Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity[⋆]

F. Betül Durak[1,2] and Serge Vaudenay[1]

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
[2] Robert Bosch LLC — Research and Technology Center
Pittsburgh PA, USA

Abstract. Following up mass surveillance and privacy issues, modern secure communication protocols now seek more security such as forward secrecy and post-compromise security. They cannot rely on an assumption such as synchronization, predictable sender/receiver roles, or online availability. Ratcheting was introduced to address forward secrecy and post-compromise security in real-world messaging protocols. At CSF 2016 and CRYPTO 2017, ratcheting was studied either without zero round-trip time (0-RTT) or without bidirectional communication. At CRYPTO 2018, ratcheting with bidirectional communication was done using heavy key-update primitives. At EUROCRYPT 2019, another protocol was proposed. All those protocols use random oracles. Furthermore, exchanging n messages has complexity $\mathcal{O}(n^2)$ in general.
In this work, we define the bidirectional asynchronous ratcheted key agreement (BARK) with formal security notions. We provide a simple security model and design a secure BARK scheme using no key-update primitives, no random oracle, and with $\mathcal{O}(n)$ complexity. It is based on a public-key cryptosystem, a signature scheme, one-time symmetric encryption, and a collision-resistant hash function family. We further show that BARK (even unidirectional) implies public-key cryptography, meaning that it cannot solely rely on symmetric cryptography.

1 Introduction

In standard communication systems, protocols are designed to provide messaging services with end-to-end encryption. Essentially, secure communication reduces to continuously exchanging keys, because each message requires a new key. In bidirectional two-party secure communication, participants alternate their role as *senders* and *receivers*. The modern instant messaging protocols are substantially *asynchronous*. In other words, for a two-party communication, the messages should be transmitted (or the key exchange should be done) even though the counterpart is not online. Moreover, to be able to send the payload data without requiring online exchanges is a major design goal called *zero round trip time (0-RTT)*. Finally, the moment when a participant wants to send a message is undefined, meaning that participants use *random roles* (sender or receiver) without any synchronization. They could send messages at the same time.

Even though many systems were designed for the privacy of their users, they rapidly faced security vulnerabilities caused by the *compromises* of the participants' states. In this work, compromising a participant means to obtain some information about its internal state. We will call it *exposure*. The desired security notion is that compromised information should not uncover more than possible by trivial attacks. For instance, the compromised state of participants should not allow decryption of messages exchanged in the past. This is called *forward secrecy*. Typically, forward secrecy is obtained by updating states with a one-way function $x \to \mathsf{H}(x) \to \mathsf{H}(\mathsf{H}(x)) \to ...$ and deleting old entries. It is used, for instance, in RFID protocols [14, 15]. A popular technique in mechanics, that allows forward movement but prevents moving backward is the use of a device called *ratchet*. In the context of secure communication, a ratchet-like action is achieved by

[⋆] A short version of this paper appeared at IWSEC 2019 [9].

using randomness in every state update so that a compromised state is not sufficient for the decryption of any future communication either. This is called *future secrecy* or *backward secrecy* or *post-compromise security* or even *self-healing*. One thesis of the present work is that healing after an active attack involving a forgery is not a nice property. We show that it implies insecurity. After one participant is compromised and impersonated, if communication self-heals, it means that some adversary can make a trivial attack which is not detected. We also demonstrate other events leading to breach of security. Hence, we recommend that communication is totally cut after active attacks.

Previous work. The security of key exchange was studied by many authors. The prominent models are the CK and eCK models [4, 13].

Techniques for ratcheting first appeared in real life protocols. It appeared in the Off-the-Record (OTR) communication system by Borisov et al. [3]. The Signal protocol designed by Open Whisper Systems [17] later gained a lot of interest from message communication companies. Today, the WhatsApp messaging application has reached billions of users worldwide [20]. It uses the Signal protocol. A broad survey about various techniques and terminologies was made at S&P 2015 by Unger et al. [18]. At CSF 2016, Cohn-Gordon et al. [6] studied bidirectional ratcheted communication and proposed a protocol. However, their protocol does not offer 0-RTT and requires synchronized roles. At EuroS&P 2017, Cohn-Gordon et al. [5] formally studied Signal.

0-RTT communication with forward secrecy was achieved using puncturable encryption by Günther et al. at EUROCRYPT 2017 [10]. Later on, at EUROCRYPT 2018, Derler et al. made it reasonably practical by using Bloom filters [7].

At CRYPTO 2017, Bellare et al. [2] gave a secure ratcheting key exchange protocol. Their protocol is unidirectional and does not allow receiver exposure.

At CRYPTO 2018, Poettering and Rösler (PR) [16] studied bidirectional asynchronous ratcheted key agreement and presented a protocol which is secure in the random oracle model. Their solution further relies on hierarchical identity-based encryption (HIBE) but offers stronger security than required for practical usage, leaving ample room for improving the protocol. At the same conference, Jaeger and Stepanovs (JS) [11] had similar results but focused on secure communication rather than key agreement. They proposed another protocol relying on HIBE. In both results, HIBE is used to construct encryption/signature schemes with key-update security. This is a rather new notion allowing forward secrecy but is expensive to achieve. In both cases, it was claimed that the depth of HIBE is really small. However, when participants are disconnected and continue sending several messages, the depth increases quite rapidly. Consequently, HIBE needs unbounded depth.

Two papers appeared after the first version of the current paper was released.

At EUROCRYPT 2019, Jost, Maurer, and Mularczyk (JMM) [12] designed another ratcheting protocol which has *"near-optimal"* security and does not use HIBE. Nevertheless, it still has a huge complexity: When messages alternate well (i.e., no participant sends two messages without receiving one in between), processing n messages requires $\mathcal{O}(n)$ operations in total. However, when messages accumulate before alternating (for instance, because the participants are disconnected by the network), the complexity becomes $\mathcal{O}(n^2)$. This is also the case for PR [16] and JS [11].[3] One advantage of the JMM protocol [12] comes with the resilience with random coin leakage as discussed below.

At EUROCRYPT 2019, Alwen, Coretti, and Dodis (ACD) [1] designed two other ratcheting protocols aiming at *immediate decryption*, i.e. the ability to decrypt even though some previous messages have not been received yet. This is closer to real-life protocols but this comes with a potential threat: keys to decrypt un-delivered messages are stored until the messages are delivered. Hence, the adversary could choose to hold messages and decrypt them with future state exposure. This prevents forward secrecy. Furthermore, unless the direction of communication changes (or

[3] For JS, this is only visible in the corrected version of the paper on eprint [11]. Our complexity analysis is based on how those protocols have been implemented (`https://github.com/qantik/ratcheted`). It was presented at the WSM 2019 workshop.

more precisely, if the *epoch* increases), their protocols do not strictly adhere to the definition of ratcheting as no random coins are used to update the state. This weakens post-compromise security as well. In Table 1, we call this weaker security "*id-optimal*" (not to say "insecure" in the model we are interested in) because it is the best we can obtain with immediate decryption. The lighter of the two protocols is not competing in the same category because it mostly uses symmetric cryptography. It is more efficient but with lower security. Namely, corrupting the state of a participant A implies impersonating B to A, and also decrypting the messages that A sends. Other protocols do not have this weakness. The second ACD [1] (in the full version) uses asymmetric cryptography.

Some authors address the corruption of random coins in different ways. Bellare et al. [2] and JMM [12] allow leaking the random coins just *after* use. JS [11] allow leaking it just *before* usage only. ACD [1] allow adversarially *chosen* random coins. In most of the protocols, revealing (or choosing) the random coins imply revealing some part of the new state which allows decrypting incoming messages. It is comparable to state exposure. JMM [12] offers better security as revealing the random coins reveals the new state (and allows to decrypt) only when the previous state was already known.

Table 1: Comparison of Protocols: complexity for exchanging n messages in alternating or accumulating mode, with timing (in seconds) for $n = 900$ of comparable implementations and asymptotic; and types of coin-leakage security ($\Rightarrow$ state exposure means coins leakage implies a state exposure).

	Security	Complexity		Coins leakage resilience	Model
		alternating	accumulating		
Poettering-Rösler [16]	optimal	86.3 , $\mathcal{O}(n)$	5897 , $\mathcal{O}(n^2)$	no	ROM
Jaeger-Stepanovs [11]	optimal	58.1 , $\mathcal{O}(n)$	9087 , $\mathcal{O}(n^2)$	pre-send leakage, $\Rightarrow$ state exposure	ROM
Jost-Maurer-Mularczyk [12]	near-optimal	2.08 , $\mathcal{O}(n)$	11.4 , $\mathcal{O}(n^2)$	post-send leakage	ROM
BARK [this paper]	sub-optimal	1.46 , $\mathcal{O}(n)$	1.09 , $\mathcal{O}(n)$	no	plain
Alwen-Coretti-Dodis [1]	id-optimal	1.18 , $\mathcal{O}(n)$	0.92 , $\mathcal{O}(n)$	chosen coins, $\Rightarrow$ state exposure	plain

Our contributions. We give a definition for a bidirectional asynchronous key agreement (BARK) along with security properties. We start setting the stage with some definitions (such as *matching status*) then identify all cases leading to trivial attacks. We split them into *direct* and *indirect leakages*. Then, we define security with a KIND game (privacy). We also consider the resistance to forgery (impersonation) and the resistance to attacks which would heal after active attacks (RECOVER security). We use these two notions as building blocks to prove KIND-security. We finally construct a secure protocol. Our design choices are detailed below and compared to other papers. More comprehensive and technical comparisons between BARK and Bellare et al. [2], JS [11], and PR [16] protocols are given in Appendix C.

1. **Simplicity**. Contrary to previous work, we define KIND security in a very comprehensive way by bringing all notions under the umbrella of a *cleanness* predicate which identifies and captures all trivial ways of attacking.

2. **Strong security**. In the same line as previous works, the adversary in our model can see the entire communication between participants and control the delivery. Of course, he can replace messages with anything. Scheduling communications is under the control of the adversary. This means that the time when a participant sends or receives messages is decided by the adversary. Moreover, the adversary is capable of corrupting participants by making exposures of their internal data. We separate two types of exposures: the exposure of the state (that is kept in internal machinery of a participant) and the exposure of the key (which is produced by the key agreement and given to an external protocol). This is because states are (normally) kept secure in our protocol

while the generated key is transferred to other applications which may leak for different reasons. We do not consider exposure of the random coins.

3. **Slightly sub-optimal security**. Using the result from exposure allows the adversary to be active, e.g. by impersonating the exposed participant. However, the adversary is not allowed to use exposures to make a *trivial* attack. Identifying such trivial attacks is not easy. As a design goal, we adopt not to forbid more than what the intuitive notion of ratcheting captures. We do forbid a bit more than PR [16] and JS [11] which are considered of having *optimal* security and than JMM [12] (which has *near-optimal* security)[4], though, allowing lighter building blocks. Namely, we need no key-update primitives and have linear-time complexity in terms of the number of exchanged messages, even when the network is occasionally down. **This translates to an important speedup factor**, as shown on Table 1. We argue that this is a reasonable choice enabling ratchet security as we define it: *unless trivial leakage, a message is private as long as it is acknowledged for reception in a subsequent message from the receiver.*

4. **Sequence integrity**. We believe that duplex communication is reliably enforced by a lower level protocol. This is assumed to solve non-malicious packet losses e.g. by resend requests and also to reconstruct the correct sequence order. What we only have to care of is when an adversary prevents the delivery of a message consistently. We make the choice to make the transmission of the next messages impossible under such an attack. Contrarily, ACD [1] advocates for immediate decryption, even though one message is missing. This lowers the security and we chose not to have it.

In the BARK protocol, the correctness implies that both participants generate the same keys. We define the stages *matching status, direct leakage, indirect leakage*. We aim to separate trivial attacks and trivial forgeries from non-trivial cases with our definitions. Direct and indirect leakages define when the adversary can trivially deduce the key generated due to the exposure of a participant who can either be the same participant (direct) or their counterpart (indirect).

We construct a secure BARK protocol. We build our constructions on top of a public-key cryptosystem and a signature scheme and achieve strong security, without key-update primitives or random oracles. We further show that a weakly secure unidirectional BARK implies public-key cryptography.

Notations. We have two characters: Alice (A) and Bob (B). When P designates a participant, $\overline{\mathsf{P}}$ refers to P's counterpart. We use the roles send and rec for sender and receiver respectively. We define $\overline{\mathsf{send}} = \mathsf{rec}$ and $\overline{\mathsf{rec}} = \mathsf{send}$. When participants A and B have exclusive roles (like in unidirectional cases), we call them *sender* S and *receiver* R.

Structure of the paper. In Section 2, we define our BARK protocol along with correctness definition and KIND security. Section 3 proves that a simple unidirectional scheme implies public-key encryption. In Section 4 we define the security notions unforgeability and unrecoverability. In Section 5, we give our BARK construction. Appendix A recalls definitions for underlying primitives. Using plaintext-aware security, Appendix B shows that "optimally secure" protocols may still eliminate attacks which are of no harm, hence eliminate more than necessary. In Appendix C, we make some comments and comparison with the results of Bellare et al. [2], Poettering-Rösler [16], and Jaeger-Stepanovs [11].

2 Bidirectional Asynchronous Ratcheted Communication

2.1 BARK Definition and Correctness

Definition 1 (BARK). *A* bidirectional asynchronous ratcheted key agreement (BARK) *consists of the following polynomially bounded algorithms:*

– $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$*: This defines the common public parameters* pp.

[4] Those terms are more formally explained on p. 11.

