{\shortparallel}{\leftarrow} c$
35 · $SK_i[trr_i] \leftarrow sk_i$
36 · If $\text{CK}_i[trr_i] \neq \bot$:
37 ·     Return
38 · Return $k$

Oracle Expose$(i, tr)$
39   Require $1 \leq i \leq n$
40 · Require $SK_i[tr] \in \mathcal{SK}$
41 · $\text{XP}_i \overset{\cup}{\leftarrow} \{tr^* \in (\mathcal{AD} \cup \mathcal{C})^* :$
         $tr \prec tr^*\}$
42   Return $SK_i[tr]$
```

Fig. 4: Security experiment KUOWR, modeling one-way security of key-updatable KEM in a multi-instance/multi-challenge setting under randomness manipulation. Lines of code tagged with '·' are (substantially) modified with respect to KUOW security in [19]. Line 41 is a shortcut notion that can be implemented efficiently. CK: challenge keys, XP: exposed secret keys, *trs*, *trr*: transcripts.

secret key. As all operations, performed with an exposed secret key, can be traced by the adversary (i.e., updates and decapsulations; note that both are deterministic), all secret keys that can be derived from an exposed secret key are also marked exposed via array XP (line 41).

Finally, an adversary can solve a challenge via oracle Solve by providing a guess for the challenge key that was encapsulated for an instance i with the encapsulation that is referenced by transcript tr. Recall that the transcript, to which an encapsulation refers, is built from the associated data of updates to the respective instance's public key (via oracle Up$_S$) and the ciphertexts of encapsulations with this instance's public key (via oracle Enc) as these two operations may modify the public key for encapsulation. If the secret key for decapsulating the referenced challenge key is not marked exposed (line 23) and the guess for the challenge key is correct (line 24), then game KUOWR stops with '1' (via 'Reward') meaning that the adversary wins.

Definition 2 (KUOWR **Advantage).** *The advantage of an adversary $\mathcal{A}$ against a* kuKEM* *scheme* K *in game* KUOWR *from Figure 4 is defined as* $\mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{A}) = \Pr[\mathrm{KUOWR}_{\mathsf{K}}(\mathcal{A}) \to 1]$.

We chose to consider one-way security as opposed to key indistinguishability for the kuKEM* as it suffices to show equivalence with key indistinguishability of RKE (in the ROM). It is evident that all proofs in this work also hold for key indistinguishability of kuKEM* and one can generically derive key indistinguishability for kuKEM* via the FO transform by Hofheinz et al. [12].

Differences compared to previous Security Definition In Figure 4 we denote changes from KUOW security (cf., Figure 1 [19]) by adding '·' at the beginning of lines. Below we elaborate on these differences.

The main difference in our definition of KUOWR security compared to KUOW security is that we allow the adversary to manipulate the execution's random coins. As we define encapsulation and decapsulation to (potentially) update the used public key or secret key, another conceptual difference is that we only allow the adversary to encapsulate and decapsulate once under each public and secret key. Thus, we assume that public and secret keys are overwritten on encapsulation and decapsulation respectively. In contrast to our security definition, in the KUOW security game only the current secret key of an instance can be exposed. Even though we assume the secret key to be replaced by its newer versions on updates or decapsulations, there might be, for example, backups that store older secret key versions. As a result we view the restriction of only allowing exposures of the current secret key artificial.[7] An important notational choice is that we index the variables with transcripts trs, trr instead of integer counters. This notation reflects the idea that public key and secret key only stay compatible as long as they are used correspondingly and immediately diverge on different associated data or tampered ciphertexts.

We further highlight the fundamental difference towards HkuPke by Jost et al. [15]. Their notion of HkuPke does not allow (fully adversary-controlled) associated data on public and secret key updates and additionally requires (authenticated) interaction between the holders of the key parts thereby. Looking ahead, this makes this primitive insufficient for diverging the public key from the secret key (in the states) of users A and B during an impersonation of A towards B in (U)RKE (especially under randomness manipulation). This is, however, required in an optimal security definition but explicitly excluded in the sub-optimal RKE notion by Jost et al. [15]. Since the syntax of HkuPke is already inadequate to reflect this security property, we cannot provide a separating attack. Nevertheless, we further expound this weakness in Appendix D.

[7] It is important to note that the equivalence between KUOWR security of kuKEM* and KINDR security of URKE is independent of this definitional choice – if either both definitions allow or both definitions forbid the exposure of also past secret keys or states respectively, equivalence can be shown.

Instantiation A kuKEM* scheme, secure in the KUOWR game, can be generically constructed from an OW-CCA adaptively secure hierarchical identity based key encapsulation mechanism (HIB-KEM). The construction – the same as in [19] – is provided for completeness in Figure 5. The update of public keys is the concatenation of associated data (interpreted as identities in the HIB-KEM) and the update of secret keys is the delegation to lower level secret keys in the identity hierarchy. The reduction is immediate: After guessing for which public key and after how many updates the challenge encapsulation that is solved by the adversary is queried, the challenge from the OW-CCA game is embedded into the respective KUOWR challenge.

```
Proc gen_K
  $(pk_{\mathsf{ID}}, sk_{\mathsf{ID}}) \leftarrow_{\$} \mathrm{gen}_{\mathsf{ID}}$
  $id \leftarrow \epsilon$
  $pk \leftarrow (pk_{\mathsf{ID}}, id)$
  $sk \leftarrow sk_{\mathsf{ID}}$
  Return $(pk, sk)$

Proc enc($pk$)
  $(pk_{\mathsf{ID}}, id) \leftarrow pk$
  $(c, k) \leftarrow_{\$} \mathrm{enc}_{\mathsf{ID}}(pk_{\mathsf{ID}}, id)$
  $id \xleftarrow{\shortparallel} c$
  $pk \leftarrow (pk_{\mathsf{ID}}, id)$
  Return $(pk, k, c)$

Proc up($sk, ad$)
  $sk \leftarrow \mathrm{del}_{\mathsf{ID}}(sk, ad)$
  Return $sk$

Proc up($pk, ad$)
  $(pk_{\mathsf{ID}}, id) \leftarrow pk$
  $id \xleftarrow{\shortparallel} ad$
  $pk \leftarrow (pk_{\mathsf{ID}}, id)$
  Return $pk$

Proc dec($sk, c$)
  $k \leftarrow \mathrm{dec}_{\mathsf{ID}}(sk, c)$
  $sk \leftarrow \mathrm{del}_{\mathsf{ID}}(sk, c)$
  Return $(sk, k)$
```

Fig. 5: Generic construction of a kuKEM* from a hierarchical identity based KEM $\mathsf{HK} = (\mathrm{gen}_{\mathsf{ID}}, \mathrm{del}_{\mathsf{ID}}, \mathrm{enc}_{\mathsf{ID}}, \mathrm{dec}_{\mathsf{ID}})$ (slightly differing from construction in [19] Figure 2 by adding an internal key update in encapsulation and decapsulation respectively).

Sufficiency of KUOWR *for SRKE* Before proving equivalence between security of key-updatable KEM and ratcheted key exchange, we shed a light on implications due to the differences between our notion of kuKEM* and its KUOWR security and the notion of kuKEM and its KUOW security in [19].

Remark 1. Even though the KUOWR game provides more power to the adversary in comparison to the KUOW game by allowing manipulation of random coins, exposures of past secret keys, and providing an explicit decapsulation oracle (instead of an oracle that only allows for checks of ciphertext-key pairs; cf., Figure 1 [19]), the game also restricts the adversary's power by only allowing decapsulations under the current secret key of an instance (as opposed to also checking ciphertext-key pairs for past secret keys of an instance as in the KUOW game). One can exploit this and define protocols that are secure with respect to one game definition but allow for attacks in the other game. Consequently, neither of both security definitions implies the other one.

Despite the above described distinction between both security definitions, KUOWR security suffices to build sesquidirectional RKE according to the KIND definition in [20] – which was yet the weakest notion of security of RKE for which a construction was built from a key-updatable public key primitive. The ability to check ciphertext-key pairs under past versions of secret keys of an instance is actually never used in the proof of Poettering and Rösler [19]. The only case in which this Check oracle is used in their proof is B's receipt of a manipulated ciphertext from the adversary. Checking whether a ciphertext-key pair for the current version of a secret key of an instance is valid, can of course be conducted by using the Dec oracle of our KUOWR notion. For full details on their proof we refer the reader to Appendix C in [19].

Consequently, for the construction of KIND secure sesquidirectional RKE (according to [20] Figure 8) from Poettering and Rösler [20], the used kuKEM must either be KUOW secure (see [20] Figure 1) or KUOWR secure (see Figure 4), which is formally depicted in the following observation. Thus, even though these notions are not equivalent, they both suffice for constructing KIND secure sesquidirectional RKE.

Observation 1 *The sesquidirectional RKE protocol* R *from [20] Figure 6 offers* key indistinguishability *according to [20] Figure 8 if function* H *is modeled as a random oracle, the* kuKEM* *provides* KUOWR *security according to Figure 4, the one-time signature scheme provides* SUF *security according to [19] Figure 22, the* MAC *scheme* M *provides* SUF *security according to Figure 14, and the symmetric-key space of the* kuKEM* *is sufficiently large.*

4 Unidirectional RKE under Randomness Manipulation

Unidirectional RKE (URKE) is the simplest variant of ratcheted key exchange. After a common initialization of a session between two parties A and B, it enables the continuous establishment of keys within this session. In this unidirectional setting, A can initiate the computation of keys repeatedly. With each computation, a ciphertext is generated that is sent to B, who can then comprehend the computation and output (the same) key. Restricting A and B to this unidirectional communication setting, in which B cannot respond, allows to understand the basic principles of ratcheted key exchange. For the same reasons we provided the whole definition of kuKEM* before (i.e., we see our changes as a significant contribution and aim for a coherent depiction), we fully define URKE under randomness manipulation below.

Syntax We recall that URKE is a set of algorithms $\mathsf{UR} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$ defined over sets of A's and B's states $\mathcal{S}_A$ and $\mathcal{S}_B$ respectively, a set of associated data $\mathcal{AD}$, a set of ciphertexts $\mathcal{C}$, and a set of keys $\mathcal{K}$ established between A and B. We extend the syntax of URKE by explicitly regarding the utilized randomness of the snd algorithm. Consequently we define $\mathcal{R}$ as the set of $\underline{\text{r}}$andom $\underline{\text{c}}$oins $rc \in \mathcal{R}$ used in snd. To highlight that A only sends and B only receives in URKE, we may add 'A' and 'B' as handles to the index of snd, and rcv respectively.

$$\begin{aligned}
&\text{init} \rightarrow_{\$} \mathcal{S}_A \times \mathcal{S}_B \\
&\mathcal{S}_A \times \mathcal{AD} \times \mathcal{R} \rightarrow \text{snd} \rightarrow \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \text{ or } \mathcal{S}_A \times \mathcal{AD} \rightarrow \text{snd} \rightarrow_{\$} \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\
&\mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} \rightarrow \text{rcv} \rightarrow \mathcal{S}_B \times \mathcal{K} \cup \{(\perp, \perp)\}
\end{aligned}$$

Please note that de-randomizing (or explicitly considering the randomness of) the initialization of URKE is of little value since an adversary, when controlling the random coins of init, obtains all information necessary to compute all keys between A and B.

Correctness Below we define correctness for URKE. Intuitively a URKE scheme is correct, if all keys produced with send operations of A can also be obtained with the resulting ciphertext by the respective receive operations of B.

Definition 3 (URKE **Correctness).** *Let $\{ad_i \in \mathcal{AD}\}_{i\geq 1}$ be a sequence of associated data. Let $\{s_{A,i}\}_{i\geq 0}, \{s_{B,i}\}_{i\geq 0}$ denote the sequences of A's and B's states generated by applying* $\text{snd}(\cdot, ad_i)$ *and* $\text{rcv}(\cdot, ad_i, \cdot)$ *operations iteratively for $i \geq 1$, that is,*

$$\begin{aligned}
(s_{A,i}, k_i, c_i) &\leftarrow_{\$} \text{snd}(s_{A,i-1}, ad_i) \\
(s_{B,i}, k'_i) &\leftarrow \text{rcv}(s_{B,i-1}, ad_i, c_i).
\end{aligned}$$

We say URKE *scheme* $\mathsf{UR} = (\text{init}, \text{snd}, \text{rcv})$ *is correct if for all $s_{A,0}, s_{B,0} \leftarrow_{\$} \text{init}$, for all associated data sequences $\{ad_i\}_{i\geq 1}$, and for all random coins used for* snd *calls, the key sequences $\{k_i\}_{i\geq 1}$ and $\{k'_i\}_{i\geq 1}$ generated as above are equal.*

Security For security, we provide the KINDR game for defining key indistinguishability under randomness manipulation of URKE in Figure 6. In this game, the adversary can let the session participants A and B send and receive ciphertexts via SndA and RcvB oracle queries respectively to establish keys between them. By querying the Reveal or Challenge oracles, the adversary can obtain these established keys or receive a challenge key (that is either the real established key or a randomly sampled element from the key space) respectively. Finally, the adversary can expose A's and B's state as the output of a specified send or receive operation respectively via oracles ExposeA or ExposeB.

When querying the SndA oracle, the adversary can specify the random coins for the invocation of the snd algorithm from the set $\mathcal{R}$ or indicate that it wants the random coins to be sampled uniformly at random by letting $rc = \epsilon$. By allowing the adversary to set the randomness for the invocations of the snd algorithm and exposing past states (which was not permitted in the definition of Poettering and Rösler [20]), new trivial attacks arise.

Below we review and explain the trivial attacks of the original URKE KIND game, map them to our version, and then introduce new trivial attacks that arise due to randomness manipulation.

A conceptual difference between our game definition and the games by Poettering and Rösler [20] is the way variables (especially arrays) are indexed.