– $\mathsf{Gen}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$*: This generates the secret key* sk *and the public key* pk *of a participant.*
– $\mathsf{Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_\mathsf{P}, \mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{P}) \to \mathsf{st}_\mathsf{P}$*: This sets up the initial state* st_P *of* P *given his secret key and the public key of his counterpart.*
– $\mathsf{Send}(\mathsf{st}_\mathsf{P}) \xrightarrow{\$} (\mathsf{st}'_\mathsf{P}, \mathsf{upd}, \mathsf{k})$*: The algorithm inputs a current state* st_P *for* $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$*. It outputs a tuple* $(\mathsf{st}'_\mathsf{P}, \mathsf{upd}, \mathsf{k})$ *with an updated state* st'_P*, a message* upd*, and a key* k*.*
– $\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{upd}) \to (\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{k})$*: The algorithm inputs* $(\mathsf{st}_\mathsf{P}, \mathsf{upd})$ *where* $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$*. It outputs a triple consisting of a flag* $\mathsf{acc} \in \{\mathsf{true}, \mathsf{false}\}$ *to indicate an accept or reject of* upd *information, an updated state* st'_P*, and a key* k *i.e.* $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{k})$*.*

For convenience, we define the following initialization procedure for all games. It returns the initial states as well as some publicly available information z*.*

```
$\mathsf{Initall}(1^\lambda, \mathsf{pp})$:
 1: $\mathsf{Gen}(1^\lambda, \mathsf{pp}) \to (\mathsf{sk}_\mathsf{A}, \mathsf{pk}_\mathsf{A})$
 2: $\mathsf{Gen}(1^\lambda, \mathsf{pp}) \to (\mathsf{sk}_\mathsf{B}, \mathsf{pk}_\mathsf{B})$
 3: $\mathsf{st}_\mathsf{A} \leftarrow \mathsf{Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_\mathsf{A}, \mathsf{pk}_\mathsf{B}, \mathsf{A})$
 4: $\mathsf{st}_\mathsf{B} \leftarrow \mathsf{Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_\mathsf{B}, \mathsf{pk}_\mathsf{A}, \mathsf{B})$
 5: $z \leftarrow (\mathsf{pp}, \mathsf{pk}_\mathsf{A}, \mathsf{pk}_\mathsf{B})$
 6: return $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z)$
```

Initialization is *splittable* in the sense that private keys can be generated by their holders with no need to rely on an authority (except maybe for authentication of pk_A and pk_B). Other protocols from the literature assume a trusted initialization.

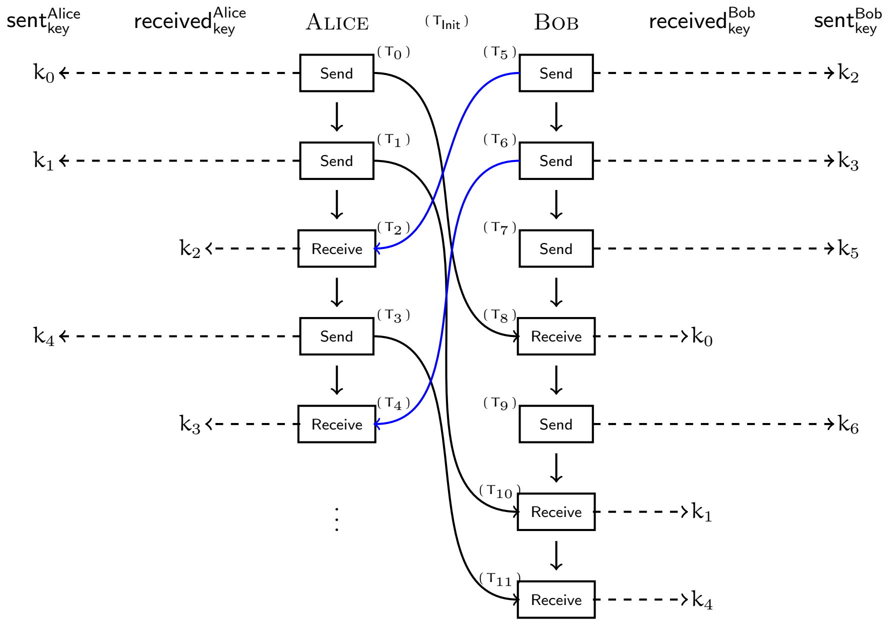
We consider bidirectional asynchronous communications. We can see, in Fig. 1, Alice and Bob running some sequences of Send and $\mathsf{Receive}$ operations without any prior agreement. Their *time scale* is different. This means that Alice and Bob run algorithms in an asynchronous way. We consider a notion of *time* relative to a participant P. Formally, the time t for P is the number of elementary steps that P executed since the beginning of the game. We assume no common clock. However, events occur in a *game* and we may have to compare the time of two different participants by reference to the scheduling of the game. E.g., we could say that time t_A for A happens *before* time t_B for B. Normally, scheduling is under the control of the adversary except in the $\mathsf{CORRECT}$ game in which there is no adversary. There, we define the scheduling by a sequence of actions. Reading the sequence tells who executes a new step of the protocol.

The protocol also uses random roles. Alice and Bob can both send and receive messages. They take their role (sender or receiver) in a sequence, but the sequences of roles of Alice and Bob are not necessarily synchronized. Sending/receiving is refined by the $\mathsf{RATCH}(\mathsf{P}, \mathsf{role}, [\mathsf{upd}])$ call in Fig. 2.

Correctness. We say that a ratcheted communication protocol functions correctly if the receiver accepts the update information upd and generates the same key as its counterpart. Correctness implies that the received keys for participant P have been generated in the same order as sent keys of participant $\overline{\mathsf{P}}$. We formally define the $\mathsf{CORRECT}$ game in Fig. 2. We define variables. $\mathsf{received}^\mathsf{P}_\mathsf{key}$ (respectively $\mathsf{sent}^\mathsf{P}_\mathsf{key}$) keeps a list of secret keys that are generated by P when running $\mathsf{Receive}$ (respectively, Send). Similarly, $\mathsf{received}^\mathsf{P}_\mathsf{msg}$ (respectively $\mathsf{sent}^\mathsf{P}_\mathsf{msg}$) keeps a list of upd information that are received (respectively sent) by P and accepted by $\mathsf{Receive}$. The $\mathsf{received}$ sequences only keep values for which $\mathsf{acc} = \mathsf{true}$.

Each variable ν such as $\mathsf{received}^\mathsf{P}_\mathsf{msg}$, k_P, or st_P is relative to a participant P. We denote by $\nu(\mathsf{t})$ the value of ν at time t for P. For instance, $\mathsf{received}^\mathsf{A}_\mathsf{msg}(\mathsf{t})$ is the sequence of upd which were received by A at time t for A.

We initialize the two participants in the $\mathsf{CORRECT}$ game in Fig. 2. The scheduling is defined by a sequence sched of tuples of form either $(\mathsf{P}, \mathsf{send})$ (saying that P must send) or $(\mathsf{P}, \mathsf{rec})$ (saying that P must receive). In this game, communication between the participants uses a waiting queue for messages in each direction. Each participant has a queue of incoming messages and is pulling them in the order they have been pushed in. Sent messages from P are buffered in the queue of $\overline{\mathsf{P}}$.



$\mathsf{sent}^{\mathsf{Alice}}_{\mathsf{key}}$
$\mathsf{received}^{\mathsf{Alice}}_{\mathsf{key}}$
ALICE
$(\mathsf{T}_{\mathsf{Init}})$
BOB
$\mathsf{received}^{\mathsf{Bob}}_{\mathsf{key}}$
$\mathsf{sent}^{\mathsf{Bob}}_{\mathsf{key}}$
Send
Receive
k_0 k_1 k_2 k_3 k_4 k_5 k_6
(T_0) (T_1) (T_2) (T_3) (T_4) (T_5) (T_6) (T_7) (T_8) (T_9) (T_{10}) (T_{11})

Fig. 1: The message exchange between Alice and Bob.

Definition 2 (Correctness of BARK). *We say that* BARK *is* correct *if for all sequence* sched*, the* CORRECT *game of Fig. 2 never returns 1. Namely, for each* P*,* $\mathsf{received}^{\mathsf{P}}_{\mathsf{key}}$ *is always prefix of* $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{key}}$[5] *and each* $\mathsf{RATCH}(., \mathsf{rec}, .)$ *call accepts.*

Security. We model our security notion with an active adversary who can have access to some of the states of Alice or Bob along with access to their secret keys enabling them to act both as a sender and as a receiver. For simplicity, we have only Alice and Bob as participants. (Models with more participants would be asymptotically equivalent.) We focus on three main security notions which are *key indistinguishability* (denoted as KIND) under the compromise of states or keys, *unforgeability* of upd information (FORGE) by the adversary which will be accepted, and *recovery from impersonation* (RECOVER) which will make the two participants restore secure communication without noticing a (trivial) impersonation resulting from a state exposure. A challenge in these notions is to eliminate the trivial attacks. FORGE and RECOVER security will be useful to prove KIND security.

2.2 KIND Security

The adversary can access four oracles called RATCH, $\mathsf{EXP}_{\mathsf{st}}$, $\mathsf{EXP}_{\mathsf{key}}$, and TEST.

RATCH. This is essentially the message exchange procedure. It is defined in Fig. 2. The adversary can call it with three inputs, a participant P, where $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$; a role of P; and an upd information if the role is rec. The adversary gets upd (for $\mathsf{role} = \mathsf{send}$) or acc (for $\mathsf{role} = \mathsf{rec}$) in return.

$\mathsf{EXP}_{\mathsf{st}}$. The adversary can expose the state of Alice or Bob. It inputs $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ to the $\mathsf{EXP}_{\mathsf{st}}$ oracle and it receives the full state st_{P} of P.

$\mathsf{EXP}_{\mathsf{key}}$. The adversary can expose the generated key by calling this oracle. Upon inputting P, it gets the last key k_{P} generated by P. If no key was generated, $\perp$ is returned.

[5] By saying that $\mathsf{received}^{\mathsf{P}}_{\mathsf{key}}$ is prefix of $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{key}}$, we mean that when n is the number of keys generated by P running Receive, then these keys are the first n keys generated by $\overline{\mathsf{P}}$ running Send.

```
Oracle $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$
 1: $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{k}) \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{upd})$
 2: if $\mathsf{acc}$ then
 3:     $\mathsf{upd}_\mathsf{P} \leftarrow \mathsf{upd}$
 4:     $\mathsf{k}_\mathsf{P} \leftarrow \mathsf{k}$
 5:     $\mathsf{st}_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
 6:     append $\mathsf{k}_\mathsf{P}$ to $\mathsf{received}^\mathsf{P}_\mathsf{key}$
 7:     append $\mathsf{upd}_\mathsf{P}$ to $\mathsf{received}^\mathsf{P}_\mathsf{msg}$
 8: end if
 9: return $\mathsf{acc}$

Oracle $\mathsf{RATCH}(\mathsf{P}, \mathsf{send})$
10: $(\mathsf{st}'_\mathsf{P}, \mathsf{upd}_\mathsf{P}, \mathsf{k}_\mathsf{P}) \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{P})$
11: $\mathsf{st}_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
12: append $\mathsf{k}_\mathsf{P}$ to $\mathsf{sent}^\mathsf{P}_\mathsf{key}$
13: append $\mathsf{upd}_\mathsf{P}$ to $\mathsf{sent}^\mathsf{P}_\mathsf{msg}$
14: return $\mathsf{upd}_\mathsf{P}$
```

```
Game $\mathsf{CORRECT}(1^\lambda, \mathsf{sched})$
 1: set all $\mathsf{sent}^*_*$ and $\mathsf{received}^*_*$ variables to $\emptyset$
 2: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
 3: $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
 4: initialize two FIFO lists $\mathsf{incoming}_A$ and $\mathsf{incoming}_B$ to empty
 5: $i \leftarrow 1$
 6: while $\mathsf{sched}_i$ exists do
 7:     $(\mathsf{P}, \mathsf{role}) \leftarrow \mathsf{sched}_i$
 8:     if $\mathsf{role} = \mathsf{rec}$ then
 9:         if $\mathsf{incoming}_\mathsf{P}$ is empty then return 0
10:         pull $\mathsf{upd}$ from $\mathsf{incoming}_\mathsf{P}$
11:         $\mathsf{acc} \leftarrow \mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$
12:         if $\mathsf{acc} = \mathsf{false}$ then return 1
13:     else
14:         $\mathsf{upd} \leftarrow \mathsf{RATCH}(\mathsf{P}, \mathsf{send})$
15:         push $\mathsf{upd}$ to $\mathsf{incoming}_{\overline{\mathsf{P}}}$
16:     end if
17:     if $\mathsf{received}^A_\mathsf{key}$ not prefix of $\mathsf{sent}^B_\mathsf{key}$ then return 1
18:     if $\mathsf{received}^B_\mathsf{key}$ not prefix of $\mathsf{sent}^A_\mathsf{key}$ then return 1
19:     $i \leftarrow i + 1$
20: end while
21: return 0
```

Fig. 2: The $\mathsf{CORRECT}$ game.

TEST. This oracle can be called only once to receive a challenge key which is generated either uniformly at random (if the challenge bit is $b = 0$) or given as the last generated key of a participant P specified as input (if the challenge bit is $b = 1$). The oracle cannot be queried if no key was generated yet.

We specifically separate $\mathsf{EXP}_\mathsf{key}$ from EXP_st because the key k generated by BARK will be used by an external process which may leak the key. Thus, $\mathsf{EXP}_\mathsf{key}$ can be more frequent than EXP_st, however it harms security less.

To define security, we avoid trivial attacks. Capturing the trivial cases in a broad sense requires a new set of definitions. All of them are intuitive.

Intuitively, P is in a matching status at a given time if his state is not dependent on an "active" attack (i.e. could result from a $\mathsf{CORRECT}$ game).

Definition 3 (Matching status). *We say that* P *is in a* matching status *at time* t *for* P *if*

1. *at any moment of the game before time* t *for* P*,* $\mathsf{received}^\mathsf{P}_\mathsf{msg}$ *is a prefix of* $\mathsf{sent}^{\overline{\mathsf{P}}}_\mathsf{msg}$ *— this defines the time* $\overline{t}$ *for* $\overline{\mathsf{P}}$ *when* $\overline{\mathsf{P}}$ *sent the last message in* $\mathsf{received}^\mathsf{P}_\mathsf{msg}(t)$*;*
2. *at any moment of the game before time* $\overline{t}$ *for* $\overline{\mathsf{P}}$*,* $\mathsf{received}^{\overline{\mathsf{P}}}_\mathsf{msg}$ *is a prefix of* $\mathsf{sent}^\mathsf{P}_\mathsf{msg}$.