```
Game KINDR$^b_{\mathsf{UR}}(\mathcal{A})$
00   $\mathrm{XP}_A \leftarrow \emptyset$; $\mathrm{MR} \leftarrow \emptyset$
01   $\mathrm{KN} \leftarrow \emptyset$; $\mathrm{CH} \leftarrow \emptyset$
02   $trs \leftarrow \epsilon$; $trr \leftarrow \epsilon$
03   $S_A[\cdot] \leftarrow \bot$; $S_B[\cdot] \leftarrow \bot$
04   $key[\cdot] \leftarrow \bot$;
05   $(s_A, s_B) \leftarrow_\$ \mathrm{init}$
06   $S_A[trs] \leftarrow s_A$; $S_B[trr] \leftarrow s_B$
07   $b' \leftarrow_\$ \mathcal{A}$
08 · Require $\mathrm{KN} \cap \mathrm{CH} = \emptyset$
09   Stop with $b'$

Oracle SndA$(ad, rc)$
10   Require $ad \in \mathcal{AD} \wedge rc \in \mathcal{R} \cup \{\epsilon\}$
11   If $rc = \epsilon$:
12     $(s_A, k, c) \leftarrow_\$ \mathrm{snd}(s_A, ad)$
13   Else:
14     $(s_A, k, c) \leftarrow \mathrm{snd}(s_A, ad; rc)$
15 ○   $\mathrm{MR} \overset{\cup}{\leftarrow} \{trs \| (ad, c)\}$
16 ○   If $trs \in \mathrm{XP}_A$:
17 ○     $\mathrm{KN} \overset{\cup}{\leftarrow} \{trs \| (ad, c)\}$
18 ○     $\mathrm{XP}_A \overset{\cup}{\leftarrow} \{trs \| (ad, c)\}$
19   $trs \overset{\shortparallel}{\leftarrow} (ad, c)$
20   $key[trs] \leftarrow k$; $S_A[trs] \leftarrow s_A$
21   Return $c$

Oracle Reveal$(tr)$
22   Require $key[tr] \in \mathcal{K}$
23 · $\mathrm{KN} \overset{\cup}{\leftarrow} \{tr\}$
24   Return $key[tr]$

Oracle RcvB$(ad, c)$
25   Require $ad \in \mathcal{AD} \wedge c \in \mathcal{C} \wedge s_B \neq \bot$
26 · If $trr \| (ad, c) \not\preceq trs$
       $\wedge \mathrm{LCP}(trs, trr) \in \mathrm{XP}_A$:
27 ·   $\mathrm{KN} \overset{\cup}{\leftarrow} \{trr \| (ad, c)\}$
28   $(s_B, k) \leftarrow \mathrm{rcv}(s_B, ad, c)$
29   If $k = \bot$: Return $\bot$
30   $trr \overset{\shortparallel}{\leftarrow} (ad, c)$
31   $key[trr] \leftarrow k$; $S_B[trr] \leftarrow s_B$
32   Return

Oracle ExposeA$(tr)$
33   Require $S_A[tr] \in \mathcal{S}_A$
34 · $\mathrm{XP}_A \overset{\cup}{\leftarrow} \{tr\}$
35 ○ $trace \leftarrow \{tr^* \in \mathcal{TR}^* : \forall tr' \in \mathcal{TR}^*$
       $(tr \prec tr' \preceq tr^* \implies tr' \in \mathrm{MR})\}$
36 ○ $\mathrm{KN} \overset{\cup}{\leftarrow} trace$; $\mathrm{XP}_A \overset{\cup}{\leftarrow} trace$
37   Return $S_A[tr]$

Oracle ExposeB$(tr)$
38   Require $S_B[tr] \in \mathcal{S}_B$
39 · $\mathrm{KN} \overset{\cup}{\leftarrow} \{tr^* \in \mathcal{TR}^* : tr \prec tr^*\}$
40   Return $S_B[tr]$

Oracle Challenge$(tr)$
41   Require $key[tr] \in \mathcal{K}$
42 · Require $tr \notin \mathrm{CH}$
43   $k \leftarrow b\ ?\ key[tr] : \$(\mathcal{K})$
44 · $\mathrm{CH} \overset{\cup}{\leftarrow} \{tr\}$
45   Return $k$
```

Fig. 6: Games KINDR^b, $b \in \{0, 1\}$, for URKE scheme UR. Lines of code tagged with a '$\cdot$' denote mechanisms to prevent or detect trivial attacks without randomness manipulation; trivial attacks caused by randomness manipulation are detected and prevented by lines tagged with '$\circ$'. We define LCP(X, Y) to return the longest common prefix between X and Y, which are lists of atomic elements $z_i \in (\mathcal{AD} \times \mathcal{C})$. By longest common prefix we mean the longest list $Z = z_0 \| \ldots \| z_n$ for which $Z \preceq X \wedge Z \preceq Y$. We further define $\mathcal{TR} = \mathcal{AD} \times \mathcal{C}$. Line 39 is a shortcut notion that can be implemented efficiently. XP: exposed states, MR: states and keys affected by manipulated randomness, KN: known keys, CH: challenge keys, *trs*, *trr*: transcripts.

While the KIND games of [20] make use of counters (of send and receive operations) to index computed keys and adversarial events, we use the communicated transcripts, sent and received by A and B respectively, as indices. We thereby heavily exploit the fact that synchronicity (and divergence) of the communication between A and B are defined over these transcripts, which results in a more comprehensible (but equivalent) game notation. Please note that, due to our indexing scheme, it suffices for our game definition to maintain a common key array $key[\cdot]$ and common sets of known keys KN and challenged keys CH for A and B (as opposed to arrays and sets for each party).[8]

The lines marked with '$\cdot$' in Figure 6 denote the handling of trivial attacks without randomness manipulation (as in [20]). Lines marked with '$\circ$' introduce modifications that become necessary due the new trivial attacks based on manipulation of randomness.

Trivial attacks without randomness manipulations are:

(a) If the adversary reveals a key via oracle Reveal, then challenging this key via oracle Challenge is trivial. In order to prevent reveal and challenge of the same key, sets KN and CH trace which keys have been revealed (line 23) and challenged (line 44). The adversary only wins, if the intersection of both sets is empty (line 08). Additionally, a key must only be challenged once as otherwise bit b can be obtained trivially (line 42).
Example: $c \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$; $k \leftarrow \mathrm{Reveal}((\epsilon, c))$; Return $k = \mathrm{Challenge}((\epsilon, c))$
(b) As keys, that are computed by both parties (because ciphertexts between them have not been manipulated yet), are stored only once in array key (due to the indexing of arrays with transcripts instead of pure counters), the adversary cannot reveal these keys on one side of the communication (e.g., at A) and then challenge them on the other side (e.g., at B). Consequently, this trivial attack (which was explicitly considered in [20]) is implicitly handled by our game definition.
(c) After exposing B's state via oracle ExposeB, the adversary can comprehend all future computations of B. Consequently, all keys that can be received by B in the future are marked known (line 39).
Example: $s_B \leftarrow \mathrm{ExposeB}(\epsilon)$; $c \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$; $\mathrm{RcvB}(\epsilon, c)$; $(s_B, k) \leftarrow \mathrm{rcv}(s_B, \epsilon, c)$; Return $k = \mathrm{Challenge}((\epsilon, c))$
(d) Exposing B's state, as long as the communication between A and B has not yet been manipulated by the adversary, allows the adversary also to compute all future keys established by A (which is also implicitly handled by our indexing of arrays via transcripts).
(e) Exposing A's state via oracle ExposeA allows the adversary to impersonate A towards B by using the exposed state to create and send own valid ciphertexts to B. As creating a forged ciphertext reveals the key that is computed by B on receipt, such keys are marked known (lines 26-27). The detection of this trivial attack works as follows: As soon as B receives a ciphertext that was not sent by A (i.e., B's transcript together with the received ciphertext is not a prefix of A's transcript) and A was exposed after A sent the last ciphertext that was also received by B (i.e., after the last common prefix LCP), the adversary is able to create this ciphertext validly on its own.[9]
Example: $s_A \leftarrow \mathrm{ExposeA}$; $(s_A, k, c) \leftarrow \mathrm{snd}(s_A, \epsilon)$; $\mathrm{RcvB}(\epsilon, c)$; Return $k = \mathrm{Challenge}((\epsilon, c))$

[8] This is because a key, computed during the sending of A and the corresponding receiving of B, only differs between A and B if the received transcript of B diverged from the sent transcript of A.

[9] Please note that we need to detect this trivial attack this way (in contrast to the game in [20]) because the adversary can forge ciphertexts to B without letting the communication between A and B actually diverge. It can do so by creating

Due to randomness manipulations, the adversary can additionally conduct the following attacks trivially:

(f) If the randomness for sending is set by the adversary (via $\mathrm{SndA}(ad, rc)$, $rc \neq \epsilon$) and the state, used for this sending, is exposed (via ExposeA), then also the next state of A, output by this send operation, will be known (and marked as exposed) as sending is thereby deterministically computed on inputs that are known by the adversary (lines 16,18). Since the adversary can also retrospectively expose A's state, all computations that can be traced, due to continuous $\underline{\text{m}}$anipulated $\underline{\text{r}}$andomness of subsequent SndA oracle queries (unified in set MR) after such an exposure, are also marked as exposed (lines 35-36).
Example: $rc \leftarrow_\$ \mathcal{R}$; $c' \leftarrow \mathrm{SndA}(\epsilon, rc)$; $\mathrm{RcvB}(\epsilon, c')$; $s_A \leftarrow \mathrm{ExposeA}(\epsilon)$; $(s_A, k', c') \leftarrow \mathrm{snd}(s_A, \epsilon; rc)$; $(s_A, k, c) \leftarrow_\$ \mathrm{snd}(s_A, \epsilon)$; $\mathrm{RcvB}(\epsilon, c)$; Return $k = \mathrm{Challenge}((\epsilon, c')\|(\epsilon, c))$

(g) Similarly, if the randomness for sending is set by the adversary and the state that A uses during this send operation is exposed, then the key, computed during sending, is known by the adversary since its computation is thereby deterministic (lines 16-17,35-36).
Example: $rc \leftarrow_\$ \mathcal{R}$; $c \leftarrow \mathrm{SndA}(\epsilon, rc)$; $s_A \leftarrow \mathrm{ExposeA}(\epsilon)$; $(s_A, k, c) \leftarrow \mathrm{snd}(s_A, \epsilon; rc)$; Return $k = \mathrm{Challenge}((\epsilon, c))$

Based on this game, we define the advantage of an adversary in breaking the security of an URKE scheme as follows.

Definition 4 (KINDR **Advantage).** *The advantage of an adversary $\mathcal{A}$ against a* URKE *scheme* UR *in game* KINDR *from Figure 6 is defined as* $\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{A}) = \left|\Pr[\mathrm{KINDR}^0_{\mathsf{UR}}(\mathcal{A}) = 1] - \Pr[\mathrm{KINDR}^1_{\mathsf{UR}}(\mathcal{A}) = 1]\right|$.

We say that an URKE scheme UR is secure if the advantage is negligible for all probabilistic polynomial time adversaries $\mathcal{A}$.

Please note that KINDR security of URKE is strictly stronger than both KIND security notions of URKE, defined by Bellare et al. [4] and Poettering and Rösler [20] (which themselves are incomparable among each other).

5 kuKEM* to URKE

Since our ultimate goal is to show that existence of a kuKEM* primitive is a necessary and sufficient condition to construct a URKE primitive – albeit requiring the help of other common cryptographic primitives such as hash functions (modeled as random oracle) and message authentication codes –, we dedicate this section to proving the latter of these implications.

an own valid ciphertext which it sends to B (via $s_A \leftarrow \mathrm{ExposeA}(\epsilon)$; $rc \leftarrow_\$ \mathcal{R}$; $(s_A, k, c) \leftarrow \mathrm{snd}(s_A, \epsilon; rc)$; $\mathrm{RcvB}(\epsilon, c)$) but then it lets A compute the same ciphertext (via $\mathrm{SndA}(\epsilon, rc)$). As a result, A and B are still in sync.

Construction of URKE *from* kuKEM* We give a generic way to construct a URKE scheme UR from a kuKEM* scheme K with the help of random oracle H and MAC scheme M. This transformation $\mathsf{K} \to \mathsf{UR}$ is fully depicted in Figure 7. Below we briefly describe the algorithms of URKE scheme $\mathsf{UR} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$.

During the state initiation algorithm init, a kuKEM* key pair (sk, pk) is generated such that the encapsulation key pk is embedded into the sender state s_A, and the decapsulation key sk into the receiver state s_B. The remaining state variables are exactly same for A and B. More specifically, two further keys are generated during initialization: the symmetric state key K and a MAC key $k.m$. Furthermore the sent or received transcript (initialized with an empty string ϵ) is stored in each state. For brevity, we assume that K, $k.m$, and the update key $k.u$ (used during sending and receiving; see below) all belong to the same key domain $\mathcal{K}$ that is sufficiently large.