We further say that time t *for* P originates *from time* $\overline{t}$ *for* $\overline{\mathsf{P}}$.

The first condition clearly states that each of the received (and accepted) upd message was sent before by the counterpart of P, in the same order, without any loss in between. The second condition similarly verifies that those messages from $\overline{\mathsf{P}}$ only depend on information coming from P. In Fig. 1, Bob is in a matching status with Alice because he receives the upd information in the exact order as they have sent by Alice (i.e. Bob generates k_2 after k_1 and k_4 after k_2 same as it has sent by Alice). In general, as long as no adversary switches the order of messages or creates fake messages successfully for either party, the participants are always in a matching status.

The key exchange literature often defines a notion of partnering which is simpler. Asynchronous random roles makes it more complicated.

Here is an easy property of the notion of matching status.

Lemma 4. *If* P *is in a matching status at time* t*, then* P *is also in a matching status at any time* $\mathsf{t}_0 \leqslant \mathsf{t}$*. Similarly, if* P *is in a matching status at time* t *and* t *for* P *originates from* $\bar{\mathsf{t}}$ *for* $\overline{\mathsf{P}}$*, then* $\overline{\mathsf{P}}$ *is in a matching status at time* $\bar{\mathsf{t}}$*.*

Proof. In Def. 3, it is clear that if the first property holds with time t for P, then it holds for any time $\mathsf{t}_0 < \mathsf{t}$ for P. Similarly, if the second property holds with time $\bar{\mathsf{t}}$ for $\overline{\mathsf{P}}$, then it holds for any time $\bar{\mathsf{t}}_0 < \bar{\mathsf{t}}$ for $\overline{\mathsf{P}}$.

Hence, if P is in a matching status at time t, then he is in a matching status at any time $\mathsf{t}_0 < \mathsf{t}$. Furthermore, if P is in a matching status at time t and time t for P originates from time $\bar{\mathsf{t}}$ for $\bar{\mathsf{t}}$, we can exchange the roles of P and $\overline{\mathsf{P}}$, obtain the first property with time $\bar{\mathsf{t}}$ instead of t, and deduce the second property for some time which is even before. Hence, $\overline{\mathsf{P}}$ is in a matching status at time $\bar{\mathsf{t}}$. □

Definition 5 (Forgery). *Given a participant* P *in a game, we say that* $\mathsf{upd} \in \mathsf{received}^{\mathsf{P}}_{\mathsf{msg}}$ *is a forgery if at the moment of the game just before* P *received* upd*,* P *was in a matching status, but no longer after receiving* upd*.*

In a matching status, any upd received by P must correspond to an upd sent by $\overline{\mathsf{P}}$ and the sequences must match. This implies the following notion.

Definition 6 (Corresponding RATCH calls). *Let* P *be a participant. We consider only the* $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, .)$ *calls by* P *returning* true*. We say that the* i^{th} *receiving call* corresponds *to the* j^{th} *sending* $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send})$ *call by* $\overline{\mathsf{P}}$ *if* $\mathsf{i} = \mathsf{j}$ *and* P *is in matching status at the time of this* i^{th} *accepting* $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, .)$ *call.*

Lemma 7. *In a correct* BARK *protocol, two corresponding* $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$ *and* $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send})$ *calls generate the same key* $\mathsf{k}_{\mathsf{P}} = \mathsf{k}_{\overline{\mathsf{P}}}$*.*

Proof. We let t be the time of the $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$ call and $\bar{\mathsf{t}}$ be the time of the $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send})$. If $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$ and $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send})$ correspond to each other, then P is in matching status and t originates from $\bar{\mathsf{t}}$. We follow the sequence of RATCH calls made by the game until time $\bar{\mathsf{t}}$ for $\overline{\mathsf{P}}$ and time t for P (i.e., we ignore what happens to $\overline{\mathsf{P}}$ after time $\bar{\mathsf{t}}$ and to P after time t). Due to the properties of the matching status, we can model them as a sequence sched as in the $\mathsf{CORRECT}$ game, where the upd messages are buffered. Using the same random coins, this game produces the same keys. Due to correctness, we have $\mathsf{received}^{\mathsf{P}}_{\mathsf{key}} = \mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{key}}$. The i^{th} element of $\mathsf{received}^{\mathsf{P}}_{\mathsf{key}}$ is $\mathsf{k}_{\mathsf{P}}(\mathsf{t})$. The j^{th} element of $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{key}}$ is $\mathsf{k}_{\overline{\mathsf{P}}}(\bar{\mathsf{t}})$. Since $\mathsf{i} = \mathsf{j}$, we have $\mathsf{k}_{\mathsf{P}}(\mathsf{t}) = \mathsf{k}_{\overline{\mathsf{P}}}(\bar{\mathsf{t}})$. □

Definition 8 (Ratcheting period of P). *A maximal time interval during which there is no* $\mathsf{RATCH}(\mathsf{P}, \mathsf{send})$ *call is called a* ratcheting period *of* P*.*

Consequently, a $\mathsf{RATCH}(\mathsf{P}, \mathsf{send})$ call ends a ratcheting period for P and starts a new one. In Fig. 1, the time between T_1 and T_3 or the interval $\mathsf{T}_5 - \mathsf{T}_6$ are called ratcheting period of Alice and Bob respectively.

We now define when the adversary can trivially obtain a key generated by P due to an exposure. We distinguish the case when the exposure was done on P (direct leakage) and on $\overline{\mathsf{P}}$ (indirect leakage).

Definition 9 (Direct leakage). *Let* t *be a time and* P *be a participant. We say that* $\mathsf{k}_{\mathsf{P}}(\mathsf{t})$ *has a* direct leakage *if one of the following conditions is satisfied:*

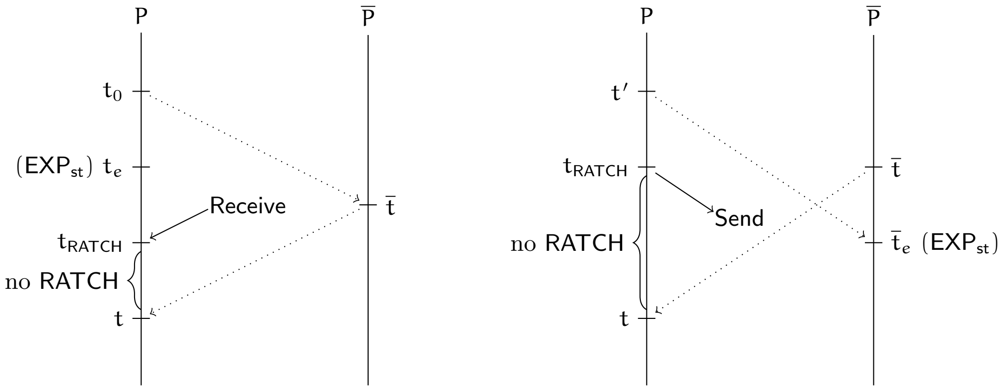
– *There is an* $\mathsf{EXP}_{\mathsf{key}}(\mathsf{P})$ *at a time* t_{e} *such that the last* RATCH *call which is executed by* P *before time* t *and the last* RATCH *call which is executed by* P *before time* t_{e} *are the same.*

– P *is in a matching status and there exists* $\mathsf{t}_0 \leqslant \mathsf{t}_e \leqslant \mathsf{t}_{\mathsf{RATCH}} \leqslant \mathsf{t}$ *and* $\overline{\mathsf{t}}$ *such that time* t *originates from time* $\overline{\mathsf{t}}$*; time* $\overline{\mathsf{t}}$ *originates from time* t_0*; there is one* $\mathsf{EXP}_{\mathsf{st}}(\mathsf{P})$ *at time* t_e*; there is one* $\mathsf{RATCH}(\mathsf{P},\mathsf{rec},.)$ *at time* $\mathsf{t}_{\mathsf{RATCH}}$*; and there is no* $\mathsf{RATCH}(\mathsf{P},.,.)$ *between time* $\mathsf{t}_{\mathsf{RATCH}}$ *and time* t*.*

In the first case, it is clear that $\mathsf{EXP}_{\mathsf{key}}(\mathsf{P})$ gives $\mathsf{k}_\mathsf{P}(\mathsf{t}_e) = \mathsf{k}_\mathsf{P}(\mathsf{t})$. In the second case (in Fig. 3)[6], the state which leaks from $\mathsf{EXP}_{\mathsf{st}}(\mathsf{P})$ at time t_e allows to simulate all deterministic Receive (by skipping all Send) and to compute the key $\mathsf{k}_\mathsf{P}(\mathsf{t}_{\mathsf{RATCH}}) = \mathsf{k}_\mathsf{P}(\mathsf{t})$. The reason why we can allow the adversary to skip all Send is that they make messages which are supposed to be delivered to $\overline{\mathsf{P}}$ after time $\overline{\mathsf{t}}$, so they have no impact on $\mathsf{k}_\mathsf{P}(\mathsf{t})$.

Consider Fig. 1. Suppose t is in between time T_3 and T_4. According to our definition $\mathsf{P} = \mathsf{A}$ and the last RATCH call is at time T_3. It is a Send, thus the second case cannot apply. The next RATCH call is at time T_4. In this case, $\mathsf{k}_\mathsf{A}(\mathsf{t})$ has a direct leakage if there is a key exposure of Alice between T_3 and T_4.

Suppose now that $\mathsf{T}_8 < \mathsf{t} < \mathsf{T}_9$. We have $\mathsf{P} = \mathsf{B}$, the last RATCH call is a Receive, it is at time $\mathsf{t}_{\mathsf{RATCH}} = \mathsf{T}_8$, and t originates from time $\overline{\mathsf{t}} = \mathsf{T}_0$ which itself originates from the origin time $\mathsf{t}_0 = \mathsf{T}_{\mathsf{Init}}$ for B. We say that t has a direct leakage if there is a key exposure between $\mathsf{T}_8 - \mathsf{T}_9$ or a state exposure of Bob before time T_8. Indeed, with this last state exposure, the adversary can ignore all Send and simulate all Receive to derive k_0.



P
$\overline{\mathsf{P}}$
t_0
$(\mathsf{EXP}_{\mathsf{st}})\ \mathsf{t}_e$
Receive
$\overline{\mathsf{t}}$
$\mathsf{t}_{\mathsf{RATCH}}$
no RATCH
t
P
$\overline{\mathsf{P}}$
t'
$\mathsf{t}_{\mathsf{RATCH}}$
$\overline{\mathsf{t}}$
Send
no RATCH
$\overline{\mathsf{t}}_e\ (\mathsf{EXP}_{\mathsf{st}})$
t

Fig. 3: Direct (left) and indirect (right) leakage

Definition 10 (Indirect leakage). *We consider a time* t *and a participant* P*. Let* $\mathsf{t}_{\mathsf{RATCH}}$ *be the time of the last successful* RATCH *call and* role *be its input role. (We have* $\mathsf{k}_\mathsf{P}(\mathsf{t}_{\mathsf{RATCH}}) = \mathsf{k}_\mathsf{P}(\mathsf{t})$*.) We say that* $\mathsf{k}_\mathsf{P}(\mathsf{t})$ *has an* indirect leakage *if* P *is in matching status at time* t *and one of the following conditions is satisfied*

– *There exists a* $\mathsf{RATCH}(\overline{\mathsf{P}},\overline{\mathsf{role}},.)$ *corresponding to that* $\mathsf{RATCH}(\mathsf{P},\mathsf{role},.)$ *and making a* $\mathsf{k}_{\overline{\mathsf{P}}}$ *which has a direct leakage for* $\overline{\mathsf{P}}$*.*
– *There exists* $\mathsf{t}' \leqslant \mathsf{t}_{\mathsf{RATCH}} \leqslant \mathsf{t}$ *and* $\overline{\mathsf{t}} \leqslant \overline{\mathsf{t}}_e$ *such that* $\overline{\mathsf{P}}$ *is in a matching status at time* $\overline{\mathsf{t}}_e$*,* t *originates from* $\overline{\mathsf{t}}$*,* $\overline{\mathsf{t}}_e$ *originates from* t'*, there is one* $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ *at time* $\overline{\mathsf{t}}_e$*, and* $\mathsf{role} = \mathsf{send}$*.*

In the first case, $\mathsf{k}_\mathsf{P}(\mathsf{t}) = \mathsf{k}_\mathsf{P}(\mathsf{t}_{\mathsf{RATCH}})$ is also computed by $\overline{\mathsf{P}}$ and leaks from there. The second case (in Fig. 3) is more complicated: it corresponds to an adversary who can get the internal state of $\overline{\mathsf{P}}$ by $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ then simulate all Receive with messages from P until the one sent at time $\mathsf{t}_{\mathsf{RATCH}}$, ignoring all Send by $\overline{\mathsf{P}}$, to recover $\mathsf{k}_\mathsf{P}(\mathsf{t})$.

[6] Origin of dotted arrows indicate when a time originates from.

For example, let t be a time between T_1 and T_2 in Fig. 1. We take $\mathsf{P} = \mathsf{A}$. The last RATCH call is at time $\mathsf{t}_{\mathsf{RATCH}} = \mathsf{T}_1$, it is a Send and corresponds to a $\mathsf{Receive}$ at time T_{10}, but t originates from time $\overline{\mathsf{t}} = \mathsf{T}_{\mathsf{Init}}$. We say that t has an indirect leakage for A if there exists a direct leakage for $\overline{\mathsf{P}} = \mathsf{B}$ at a time between T_{10} and T_{11} (first condition) or there exists a $\mathsf{EXP}_{\mathsf{st}}(\mathsf{B})$ call at a time $\overline{\mathsf{t}}_e$ (after time $\overline{\mathsf{t}} = \mathsf{T}_{\mathsf{Init}}$), originating from a time t' before time T_1, so $\overline{\mathsf{t}}_e < \mathsf{T}_{10}$ (second condition). In the latter case, the adversary can simulate $\mathsf{Receive}$ with the updates sent at time T_0 and T_1 to derive the key k_1.

Exposing the state of a participant gives certain advantages to the attacker and make trivial attacks possible. In our security game, we avoid those attack scenarios. In the following lemma, we show that direct and indirect leakage capture the times when the adversary can trivially win. The proof is straightforward.

Lemma 11 (Trivial attacks). *Assume that* BARK *is correct. For any* t *and* P*, if* $\mathsf{k}_{\mathsf{P}}(\mathsf{t})$ *has a direct or indirect leakage, the adversary can compute* $\mathsf{k}_{\mathsf{P}}(\mathsf{t})$*.*

Proof. We use correctness, Lemma 7, and the explanations given after Def. 9 and Def. 10. □

So far, we mostly focused on matching status cases but there could be situations with forgeries. Some are unavoidable. We call them *trivial* forgeries.

Definition 12 (Trivial forgery). *Let* upd *be a forgery received by* P*. At the time* t *just before the* $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$ *call,* P *was in a matching status. We assume that time* t *for* P *originates from time* $\overline{\mathsf{t}}$ *for* $\overline{\mathsf{P}}$*. If there is an* $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ *call during the ratcheting period of* $\overline{\mathsf{P}}$ *starting at time* $\overline{\mathsf{t}}$*, we say that* upd *is a* trivial forgery*.*

We define the KIND security game in Fig. 4. Essentially, the adversary plays with all oracles. At some point, he does one $\mathsf{TEST}(\mathsf{P})$ call which returns either the same result as $\mathsf{EXP}_{\mathsf{key}}(\mathsf{P})$ (case $b = 1$) or some random value (case $b = 0$). The goal of the adversary is to guess b. The TEST call can be done only once and it defines the participant $\mathsf{P}_{\mathsf{test}} = \mathsf{P}$ and the time $\mathsf{t}_{\mathsf{test}}$ at which this call is made. It also defines $\mathsf{upd}_{\mathsf{test}}$, the last upd which was used (either sent or received) to carry $\mathsf{k}_{\mathsf{P}_{\mathsf{test}}}(\mathsf{t}_{\mathsf{test}})$ from the sender to the receiver. It is not allowed to make this call at the beginning, when P did not generate a key yet. It is not allowed to make a trivial attack as defined by a cleanness predicate $\mathsf{C}_{\mathsf{clean}}$ appearing on Step 6 in the KIND game in Fig. 4. Identifying the appropriate *cleanness predicate* $\mathsf{C}_{\mathsf{clean}}$ is not easy. It must clearly forbid trivial attacks but also allow efficient protocols. In what follows we use the following predicates:

- $\mathsf{C}_{\mathsf{leak}}$: $\mathsf{k}_{\mathsf{P}_{\mathsf{test}}}(\mathsf{t}_{\mathsf{test}})$ has no direct or indirect leakage.
- $\mathsf{C}^{\mathsf{P}}_{\mathsf{trivial\ forge}}$: P received no trivial forgery until P has seen $\mathsf{upd}_{\mathsf{test}}$.
  (This implies that $\mathsf{upd}_{\mathsf{test}}$ is not a trivial forgery. It also implies that if P never sees $\mathsf{upd}_{\mathsf{test}}$, then P received no trivial forgery at all.)
- $\mathsf{C}^{\mathsf{P}}_{\mathsf{forge}}$: P received no forgery until P has seen $\mathsf{upd}_{\mathsf{test}}$.
- $\mathsf{C}_{\mathsf{ratchet}}$: $\mathsf{upd}_{\mathsf{test}}$ was sent by a participant P, then received and accepted by $\overline{\mathsf{P}}$, then some $\mathsf{upd}_{\mathsf{ack}}$ was sent by $\overline{\mathsf{P}}$, then $\mathsf{upd}_{\mathsf{ack}}$ was received and accepted by P.
  (Here, P could be $\mathsf{P}_{\mathsf{test}}$ or his counterpart. This accounts for the receipt of $\mathsf{upd}_{\mathsf{test}}$ being acknowledged by $\overline{\mathsf{P}}$ through $\mathsf{upd}_{\mathsf{ack}}$.)
- $\mathsf{C}_{\mathsf{noEXP}(\mathsf{R})}$: there is no $\mathsf{EXP}_{\mathsf{st}}(\mathsf{R})$ and no $\mathsf{EXP}_{\mathsf{key}}(\mathsf{R})$ query. (R is the receiver.)

Lemma 11 says that the adopted cleanness predicate $\mathsf{C}_{\mathsf{clean}}$ must imply $\mathsf{C}_{\mathsf{leak}}$ in all considered games. Otherwise, no security is possible. It is however not sufficient as it hardly covers trivial attacks with forgeries.

$\mathsf{C}_{\mathsf{ratchet}}$ targets that any acknowledged sent message is secure. Another way to say is that a key generated by one Send starting a round trip must be safe. This is the notion of healing by ratcheting. Intuitively, the security notion from $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}_{\mathsf{ratchet}}$ is fair enough.

Bellare et al. [2] consider unidirectional BARK with $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{P}_{\mathsf{test}}}_{\mathsf{trivial\ forge}} \wedge \mathsf{C}_{\mathsf{noEXP}(\mathsf{R})}$. Other papers like PR [16] and JS [11] implicitly use $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{P}_{\mathsf{test}}}_{\mathsf{trivial\ forge}}$ as cleanness predicate. They show that this is sufficient to build secure protocols but it is probably not the minimal cleanness predicate (it is nevertheless called "optimal"). JMM [12] excludes cases where $\overline{\mathsf{P}}_{\mathsf{test}}$ received a (trivial) forgery then had an $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}}_{\mathsf{test}})$ before receiving $\mathsf{upd}_{\mathsf{test}}$. Actually, they use a cleanness predicate ("near-optimal" security) which is somewhere between $\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{P}_{\mathsf{test}}}_{\mathsf{trivial\ forge}}$ and $\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{A}}_{\mathsf{trivial\ forge}} \wedge \mathsf{C}^{\mathsf{B}}_{\mathsf{trivial\ forge}}$: this cleanness implies the JMM cleanness which itself implies the PR/JS cleanness.

In our construction ("sub-optimal"), we use the predicate $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{A}}_{\mathsf{forge}} \wedge \mathsf{C}^{\mathsf{B}}_{\mathsf{forge}}$. However, in Section 4.1, we define the FORGE security (unforgeability) which implies that $(\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{A}}_{\mathsf{forge}} \wedge \mathsf{C}^{\mathsf{B}}_{\mathsf{forge}})$-KIND security and $(\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{A}}_{\mathsf{trivial\ forge}} \wedge \mathsf{C}^{\mathsf{B}}_{\mathsf{trivial\ forge}})$-KIND security are equivalent. (See Th. 16.) One drawback is that it forbids more than $(\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{P}_{\mathsf{test}}}_{\mathsf{trivial\ forge}})$-KIND security. The advantage is that we can achieve security without key-update primitives. We will prove in Th. 19 that this security is enough to achieve security with the predicate $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}_{\mathsf{ratchet}}$, thanks to RECOVER-security which we define in Section 4.2. Thus, our cleanness notion is fair enough.

```
Game $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C}_{\mathsf{clean}}}(1^\lambda)$
  $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
  $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
  set all $\mathsf{sent}^*_*$ and $\mathsf{received}^*_*$ variables to $\emptyset$
  set $\mathsf{t}_{\mathsf{test}}$, $\mathsf{k}_A$, $\mathsf{k}_B$ to $\perp$
  $b' \leftarrow \mathcal{A}^{\mathsf{RATCH},\mathsf{EXP}_{\mathsf{st}},\mathsf{EXP}_{\mathsf{key}},\mathsf{TEST}}(z)$
  if $\neg\mathsf{C}_{\mathsf{clean}}$ then return $\perp$
  return $b'$

Oracle $\mathsf{EXP}_{\mathsf{st}}(\mathsf{P})$
  return $\mathsf{st}_{\mathsf{P}}$
```

```
Oracle $\mathsf{TEST}(\mathsf{P})$
  if $\mathsf{t}_{\mathsf{test}} \neq \perp$ then return $\perp$
  if $\mathsf{k}_{\mathsf{P}} = \perp$ then return $\perp$
  $\mathsf{t}_{\mathsf{test}} \leftarrow \mathsf{time}$, $\mathsf{P}_{\mathsf{test}} \leftarrow \mathsf{P}$, $\mathsf{upd}_{\mathsf{test}} \leftarrow \mathsf{upd}_{\mathsf{P}}$
  if $b = 1$ then
      return $\mathsf{k}_{\mathsf{P}}$
  else
      return random $\{0,1\}^{|\mathsf{k}_{\mathsf{P}}|}$
  end if

Oracle $\mathsf{EXP}_{\mathsf{key}}(\mathsf{P})$
  return $\mathsf{k}_{\mathsf{P}}$
```

Fig. 4: $\mathsf{C}_{\mathsf{clean}}$-KIND game.
(Oracle RATCH is defined in Fig. 2.)

Definition 13 ($\mathsf{C}_{\mathsf{clean}}$-KIND security). *Let* $\mathsf{C}_{\mathsf{clean}}$ *be a cleanness predicate. We consider the* $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C}_{\mathsf{clean}}}$ *game of Fig. 4. We say that the ratcheted key agreement* BARK *is* $(\lambda, q, T, \varepsilon)$-$\mathsf{C}_{\mathsf{clean}}$-KIND-secure *if for any adversary limited to* q *queries and time complexity* T*, the advantage*

$$\mathsf{Adv}_{\mathcal{A}}(1^\lambda) = \left|\Pr\left[\mathsf{KIND}^{\mathcal{A}}_{0,\mathsf{C}_{\mathsf{clean}}}(1^\lambda) \to 1\right] - \Pr\left[\mathsf{KIND}^{\mathcal{A}}_{1,\mathsf{C}_{\mathsf{clean}}}(1^\lambda) \to 1\right]\right|$$

of $\mathcal{A}$ *in* $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C}_{\mathsf{clean}}}(1^\lambda)$ *security game is bounded by* ε.

3 uniARK Implies KEM

We now prove that a weakly secure uniARK (a unidirectional asynchronous ratcheted key exchange — a straightforward variant of BARK in which messages can only be sent from a participant whom we call S and can only be received by another participant whom we call R) implies public key encryption. Namely, we can construct a key encapsulation mechanism (KEM) out of it. We recall the KEM definition and its IND-CPA security in Appendix A.

We consider a uniARK which is KIND-secure for the following cleanness predicate:

$\mathsf{C}_{\mathsf{weak}}$: the adversary makes only three oracle calls which are, in order, $\mathsf{EXP}_{\mathsf{st}}(\mathsf{S})$, $\mathsf{RATCH}(\mathsf{S}, \mathsf{send})$, and $\mathsf{TEST}(\mathsf{S})$.

(Note that R is never used.) $\mathsf{C}_{\mathsf{weak}}$ implies cleanness for all other considered predicates. Hence, it is more restrictive. Our result implies that it is unlikely to construct even such weakly secure uniARK from symmetric cryptography.

Theorem 14. *Given a* uniARK *protocol, we can construct a* KEM *with the following properties. The correctness of* uniARK *implies the correctness of* KEM*. The* $\mathsf{C_{weak}}$*-*KIND*-security of* uniARK *implies the* $\mathsf{IND\text{-}CPA}$ *security of* KEM*.*

Proof. Assuming a uniARK protocol, we construct a KEM as follows:

$\mathsf{KEM.Gen}(1^\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$**:** run $\mathsf{uniARK.Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$, $\mathsf{uniARK.InitAll}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st_S}, \mathsf{st_R}, z)$ and set $\mathsf{pk} = \mathsf{st_S}$, $\mathsf{sk} = \mathsf{st_R}$.

$\mathsf{KEM.Enc}(\mathsf{pk}) \xrightarrow{\$} (\mathsf{k}, \mathsf{ct})$**:** run $\mathsf{uniARK.Send}(\mathsf{pk}) \xrightarrow{\$} (., \mathsf{upd}, \mathsf{k})$ and set $\mathsf{ct} = \mathsf{upd}$.

$\mathsf{KEM.Dec}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{k}$**:** run $\mathsf{uniARK.Receive}(\mathsf{sk}, \mathsf{upd}) \to (., ., \mathsf{k})$.

The $\mathsf{IND\text{-}CPA}$ security game with adversary $\mathcal{A}$ works as in the left-hand side below. We transform $\mathcal{A}$ into a KIND adversary $\mathcal{B}$ in the right-hand side below.

```
Game IND-CPA:
 1: $\mathsf{KEM.Gen} \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$
 2: $\mathsf{KEM.Enc}(\mathsf{pk}) \xrightarrow{\$} (\mathsf{k}, \mathsf{ct})$
 3: if $b = 0$ then set $k$ to random
 4: $\mathcal{A}(\mathsf{pk}, \mathsf{ct}, \mathsf{k}) \xrightarrow{\$} b'$
 5: return $b'$
```

```
Adversary $\mathcal{B}(z)$:
 1: call $\mathsf{EXP_{st}}(S) \to \mathsf{pk}$
 2: call $\mathsf{RATCH}(S, \mathsf{send}) \to \mathsf{ct}$
 3: call $\mathsf{TEST}(S) \to \mathsf{k}$
 4: run $\mathcal{A}(\mathsf{pk}, \mathsf{ct}, \mathsf{k}) \to b'$
 5: return $b'$
```

We can check that $\mathsf{C_{weak}}$ is satisfied. The KIND game with $\mathcal{B}$ simulates perfectly the $\mathsf{IND\text{-}CPA}$ game with $\mathcal{A}$. So, the KIND-security of uniARK implies the $\mathsf{IND\text{-}CPA}$ security of KEM. □

4 FORGE and RECOVER Security

4.1 Unforgeability

Another security aspect of the key agreement BARK is to have that no upd information is forgeable by any bounded adversary except trivially by state exposure. This security notion is independent from KIND security but is certainly nice to have for explicit authentication in key agreement. Besides, it is easy to achieve. We will also use it as a helper to prove KIND security: to reduce $\mathsf{C^P_{trivial\ forge}}$-cleanness to $\mathsf{C^P_{forge}}$-cleanness.

Let the adversary interact with the oracles $\mathsf{RATCH}, \mathsf{EXP_{st}}, \mathsf{EXP_{key}}$ in any order. For BARK to have unforgeability, we eliminate the trivial forgeries (as defined in Def. 12). The FORGE game is defined in Fig. 5.