```
**Proc** init
$(pk, sk) \leftarrow_\$ \mathrm{gen}_{\mathsf{K}}$
$K \leftarrow_\$ \mathcal{K}$; $k.m \leftarrow_\$ \mathcal{K}$
$t \leftarrow \epsilon$
$S_A \leftarrow (pk, K, k.m, t)$
$S_B \leftarrow (sk, K, k.m, t)$
Return $(S_A, S_B)$
```

```
**Proc** snd$(S_A, ad)$
$(pk, K, k.m, t) \leftarrow S_A$
$(pk, k, c) \leftarrow_\$ \mathrm{enc}(pk)$
$\tau \leftarrow \mathrm{tag}(k.m, (ad, c))$
$C \leftarrow (c, \tau)$
$t \overset{\shortparallel}{\leftarrow} (ad, c)$
$(k.o, K, k.m, k.u) \leftarrow$
        $\mathrm{H}(K, k, t)$
$pk \leftarrow \mathrm{up}(pk, k.u)$
$S_A \leftarrow (pk, K, k.m, t)$
Return $(S_A, k.o, C)$
```

```
**Proc** rcv$(S_B, ad, C)$
$(sk, K, k.m, t) \leftarrow S_B$
$(c, \tau) \leftarrow C$
Require $\mathrm{vfy}_{\mathsf{M}}(k.m, (ad, c), \tau)$
$(sk, k) \leftarrow \mathrm{dec}(sk, c)$
Require $k \neq \bot$
$t \overset{\shortparallel}{\leftarrow} (ad, c)$
$(k.o, K, k.m, k.u) \leftarrow$
        $\mathrm{H}(K, k, t)$
$sk \leftarrow \mathrm{up}(sk, k.u)$
$S_B \leftarrow (sk, K, k.m, t)$
Return $(S_B, k.o)$
```

Fig. 7: Construction of a URKE scheme from a kuKEM* scheme $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{up}, \mathrm{enc}, \mathrm{dec})$, 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 the symmetric key K in s_A with the same symbol $\mathcal{K}$.

On sending, public key pk in A's state is used by the encapsulation algorithm to generate key k and ciphertext c. Then, MAC key $k.m$, contained in the current state of A, is used to issue a tag τ over the tuple of associated data ad and encapsulation ciphertext c. The finally sent ciphertext, denoted by C, is a concatenation of c and τ. The output key $k.o$, as well as the symmetric keys of the next state of A are obtained from the random oracle, on input of the symmetric state key K, the freshly encapsulated key k, and the history of sent transcript t. Finally, a kuKEM* update is applied on pk under associated data that is derived from the random oracle output (denoted by $k.u$). Please note that the encapsulation algorithm is the only randomized operation inside snd. Hence the random coins of the latter are only used by the encapsulation.

On receiving, the operations are on par with the sending algorithm. Namely, the received ciphertext C is parsed as the encapsulation ciphertext c and the

MAC tag τ. The latter is verified with regards to the MAC key $k.m$, stored in the state of B. After the key k is decapsulated, the same input to the random oracle H is composed. The symmetric components of the next state and $k.o$ are derived from the random oracle's output. Finally, the secret key *sk* is updated with $k.u$, so that it is in-sync with the update of *pk*.

We remark that our construction in Figure 7 differs from the unidirectional RKE scheme by Poettering and Rösler [20] only in the output of the random oracle and in the subsequent use of the kuKEM*'s update algorithm (instead they freshly generated a new KEM key pair from the random oracle output). These changes are, nevertheless, significant as their scheme is insecure when the adversary is able to (reveal or) manipulate the random coins for invocations of the snd algorithm. We give a detailed attack description against their scheme in our model in Appendix C.

Theorem 1. *If* kuKEM* *scheme* K *is* KUOWR *secure according to Figure 4,* MAC *scheme* M *is* SUF *secure according to Figure 14, and* H *is a hash function modeled as random oracle, then* URKE *scheme* UR *from Figure 7 is* KINDR *secure according to Figure 6 with*

$$\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{q_{\mathrm{H}} \cdot (q_{\mathrm{SndA}} + q_{\mathrm{RcvB}})}{|\mathcal{K}|}$$

where $\mathcal{A}$ *is an adversary against* KINDR *security,* $\mathcal{B}_{\mathsf{K}}$ *is an adversary against* KUOWR *security,* $\mathcal{B}_{\mathsf{M}}$ *is an adversary against* SUF *security,* $\mathcal{K}$ *is the key domain in the construction* UR*,* q_{SndA}*,* q_{RcvB}*, and* q_{H} *are the number of* SndA*,* RcvB *and* H *queries respectively by* $\mathcal{A}$*, and the running time of* $\mathcal{A}$ *is approximately the running time of* $\mathcal{B}_{\mathsf{K}}$ *and* $\mathcal{B}_{\mathsf{M}}$*.*

Proof (Sketch, Theorem 1). We here give the sketch of the full proof that is in Appendix B. Our idea is to design a series of games **Game 0-5**, in which differences between subsequent games are only syntactical and the advantage of the adversary $\mathcal{A}$ remains same. From this fifth game we are then ultimately able to reduce either of the following cases, that are explained below, to one of the hardness assumptions.

Consider the following scenarios which lead to a win for the adversary $\mathcal{A}$. Since the challenged keys are derived from the random oracle, we argue that, if $\mathcal{A}$ does not make a random oracle query $\mathrm{H}(K, k, t)$ for any of the challenged keys, then its advantage in guessing the challenge bit correctly remains negligible. We do not consider random oracle queries to keys that are trivially revealed to the adversary, as they do not lead to a win in the KINDR game (e.g., if the exposed state of B helps the adversary to trivially query H). Therefore, we regard the following three events in which $\mathcal{A}$ makes such *special* random oracle queries:

– The random oracle query $\mathrm{H}(K, k, t)$ belongs to one of the keys derived by the sender, in which fresh random coins, unknown to the adversary, are used for sending (and hence for encapsulation). In this case, we can give a reduction to the KUOWR game with respect to kuKEM* scheme K, in which the reduction wins the KUOWR game by using the encapsulated key k as the solution.

- The random oracle query $\mathrm{H}(K, k, t)$ belongs to one of the keys, derived from the sender where the used random coins are chosen by the adversary. We know that $\mathcal{A}$ did not expose the respectively used states of A or B as this leads to a trivial win. Therefore, we can show that the symmetric state keys K in these cases are independent from the view of $\mathcal{A}$. This implies that making such special $\mathrm{H}(K, k, t)$ query requires a collision in the key domain $\mathcal{K}$, whose probability is bounded by $q_{\mathrm{H}} \cdot (q_{\mathrm{SndA}} + q_{\mathrm{RcvB}})/|\mathcal{K}|$.
- The random oracle query $\mathrm{H}(K, k, t)$ belongs to one of the keys, derived by the receiver B, who reaches to an out-of-sync status (if B is still in-sync with A, then one of the two cases above are relevant). Since each received ciphertext contains a MAC tag, we can show that the first received ciphertext by B that is different from the sent ciphertext by A either corresponds to a trivial impersonation or can be used to reduce this event to a forgery in the SUF game with respect to MAC scheme M.

Therefore, by bounding the probability of these three cases, we can deduce the adversary's advantage (which is negligible under the named assumptions). □

6 URKE to kuKEM*

In order to show that public key encryption with independently updatable key pairs (in our case kuKEM*) is a necessary building block for ratcheted key exchange, we build the former from the latter. The major obstacle is that the updates of public key and secret key of a kuKEM* are conducted independently – consequently no communication between holder of the public key and holder of the secret key can be assumed for updates. In contrast, all actions in ratcheted key exchange are based on communication (i.e., sent or received ciphertexts). Another property that public key updates for kuKEM* must fulfill – in contrast to state updates in KIND secure unidirectional RKE as in [20] – is that they must not leak any information on the according secret key during the update computation. In the following we first explain these sketched issues (and their origin) in more detail, then describe how we solve it, and present a reduction of KUOWR security to KINDR security of a generic URKE scheme.

Crucial Properties of kuKEM* Syntax and KUOWR security of kuKEM* (as well as KUOW security of kuKEM) have several implications that we explain below. As described before, the syntax of kuKEM* does not allow interactions between secret key holder and public key holder(s) to communicate information for the key parts' updates (see Figure 1). This condition originates from the utilization of kuKEM as a building block for the instantiation of sesquidirectional ratcheted key exchange (SRKE) [20]. This extended RKE notion requires the two communication participants' states to immediately become incompatible as soon as one of the participants receives a ciphertext that was manipulated by the adversary. Public key and secret key of the used kuKEM, as part of the respective state, are therefore updated independently in order to cause an immediate divergence between these key pair parts.[10]

A second property that immediately follows from the first one is that, for all public keys that are updated equally, a compatible secret key can be used to decapsulate ciphertexts from all these public keys. As a public key update can also be conducted by an adversary, the computation of this update itself must not reveal any information on encapsulated keys – especially not on a compatible secret key. We will further comment on this property when explaining that, even though KINDR security of URKE implies KUOWR security of kuKEM*, one can instantiate URKE KIND securely with standard key encapsulation mechanisms. In Appendix C, we describe why these prior instantiations of KIND secure URKE are insecure according to KINDR security.

When deriving the notion of kuKEM* and its KUOWR security, we take these properties into account, as the goal of this work is not to find the minimal building block for unidirectional RKE, but for RKE in general (e.g., also for sesquidirectional RKE).

Construction of kuKEM* *from* URKE The weaker KIND security of URKE (as in [20]) already allows that the sender's state s_A can always be exposed without affecting the security of any established keys (as long as this exposed state is not used to impersonate A towards B). Consequently, A's pure state reveals no information on encapsulated keys nor on B's secret key(s). KIND security of URKE further implies that B's state only reveals information on keys that have not yet been computed by B (while earlier computed keys stay secure). One can imagine A's state consequently as the public part of a (stateful) key pair and B's state as the secret counterpart.

The two above mentioned crucial properties of KUOW(R) security are, however, not implied by KIND security when using s_A as the public key and s_B as the secret key of a kuKEM. Firstly, updating s_B (as part of receiving a ciphertext) requires that the ciphertext, generated during sending of A (and updating of s_A), is known by B but the syntax of kuKEM does not allow an interaction between public key holder and secret key holder. This issue can be solved by de-randomizing the snd algorithm. If A's state as part of the public key is updated via a de-randomized invocation of snd, the secret key holder can also obtain the ciphertext that A would produce for the same update (by invoking the de-randomized/deterministic snd) and then update s_B with this ciphertext via rcv. A conceptional depiction of this is in Figure 8. Thereby the secret key is defined to contain s_A in addition to s_B.

Secondly, in the URKE construction of Poettering and Rösler [20] A temporarily computes secrets of B that match A's updated values during sending. As a result, normal KIND security allows that a de-randomized snd invocation reveals the secrets of B to an adversary if s_A is known (see Appendix C for a detailed description of this attack). In order to solve this issue, the security

[10] A full description of the attack that is prevented by independent key updates is in [20] Appendix A.2.

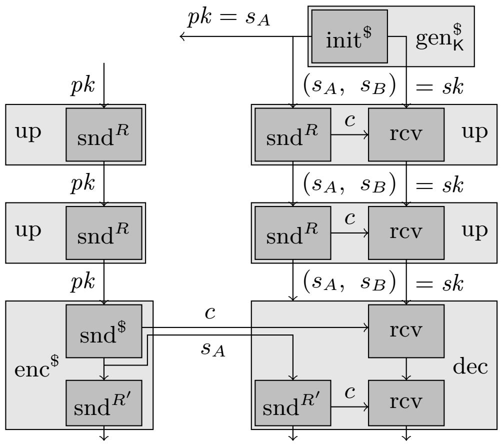


Fig. 8: Conceptual depiction of kuKEM* construction from generic URKE scheme. The symbol in the upper index of an algorithm name denotes the source of random coins ('\$' indicates uniformly sampled). R is a fixed value. For clarity we omit ad inputs and k outputs (cf. Figure 1).

definition of URKE must ensure that future encapsulated keys' security is not compromised if snd is invoked under a known state s_A and with random coins that are chosen by an adversary (i.e., KINDR security).

Our generic construction of a KUOWR secure kuKEM* from a generic KINDR secure URKE scheme is depicted in Figure 9. As described before, the public key contains state s_A and the secret key contains both states (s_A, s_B) that are derived from the init algorithm. In order to update the public key, the snd algorithm is invoked on state s_A, with the update associated data, and fixed randomness. The output key and ciphertext are thereby ignored. Accordingly, the secret key is updated by first invoking the snd algorithm on state s_A with the same fixed randomness and the update associated data. This time the respective ciphertext from A to B is not omitted but used as input to rcv algorithm with the same associated data under s_B.

Encapsulation and decapsulation are conducted by invoking snd probabilistically and rcv respectively. In order to separate updates from en-/decapsulation, a '0' or '1' is prepended to the associated data input of snd and rcv respectively. For bounding the probability of a ciphertext collision in the proof, a randomly sampled '$\underline{\text{c}}$ollision $\underline{\text{k}}$ey' ck is attached to the associated data of the snd invocation in encapsulation. In order to accordingly add ck to the associated data of rcv as part of the decapsulation, ck is appended to the ciphertext. Since state s_A, output by the snd algorithm during the encapsulation, is computed probabilistically, it is also attached to the encapsulation ciphertext, so that (the other) s_A, embedded in the secret key, can be kept compatible with the public key holder's. To bind ck and s_A to the ciphertext, both are integrity protected by a message authentication code (MAC) that takes one part of the key from the snd invocation as MAC key (only the remaining key bytes are output as the encapsulated kuKEM* key). Additionally the whole ciphertext (i.e., URKE ciphertext, collision key, state s_A, and MAC tag) is used as associated data for an additional 'internal update' of the public key and the secret key in encapsulation and decapsulation respectively. This is done to escalate manipulations of collision key, state s_A, or MAC tag (as part of the ciphertext) back into the URKE states s_A and s_B (as part of public key and secret key). For full details on the rationales behind these two *binding* steps we refer the reader to the proof.