Definition 15 (FORGE **security).** *Consider* $\mathsf{FORGE}^{\mathcal{A}}(1^\lambda)$ *game in Fig. 5 associated to the adversary* $\mathcal{A}$*. Let the advantage of* $\mathcal{A}$ *be the probability that the game outputs 1. We say that* BARK *is* $(\lambda, q, T, \varepsilon)$-$\mathsf{FORGE}$-*secure if, for any adversary limited to* q *queries and time complexity* T*, the advantage is bounded by* ε*.*

We can now justify why forgeries in the KIND game must be trivial for a BARK with unforgeability.

Theorem 16. *If a* BARK *is* $(\lambda, q, T, \varepsilon)$-$\mathsf{FORGE}$*-secure, then* $(\lambda, q, T, \varepsilon')$-$(\mathsf{C_{leak}} \wedge \mathsf{C^{P_{test}}_{forge}})$-$\mathsf{KIND}$*-security implies* $(\lambda, q, T, 2q\varepsilon+\varepsilon')$-$(\mathsf{C_{leak}} \wedge \mathsf{C^{P_{test}}_{trivial\ forge}})$-$\mathsf{KIND}$*-security and* $(\lambda, q, T, \varepsilon')$-$(\mathsf{C_{leak}} \wedge \mathsf{C^A_{forge}} \wedge \mathsf{C^B_{forge}})$-$\mathsf{KIND}$*-security implies* $(\lambda, q, T, 2q\varepsilon + \varepsilon')$-$(\mathsf{C_{leak}} \wedge \mathsf{C^A_{trivial\ forge}} \wedge \mathsf{C^B_{trivial\ forge}})$-$\mathsf{KIND}$*-security.*

Proof. Let us assume that we have FORGE-security and $(\mathsf{C_{leak}} \wedge \mathsf{C^{P_{test}}_{forge}})$-$\mathsf{KIND}$-security. To prove $(\mathsf{C_{leak}} \wedge \mathsf{C^{P_{test}}_{trivial\ forge}})$-$\mathsf{KIND}$-security, we consider an adversary $\mathcal{A}$. Let $\mathsf{C_{clean}} = \mathsf{C_{leak}} \wedge \mathsf{C^{P_{test}}_{trivial\ forge}}$ and $\mathsf{C'_{clean}} = \mathsf{C_{leak}} \wedge \mathsf{C^{P_{test}}_{forge}}$. We transform the game $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C_{clean}}}$ into a $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C'_{clean}}}$ game by adding a failure event F which is true if and only if $\mathsf{P_{test}}$ received a non-trivial forgery before he has seen $\mathsf{upd_{test}}$. By using the Difference Lemma, we have

$$|\Pr[\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C_{clean}}} \to 1] - \Pr[\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C'_{clean}}} \to 1]| \leqslant \Pr[\mathsf{F}]$$

```
Game $\mathsf{FORGE}^{\mathcal{A}}(1^\lambda)$
1: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
2: $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
3: $(\mathsf{P}, \mathsf{upd}) \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP}_{\mathsf{st}}, \mathsf{EXP}_{\mathsf{key}}}(z)$
4: $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd}) \rightarrow \mathsf{acc}$
5: if $\mathsf{acc} = \mathsf{false}$ then return 0
6: if $\mathsf{upd}$ is not a forgery for $\mathsf{P}$ then return 0
7: if $\mathsf{upd}$ is a trivial forgery for $\mathsf{P}$ then return 0
8: return 1
```

```
Game $\mathsf{RECOVER}^{\mathcal{A}}_{\mathsf{BARK}}(1^\lambda)$
1: $\mathsf{win} \leftarrow 0$
2: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
3: $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
4: set all $\mathsf{sent}^*_*$ and $\mathsf{received}^*_*$ variables to $\emptyset$
5: $\mathsf{P} \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP}_{\mathsf{st}}, \mathsf{EXP}_{\mathsf{key}}}(z)$
6: if we can parse $\mathsf{received}^{\mathsf{P}}_{\mathsf{msg}} = (\mathsf{seq}_1, \mathsf{upd}, \mathsf{seq}_2)$ and
   $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{msg}} = (\mathsf{seq}_3, \mathsf{upd}, \mathsf{seq}_4)$ with $\mathsf{seq}_1 \neq \mathsf{seq}_3$ (where $\mathsf{upd}$
   is a single message and all $\mathsf{seq}_i$ are finite sequences of
   single messages) then $\mathsf{win} \leftarrow 1$
7: return $\mathsf{win}$
```

```
Game $\mathsf{PREDICT}^{\mathcal{A}}_{\mathsf{BARK}}(1^\lambda)$
1: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
2: $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
3: $\mathsf{P} \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP}_{\mathsf{st}}, \mathsf{EXP}_{\mathsf{key}}}(z)$
4: $\mathsf{RATCH}(\mathsf{P}, \mathsf{send}) \rightarrow \mathsf{upd}$
5: if $\mathsf{upd} \in \mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{msg}}$ then return 1
6: return 0
```

Fig. 5: FORGE, RECOVER, and PREDICT games.
(Oracle RATCH, $\mathsf{EXP}_{\mathsf{st}}$, $\mathsf{EXP}_{\mathsf{key}}$ are defined in Fig. 2 and Fig. 4.)

We show below that $\Pr[\mathsf{F}] \leqslant q\varepsilon$, we deduce

$$\mathsf{Adv}(\mathsf{KIND}^{\mathcal{A}}_{C'_{\mathsf{clean}}}) \leqslant \mathsf{Adv}(\mathsf{KIND}^{\mathcal{A}}_{C'_{\mathsf{clean}}}) + 2q\varepsilon$$

and conclude.

To show $\Pr[\mathsf{F}] \leqslant q\varepsilon$, for $i = 1, \ldots, q$, we transform $\mathcal{A}$ into a FORGE-adversary $\mathcal{A}_i$ as follows: $\mathcal{A}_i$ simulates $\mathcal{A}$ until $\mathcal{A}$ asks for the i^{th} receive RATCH call. We denote by P and upd the parameters of this i^{th} $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$ call by $\mathcal{A}$. The adversary $\mathcal{A}_i$ just stops and outputs $(\mathsf{P}, \mathsf{upd})$. The FORGE game goes on by making this $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd})$ call in line 4 of the FORGE game. and returns 1 if and only if it is accepted as a non-trivial forgery. When F occurs, at least one of the $\mathcal{A}_i$ wins in the FORGE game. However, forgery implies that $\Pr[\mathsf{FORGE}^{\mathcal{A}_i} \rightarrow 1] \leqslant \varepsilon$. Hence, $\Pr[\mathsf{F}] \leqslant q\varepsilon$.

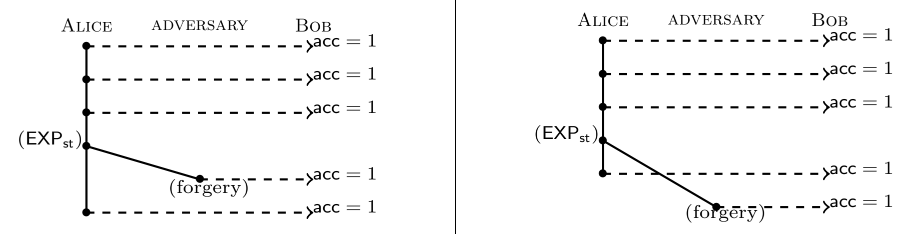
The same proof works for the other cleanness predicates in the statement. $\square$

4.2 Recovery from Impersonation

A priori, it seems nice to be able to restore a secure state when a state exposure of a participant takes place. We show here that it is not a good idea.

Let $\mathcal{A}$ be an adversary playing the two games in Fig. 6. On the left strategy, $\mathcal{A}$ exposes A with an $\mathsf{EXP}_{\mathsf{st}}$ query (Step 2). Then, the adversary $\mathcal{A}$ impersonates A by running the Send algorithm on its own (Step 3). Next, the adversary $\mathcal{A}$ "sends" a message to B which is accepted due to correctness because it is generated with A's state. In Step 5, $\mathcal{A}$ lets the legitimate sender generate upd' by calling RATCH oracle. In this step, *if* security self-restores, then B accepts upd' which is sent by A, hence $\mathsf{acc}' = 1$. It is clear that the strategy shown on the left side in Fig. 6 is equivalent to the strategy shown on the right side of the same figure (which only switches Alice and the adversary who run the same algorithm). Hence, both lead to $\mathsf{acc}' = 1$ with the same probability p. The crucial point is that the forgery in the right-hand strategy becomes non-trivial, which implies that the protocol is not FORGE-secure. In addition to this, if such phenomenon occurs, we can make a KIND adversary passing the $C_{\mathsf{leak}} \wedge C^{\mathsf{P}_{\mathsf{test}}}_{\mathsf{trivial\ forge}}$ condition. Thus, we lose KIND-security. Consequently, security should *not* self-restore.

We define the RECOVER security notion with another game in Fig. 5. Essentially, in the game, we require the receiver P to accept some messages upd sent by the sender after the adversary makes successful forgeries in seq_1. We further use it as a second helper to prove KIND security with C_{ratchet}-cleanness.



```
1: ··· (normal communications) ···
2: $\mathsf{EXP_{st}}(A) \to \mathsf{st}_A$
3: $\mathsf{Send}(\mathsf{st}_A) \to (\mathsf{st}'_A, \mathsf{upd}, \mathsf{k}_A)$
4: $\mathsf{RATCH}(B, \mathsf{rec}, \mathsf{upd}) \to \mathsf{acc}$
5: $\mathsf{RATCH}(A, \mathsf{send}) \to \mathsf{upd}'$
6: $\mathsf{RATCH}(B, \mathsf{rec}, \mathsf{upd}') \to \mathsf{acc}'$
```

```
1: ··· (normal communications) ···
2: $\mathsf{EXP_{st}}(A) \to \mathsf{st}_A$
3: $\mathsf{RATCH}(A, \mathsf{send}) \to \mathsf{upd}$
4: $\mathsf{RATCH}(B, \mathsf{rec}, \mathsf{upd}) \to \mathsf{acc}$
5: $\mathsf{Send}(\mathsf{st}_A) \to (\mathsf{st}'_A, \mathsf{upd}', \mathsf{k}'_A)$
6: $\mathsf{RATCH}(B, \mathsf{rec}, \mathsf{upd}') \to \mathsf{acc}'$
```

Fig. 6: Two recoveries succeeding with the same probability.

Definition 17 **(RECOVER security).** *Consider* $\mathsf{RECOVER}^{\mathcal{A}}_{\mathsf{BARK}}(1^\lambda)$ *game in Fig. 5 associated to the adversary* $\mathcal{A}$*. Let the advantage of* $\mathcal{A}$ *in succeeding playing the game be* $\Pr(\mathsf{win} = 1)$*. We say that the ratcheted communication protocol is* $(\lambda, q, T, \varepsilon)$-RECOVER-secure*, if for any adversary limited to* q *queries and time complexity* T*, the advantage is bounded by* ε*.*

RECOVER-security is easy to achieve using a collision-resistant hash function.

To be sure that no message was received before it was sent, we need the following security notion. In the PREDICT game, the adversary tries to make $\overline{\mathsf{P}}$ receive a message upd before it was sent by P.

Definition 18 **(PREDICT security).** *Consider* $\mathsf{PREDICT}^{\mathcal{A}}_{\mathsf{BARK}}(1^\lambda)$ *game in Fig. 5 associated to the adversary* $\mathcal{A}$*. Let the advantage of* $\mathcal{A}$ *in succeeding playing the game be the probability that* 1 *is returned. We say that the ratcheted communication protocol is* $(\lambda, q, \varepsilon)$-PREDICT-secure*, if for any adversary limited to* q *queries, the advantage is bounded by* ε*.*

Theorem 19. *If a* BARK *is* $(\lambda, q, T, \varepsilon)$-RECOVER*-secure,* (λ, ε')-PREDICT*-secure, and* $(\lambda, q, T, \varepsilon'')$-$(\mathsf{C_{leak}} \wedge \mathsf{C^A_{forge}} \wedge \mathsf{C^B_{forge}})$-KIND *secure, then it is* $(\lambda, q, T, 4\varepsilon + 2q\varepsilon' + \varepsilon'')$-$(\mathsf{C_{leak}} \wedge \mathsf{C_{ratchet}})$-KIND *secure.*

Proof. Let $\mathsf{C_{clean}} = \mathsf{C_{leak}} \wedge \mathsf{C_{ratchet}}$, and let us consider a $\mathsf{C_{clean}}$-KIND game with an adversary $\mathcal{A}$. In this game, we define the failure event F that $\mathsf{C_{ratchet}}$ holds and the following happens. We denote by P the participant who sent $\mathsf{upd_{test}}$. Since $\mathsf{C_{ratchet}}$ holds, we know that $\mathsf{upd_{test}}$ and its acknowledgement $\mathsf{upd_{ack}}$ are genuine messages. We consider the failure event that, at the end of the game, we can parse

- either $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{msg}} = (\mathsf{seq}_1, \mathsf{upd_{test}}, \mathsf{seq}_2)$, $\mathsf{sent}^{\mathsf{P}}_{\mathsf{msg}} = (\mathsf{seq}_3, \mathsf{upd_{test}}, \mathsf{seq}_4)$ with $\mathsf{seq}_1 \neq \mathsf{seq}_3$,
- or $\mathsf{received}^{\mathsf{P}}_{\mathsf{msg}} = (\mathsf{seq}_1, \mathsf{upd_{ack}}, \mathsf{seq}_2)$, $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{msg}} = (\mathsf{seq}_3, \mathsf{upd_{ack}}, \mathsf{seq}_4)$ with $\mathsf{seq}_1 \neq \mathsf{seq}_3$.

We define four RECOVER adversaries $\mathcal{A}_{b,P}$ for $b \in \{0, 1\}$ and $P \in \{A, B\}$ to transform the $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C_{clean}}}$ game into a $\mathsf{RECOVER}^{\mathcal{A}_{b,P}}$ game. More precisely, $\mathcal{A}_{b,P}$ essentially simulates $\mathcal{A}$ except for the TEST oracle, and eventually outputs P. When $\mathcal{A}$ makes a TEST query, $\mathcal{A}_{b,P}$ simulates this oracle (it uses a $\mathsf{EXP_{key}}$ oracle call for that in the $b = 1$ case). Clearly, if the failure event happens in $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C_{clean}}}$, then either $\mathsf{RECOVER}^{\mathcal{A}_{b,A}}$ or $\mathsf{RECOVER}^{\mathcal{A}_{b,B}}$ returns 1. Due to RECOVER security, the failure event in $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C_{clean}}}$ occurs with probability bounded by 2ε.

In a KIND game where $\mathsf{C_{ratchet}}$ holds and the failure event F does not occurs, we have that

$$\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{msg}} = (\mathsf{seq}_1, \mathsf{upd_{test}}, \mathsf{seq}_2)\ ,\ \mathsf{sent}^{\mathsf{P}}_{\mathsf{msg}} = (\mathsf{seq}_1, \mathsf{upd_{test}}, \mathsf{seq}_4)$$
$$\mathsf{received}^{\mathsf{P}}_{\mathsf{msg}} = (\mathsf{seq}'_1, \mathsf{upd_{ack}}, \mathsf{seq}'_2)\ ,\ \mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{msg}} = (\mathsf{seq}'_1, \mathsf{upd_{ack}}, \mathsf{seq}'_4)$$

Namely, all messages received by $\overline{\mathsf{P}}$ until $\mathsf{upd}_{\mathsf{test}}$ were sent in the same order by P, and all messages received by P until $\mathsf{upd}_{\mathsf{ack}}$ were sent in the same order by $\overline{\mathsf{P}}$.

We define another failure event F' that $\mathsf{C}_{\mathsf{ratchet}}$ holds, F does not occur, and either P is not in a matching status when receiving $\mathsf{upd}_{\mathsf{ack}}$, or $\overline{\mathsf{P}}$ is not in a matching status when receiving $\mathsf{upd}_{\mathsf{test}}$. For each $i = 1, \ldots, q$ and each $b = 0, 1$, we define a $\mathsf{PREDICT}$ adversary $\mathcal{A}'_{b,i}$ who simulates $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C}_{\mathsf{clean}}}$ until it makes the i^{th} send RATCH call. If this call is made with a participant Q, $\mathcal{A}'_{b,\mathsf{P}}$ just stops and outputs Q. The simulation of TEST works like for $\mathcal{A}_{b,\mathsf{P}}$. If the failure event F' happens, it must be the case that one of the $\mathcal{A}'_{b,\mathsf{P}}$ adversaries wins the $\mathsf{PREDICT}$ game because one message was received before it was sent. Hence, F' happens with probability bounded by $q\varepsilon'$.

When neither F nor F' happens, the matching status of both participants imply that $\mathsf{C}^{\mathsf{A}}_{\mathsf{forge}} \wedge \mathsf{C}^{\mathsf{B}}_{\mathsf{forge}}$ holds. Hence, by letting $\mathsf{C}'_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{A}}_{\mathsf{forge}} \wedge \mathsf{C}^{\mathsf{B}}_{\mathsf{forge}}$, we have

$$|\Pr[\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C}_{\mathsf{clean}}} \to 1] - \Pr[\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C}'_{\mathsf{clean}}} \to 1]| \leqslant \Pr[\mathsf{F}] + \Pr[\mathsf{F}'] \leqslant 2\varepsilon + q\varepsilon'$$

Hence

$$\mathsf{Adv}(\mathsf{KIND}^{\mathcal{A}}_{\mathsf{C}_{\mathsf{clean}}}) \leqslant \mathsf{Adv}(\mathsf{KIND}^{\mathcal{A}}_{\mathsf{C}'_{\mathsf{clean}}}) + 4\varepsilon + 2q\varepsilon' \leqslant 4\varepsilon + 2q\varepsilon' + \varepsilon''$$

□

5 Our BARK Protocol

We construct a BARK in Fig. 7. We use a public-key cryptosystem PKC, a digital signature scheme DSS, a one-time symmetric encryption Sym, and a collision-resistant hash function H. They are all defined in Appendix A. First, we construct a naive signcryption SC from PKC and DSS by

$$\mathsf{SC.Enc}(\overbrace{\mathsf{sk}_\mathsf{S}, \mathsf{pk}_\mathsf{R}}^{\mathsf{st}_\mathsf{S}}, \mathsf{ad}, \mathsf{pt}) = \mathsf{PKC.Enc}(\mathsf{pk}_\mathsf{R}, (\mathsf{pt}, \mathsf{DSS.Sign}(\mathsf{sk}_\mathsf{S}, (\mathsf{ad}, \mathsf{pt}))))$$
$$\mathsf{SC.Dec}(\underbrace{\mathsf{sk}_\mathsf{R}, \mathsf{pk}_\mathsf{S}}_{\mathsf{st}_\mathsf{R}}, \mathsf{ad}, \mathsf{ct}) = (\mathsf{pt}, \sigma) \leftarrow \mathsf{PKC.Dec}(\mathsf{sk}_\mathsf{R}, \mathsf{ct})\ ;\ \mathsf{DSS.Verify}(\mathsf{pk}_\mathsf{S}, (\mathsf{ad}, \mathsf{pt}), \sigma)\ ?\ \mathsf{pt}\ :\ \perp$$

Second, we extend SC to multi-key encryption called onion due to the multiple layers of keys. Third, we transform onion into a unidirectional ratcheting scheme uni. Finally, we design BARK. (See Fig. 7.)

The state of a participant is a tuple $\mathsf{st} = (\lambda, \mathsf{hk}, \mathsf{List}_\mathsf{S}, \mathsf{List}_\mathsf{R}, \mathsf{Hsent}, \mathsf{Hreceived})$ where hk is the hashing key, Hsent is the iterated hash of all sent messages, and $\mathsf{Hreceived}$ is the iterated hash of all received messages. We also have two lists List_S and List_R. They are lists of states to be used for unidirectional communication: sending and receiving. Both lists are growing but entries are eventually erased. Thus, they can be compressed. (Typically, only the last entry is not erased.)

The idea is that the i^{th} entry of List_S for a participant P is associated to the i^{th} entry of List_R for its counterpart $\overline{\mathsf{P}}$. Every time a participant P sends a message, it creates a new pair of states for sending and receiving and sends the sending state to his counterpart $\overline{\mathsf{P}}$, to be used in the case $\overline{\mathsf{P}}$ wants to respond. If the same participant P keeps sending without receiving anything, he accumulates some receiving states this way. Whenever a participant $\overline{\mathsf{P}}$ who received many messages starts sending, he also accumulated many sending states. His message is sent using *all* those states in the $\mathsf{uni.Send}$ procedure. After sending, all but the last send state are erased, and the message shall indicate the erasures to the counterpart P, who shall erase corresponding receiving states accordingly. Our onion encryption needs to ensure $\mathcal{O}(n)$ complexity (we cannot compose SC encryptions as ciphertext overheads would produce a $\mathcal{O}(n^2)$ complexity). For that, we use a one-time symmetric encryption Sym using a key k in $\{0,1\}^{\mathsf{Sym.kl}}$ which is split into shares $\mathsf{k}_1, \ldots, \mathsf{k}_n$. Each share is SC-encrypted in one state. Only the last state is updated (as others are meant to be erased).

The protocol is quite efficient when participants alternate their roles well, because the lists are often flushed to contain only one unerased state. It also becomes more secure due to ratcheting:

```
$\mathsf{onion.Enc}(1^\lambda, \mathsf{hk}, \mathsf{st}_S^1, \ldots, \mathsf{st}_S^n, \mathsf{ad}, \mathsf{pt})$
 1: pick $k_1, \ldots, k_n$ in $\{0,1\}^{\mathsf{Sym.kl}(\lambda)}$
 2: $k \leftarrow k_1 \oplus \cdots \oplus k_n$
 3: $\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt})$
 4: $\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
 5: for $i = n$ down to 1 do
 6:     $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
 7:     $\mathsf{ct}_i \leftarrow \mathsf{SC.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k_i)$
 8: end for
 9: return $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$
```

```
$\mathsf{onion.Dec}(\mathsf{hk}, \mathsf{st}_R^1, \ldots, \mathsf{st}_R^n, \mathsf{ad}, \vec{\mathsf{ct}})$
 1: if $|\vec{\mathsf{ct}}| \neq n+1$ then return $\perp$
 2: parse $\vec{\mathsf{ct}} = (\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$
 3: $\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
 4: for $i = n$ down to 1 do
 5:     $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
 6:     $\mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \rightarrow k_i$
 7:     if $k_i = \perp$ then return $\perp$
 8: end for
 9: $k \leftarrow k_1 \oplus \cdots \oplus k_n$
10: $\mathsf{pt} \leftarrow \mathsf{Sym.Dec}(k, \mathsf{ct}_{n+1})$
11: return $\mathsf{pt}$
```

```
$\mathsf{uni.Init}(1^\lambda)$
 1: $\mathsf{SC.Gen}_S(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_S, \mathsf{pk}_S)$
 2: $\mathsf{SC.Gen}_R(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_R, \mathsf{pk}_R)$
 3: $\mathsf{st}_S \leftarrow (\mathsf{sk}_S, \mathsf{pk}_R)$
 4: $\mathsf{st}_R \leftarrow (\mathsf{sk}_R, \mathsf{pk}_S)$
 5: return $(\mathsf{st}_S, \mathsf{st}_R)$
```

```
$\mathsf{uni.Send}(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_S, \mathsf{ad}, \mathsf{pt})$
 1: $\mathsf{SC.Gen}_S(1^\lambda) \xrightarrow{\$} (\mathsf{sk}'_S, \mathsf{pk}'_S)$
 2: $\mathsf{SC.Gen}_R(1^\lambda) \xrightarrow{\$} (\mathsf{sk}'_R, \mathsf{pk}'_R)$
 3: $\mathsf{st}'_S \leftarrow (\mathsf{sk}'_S, \mathsf{pk}'_R)$
 4: $\mathsf{st}'_R \leftarrow (\mathsf{sk}'_R, \mathsf{pk}'_S)$
 5: $\mathsf{pt}' \leftarrow (\mathsf{st}'_R, \mathsf{pt})$
 6: $\mathsf{onion.Enc}(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_S, \mathsf{ad}, \mathsf{pt}') \rightarrow \vec{\mathsf{ct}}$
 7: return $(\mathsf{st}'_S, \vec{\mathsf{ct}})$
```

```
$\mathsf{uni.Receive}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}})$
 1: $\mathsf{onion.Dec}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}}) \rightarrow \mathsf{pt}'$
 2: if $\mathsf{pt}' = \perp$ then return $(\mathsf{false}, \perp, \perp)$
 3: parse $\mathsf{pt}' = (\mathsf{st}'_R, \mathsf{pt})$
 4: return $(\mathsf{true}, \mathsf{st}'_R, \mathsf{pt})$
```

```
$\mathsf{BARK.Setup}(1^\lambda)$
 1: $\mathsf{H.Gen}(1^\lambda) \xrightarrow{\$} \mathsf{hk}$
 2: return $\mathsf{hk}$
```

```
$\mathsf{BARK.Gen}(1^\lambda, \mathsf{hk})$
 1: $\mathsf{SC.Gen}_S(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_S, \mathsf{pk}_S)$
 2: $\mathsf{SC.Gen}_R(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_R, \mathsf{pk}_R)$
 3: $\mathsf{sk} \leftarrow (\mathsf{sk}_S, \mathsf{sk}_R)$
 4: $\mathsf{pk} \leftarrow (\mathsf{pk}_S, \mathsf{pk}_R)$
 5: return $(\mathsf{sk}, \mathsf{pk})$
```

```
$\mathsf{BARK.Init}(1^\lambda, \mathsf{hk}, \mathsf{sk}_P, \mathsf{pk}_{\overline{P}}, P)$
 1: parse $\mathsf{sk}_P = (\mathsf{sk}_S, \mathsf{sk}_R)$
 2: parse $\mathsf{pk}_{\overline{P}} = (\mathsf{pk}_S, \mathsf{pk}_R)$
 3: $\mathsf{st}_P^{\mathsf{send}} \leftarrow (\mathsf{sk}_S, \mathsf{pk}_R)$
 4: $\mathsf{st}_P^{\mathsf{rec}} \leftarrow (\mathsf{sk}_R, \mathsf{pk}_S)$
 5: $\mathsf{st}_P \leftarrow (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send}}), (\mathsf{st}_P^{\mathsf{rec}}), \perp, \perp)$
 6: return $\mathsf{st}_P$
```

```
$\mathsf{BARK.Send}(\mathsf{st}_P)$
 1: pick $k$ at random in $\{0,1\}^{\mathsf{BARK.kl}}$
 2: parse $\mathsf{st}_P = (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v}), \mathsf{Hsent}, \mathsf{Hreceived})$
 3: $\mathsf{uni.Init}(1^\lambda) \xrightarrow{\$} (\mathsf{st}_{S\mathsf{new}}, \mathsf{st}_P^{\mathsf{rec},v+1})$      ▷ append a new receive state to the $\mathsf{st}_P^{\mathsf{rec}}$ list
 4: $\mathsf{pt} \leftarrow (\mathsf{st}_{S\mathsf{new}}, k)$      ▷ $\mathsf{st}_{S\mathsf{new}}$ could be deleted immediately to avoid leaking
 5: take the smallest $i$ s.t. $\mathsf{st}_P^{\mathsf{send},i} \neq \perp$      ▷ $i = u - n$ if we had $n$ Receive since the last Send
 6: $\mathsf{uni.Send}(1^\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},i}, \ldots, \mathsf{st}_P^{\mathsf{send},u}), \mathsf{Hsent}, \mathsf{pt}) \xrightarrow{\$} (\mathsf{st}_P^{\mathsf{send},u}, \vec{\mathsf{ct}})$      ▷ update $\mathsf{st}_P^{\mathsf{send},u}$
 7: $\mathsf{st}_P^{\mathsf{send},i}, \ldots, \mathsf{st}_P^{\mathsf{send},u-1} \leftarrow \perp$      ▷ flush the send state list: only $\mathsf{st}_P^{\mathsf{send},u}$ remains
 8: $\mathsf{upd} \leftarrow (\mathsf{Hsent}, \vec{\mathsf{ct}})$      ▷ $\vec{\mathsf{ct}}$ has $u - i + 2 (= n+1)$ components
 9: $\mathsf{Hsent}' \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{upd})$
10: $\mathsf{st}'_P \leftarrow (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v+1}), \mathsf{Hsent}', \mathsf{Hreceived})$
11: return $(\mathsf{st}'_P, \mathsf{upd}, k)$

$\mathsf{BARK.Receive}(\mathsf{st}_P, \mathsf{upd})$
12: parse $\mathsf{st}_P = (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v}), \mathsf{Hsent}, \mathsf{Hreceived})$
13: parse $\mathsf{upd} = (h, \vec{\mathsf{ct}})$
14: set $n+1$ to the number of components in $\vec{\mathsf{ct}}$      ▷ the onion has $n$ layers
15: if $h \neq \mathsf{Hreceived}$ then return $(\mathsf{false}, \mathsf{st}_P, \perp)$
16: set $i$ to the smallest index such that $\mathsf{st}_P^{\mathsf{rec},i} \neq \perp$
17: if $i + n - 1 > v$ then return $(\mathsf{false}, \mathsf{st}_P, \perp)$
18: $\mathsf{uni.Receive}(\mathsf{hk}, (\mathsf{st}_P^{\mathsf{rec},i}, \ldots, \mathsf{st}_P^{\mathsf{rec},i+n-1}), \mathsf{Hreceived}, \vec{\mathsf{ct}}) \rightarrow (\mathsf{acc}, {\mathsf{st}'_P}^{\mathsf{rec},i+n-1}, \mathsf{pt})$
19: if $\mathsf{acc} = \mathsf{false}$ then return $(\mathsf{false}, \mathsf{st}_P, \perp)$
20: parse $\mathsf{pt} = (\mathsf{st}_P^{\mathsf{send},u+1}, k)$      ▷ a new send state is added in the list
21: $\mathsf{st}_P^{\mathsf{rec},i}, \ldots, \mathsf{st}_P^{\mathsf{rec},i+n-2} \leftarrow \perp$      ▷ update $\mathsf{st}_P^{\mathsf{rec}}$ stage 1: clean up
22: $\mathsf{st}_P^{\mathsf{rec},i+n-1} \leftarrow {\mathsf{st}'_P}^{\mathsf{rec},i+n-1}$      ▷ update $\mathsf{st}_P^{\mathsf{rec}}$ stage 2: update $\mathsf{st}_P^{\mathsf{rec},i+n-1}$
23: $\mathsf{Hreceived}' \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{upd})$
24: $\mathsf{st}'_P \leftarrow (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u+1}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v}), \mathsf{Hsent}, \mathsf{Hreceived}')$
25: return $(\mathsf{acc}, \mathsf{st}'_P, k)$
```