```
Proc $\mathrm{gen}_{\mathsf{K}}$
$(s_A, s_B) \leftarrow_{\$} \mathrm{init}$
$pk \leftarrow s_A$
$sk \leftarrow (s_A, s_B)$
Return $(pk, sk)$

Proc $\mathrm{up}(pk, ad)$
$(pk, \_, \_) \leftarrow \mathrm{snd}(pk, (0, ad); 0)$
Return $pk$

Proc $\mathrm{enc}(pk)$
$ck \leftarrow_{\$} \mathcal{K}$
$(pk, (k, k.m), c') \leftarrow_{\$} \mathrm{snd}(pk, (1, ck))$
$\tau \leftarrow \mathrm{tag}(k.m, (ck, pk, c'))$
$c \leftarrow (ck, pk, c', \tau)$
$(pk, \_, \_) \leftarrow \mathrm{snd}(pk, (2, c); 0)$
Return $(pk, k, c)$

Proc $\mathrm{up}(sk, ad)$
$(s_A, s_B) \leftarrow sk$
$(s_A, \_, c) \leftarrow \mathrm{snd}(s_A, (0, ad); 0)$
$(s_B, \_) \leftarrow \mathrm{rcv}(s_B, (0, ad), c)$
$sk \leftarrow (s_A, s_B)$
Return $sk$

Proc $\mathrm{dec}(sk, c)$
$(s_A, s_B) \leftarrow sk$
$(ck, pk, c', \tau) \leftarrow c$
$(s_B, (k, k.m)) \leftarrow \mathrm{rcv}(s_B, (1, ck), c')$
Require $\mathrm{vfy}_{\mathsf{M}}(k.m, (ck, pk, c'), \tau)$
$(s_A, \_, c'') \leftarrow \mathrm{snd}(pk, (2, c); 0)$
$(s_B, \_) \leftarrow \mathrm{rcv}(s_B, (2, c), c'')$
$sk \leftarrow (s_A, s_B)$
Return $(sk, k)$
```

Fig. 9: Construction of a key-updatable KEM from a generic URKE scheme $\mathsf{UR} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$ and one-time message authentication code $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_{\mathsf{M}})$.

Interestingly, the public key holder can postpone the de-randomized snd invocation for public key updates until encapsulation and instead only remember the updates' associated data without compromising security. However, the updates of the secret key must be performed immediately as otherwise an exposure of the current secret key reveals also information on its past versions. Thereby the computation of snd in the up algorithm must be conducted during the secret key update without interaction between public key holder and secret key holder.

Theorem 2. *If* URKE *scheme* UR *is* KINDR *secure according to Figure 6, one-time* MAC M *is* SUF *secure according to Figure 14, and for all* $(k, k.m) \in \mathcal{K}_{\mathsf{UR}}$ *it holds that* $k \in \mathcal{K}_{\mathsf{K}}$ *and* $k.m \in \mathcal{K}_{\mathsf{M}}$*, then* kuKEM* *scheme* K *from Figure 9 is* KUOWR *secure according to Figure 4 with*

$$\begin{aligned}\mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{A}) &\leq q_{\mathrm{Gen}} q_{\mathrm{Enc}} \cdot \left(\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{1}{|\mathcal{K}|}\right) \\ &\leq q_{\mathrm{Gen}} q_{\mathrm{Enc}} \cdot \left(2 \cdot \mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}}) + \frac{1}{|\mathcal{K}|}\right)\end{aligned}$$

where $\mathcal{A}$ is an adversary against KUOWR *security, $\mathcal{B}_{\mathsf{UR}}$ is an adversary against* KINDR *security of* UR*, $\mathcal{B}_{\mathsf{M}}$ is an adversary against* SUF *security of* M*, q_{Gen} and q_{Enc} are the number of* Gen *and* Enc *queries by $\mathcal{A}$ respectively, $\mathcal{K}$ is the space from which ck is sampled, and the running time of $\mathcal{A}$ is approximately the running time of $\mathcal{B}_{\mathsf{UR}}$ and $\mathcal{B}_{\mathsf{M}}$.*

In Appendix E we show how to construct an SUF secure one-time MAC from a generic KINDR secure URKE scheme, which implies the second term in Theorem 2. We prove Theorem 2 below and provide a formal pseudo-code version of the simulation's game hops in Figure 10.

Proof (Theorem 2).
We conduct the proof in four game hops: In the first game hop we guess for which instance the first valid Solve oracle query is provided by the adversary; in the second game hop, we guess for which Enc oracle query of the previously guessed instance the first valid Solve oracle query is provided; additionally the simulation aborts in this game hop if the adversary crafts this first valid ciphertext and provides it to the Dec oracle before it is output by the Enc oracle; in the third game hop, we replace the key, output by the first snd invocation in this guessed Enc oracle query by a randomly sampled key (which is reduced to KINDR security of UR); in the final game hop, we abort on a MAC forgery, provided to the Dec oracle, that belongs to the ciphertext that is output by the guessed Enc oracle query (which is reduced to the SUF security of M).

Game 0 This game is equivalent to the original KUOWR game.

Game 1 The simulation guesses for which instance n_{Gen} the first key k^* is provided to the Solve oracle such that the secret key for decapsulation is not marked exposed (i.e., $tr^* \notin \mathrm{XP}_{n_{\mathrm{Gen}}}$) and the provided key equals the indicated challenge key (i.e., $k^* = \mathrm{CK}_{n_{\mathrm{Gen}}}[tr^*]$). Therefore n_{Gen} is randomly sampled from $[q_{\mathrm{Gen}}]$, where q_{Gen} is the number of Gen oracle queries by the adversary. The reduction aborts if n_{Gen} is not the instance for which the first valid Solve oracle query is provided (see Figure 10 lines 47,50).

Consequently we have $\mathrm{Adv}^{G_0} = q_{\mathrm{Gen}} \cdot \mathrm{Adv}^{G_1}$.

Game 2 The simulation guesses in which of n_{Gen}'s Enc queries the challenge is created, that is the first valid query to the Solve oracle by the adversary. Therefore n_{Enc} is randomly sampled from $[q_{\mathrm{Enc}}]$ and the simulation aborts if either the randomness for the n_{Enc}'s Enc query is manipulated as thereby no challenge would be created (lines 28,29), or the first valid query to the Solve oracle is for another challenge than the one created by n_{Gen}'s n_{Enc}th Enc query (lines 48,50), or a secret key that helps to trivially solve the challenge from n_{Gen}'s n_{Enc}th Enc query is exposed (lines 29,81).

In addition, the simulation aborts if, before the n_{Gen}'s n_{Enc}th Enc query was made, Dec was queried on a ciphertext (with the same preceding transcript) that contains the same URKE ciphertext and 'collision key' *ck* as n_{Gen}'s n_{Enc}th Enc query (lines 28,29). As the probability of a collision in the URKE transcript (i.e., associated data and ciphertext of the first snd invocation of n_{Gen}'s n_{Enc}th Enc query were previously already provided to n_{Gen}'s n_{Enc}th Dec query under the same preceding transcript) is bounded by a collision in the the key space $\mathcal{K}$ (as thereby ck as associated data must collide), we have $\mathrm{Adv}^{G_1} = q_{\mathrm{Enc}} \cdot \left(\mathrm{Adv}^{G_2} + \frac{1}{|\mathcal{K}|}\right)$.

Game 3 The simulation replaces the output $(k, k.m)$ from the first snd invocation of n_{Gen}'s n_{Enc}th Enc query by values randomly sampled.

An adversary that can distinguish between **Game 2** and **Game 3** can be turned into an adversary that breaks KINDR security of URKE scheme UR. We describe the reduction below: The reduction obtains n_{Gen}'s public key in oracle Gen via oracle ExposeA from the KINDR game. Invocations of snd in Up_S to n_{Gen} are replaced by SndA and ExposeA queries. Invocations of snd in Up_R to n_{Gen} are processed by the reduction itself and the subsequent rcv invocations are replaced by RcvB queries. The state s_B in queries to Expose for n_{Gen} is obtained via ExposeB queries to the KINDR game. For all queries to Enc of n_{Gen} the snd invocations are replaced by SndA and ExposeA queries. kuKEM* key and MAC key $(k, k.m)$ for n_{Gen}'s Enc oracle queries are obtained via Reveal – except for n_{Gen}'s n_{Enc}th Enc query, in which these two keys are obtained from the Challenge oracle in the KINDR game. Invocations of rcv in the Dec oracle for n_{Gen} are replaced by RcvB queries and Reveal queries (in case the respective key was not already computed in the Enc oracle). The snd invocation in oracle Dec is directly computed by the reduction.

In order to show that manipulations of transcripts in the KUOWR game manipulate equivalently the transcripts in the KINDR game (such that the state s_A in the public key diverges from state s_B in the secret key iff the transcripts $trs_{n_{\mathrm{Gen}}}$ and $trr_{n_{\mathrm{Gen}}}$ diverge), we define the translation array $\mathrm{TR}[\cdot]$ that maps the transcript of n_{Gen} in the KUOWR game to the according transcripts in the KINDR game.

As **Game 2** aborts if n_{Gen}'s n_{Enc}th Enc query entails no valid KINDR challenge, or if the respective ciphertext was already crafted by the adversary (and provided to the Dec oracle), an adversary, distinguishing the real key pair $(k, k.m)$ from the randomly sampled one, breaks KINDR security. Formally, the solution for n_{Gen}'s n_{Enc}th Enc query to the Solve oracle is compared with the challenge key k from the KINDR Challenge oracle (which is obtained during n_{Gen}'s n_{Enc}th Enc query): If the keys equal, the reduction terminates with $b' = 0$ (as thereby the KINDR game's challenge entailed the real key), otherwise it terminates with $b' = 1$.

Consequently we have $\mathrm{Adv}^{G_2} \leq \mathrm{Adv}^{G_3} + \mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}})$.

Game 4 The only way, the adversary can win in **Game 3**, is to keep secret key and public key of n_{Gen} compatible (by updating them equivalently and forwarding all Enc queries to the Dec oracle) and then forwarding only the URKE ciphertext c' of n_{Gen}'s n_{Enc}th Enc query to the Dec oracle while manipulating parts of the remaining challenge ciphertext. Thereby the Dec oracle outputs the correct challenge key such that the adversary trivially wins.[11]

[11] Please note that after this manipulation, the states s_A and s_B in the public key and secret key respectively diverge, but the key, output by the Dec oracle, still equals the challenge key. In case, the URKE ciphertext c' from the challenge ciphertext

We therefore define **Game 4** to let the simulation abort if a forgery of the MAC tag for the challenge ciphertext is provided to the Dec oracle. Distinguishing between **Game 3** and **Game 4** can hence be reduced to the SUF security of the one-time MAC M. We describe the reduction below: Instead of sampling $k.m$ randomly, the MAC tag for n_{Gen}'s n_{Enc}th Enc query is derived from the Tag oracle of the SUF game. Since an abort requires that the URKE challenge ciphertext c' is indeed received in oracle Dec (and also the transcripts prior to this ciphertext equal for $trs_{n_{\mathrm{Gen}}}$ and $trr_{n_{\mathrm{Gen}}}$), the URKE key (containing $k.m$) equals. As a consequence, a crafted ciphertext (pk, c', τ), provided to the Dec oracle, is a forgery τ for message (pk, c') in the SUF game.

Consequently we have $\mathrm{Adv}^{G_3} \leq \mathrm{Adv}^{G_4} + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}})$.

As the challenge key from n_{Gen}'s n_{Enc}th Enc query is randomly sampled and cannot be derived from any other oracle, the advantage of winning in **Game 4** is $\mathrm{Adv}^{G_4} = 0$.

Summing up the advantages above, we have:

$$\begin{aligned}\mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{A}) &\leq q_{\mathrm{Gen}} q_{\mathrm{Enc}} \cdot \left(\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{1}{|\mathcal{K}|}\right) \\ &\leq q_{\mathrm{Gen}} q_{\mathrm{Enc}} \cdot \left(2 \cdot \mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}}) + \frac{1}{|\mathcal{K}|}\right)\end{aligned}$$

where the latter follows from an SUF secure one-time MAC construction from a generic KINDR secure URKE scheme UR (which is described in Appendix E).

□

7 Discussion

Our results clearly show that key-updatable key encapsulation is a necessary building block for optimally secure ratcheted key exchange, if the security definition of the latter regards manipulation of the algorithm invocations' random coins. As unidirectional RKE can naturally be built from sesquidirectional RKE, which in turn can be built from bidirectional RKE (which can be derived from optimally secure group RKE), our results are expected to hold also for the according security definitions under these extended communication settings. In contrast, security definitions of ratcheting that restrict the adversary more than necessary in exposing the local state or in solving embedded game challenges (i.e., by excluding more than unpreventable attacks) allow for instantiations that can dispense with these inefficient building blocks.

However, the two previous security definitions fulfilled by constructions that use kuKEM as a building block (cf. Table 1) consider only *randomness reveal* [13] or even *secure randomness* [20]. This raises the question whether using kuKEM in these cases was indeed necessary (or not). The resulting gap between the notions of ratcheting that can be built from only standard PKC and our optimally

is already provided manipulately to the Dec oracle, the challenge key is already independent from the key, computed in the Dec oracle.

secure URKE definition with *randomness manipulation*, implying kuKEM, will be discussed in the following.

Implications under Randomness Reveal The core of our proof (showing that URKE implies kuKEM under randomness manipulation) is to utilize URKE's state update in algorithms snd and rcv for realizing public key and secret key updates in kuKEM's up algorithm. In order to remove the otherwise necessary communication between snd and rcv algorithms of RKE, snd is de-randomized by fixing its random coins to a static value. While this de-randomization *trick* is not immediately possible if the reduction to URKE KIND security cannot manipulate the randomness of snd invocations, one can utilize a programmable random oracle to emulate it: instead of fixing the (input) random coins of snd invocations to a static value, one could derive these coins from the output of a random oracle on input of the respective update's associated data (i.e., *ad* input of algorithm up). Additionally, instead of directly forwarding the update's associated data to the associated data input of snd, another random oracle could be interposed between them. The reduction then simply pre-computes all kuKEM up invocations independent of associated data inputs by querying the SndA oracle in the URKE KIND game on random associated data strings. Then the reduction reveals all used random coins in the URKE KIND game and programs them as output into the random oracle lazily (i.e., as soon as the adversary queries the random oracle on update associated data strings). By correctly guessing, which of the adversary's random oracle queries fit its queried kuKEM update invocations, the reduction can perform the same de-randomization trick as in our proof. The probability of guessing correctly is, however, exponential in the number of queried kuKEM updates such that a useful implication may only be derivable for a constant number of queried updates.

In conclusion, we conjecture that URKE under randomness reveal already requires the use of a kuKEM-like building block with a constantly bounded number of public key and secret key updates. Thereby we argue that our proof approach partially carries over to the case of randomness reveal. This would indicate that the use of a kuKEM-like building block in the construction of Jaeger and Stepanovs [13] is indeed necessary. The formal analysis of this conjecture is left as an open question for future work.

Implications under Secure Randomness For optimal security under secure randomness, Poettering and Rösler [20] show that URKE can be instantiated from standard PKC only (cf. Table 1). In contrast, their construction for sesquidirectional RKE (SRKE: a restricted interactive RKE variant) uses kuKEM for satisfying optimal security under secure randomness. Since a reduction towards SRKE (under KIND security with secure randomness) has no access to random coins respectively used in the RKE algorithms, our de-randomization trick seems inapplicable. Furthermore, while the RKE algorithms snd and rcv can use exchanged ciphertexts for their state updates, generically transforming this state update to realize a 'silent', non-interactive key update needed for kuKEM without our trick appears (at least) problematic.

Nevertheless, it is likely that SRKE KIND security under secure randomness requires kuKEM-like building blocks. This intuition is based on an example attack by Poettering and Rösler [19, Appendix B.2]. It illustrates that a key k^*, computed by any secure SRKE construction under the following attack, needs to be indistinguishable from a random key according to this security notion. The attack proceeds as follows: 1. Alice's and Bob's states are exposed ($s_A \leftarrow \mathrm{ExposeA}(\epsilon)$; $s_B \leftarrow \mathrm{ExposeB}(\epsilon)$), 2. Bob sends update information to Alice (which is possible in SRKE) to recover from his exposure ($c \leftarrow \mathrm{SndB}(\epsilon, \epsilon)$; $\mathrm{RcvA}(\epsilon, c)$). Keys established by Alice after receiving the update information are required to be secure again. Translated to the kuKEM setting, this step corresponds to Bob generating a new key pair and publishing the respective public key. 3. Simultaneously Alice is impersonated towards Bob ($(s'_A, k', c') \leftarrow_\$ \mathrm{snd}_A(s_A, \epsilon)$; $\mathrm{RcvB}(\epsilon, c')$). This requires Bob's state to become incompatible with Alice's state. In the kuKEM setting, this corresponds to the secret key being updated with c' as associated data. Note that c' can be independent of Bob's state update from step 2 via c, and the computation of c' is controlled by the adversary. 4. Afterwards Bob's state is again exposed ($s'_B \leftarrow \mathrm{ExposeB}((\epsilon, c)\|(\epsilon, c'))$). 5. Finally, Alice sends and establishes key k^* which is required to be secure ($c'' \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$). 6. Exposing Alice's state thereafter should not harm security of k^* ($s''_A \leftarrow \mathrm{ExposeA}((\epsilon, c''))$).

We observe that, as with a kuKEM public key, Alice's state is publicly known during the entire attack. Only Alice's random coins when establishing k^* and updating her state, and Bob's random coins when sending, as well as his resulting state until he receives c' are hidden towards the adversary. We furthermore note that, by computing ciphertext c', the adversary controls Bob's state update. As a consequence, Bob's state update must reach forward-secrecy for key k^* with respect to adversarially chosen associated update data c' and Bob's resulting (diverged) state s'_B.

All in all, the security requirements highlighted by this attack emphasize the similarity of kuKEM's and SRKE's security. Nevertheless, we note that all our attempts to apply our proof technique for this case failed due to the above mentioned problems. Therefore, formally substantiating or disproving the intuition conveyed by this attack remains an open question for future work.

Open Questions and Impact With our work we aim to motivate research on another remaining open problem: can key-updatable KEM be instantiated more efficiently than generically from HIBE? It is, in contrast, evident that equivalence between HIBE and RKE is unlikely as constructions of the latter only utilize "one identity path" of the whole "identity tree" of the former.

Conclusively, we note that defining security for, and constructing schemes of interactive ratcheted key exchange variants (i.e., under bidirectional communication) is highly complicated and consequently error-prone.[5] By providing generic constructions (instead of ad-hoc designs) and grasping core components and concepts of ratcheted key exchange, complexity is reduced and sources of errors are eliminated. Additionally, our equivalence result serves as a benchmark for current and future designs of ratcheted key exchange – especially group RKE. For future constructions that only rely on standard public key cryptography either of the following questions may arise: how far is the adversary restricted such that our implication is circumvented, or how far is the construction secure under the respective security definition?

References

1. Alwen, J., Capretto, M., Cueto, M., Kamath, C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol. Cryptology ePrint Archive, Report 2019/1489 (2019), `https://eprint.iacr.org/2019/1489`
2. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019)
3. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. Cryptology ePrint Archive, Report 2019/1189 (2019), `https://eprint.iacr.org/2019/1189`
4. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 619–650. Springer, Heidelberg (Aug 2017)
5. 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, October 28, 2004. pp. 77–84. ACM (2004)
6. Caforio, A., Durak, F.B., Vaudenay, S.: On-demand ratcheting with security awareness. Cryptology ePrint Archive, Report 2019/965 (2019), `https://eprint.iacr.org/2019/965`
7. Checkoway, S., Niederhagen, R., Everspaugh, A., Green, M., Lange, T., Ristenpart, T., Bernstein, D.J., Maskiewicz, J., Shacham, H., Fredrikson, M.: On the practical exploitability of dual EC in TLS implementations. In: Fu, K., Jung, J. (eds.) USENIX Security 2014. pp. 319–335. USENIX Association (Aug 2014)
8. Cohn-Gordon, K., Cremers, C.J.F., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017. pp. 451–466 (2017)
9. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. Cryptology ePrint Archive, Report 2018/889 (2018), `https://eprint.iacr.org/2018/889`
10. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. In: Attrapadung, N., Yagi, T. (eds.) IWSEC 19. LNCS, vol. 11689, pp. 343–362. Springer, Heidelberg (Aug 2019)
11. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your ps and qs: Detection of widespread weak keys in network devices. In: Kohno, T. (ed.) USENIX Security 2012. pp. 205–220. USENIX Association (Aug 2012)
12. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017, Part I. LNCS, vol. 10677, pp. 341–371. Springer, Heidelberg (Nov 2017)

13. Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 33–62. Springer, Heidelberg (Aug 2018)
14. Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. Cryptology ePrint Archive, Report 2018/553 (2018), `https://eprint.iacr.org/2018/553`
15. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 159–188. Springer, Heidelberg (May 2019)
16. Langley, A.: Source code of Pond (05 2016), `https://github.com/agl/pond`
17. Marlinspike, M., Perrin, T.: The double ratchet algorithm (11 2016), `https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf`
18. Off-the-Record Messaging. `http://otr.cypherpunks.ca` (2016)
19. Poettering, B., Rösler, P.: Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296 (2018), `https://eprint.iacr.org/2018/296`
20. Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 3–32. Springer, Heidelberg (Aug 2018)
21. Raimondo, M.D., Gennaro, R., Krawczyk, H.: Secure off-the-record messaging. In: Atluri, V., di Vimercati, S.D.C., Dingledine, R. (eds.) Proceedings of the 2005 ACM WPES 2005, Alexandria, VA, USA, November 7, 2005. pp. 81–89. ACM (2005)
22. Rescorla, E., Salter, M.: Extended random values for tls (2009), `https://tools.ietf.org/html/draft-rescorla-tls-extended-random-02`
23. Yilek, S., Rescorla, E., Shacham, H., Enright, B., Savage, S.: When private keys are public: results from the 2008 debian openssl vulnerability. In: Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009, Chicago, Illinois, USA, November 4-6, 2009. pp. 15–27 (2009)

A Proof Figure for KUOWR Security under KINDR Security