Fig. 7: Our BARK Protocol.

any exposure has very limited impact. If there are unidirectional sequences, the protocol becomes less and less efficient due to the growth of the lists.

We note that our protocol does *not* offer $(\mathsf{C_{leak}} \wedge \mathsf{C_{forge}^{P_{test}}})$-KIND security due to the following attack:

```
1: $\mathsf{EXP_{st}}(A) \rightarrow \mathsf{st}_A$
2: $\mathsf{EXP_{st}}(B) \rightarrow \mathsf{st}_B$      ▷ this reveals $\mathsf{sk}_B^{\mathsf{rec},1}$ to be used later on
3: $\mathsf{RATCH}(B, \mathsf{send}) \rightarrow \mathsf{upd}_B$
```

```
4:  $\mathsf{RATCH}(A, \mathsf{rec}, \mathsf{upd}_B) \to \mathsf{true}$
5:  $\mathsf{RATCH}(A, \mathsf{send}) \to \mathsf{upd}$
6:  $\mathsf{TEST}(A) \to k$
7:  $\mathsf{Send}(\mathsf{st}_A) \to \mathsf{upd}_A$                     ▷ this creates a trivial forgery
8:  $\mathsf{RATCH}(B, \mathsf{rec}, \mathsf{upd}_A) \to \mathsf{true}$    ▷ this makes B out-of-sync and updates $\mathsf{sk}_B^{\mathsf{rec},1}$
9:  $\mathsf{EXP}_{\mathsf{st}}(B) \to \mathsf{st}'_B$                   ▷ this reveals $\mathsf{sk}_B^{\mathsf{rec},2}$ and $\mathsf{sk}_B^{\mathsf{rec},1}$ (updated)
10: use $\mathsf{sk}_B^{\mathsf{rec},1}$ (from Step 2) and $\mathsf{sk}_B^{\mathsf{rec},2}$ to decrypt $\mathsf{upd}$
11: compare the result with $k$
```

Note that the trivial forgery is here to make the following $\mathsf{EXP}_{\mathsf{st}}(B)$ a non-trivial leakage for $\mathsf{sk}_B^{\mathsf{rec},2}$ ($\mathsf{sk}_B^{\mathsf{rec},1}$ is already known).

The attack is ruled out in the $(\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}_{\mathsf{forge}}^A \wedge \mathsf{C}_{\mathsf{forge}}^B)$-KIND security which does not allow forgeries until upd is received.

Theorem 20 (Correctness). *BARK in Fig. 7 is correct.*

Proof. To prove correctness, we first show that the underlying constructions in BARK are correct. This is quite straightforward for the signcryption SC and the multi-key extension onion of it. When two SC states st_S and st_R are generated, we call them *associated* SC *states* and we denote $\mathsf{st}_S \triangleright \mathsf{st}_R$ (or $\mathsf{st}_R \triangleleft \mathsf{st}_S$). By convention, we say that $\perp$ and $\perp$ are associated SC states as well: $\perp \triangleright \perp$. We extend this notion to vectors of SC states. We say that a vector $\mathsf{st}_P^{\mathsf{send}}$ of sending states is associated to a vector $\mathsf{st}_{\overline{P}}^{\mathsf{rec}}$ of receiving states if we have $u \leqslant v$, where u (resp. v) denotes the size of $\mathsf{st}_P^{\mathsf{send}}$ (resp. $\mathsf{st}_{\overline{P}}^{\mathsf{rec}}$), and we have $\mathsf{st}_P^{\mathsf{send},j} \triangleright \mathsf{st}_{\overline{P}}^{\mathsf{rec},j}$ for $j = 1, \ldots, u$. We denote $\mathsf{st}_P^{\mathsf{send}} \triangleright \mathsf{st}_R^{\mathsf{rec}}$ or $\mathsf{st}_P^{\mathsf{rec}} \triangleleft \mathsf{st}_R^{\mathsf{send}}$.

For BARK, we first observe that Hsent is the hash of $\mathsf{sent}_{\mathsf{msg}}$ and Hreceived is the hash of $\mathsf{received}_{\mathsf{msg}}$. In the CORRECT game, the participants are always in a matching status, thus the hashes match. We ignore them below.

For every participant P and every integer i, let $\mathsf{st}_{P,i}$ be the state of P after the i^{th} step of the loop in the CORRECT game. Let $u_{P,i}$ be the size of $\mathsf{st}_{P,i}^{\mathsf{send}}$ and $v_{P,i}$ be the size of $\mathsf{st}_{P,i}^{\mathsf{rec}}$. Similarly, let $k_{P,i}$ be the key generated by P at step i.

Each new entry $\mathsf{st}_{P,i}^{\mathsf{rec},j}$ is generated as some st_R from a uni.Init in BARK.Send. Each new entry $\mathsf{st}_{P,i}^{\mathsf{send},j}$ is the result of a decryption (which is presumably some st_S generated by $\overline{P}$ from a uni.Init before encryption). Entries $\mathsf{st}_{P,i}^{\mathsf{rec},j}$ are updated with outputs from $\mathsf{SC.Gen}_S$ in uni.Send. Conversely, entries $\mathsf{st}_{P,i}^{\mathsf{send},j}$ are updated with the result of decryptions (which are presumably some outputs from $\mathsf{SC.Gen}_R$ in uni.Send by $\overline{P}$). We have to identify which $\mathsf{st}_{P,i}^{\mathsf{send},j}$ is associated with which $\mathsf{st}_{\overline{P},i'}^{\mathsf{rec},j}$.

We can see by looking at the Send and Receive procedures that exactly $u_{P,i} - 1$ messages were received by P after the i^{th} step of the game ($\mathsf{st}_P^{\mathsf{send}}$ grows by 1 at every Receive and its size remains unchanged at every Send) and exactly $v_{P,i} - 1$ messages were sent by P ($\mathsf{st}_P^{\mathsf{rec}}$ grows by 1 at every Send and its size remains unchanged at every Receive). Actually, $\mathsf{st}_P^{\mathsf{rec},j+1}$ was created when P sent his j^{th} message and $\mathsf{st}_{\overline{P}}^{\mathsf{send},j+1}$ was set when $\overline{P}$ decrypted the j^{th} message from P. Due to the structure of the CORRECT game, a participant P cannot receive more messages than what $\overline{P}$ has sent. Hence, we have the property that

$$\forall P, m, m' \quad m \leqslant m' \Longrightarrow u_{P,m} \leqslant v_{\overline{P},m'} \tag{1}$$

If $\mathsf{sched}_{i+1} = (P, \mathsf{send})$, we denote $\mathsf{st}_{P,i}^{\mathsf{role}} \xrightarrow{\mathsf{send}} \mathsf{st}_{P,i+1}^{\mathsf{role}}$ for $\mathsf{role} \in \{\mathsf{send}, \mathsf{rec}\}$. Similarly, if $\mathsf{sched}_{i+1} = (P, \mathsf{rec})$, we denote $\mathsf{st}_{P,i}^{\mathsf{role}} \xrightarrow{\mathsf{rec}} \mathsf{st}_{P,i+1}^{\mathsf{role}}$ for $\mathsf{role} \in \{\mathsf{send}, \mathsf{rec}\}$.

By inspection on the Send and Receive procedures, we easily show that

$$\left(\mathsf{st}_{P,m}^{\mathsf{send}} \triangleright \mathsf{st}_{\overline{P},m'}^{\mathsf{rec}} \wedge \mathsf{st}_{P,m}^{\mathsf{send}} \xrightarrow{\mathsf{send}} \mathsf{st}_{P,m+1}^{\mathsf{send}} \wedge \mathsf{st}_{\overline{P},m'}^{\mathsf{rec}} \xrightarrow{\mathsf{rec}} \mathsf{st}_{\overline{P},m'+1}^{\mathsf{rec}}\right) \Longrightarrow \begin{cases} \mathsf{st}_{P,m+1}^{\mathsf{send}} \triangleright \mathsf{st}_{\overline{P},m'+1}^{\mathsf{rec}} \\ \mathsf{st}_{P,m+1}^{\mathsf{rec},v_{P,m+1}} \triangleleft \mathsf{st}_{\overline{P},m'+1}^{\mathsf{send},u_{\overline{P},m'+1}} \\ k_{P,m+1} = k_{\overline{P},m'+1} \end{cases}$$

This is essentially the correctness of the uni procedure. We call this property the correctness of the send/rec transition.

Similarly,

$$\left(\mathsf{st}^{\mathsf{send}}_{P,m} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{P},m'} \wedge \mathsf{st}^{\mathsf{rec}}_{\overline{P},m'} \xrightarrow{\mathsf{send}} \mathsf{st}^{\mathsf{rec}}_{\overline{P},m'+1}\right) \Longrightarrow \mathsf{st}^{\mathsf{send}}_{P,m} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{P},m'+1}$$

because the send operation only adds one extra rec state $\mathsf{st}^{\mathsf{rec},\nu_{\overline{P},m'+1}}_{\overline{P},m'+1}$, which cannot be taken into account in the $\triangleright$ definition since $u_{P,m} \leqslant \nu_{\overline{P},m'} < \nu_{\overline{P},m'+1}$. We call this property the correctness of the $\bot/\mathsf{send}$ transition.

We show by induction on i that

$$\forall P, m, m' \quad (0 \leqslant m \leqslant i, 0 \leqslant m' \leqslant i, u_{P,m} \leqslant \nu_{\overline{P},m'}, \nu_{P,m} = u_{\overline{P},m'}) \Longrightarrow \mathsf{st}^{\mathsf{send}}_{P,m} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{P},m'}$$

We denote this property by R_i.

First of all, R_0 is trivial: it is equivalent to $\forall P \quad \mathsf{st}^{\mathsf{send},1}_{P,0} \triangleright \mathsf{st}^{\mathsf{rec},1}_{\overline{P},0}$ which is true due to how $\mathsf{InitAll}$ works. We now assume that R_i is true and we prove R_{i+1}. We consider the operation done in this iteration number $i+1$, following sched_{i+1}. We assume it involves a participant Q. Since sched_{i+1} only involves Q, we have $\mathsf{st}_{\overline{Q},i+1} = \mathsf{st}_{\overline{Q},i}$. Thus, to prove R_{i+1}, we only have to consider $P = Q$ and $m = i+1$, or $P = \overline{Q}$ and $m' = i+1$. Hence, we only have to prove the two following properties:

$$R^1_{i+1} : \forall m' \quad (0 \leqslant m' \leqslant i, u_{Q,i+1} \leqslant \nu_{\overline{Q},m'}, \nu_{Q,i+1} = u_{\overline{Q},m'}) \Longrightarrow \mathsf{st}^{\mathsf{send}}_{Q,i+1} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{Q},m'} \qquad (2)$$

$$R^2_{i+1} : \forall m \quad (0 \leqslant m \leqslant i, u_{\overline{Q},m} \leqslant \nu_{Q,i+1}, \nu_{\overline{Q},m} = u_{Q,i+1}) \Longrightarrow \mathsf{st}^{\mathsf{rec}}_{Q,i+1} \triangleleft \mathsf{st}^{\mathsf{send}}_{\overline{Q},m} \qquad (3)$$

Case 1: $\mathsf{sched}_{i+1} = (Q, \mathsf{send})$. We first prove R^1_{i+1}: Let m' be such that $0 \leqslant m' \leqslant i$. Due to the send operation, we have $\nu_{Q,i+1} = \nu_{Q,i} + 1$. Since $m' \leqslant i$, due to (1) we have $u_{\overline{Q},m'} \leqslant \nu_{Q,i}$ thus $u_{\overline{Q},m'} < \nu_{Q,i+1}$. Hence, we cannot have $\nu_{Q,i+1} = u_{\overline{Q},m'}$. Therefore, R^1_{i+1} is true.