```
**Simulation** $\mathcal{S}$
$n \leftarrow 0$
$n_{\text{Gen}} \leftarrow_\$ [q_{\text{Gen}}]$                                   $\mathbf{G_{\geq 1}}$
$n_{\text{Enc}} \leftarrow_\$ [q_{\text{Enc}}]$; $e \leftarrow 0$                  $\mathbf{G_{\geq 2}}$
$tr^\circ \leftarrow \epsilon$; $c^\circ \leftarrow \perp$                          $\mathbf{G_{\geq 2}}$
$k^\bullet \leftarrow \perp$; $k.m^\bullet \leftarrow \perp$; $ck^\bullet \leftarrow \perp$; $c^\bullet \leftarrow \perp$   $\mathbf{G_{\geq 3}}$
Invoke $\mathcal{A}$
Stop with 0

**Oracle** Gen
$n \leftarrow n + 1$
If $n = n_{\text{Gen}}$: $\mathrm{T}[\cdot] \leftarrow \perp$                       $\mathbf{G_{=3}}$
$(s_A, s_B) \leftarrow_\$$ init
$pk_n \leftarrow s_A$; $sk_n \leftarrow (s_A, s_B)$
$\mathrm{CK}_n[\cdot] \leftarrow \perp$; $\mathrm{XP}_n \leftarrow \emptyset$
$trs_n \leftarrow \epsilon$; $trr_n \leftarrow \epsilon$
$SK_n[\cdot] \leftarrow \perp$; $SK_n[trr_n] \leftarrow sk_n$
Return $pk_n$

**Oracle** $\mathrm{Up}_S(i, ad)$
Require $1 \leq i \leq n \wedge ad \in \mathcal{AD}$
$(pk_i, k, c) \leftarrow \mathrm{snd}(pk_i, (0, ad); 0)$
If $i = n_{\text{Gen}}$:                                                             $\mathbf{G_{=3}}$
    $\mathrm{T}[trs_i \| ad] \leftarrow \mathrm{T}[trs_i] \| ((0, ad), c)$         $\mathbf{G_{=3}}$
$trs_i \xleftarrow{\shortparallel} ad$
Return $pk_i$

**Oracle** $\mathrm{Enc}(i, rc)$
Require $1 \leq i \leq n$
Require $rc \in \mathcal{R} \cup \{\epsilon\}$
If $rc = \epsilon$: $mr \leftarrow \mathtt{F}$; $rc \leftarrow_\$ \mathcal{R}$
Else: $mr \leftarrow \mathtt{T}$
$ck \leftarrow_\$ \mathcal{K}$
$(pk_i, (k, k.m), c') \leftarrow_\$ \mathrm{snd}(pk_i, (1, ck); rc)$
If $i = n_{\text{Gen}} \wedge e = n_{\text{Enc}}$:                                  $\mathbf{G_{\geq 2}}$
    If $mr = \mathtt{T} \vee (\exists pk' \in \mathcal{PK}, \tau' \in \mathcal{T}$ :
        $trs_i \| (ck, pk', c', \tau') \in \mathrm{XP}_i$
        $\vee trs_i \| (ck, pk', c', \tau') \preceq trr_i)$:                          $\mathbf{G_{\geq 2}}$
        Abort                                                                          $\mathbf{G_{\geq 2}}$
    $(ck^\bullet, c^\bullet) \leftarrow (ck, c')$                                     $\mathbf{G_{\geq 3}}$
    $(k^\bullet, k.m^\bullet) \leftarrow_\$ \mathcal{K}_{\mathsf{UR}}$; $(k, k.m) \leftarrow (k^\bullet, k.m^\bullet)$   $\mathbf{G_{\geq 3}}$
$\tau \leftarrow_\$ \mathrm{tag}(k.m, (ck, pk_i, c'))$
$c \leftarrow (ck, pk_i, c', \tau)$
$(pk_i, c'', k'') \leftarrow \mathrm{snd}(pk_i, (2, c); 0)$
If $i = n_{\text{Gen}}$:                                                             $\mathbf{G_{\geq 2}}$
    If $e = n_{\text{Enc}}$:                                                         $\mathbf{G_{\geq 2}}$
        $tr^\circ \leftarrow trs_i$; $c^\circ \leftarrow c$                         $\mathbf{G_{\geq 2}}$
    $e \leftarrow e + 1$                                                             $\mathbf{G_{\geq 2}}$
    $\mathrm{T}[trs_i \| c] \leftarrow \mathrm{T}[trs_i] \| ((1, ck), c') \| ((2, c), c'')$   $\mathbf{G_{=3}}$
$trs_i \xleftarrow{\shortparallel} c$
If $mr = \mathtt{F}$: $\mathrm{CK}_i[trs_i] \leftarrow k$
Return $(pk_i, c)$

**Oracle** $\mathrm{Solve}(i, tr, k)$
Require $1 \leq i \leq n$
Require $tr \notin \mathrm{XP}_i$
Require $\mathrm{CK}_i[tr] \neq \perp$
Reward $k = \mathrm{CK}_i[tr]$                                                       $\mathbf{G_{<1}}$
If $i = n_{\text{Gen}}$:                                                             $\mathbf{G_{\geq 1}}$
    If $tr = tr^\circ \| c^\circ$:                                                   $\mathbf{G_{\geq 2}}$
        Reward $k = \mathrm{CK}_i[tr]$                                               $\mathbf{G_{\geq 1,2}}$
Else if $k = \mathrm{CK}_i[tr]$: Abort                                               $\mathbf{G_{\geq 1}}$
Return

**Oracle** $\mathrm{Up}_R(i, ad)$
Require $1 \leq i \leq n \wedge ad \in \mathcal{AD}$
$(s_A, s_B) \leftarrow SK_i[trr_i]$
$(s_A, k, c) \leftarrow \mathrm{snd}(s_A, (0, ad); 0)$
$(s_B, k) \leftarrow \mathrm{rcv}(s_B, (0, ad), c)$
If $i = n_{\text{Gen}}$:                                                             $\mathbf{G_{=3}}$
    $\mathrm{T}[trr_i \| ad] \leftarrow \mathrm{T}[trr_i] \| ((0, ad), c)$         $\mathbf{G_{=3}}$
$trr_i \xleftarrow{\shortparallel} ad$
$SK_i[trr_i] \leftarrow (s_A, s_B)$
Return

**Oracle** $\mathrm{Dec}(i, c)$
Require $1 \leq i \leq n \wedge c \in \mathcal{C}$
$(s_A, s_B) \leftarrow SK_i[trr_i]$
$(ck, pk, c', \tau) \leftarrow c$
$(s_B, (k, k.m)) \leftarrow \mathrm{rcv}(s_B, (1, ck), c')$
If $trr_i = tr^\circ \wedge (ck^\bullet, c^\bullet) = (ck, c') \neq (\perp, \perp)$:   $\mathbf{G_{\geq 3}}$
    $(k, k.m) \leftarrow (k^\bullet, k.m^\bullet)$                                   $\mathbf{G_{\geq 3}}$
Require $\mathrm{vfy}_{\mathsf{M}}(k.m, (ck, pk, c'), \tau)$
If $trr_i = tr^\circ \wedge c \neq c^\circ$
    $\wedge (ck^\bullet, c^\bullet) = (ck, c') \neq (\perp, \perp)$:                 $\mathbf{G_{\geq 4}}$
    Abort                                                                              $\mathbf{G_{\geq 4}}$
$(s_A, k'', c'') \leftarrow \mathrm{snd}(pk, (2, c); 0)$
$(s_B, k''') \leftarrow \mathrm{rcv}(s_B, (2, c), c'')$
If $i = n_{\text{Gen}}$:                                                             $\mathbf{G_{=3}}$
    $\mathrm{T}[trr_i \| c] \leftarrow \mathrm{T}[trr_i] \| ((1, ck), c') \| ((2, c), c'')$   $\mathbf{G_{=3}}$
$trr_i \xleftarrow{\shortparallel} c$
$SK_i[trr_i] \leftarrow (s_A, s_B)$
If $\mathrm{CK}_i[trr_i] \neq \perp$:
    Return
Return $k$

**Oracle** $\mathrm{Expose}(i, tr)$
Require $1 \leq i \leq n$
Require $SK_i[tr] \in \mathcal{PK} \times \mathcal{SK}$
If $tr \preceq tr^\circ$: Abort                                                      $\mathbf{G_{\geq 2}}$
$\mathrm{XP}_i \xleftarrow{\cup} \{tr^* \in (\mathcal{AD} \cup \mathcal{C})^* : tr \prec tr^*\}$
$(s_A, s_B) \leftarrow SK_i[tr]$
Return $(s_A, s_B)$
```

Fig. 10: Games of simulation for proof of KUOWR security for construction from Figure 9.

B Full Proof of URKE KINDR Security

Below we describe the full proof of Theorem 1 for which a sketch is in Section 5.

Proof (Theorem 1).

The high-level organization of our proof is as follows: We first expand the KINDR security game by replacing each generic call to URKE's algorithms with their actual instantiation in the UR construction. Then, starting from this game, we insert and remove lines of codes in the game, in such a way that the interaction of $\mathcal{A}$ with the game never changes, i.e. the game-hops are merely bridging steps. Therefore, we define new games that look different in code, but are essentially same, and $\mathcal{A}$'s advantage does not change from one to another. Our motivation behind these syntactic updates is to prepare the game for reductions to KUOWR and SUF security games of kuKEM* and MAC respectively. Finally, we describe in detail how these reductions work.

Although, the type of transcripts collected by variables $tr \in (\mathcal{AD} \times \mathcal{C} \times \mathcal{T})^*$ (after expansion of the UR construction w.r.t. the original KINDR definition) and $t \in (\mathcal{AD} \times \mathcal{C})^*$ (from the UR construction) are different, tr can be reduced to $(\mathcal{AD} \times \mathcal{C})^*$ by removing the tag part τ from each element in the list. Therefore, $t \preceq tr$, $t \prec tr$ and $\{t\} \in \mathcal{X} \subseteq (\mathcal{AD} \times \mathcal{C} \times \mathcal{T})^*$ should be interpreted as abuse of notation frequently used throughout the proof.

Game 0 The original KINDR game of URKE where invocation of UR's algorithms are expanded according to Figure 7. We also simulate the random oracle with H as described in Figure 12.

Game 1 In order to prepare the game simulation to the KUOWR reduction, we make few syntactical modifications in the game which does not affect the advantage of the adversary. Namely, the new simulation keeps two lists: CK for <u>c</u>hallengeable <u>k</u>eys and NK for <u>n</u>on-challengeable <u>k</u>eys. Intuitively, CK is synchronized with CK_i list defined in the KUOWR game in Figure 4, therefore any (non-trivial) hash query containing a key from CK will lead to a win. CK is updated only when the random coins used by the encapsulation are sampled by the oracle, but not chosen by the adversary. This condition is captured in line 24 in Figure 11.

On the other hand, NK represents the rest of the keys derived in the game and can be thought as complementary to CK. Although the keys from NK might allow the adversary $\mathcal{A}$ to win the KINDR game, it does not lead to a winning reduction in the KUOWR game, because the latter only rewards keys that belongs to CK (line 24 in Figure 4).

Game 2 We split the random oracle interfaces into two: H works with input (K, k, t) and G works only with t. The adversary uses H, and the simulator uses G for hash queries. G allows the simulator to produce keys $(k.o, K, k.m, k.u)$ by programming the hash output of the yet-unknown key k or even the symmetric state key K.

More precisely, dictionaries $L_{\text{H}}, L_{\text{G}}$ are used for bookkeeping, whose all entries are set to $\perp$ initially. Query inputs (K, k, t) and t are directly used as access keys to their corresponding entries in the dictionaries. For repeating queries, oracles return their previous answer for consistency. Otherwise, when a hash query is made with input (K, k, t) (resp. t), H (resp. G) checks whether there is a matching entry in L_{G} (resp. L_{H}), and if so, it copies the key into $L_{\text{H}}[K, k, t]$ (resp. $L_{\text{G}}[t]$). Otherwise, a fresh key is sampled for $L_{\text{H}}[K, k, t]$ (resp. $L_{\text{G}}[t]$).

We now focus on keeping G and H in harmony when queries are intertwined. This boils down to determining whether given (K, k, t) tuple matches a transcript t with respect to the history of states and keys of the game. This is captured by the Match in Figure 12 which takes (K, k, t) as input and decides whether (K, k) and t are matching each other, with the help of states S_A, S_B and derived key lists CK, NK.

Therefore, the game follows exactly same and the advantage of the adversary remains same.

Game 3 We now isolate K, which we call the symmetric state key, that resides in the both states of the sender and the receiver. Our ultimate goal is to show that the adversary cannot recover the symmetric state key K, except some trivial state exposures, therefore making a hash query with a matching K requires a collision on the domain $\mathcal{K}$.

Our argument is that since keys K are never used besides as input to random oracle, the view of $\mathcal{A}$ should be independent of choices of K values, therefore it has no information over them. This independence works so long as a symmetric state key K is not a part of states known by $\mathcal{A}$ which are due to state exposure queries and malicious random coins connected to these states. However, $\mathcal{A}$'s knowledge is strictly limited

```
Simulation $\mathrm{KINDR}^b_{\mathsf{UR}}(\mathcal{A})$
00 $trs \leftarrow \epsilon$; $trr \leftarrow \epsilon$; $\mathrm{XP}_A \leftarrow \emptyset$
01 $\mathrm{MR} \leftarrow \emptyset$; $\mathrm{KN} \leftarrow \emptyset$; $\mathrm{CH} \leftarrow \emptyset$
02 $S_A[\cdot] \leftarrow \perp$; $S_B[\cdot] \leftarrow \perp$; $key[\cdot] \leftarrow \perp$
03 $\mathrm{NK}[\cdot] \leftarrow \perp$; $\mathrm{CK}[\cdot] \leftarrow \perp$    $\mathbf{G}_{\geq 1}$
04 $\mathrm{TK}[\cdot] \leftarrow \perp$    $\mathbf{G}_{\geq 3}$
05 $\mathrm{MK}[\cdot] \leftarrow \perp$    $\mathbf{G}_{\geq 4}$
06 $kuowr \leftarrow \mathtt{F}$; $suf \leftarrow \mathtt{F}$; $coll \leftarrow \mathtt{F}$    $\mathbf{G}_{=5}$
07 $(pk, sk) \leftarrow_\$ \mathrm{gen}_{\mathsf{K}}$
08 $K \leftarrow_\$ \mathcal{K}$; $k.m \leftarrow_\$ \mathcal{K}$; $t \leftarrow \epsilon$
09 $s_A \leftarrow (pk, K, k.m, t)$
10 $s_B \leftarrow (sk, K, k.m, t)$
11 $S_A[trs] \leftarrow s_A$; $S_B[trr] \leftarrow s_B$; $b' \leftarrow_\$ \mathcal{A}$
12 Require $\mathrm{KN} \cap \mathrm{CH} = \emptyset$
13 Stop with $b'$

Oracle $\mathrm{SndA}(ad, rc)$
14 Require $ad \in \mathcal{AD} \wedge rc \in \mathcal{R} \cup \{\epsilon\}$
15 $(pk, K, k.m, t) \leftarrow s_A$
16 $pk' \leftarrow pk$    $\mathbf{G}_{\geq 1}$
17 If $rc = \epsilon$: $(pk, k, c) \leftarrow_\$ \mathrm{enc}(pk)$
18 Else: $(pk, k, c) \leftarrow \mathrm{enc}(pk; rc)$
19 $k.m \leftarrow \mathrm{MK}[t]$    $\mathbf{G}_{\geq 4}$
20 $\tau \leftarrow_\$ \mathrm{tag}(k.m, (ad, c))$
21 $C \leftarrow (c, \tau)$; $t \xleftarrow{\Vert} (ad, c)$
22 $(k.o, K, k.m, k.u) \leftarrow \mathrm{H}(K, k, t)$    $\mathbf{G}_{<2}$
23 $(k.o, K, k.m, k.u) \leftarrow \mathrm{G}(t)$    $\mathbf{G}_{\geq 2}$
24 If $rc = \epsilon$: $\mathrm{CK}[t] \leftarrow k$    $\mathbf{G}_{\geq 1}$
25 Else: $(\_, \mathrm{NK}[t], \_) \leftarrow \mathrm{enc}(pk'; rc)$    $\mathbf{G}_{\geq 1}$
26 $pk \leftarrow \mathrm{up}(pk, k.u)$
27 $s_A \leftarrow (pk, K, k.m, t)$
28 If $rc \neq \epsilon$:
29     $\mathrm{MR} \xleftarrow{\cup} \{trs \Vert (ad, C)\}$
30     If $trs \in \mathrm{XP}_A$:
31         $\mathrm{KN} \xleftarrow{\cup} \{trs \Vert (ad, C)\}$
32         $\mathrm{XP}_A \xleftarrow{\cup} \{trs \Vert (ad, C)\}$
33 $trs \xleftarrow{\Vert} (ad, C)$; $key[trs] \leftarrow k.o$; $S_A[trs] \leftarrow s_A$
34 Return $C$

Oracle $\mathrm{Reveal}(tr)$ (see Figure 6)

Oracle $\mathrm{Challenge}(tr)$ (see Figure 6)

Oracle $\mathrm{RcvB}(ad, C)$
35 Require $ad \in \mathcal{AD} \wedge C \in \mathcal{C} \wedge s_B \neq \perp$
36 If $trr \Vert (ad, C) \not\preceq trs$
       $\wedge \mathrm{LCP}(trs, trr) \in \mathrm{XP}_A$:
37     $\mathrm{KN} \xleftarrow{\cup} \{trr \Vert (ad, C)\}$
38 $(sk, K, k.m, t) \leftarrow s_B$
39 $(c, \tau) \leftarrow C$
40 $k.m \leftarrow \mathrm{MK}[t]$    $\mathbf{G}_{\geq 4}$
41 Require $\mathrm{vfy}_{\mathsf{M}}(k.m, (ad, c), \tau)$
42 If $trr \Vert (ad, C) \not\preceq trs$: $suf \leftarrow t \notin \mathrm{KN}$    $\mathbf{G}_{=5}$
43 $(sk, k) \leftarrow \mathrm{dec}(sk, c)$
44 If $\mathrm{CK}[trr \Vert (ad, C)] = \perp$:    $\mathbf{G}_{\geq 1}$
45     $\mathrm{NK}[trr \Vert (ad, C)] \leftarrow k$    $\mathbf{G}_{\geq 1}$
46 Require $k \neq \perp$; $t \xleftarrow{\Vert} (ad, c)$
47 $(k.o, K, k.m, k.u) \leftarrow \mathrm{H}(K, k, t)$    $\mathbf{G}_{<2}$
48 $(k.o, K, k.m, k.u) \leftarrow \mathrm{G}(t)$    $\mathbf{G}_{\geq 2}$
49 $sk \leftarrow \mathrm{up}(sk, k.u)$
50 $s_B \leftarrow (sk, K, k.m, t)$
51 If $s_B = \perp$: Return $\perp$
52 $trr \xleftarrow{\Vert} (ad, C)$; $key[trr] \leftarrow k.o$; $S_B[trr] \leftarrow s_B$
53 Return

Oracle $\mathrm{ExposeA}(tr)$ (see Figure 6)
54 Require $S_A[tr] \in \mathcal{S}_A$
55 $\mathrm{XP}_A \xleftarrow{\cup} \{tr\}$
56 $trace \leftarrow \{tr^* \in \mathcal{TR}^* : \forall tr' \in \mathcal{TR}^*$
       $(tr \prec tr' \preceq tr^* \implies tr' \in \mathrm{MR})\}$
57 $\mathrm{KN} \xleftarrow{\cup} trace$; $\mathrm{XP}_A \xleftarrow{\cup} trace$
58 If $\mathrm{TK}[t] = \perp$: $\mathrm{TK}[t] \leftarrow_\$ \mathcal{K}$    $\mathbf{G}_{\geq 3}$
59 $S_A[tr][1] \leftarrow \mathrm{TK}[t]$    $\mathbf{G}_{\geq 3}$
60 $S_A[tr][2] \leftarrow \mathrm{MK}[t]$    $\mathbf{G}_{\geq 4}$
61 Return $S_A[tr]$

Oracle $\mathrm{ExposeB}(tr)$
62 Require $S_B[tr] \in \mathcal{S}_B$
63 · $\mathrm{KN} \xleftarrow{\cup} \{tr^* \in \mathcal{TR}^* : tr \prec tr^*\}$
64 If $\mathrm{TK}[t] = \perp$: $\mathrm{TK}[t] \leftarrow_\$ \mathcal{K}$    $\mathbf{G}_{\geq 3}$
65 $S_B[tr][1] \leftarrow \mathrm{TK}[t]$    $\mathbf{G}_{\geq 3}$
66 $S_B[tr][2] \leftarrow \mathrm{MK}[t]$    $\mathbf{G}_{\geq 4}$
67 Return $S_B[tr]$
```

Fig. 11: The simulation of KINDR game, extended with regards to UR construction. $S_A[t][i]$ refers to i-th element of $S_A[t] = (pk, K, k.m, t)$ tuple where indexing starts from 0, and goes upto 3. Although, the type of transcripts collected by $tr \in (\mathcal{AD} \times \mathcal{C} \times \mathcal{T})^*$ and $t \in (\mathcal{AD} \times \mathcal{C})^*$ are different, tr can be reduced to $(\mathcal{AD} \times \mathcal{C})^*$ by removing the tag part τ from each element in the list.

to these chains of states. We utilize KN array introduced by the original game definition to keep track of symmetric state keys K known by $\mathcal{A}$.

In order to show independence of state keys K from rest of the values derived in the game, we use lazy sampling for K. Initially all values of TK are set to $\perp$. Then, we ignore K that resides in the state information completely, and always use the corresponding s$\underline{\text{t}}$ate $\underline{\text{k}}$ey from $\mathrm{TK}[t]$ for a transcript t. $\mathcal{A}$ can receive $\mathrm{TK}[t]$ values either by exposing the state of A or B; or through querying the hash oracle H. Therefore, we return these symmetric state keys to $\mathcal{A}$ by overwriting the values of $S_A[t]$,$S_B[t]$, in lines 59,65 of Figure 11.

```
**Oracle** $\mathrm{H}(K, k, t)$                                      $\mathbf{G}_{\geq 0}$
00  $(k.o, K, k.m, k.u) \leftarrow_\$ \mathcal{K}^4$
01  If $L_\mathrm{H}[K, k, t] \neq \bot$:
02      $(k.o, K, k.m, k.u) \leftarrow L_\mathrm{H}[K, k, t]$
03  Else if $L_\mathrm{G}[t] \neq \bot \wedge \mathrm{Match}(K, k, t)$:   $\mathbf{G}_{\geq 2}$
04      $(k.o, K, k.m, k.u) \leftarrow L_\mathrm{G}[t]$                $\mathbf{G}_{\geq 2}$
05      $K \leftarrow \mathrm{TK}[t]$                                  $\mathbf{G}_{\geq 3}$
06      $k.m \leftarrow \mathrm{MK}[t]$                                $\mathbf{G}_{\geq 4}$
07  $L_\mathrm{H}[K, k, t] \leftarrow (k.o, K, k.m, k.u)$
08  Return $(k.o, K, k.m, k.u)$

**Oracle** $\mathrm{G}(t)$                                           $\mathbf{G}_{\geq 2}$
09  $(k.o, K, k.m, k.u) \leftarrow_\$ \mathcal{K}^4$
10  If $L_\mathrm{G}[t] \neq \bot$:
11      $(k.o, K, k.m, k.u) \leftarrow L_\mathrm{G}[t]$
12  Else if $\exists k, K\ L_\mathrm{H}[K, k, t] \neq \bot$
            $\wedge \mathrm{Match}(K, k, t)$:
13      $(k.o, K, k.m, k.u) \leftarrow L_\mathrm{H}[K, k, t]$
14  Else: $\mathrm{MK}[t] \leftarrow_\$ \mathcal{K}$                     $\mathbf{G}_{\geq 4}$
15  $L_\mathrm{G}[t] \leftarrow (k.o, K, k.m, k.u)$
16  Return $(k.o, K, k.m, k.u)$

**Predicate** $\mathrm{Match}(K, k, t)$                              $\mathbf{G}_{\geq 2}$
17  If $t = \epsilon$: Return F
18  $t' \| (ad, c) \leftarrow t$
19  If $t \preceq trs$: $K' \leftarrow S_A[t][1]$                       $\mathbf{G}_{<3}$
20  Else: $K' \leftarrow S_B[t][1]$                                    $\mathbf{G}_{<3}$
21  If $\mathrm{TK}[t] = \bot$:                                        $\mathbf{G}_{\geq 3}$
22      $\mathrm{TK}[t] \leftarrow_\$ \mathcal{K}$; $K' \leftarrow \mathrm{TK}[t]$              $\mathbf{G}_{\geq 3}$
23  If $K' = K$:
24      $kuowr \leftarrow \mathrm{CK}[t] = k \wedge t \notin \mathrm{KN}$           $\mathbf{G}_{=5}$
25      $coll \leftarrow t \notin \mathrm{KN}$                              $\mathbf{G}_{=5}$
26      Return $\mathrm{CK}[t] = k \vee \mathrm{NK}[t] = k$
27  Return F
```

Fig. 12: The simulation of the random oracle during KINDR security game of UR construction through separate interfaces H (accessible by the adversary) and G (accessible by the game oracles). We assume there is mapping from the type of *tr* to the type of *t*, as explained in the caption of Figure 11.

Lastly, before returning $\mathrm{TK}[t]$ to $\mathcal{A}$, we also make sure that its value is initialized. This initialization is handled in lines 58,64 in Figure 11 and line 22 in Figure 12.

These changes are again syntactical and does not affect $\mathcal{A}$'s advantage.

Game 4 We perform the similar changes of **Game 3** for MAC keys. We add a new array MK for MAC keys. In SUF game, we recall that the adversary can request the generation of multiple MAC keys, and expose a subset of them adaptively. Its final goal is to forge a message and a MAC for one of the unexposed keys.

For ExposeA and ExposeB, we again overwrite $k.m$ keys from the array MK. For $\mathrm{G}(t)$ queries, we treat $k.m$ keys specially. Namely, for each fresh $\mathrm{G}(t)$ query, we initialize a new MAC key as $\mathrm{MK}[t]$. We again use the array KN to keep track of revealed MAC keys.

Game 5 We introduce three boolean flags, all of which are initially set to F. Conditioned on $\mathcal{A}$ making a special hash query, i.e. a hash query that reveals one of the challengeable keys, then these flags will assume the following meanings:

- *kuowr* flag in line 24 of Figure 12 captures whether the adversary made a special hash query, from which we can recover the key k to win the KUOWR game.
- *coll* flag in line 25 of Figure 12 captures $\mathcal{A}$'s correct guess on K for one of the states not exposed to $\mathcal{A}$ in one of its hash queries, which leads to a collision in $\mathcal{K}$.
- *suf* flag in line 42 of Figure 11 captures the RKE ciphertext forgery which contains a MAC forgery, and leads to a win in the SUF game.

Finally we classify all cases for which $\mathcal{A}$ wins the KINDR game by making a special hash query $\mathrm{H}(K, k, t)$ such that its matching $\mathrm{G}(t)$ query was made by the game simulation to derive either $\mathrm{CK}[t]$ or $\mathrm{NK}[t]$ with $t \notin \mathrm{KN}$. Suppose that a special hash query is made for one of:

- $\mathrm{CK}[t]$ where $t \preceq trs$. This sets the flag *kuowr* to T during the game, because the special hash query provides the correct key k of the underlying encapsulation.
- $\mathrm{NK}[t]$ where $t \preceq trs$. This implies that the last encapsulation query leading to $\mathrm{NK}[t]$ used malicious random coins chosen by $\mathcal{A}$, and $t \notin \mathrm{XP}_A$. This query sets the flag *coll* to T.

- NK$[t]$ where $t \not\preceq trs$. This implies that $t \preceq trr$ and $t \notin$ KN. Then, either the forgery flag *suf* or the collision flag *coll* is set to `T` during the game. The reason is that either B receives a MAC whose key is not exposed (meaning $coll = \texttt{F}$), or a previous hash query exposed the MAC key (meaning $coll = \texttt{T}$).

We should treat the ordering of these flags with care, as they will determine the order of abortion. By the definition of **Game 5**, since $kuowr \implies coll$ (i.e. whenever the former is triggered, the latter is also triggered), we consider $coll \wedge \neg kuowr$ as collisions over $\mathcal{K}$. Secondly, since a MAC forgery requires $coll = \texttt{F}$ (otherwise the MAC key is exposed, so the forgery becomes trivial) besides $suf = \texttt{T}$, we consider $suf \wedge \neg coll \wedge \neg kuowr$ for MAC forgeries. Thereby, we bound the probability of events, each of which is disjoint to the others.

$\mathcal{B}_{\mathsf{K}}$: *Reduction to* KUOWR. We define the reduction $\mathcal{B}_{\mathsf{K}}$ which replaces kuKEM* operations with the oracles of KUOWR game. More precisely, the difference between $\mathcal{B}_{\mathsf{K}}$ and $\mathbf{G_5}$ are as follows:

- Instead of generating key pairs, $\mathcal{B}_{\mathsf{K}}$ receives the public key from KUOWR and assigns $\perp$ to the secret key. The line 07 is replaced by $sk \leftarrow \perp$; $pk \leftarrow \mathrm{Gen}()$.
- In SndA, encapsulation and key update operations are passed to the KUOWR game. Lines 17,18 are updated as $(pk, c) \leftarrow \mathrm{Enc}(1, rc)$; and line 26 is replaced with $pk \leftarrow \mathrm{Up}_S(1, ku)$.
- In RcvB, decapsulation and key update operations are passed to the KUOWR game. Line 43 is updated as $k \leftarrow \mathrm{Dec}(1, c)$; and line 49 is updated as $\mathrm{Up}_R(1, k.u)$.
- When $\mathcal{A}$ requests ExposeB with tr, $\mathrm{Expose}(1, tr')$ query is made in KUOWR, where tr' is an array of c and $k.u$ values that matches the corresponding key in KUOWR, and its value is extracted from tr and the table of internal hash queries L_{G}. $\mathcal{B}_{\mathsf{K}}$ stores the recovered secret key sk in $S_A[tr]$ (which might previously contain $sk = \perp$).
- The predicate Match is also updated. Specifically, we only expand the condition in line 26 of Figure 12. If $t \in$ KN, $\mathcal{B}_{\mathsf{K}}$ can compute the key CK$[t]$ by iteratively applying dec algorithm by using the state information contained in S_B. If $t \notin$ KN, then $\mathcal{B}_{\mathsf{K}}$ uses the Solve oracle of KUOWR. If Solve does not abort with rewarding a win, then it can be deduced that CK$[t] \neq k$.
- If $\mathcal{A}$ terminates with failure, $\mathcal{B}_{\mathsf{K}}$ aborts with failure.

Bounding collisions over $\mathcal{K}$. The total number of keys stored in TK array is bounded by the number of states that can be generated through send and receive calls, i.e. $q_{\mathrm{SndA}} + q_{\mathrm{SndB}}$ at most. If $\mathcal{A}$ makes q_{H} hash queries, then the probability of guessing the key TK$[t]$ for some $t \notin$ KN is bounded by $\frac{q_{\mathrm{H}} \cdot (q_{\mathrm{SndA}} + q_{\mathrm{RcvB}})}{|\mathcal{K}|}$. If we treat collision finding over $\mathcal{K}$ as a game, then this reduction would abort with failure either if *kuowr* is triggered or if $\mathcal{A}$ terminates with failure. Therefore the success is captured by $coll \wedge \neg kuowr$.

$\mathcal{B}_{\mathsf{M}}$: *Reduction to* SUF. We define the reduction $\mathcal{B}_{\mathsf{M}}$ which replaces MAC key generation, tagging and verification operations with the oracles of the SUF game. More precisely, the difference between $\mathcal{B}_{\mathsf{M}}$ and $\mathbf{G_5}$ are as follows:

- During the initialization, instead of generating a MAC key, the Gen oracle of the SUF game is called, and $\perp$ is assigned to MK$[\epsilon]$.
- In the SndA oracle, the tagging algorithm tag is replaced with a Tag oracle of SUF.
- In the RcvB oracle, the tag verification algorithm $\mathrm{vfy}_{\mathsf{M}}$ is replaced with a Vfy oracle of SUF.
- In the ExposeA, ExposeB oracles, the MAC keys are first revealed via oracle Expose of the SUF game, and then the updated state is returned.
- For any update in KN, we also expose the added keys.
- Inside G(t), we request the generation of a new MAC key from the SUF game, instead of sampling a fresh $k.m$.
- $\mathcal{B}_{\mathsf{M}}$ aborts with failure if $kuowr \vee coll$ is set to `T` at any point, or $\mathcal{A}$ terminates with failure.

Since a special forgery implies $kuowr \vee coll \vee suf$, we conclude that:

$$\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{A}) \leq \Pr[kuowr] + \Pr[coll \wedge \neg kuowr] + \Pr[suf \wedge \neg kuowr \wedge \neg coll]$$

therefore:

$$\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}}) + \frac{q_{\mathrm{H}} \cdot (q_{\mathrm{SndA}} + q_{\mathrm{RcvB}})}{|\mathcal{K}|} + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}})$$

□

C Insufficiency of Previous URKE Schemes

As described before, both previous unidirectional RKE security definitions are slightly weaker than ours, allowing the instantiations to bypass our equivalence result. In the following we describe (non-trivial) attacks according to our security definition against both schemes.

As Poettering and Rösler already described [20], due to only allowing exposures of A in the unidirectional RKE security definition of Bellare et al. [4], no forward secure state update for B is required during invocations of the rcv algorithm. Accordingly, when exposing the state of B (which contains a static secret key) in the scheme by Bellare et al., all established keys between A and B become insecure (as opposed to only keys established with this exposed or future states).

Example: $c \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$; $\mathrm{RcvB}(\epsilon, c)$; $c' \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$; $\mathrm{RcvB}(\epsilon, c')$; $s_B \leftarrow \mathrm{ExposeB}((\epsilon, c)\|(\epsilon, c'))$; $(X, \sigma) \leftarrow c$; $(hk, y, i, \dots) \leftarrow s_B$; $k \leftarrow \mathrm{H}(hk, (i, \sigma, X, X^y))$; Return $k = \mathrm{Challenge}((\epsilon, c))$

While allowing the adversary to expose B's state in their unidirectional RKE security definition, Poettering and Rösler [20] do not consider randomness reveal (nor manipulation of randomness) in their model. Their unidirectional RKE scheme exploits this in the state update of A and B during the invocation of snd and rcv respectively. The new state of A during the snd invocation is derived by generating a new KEM key pair based on the randomness of this invocation together with the secrets in the previous state of A. The secret key of this KEM key pair is immediately discarded and only the public key is stored in A's state. B derives the corresponding secret key from the ciphertext that he receives from A and his previous state secrets. As a consequence, an adversary obtains B's current secret key (and thereby all his future secret keys) as soon as it once knows the current state of A and then manipulates randomness for the subsequent invocation of snd.

Example: $s_A \leftarrow \mathrm{ExposeA}(\epsilon)$; $(rc_0\|rc_1) \leftarrow_{\$} \mathcal{R}$; $c' \leftarrow \mathrm{SndA}(\epsilon, rc_0\|rc_1)$; $c \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$; $(pk, K, k_m, t) \leftarrow s_A$; $(k'_e, c'_e) \leftarrow \mathrm{enc}(pk; rc_0)$; $t \leftarrow (\epsilon, c')$ $(k', K, k_m, sk) \leftarrow \mathrm{H}(K, k_e, t)$; $(c_e, \tau) \leftarrow c$; $k_e \leftarrow \mathrm{dec}(sk, c_e)$; $t \overset{''}{\leftarrow} (\epsilon, c)$; $(k, K, k_m, sk) \leftarrow \mathrm{H}(K, k_e, t)$; Return $k = \mathrm{Challenge}((\epsilon, c')\|(\epsilon, c))$

D Insufficiency of Weakly Updatable PKE

Apart from the kuKEM notion by Poettering and Rösler [20] (similar to kuPKE by Jaeger and Stepanovs [13]) and our enhanced kuKEM* notion, Jost et al. [15] introduced the notion of healable and key-updating public-key encryption (HkuPke). The former three are instantiated from HIBE and the latter can be derived from efficient building-blocks based on Diffie-Hellman assumptions.

Intuitively, the key update mechanism in kuKEM* (and kuKEM) depicts a one-way function that can be applied independently on secret key and public key with respect to some associated data. It is yet unclear, how to implement this mechanism without relying on HIBE primitives.

In contrast, the intuition behind the update mechanism in HkuPke depicts a merging of an old key pair with some update key pair (implemented via multiplying public Diffie-Hellman shares and adding their secret exponents respectively), and a deterministic deriving of new key pairs. Problems with this mechanism are that the merging is not one-way (i.e., it can be inverted) and it either requires interaction from secret key holder to public key holder (for transmitting the update public key), or the public key holder learns the secret update exponent during the update (the former is the case for their first public key ek^{upd} which is transmitted, and the latter is the case for their second public key ek^{eph} which is derived together with its secret key on the sender side).

Why One-Wayness is Needed in URKE It is essential for KINDR security of URKE that the following attack is harmless with respect to the security of key k for all random coins $rc, rc^* \in \mathcal{R}, rc \neq rc^*$:
$s_A^0 \leftarrow \mathrm{ExposeA}(\epsilon)$; $ad \leftarrow \epsilon$; $c^1 \leftarrow \mathrm{SndA}(ad, rc)$; $(s_A^1, k^1, c^1) \leftarrow \mathrm{snd}(s_A^0, ad; rc)$; $(s_A^*, k^*, c^*) \leftarrow \mathrm{snd}(s_A^0, ad; rc^*)$; $\mathrm{RcvB}(ad, c^*)$; $s_B^* \leftarrow \mathrm{ExposeB}((ad, c^*))$; $c^2 \leftarrow \mathrm{SndA}(ad, \epsilon)$; $s_A^2 \leftarrow \mathrm{ExposeA}((ad, c^1)\|(ad, c^2))$; $k \leftarrow \mathrm{Challenge}($ $(ad, c^1)\|(ad, c^2))$

The state updates due to oracle queries $\mathrm{SndA}(ad, rc)$ and $\mathrm{RcvB}(ad, c^*)$ (i.e., in the respective algorithm invocations of snd and rcv) must diverge states s_A^1 and s_B^* such that they cannot be used by an adversary to derive the key encapsulated in c^2. Note that the random coins rc^* can be arbitrarily chosen (e.g., similar to rc) in order to let c^* differ from c^1 only in few bits. Since any difference between c^1 and c^* must result in a divergence of s_B^* from a state that can be used to derive the key encapsulated in c^2, the secrets in B's state must be updated on the entire incoming ciphertext. This requires a 'one-way' update of the secrets in B's state that takes c^* as respected (associated) data. Furthermore, all computations for deriving s_A^1 and s_A^* can be traced and determined by the adversary, since rc and rc^* are chosen by it. Hence, the public key update (in this state derivation) is computed publicly (and cannot rely on any inputs from B), and the secret key update (in the derivation of s_B^*) must respect all incoming ciphertext bits.

Consequently, the public key update is deterministic (or probabilistic on adversarially chosen random coins) and based on public inputs, and a secret key update is based on adversarially chosen (ciphertext as) associated data, which is what the notion of KUOWR secure kuKEM* (and KUOW secure kuKEM) reflects but the syntax of HkuPku does not allow.

We finally have two remarks: (1) Jost et al. [15] explicitly make it transparent that they do not aim to protect against such attacks as their main goal is an efficient protocol but not optimal security (hence we do not label it a weakness of their primitive but only an important and notable difference), but (2) their construction of HkuPke itself is actually not correct – we reported this problem to the authors.

E One-Time MAC from URKE

Here we describe the construction of a one-time message authentication code from a generic URKE scheme. The symmetric key can be thought of as the random coins for the state initialization (and the tag algorithm) of the URKE scheme. A tag is generated by taking the message as associated data for the snd algorithm (which is invoked on randomness that is derived from key k). The MAC tag then contains the resulting ciphertext as well as the output key. For verification, the rcv algorithm is invoked on the message as associated data and the ciphertext from the tag. The resulting key is then compared to the key from the MAC tag. We depict this scheme in Figure 13.

Proc $\mathrm{tag}(k, m)$	**Proc** $\mathrm{vfy}_{\mathsf{M}}(k, m, \tau)$
00 $rc_i \| rc_s \leftarrow k$	05 $rc_i \| rc_s \leftarrow k$
01 $(s_A, s_B) \leftarrow \mathrm{init}(rc_i)$	06 $(s_A, s_B) \leftarrow \mathrm{init}(rc_i)$
02 $(s_A, \kappa, c) \leftarrow \mathrm{snd}(s_A, m; rc_s)$	07 $(\kappa, c) \leftarrow \tau$
03 $\tau \leftarrow (\kappa, c)$	08 $(s_B, \kappa') \leftarrow_\$ \mathrm{rcv}(s_B, m, c)$
04 Return τ	09 Require $\kappa = \kappa'$
	10 Return $\mathtt{T}$

Fig. 13: One-time MAC scheme M from generic URKE scheme $\mathsf{UR} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$.

Corollary 1. *If* URKE *scheme* UR *is* KINDR *secure, then one-time* MAC *scheme* M *from Figure 13 is* SUF *secure with* $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{A}) \leq q_{\mathrm{Gen}} q_{\mathrm{Vfy}} \mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B})$ *where* q_{Gen} *is the number of* Gen *queries and* q_{Vfy} *is the number of* Vfy *queries in the multi-instance* SUF *notion.*

Please note that in the proof of Theorem 2, only one instance and one tag verification are necessary, resulting in a tight proof to URKE KINDR security.

The proof of single-instance SUF security with one Vfy oracle query is immediate: The Tag oracle of the SUF game is simulated by the KINDR's SndA and Challenge oracles to produce the MAC tag $\tau = (\kappa, c)$. When the (successful) adversary eventually provides a valid tag forgery $\tau^* = (\kappa^*, c^*)$ to the Vfy oracle, two cases are considered:

1. If $c \neq c^*$, then the KINDR's RcvB oracle is invoked on c^* and the Challenge oracle is queried on the resulting key. If this challenge key κ' equals κ^*, then $b' = 0$ is returned to the KINDR game. Otherwise $b' = 1$ is returned.
2. If $c = c^* \wedge \kappa \neq \kappa^*$, then $b' = 1$ is returned.

Please note that the reduction looses tightness linearly in the number of Gen and Vfy queries. As queries to Vfy either contain a known tag, an invalid tag, or a tag forgery, the simulation is straight forward: the reduction guesses, which unknown tag is the first correct tag forgery (and uses it to solve the KINDR game) and treats all remaining queries with unknown tags as invalid tags.

Below in Figure 14 we define the one-time multi-instance strong unforgeability security game SUF (which is adopted from [19] Figure 20).

```
Game SUF($\mathcal{A}$)
  $n \leftarrow 0$; $\mathrm{XP} \leftarrow \emptyset$
  Invoke $\mathcal{A}$
  Stop with 0

Oracle Gen
  $n \leftarrow n + 1$
  $k_n \leftarrow_{\$} \mathcal{K}$
  $mt_n \leftarrow \bot$
  Return

Oracle Expose($i$)
  Require $1 \leq i \leq n$
  $\mathrm{XP} \overset{\cup}{\leftarrow} \{i\}$
  Return $k_i$

Oracle Tag($i, m$)
  Require $1 \leq i \leq n$
  Require $m \in \mathcal{M}$
  Require $mt_i = \bot$
  $\tau \leftarrow \mathrm{tag}(k_i, m)$
  $mt_i \leftarrow (m, \tau)$
  Return $\tau$

Oracle Vfy($i, m, \tau$)
  Require $1 \leq i \leq n$
  Require $m \in \mathcal{M} \wedge \tau \in \mathcal{T}$
  $b \leftarrow \mathrm{vfy}_{\mathsf{M}}(k_i, m, \tau)$
  If $i \notin \mathrm{XP} \wedge (m, \tau) \neq mt_i$:
    Reward $b$
  Return $b$
```

Fig. 14: Security game SUF for a MAC scheme M, defining one-time multi-instance strong unforgeability.