We also prove R^2_{i+1}: Let m be such that $0 \leqslant m \leqslant i$, $u_{\overline{Q},m} \leqslant \nu_{Q,i+1}$, and $\nu_{\overline{Q},m} = u_{Q,i+1}$. Due to the send operation, we have $u_{Q,i+1} = u_{Q,i}$. Since $m \leqslant i$, due to (1) we have $u_{\overline{Q},m} \leqslant \nu_{Q,i}$. Hence, we have $u_{\overline{Q},m} \leqslant \nu_{Q,i}$ and $\nu_{\overline{Q},m} = u_{Q,i}$. Due to R_i, we deduce $\mathsf{st}^{\mathsf{rec}}_{Q,i} \triangleleft \mathsf{st}^{\mathsf{send}}_{\overline{Q},m}$. Since, $\mathsf{st}^{\mathsf{rec}}_{Q,i} \xrightarrow{\mathsf{send}} \mathsf{st}^{\mathsf{rec}}_{Q,i+1}$, we apply the correctness of the $\bot/\mathsf{send}$ transition to deduce $\mathsf{st}^{\mathsf{rec}}_{Q,i+1} \triangleleft \mathsf{st}^{\mathsf{send}}_{\overline{Q},m}$. Therefore, R^2_{i+1} is true.

Therefore, R_{i+1} is true in the $\mathsf{sched}_{i+1} = (Q, \mathsf{send})$ case.

Case 2: $\mathsf{sched}_{i+1} = (Q, \mathsf{rec})$. We consider the smallest $m'' \leqslant i$ such that $\nu_{\overline{Q},m''} = u_{Q,i+1}$ (i.e., the message received by Q at step $i+1$ is the message sent by $\overline{Q}$ at step m'' and $\mathsf{sched}_{m''} = (\overline{Q}, \mathsf{send})$). Due to both the send and receive operations, we have $\nu_{\overline{Q},m''} = \nu_{\overline{Q},m''-1} + 1$ and $u_{Q,i+1} = u_{Q,i} + 1$ thus $\nu_{\overline{Q},m''-1} = u_{Q,i}$. Furthermore, due to (1), since $m'' - 1 \leqslant i$, we have $u_{\overline{Q},m''-1} \leqslant \nu_{Q,i}$. We can apply R_i and deduce $\mathsf{st}^{\mathsf{send}}_{\overline{Q},m''-1} \triangleright \mathsf{st}^{\mathsf{rec}}_{Q,i}$. We can then apply the correctness of the $\mathsf{send}/\mathsf{rec}$ transition and deduce $\mathsf{st}^{\mathsf{send}}_{\overline{Q},m''} \triangleright \mathsf{st}^{\mathsf{rec}}_{Q,i+1}$ and $\mathsf{st}^{\mathsf{rec},\nu_{\overline{Q},m''}}_{\overline{Q},m''} \triangleleft \mathsf{st}^{\mathsf{send},u_{Q,i+1}}_{Q,i+1}$.

We first prove R^1_{i+1}. Let m' be such that $0 \leqslant m' \leqslant i$, $u_{Q,i+1} \leqslant \nu_{\overline{Q},m'}$, and $\nu_{Q,i+1} = u_{\overline{Q},m'}$. Due to the rec operation, we have $u_{Q,i+1} = u_{Q,i}+1$ and $\nu_{Q,i+1} = \nu_{Q,i}$. Since $m' \leqslant i$, due to (1) we have $u_{\overline{Q},m'} \leqslant \nu_{Q,i}$. thus, $u_{Q,i} < \nu_{\overline{Q},m'}$, and $\nu_{Q,i} = u_{\overline{Q},m'}$. Due to R_i, we have $\mathsf{st}^{\mathsf{send}}_{Q,i} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{Q},m'}$. The rec operation only adds a new element $\mathsf{st}^{\mathsf{send},u_{Q,i+1}}_{Q,i+1}$ in $\mathsf{st}^{\mathsf{send}}_{Q,i}$. We have shown above that this new element is such that $\mathsf{st}^{\mathsf{send},u_{Q,i+1}}_{Q,i+1} \triangleright \mathsf{st}^{\mathsf{rec},\nu_{\overline{Q},m''}}_{\overline{Q},m''}$. Since $\nu_{\overline{Q},m''} = u_{Q,i+1} \leqslant \nu_{\overline{Q},m'}$, we have $m'' \leqslant m'$. None of the rec operations for $\overline{Q}$ in between step m'' and step m' will need to use $\mathsf{st}^{\mathsf{rec},\nu_{\overline{Q},m''}}_{\overline{Q},m''}$ because Q has $u_{Q,j} \leqslant u_{Q,i} < \nu_{\overline{Q},m''}$ for all $j \leqslant i$. Hence, $\mathsf{st}^{\mathsf{rec},\nu_{\overline{Q},m''}}_{\overline{Q},m''} = \mathsf{st}^{\mathsf{rec},\nu_{\overline{Q},m''}}_{\overline{Q},m'}$. We deduce $\mathsf{st}^{\mathsf{send},u_{Q,i+1}}_{Q,i+1} \triangleright \mathsf{st}^{\mathsf{rec},\nu_{\overline{Q},m'}}_{\overline{Q},m'}$. Since $\mathsf{st}^{\mathsf{send}}_{Q,i} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{Q},m'}$ and the rec operation does not change the elements in $\mathsf{st}^{\mathsf{send}}_{Q,i}$, we deduce $\mathsf{st}^{\mathsf{send}}_{Q,i+1} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{Q},m'}$. Therefore, R^1_{i+1} is true.

We also prove R^2_{i+1}: Let m be such that $0 \leqslant m \leqslant i$, $u_{\overline{Q},m} \leqslant \nu_{Q,i+1}$, and $\nu_{\overline{Q},m} = u_{Q,i+1}$. We have already proven that $\mathsf{st}^{\mathsf{send}}_{\overline{Q},m''} \triangleright \mathsf{st}^{\mathsf{rec}}_{Q,i+1}$. We have $m \geqslant m''$ (because m'' is the smallest integer with the $\nu_{\overline{Q},m} = u_{Q,i+1}$ property). Hence, only some rec operations occur for $\overline{Q}$ between step m'' and step m. These rec only add new elements in $\mathsf{st}^{\mathsf{send}}_{\overline{Q},m''}$. By the same reasoning as before, these elements are associated to elements in $\mathsf{st}^{\mathsf{rec}}_{\mathsf{Q}}$ at the same position which remain untouched until $\mathsf{st}^{\mathsf{rec}}_{\mathsf{Q},i+1}$ because Q does not receive more messages after the one sent at step m'' by $\overline{\mathsf{Q}}$. Hence, $\mathsf{st}^{\mathsf{send}}_{\overline{\mathsf{Q}},m} \triangleright \mathsf{st}^{\mathsf{rec}}_{\mathsf{Q},i+1}$. Therefore, R^1_{i+1} is true.

Therefore, R_{i+1} is also true in the $\mathsf{sched}_{i+1} = (\mathsf{Q}, \mathsf{rec})$ case.

By induction, R_i is true for every i. Hence,

$$\forall \mathsf{P}, m, m' \quad (u_{\mathsf{P},m} \leqslant v_{\overline{\mathsf{P}},m'}, v_{\mathsf{P},m} = u_{\overline{\mathsf{P}},m'}) \Longrightarrow \mathsf{st}^{\mathsf{send}}_{\mathsf{P},m} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{\mathsf{P}},m'}$$

Let P be a participant and let j be any integer. If $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{key}}$ has an entry at position j, there must be a step m' at which $\overline{\mathsf{P}}$ received the j$^{\text{th}}$ message, and there must be a step $m \leqslant m'$ at which P sent the j$^{\text{th}}$ message. We have $v_{\mathsf{P},m-1} = u_{\overline{\mathsf{P}},m'-1}$. Due to (1), we have $u_{\mathsf{P},m-1} \leqslant v_{\overline{\mathsf{P}},m'-1}$. Hence, we have $\mathsf{st}^{\mathsf{send}}_{\mathsf{P},m-1} \triangleright \mathsf{st}^{\mathsf{rec}}_{\overline{\mathsf{P}},m'-1}$. Due to the correctness of the send/rec transition, the j$^{\text{th}}$ entry $k_{\mathsf{P},m}$ which is inserted at step m in $\mathsf{sent}^{\mathsf{P}}_{\mathsf{key}}$ is equal to the j$^{\text{th}}$ entry $k_{\overline{\mathsf{P}},m'}$ which is inserted at step m' in $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{key}}$.

Since this holds for any j for which $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{key}}$ has a j$^{\text{th}}$ entry, we deduce that $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{key}}$ is a prefix of $\mathsf{sent}^{\mathsf{P}}_{\mathsf{key}}$. Since this holds for any P, the protocol is correct. □

We state the security of our protocol below.

Theorem 21 (Unrecoverability). *For any* $\lambda, q, T, \varepsilon$, *if* H *is a* $(\lambda, T, \varepsilon)$*-collision-resistant hash function, then* BARK *in Fig. 7 is* $(\lambda, q, T, \varepsilon)$-RECOVER*-secure.*

Proof. Each upd sent must include the hash of the previous upd sent. We call them chained for this reason. If $(\mathsf{seq}_1, \mathsf{upd}, \mathsf{seq}_2)$ and $(\mathsf{seq}_3, \mathsf{upd}, \mathsf{seq}_4)$ are two validly chained list of messages with $\mathsf{seq}_1 \neq \mathsf{seq}_3$, we can easily see that $\mathsf{upd} = (h, \vec{\mathsf{ct}})$ must include a collision on h. This cannot happen thanks to collision resistance. □

Theorem 22 (Unpredictability). *For any* λ *and* q, BARK *in Fig. 7 is* $(\lambda, q, q2^{-\mathsf{Sym.kl}(\lambda)})$-PREDICT*-secure.*

Proof. We show that guessing upd before a BARK.Send call implies guessing $k \in \{0,1\}^{\mathsf{Sym.kl}}$, from an information theory point of view.

Indeed, given an adversary $\mathcal{A}$ playing the PREDICT game, we design an unbounded adversary $\mathcal{A}'$ to play the following game Γ:

```
Game $\Gamma(1^\lambda)$:
1: run $\mathcal{A}' \to k'$
2: pick $k \in \{0,1\}^{\mathsf{Sym.kl}(\lambda)}$
3: **return** $1_{k=k'}$
```

Clearly, whatever unbounded algorithm $\mathcal{A}'$ we use, the probability that Γ returns 1 is bounded by $2^{-\mathsf{Sym.kl}(\lambda)}$.

To construct $\mathcal{A}'$, the algorithm simulates the PREDICT game until $\mathcal{A}$ is done (i.e., before the final RATCH in the game). The simulation of $\mathcal{A}$ gives some P. Next, $\mathcal{A}'$ looks into the state st_{P} to see with which key in the next RATCH send call will encrypt a key k. The st_{P} includes some PKC encryption key pk in $\mathsf{st}^n_{\mathsf{S}}$ which is used by PKC in SC. By exhaustive search, $\mathcal{A}'$ can find some associated decryption key for PKC to be able to decrypt. This is done by finding coins ρ such that $\mathsf{PKC.Gen}(\rho) = (\mathsf{sk}, \mathsf{pk})$. Then, $\mathcal{A}'$ guesses which upd from $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{msg}}$ will come out, in the case the PREDICT game succeeds. This gives a upd to decrypt. If P ran $\mathsf{PKC.Enc}(\mathsf{pk}, x) \to \mathsf{ct}_n$ for some x and $\mathcal{A}'$ runs $\mathsf{PKC.Dec}(\mathsf{sk}, \mathsf{ct}_n)$ with ct_n from upd, due to the correctness of PKC, this must give x from which $\mathcal{A}'$ can extract k_n. Hence, the decryption gives one key k' which is the output of $\mathcal{A}'$.

Let p be the probability that the PREDICT game outputs 1. We note that $\mathcal{A}'$ guesses the correct upd in $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{msg}}$ with probability $\frac{1}{q}$. Hence, with probability at least $\frac{p}{q}$, the adversary $\mathcal{A}'$ predicts the value of $k = k'$ before it is selected at random. Hence, Γ outputs 1 with probability $\frac{p}{q}$.

Since $\frac{p}{q} \leqslant 2^{-\mathsf{Sym.kl}}$, we have $p \leqslant q2^{-\mathsf{Sym.kl}}$. □

Theorem 23 (Unforgeability). *For any* $\lambda, q, T, \varepsilon$*, assuming that* DSS *is* $(\lambda, T', \varepsilon)$-$\mathsf{SEF\text{-}OTCMA}$-*secure and* H *is a* $(\lambda, T, \varepsilon')$*-collision-resistant hash function, then* BARK *in Fig. 7 is* $(\lambda, q, T, 2(q+1)\varepsilon + \varepsilon')$-$\mathsf{FORGE}$-*secure. We let* $T' = T + T_{\mathsf{Init}} + qT_{\mathsf{Send,Receive}}$ *where* T_{Init} *denotes a complexity upper bound of* Setup *and* $\mathsf{InitAll}$ *and* $T_{\mathsf{Send,Receive}}$ *denotes a complexity upper bound of both* Send *and* $\mathsf{Receive}$.

Proof. It is quite clear that SC is $(\lambda, T', \varepsilon)$-$\mathsf{EF\text{-}OTCPA}$-secure (as defined in Def. 27 in Appendix A) if and only if DSS is $(\lambda, T', \varepsilon)$-$\mathsf{SEF\text{-}OTCMA}$-secure.

We consider an adversary $\mathcal{A}$ making a forgery $\mathsf{upd} = (h, \vec{ct})$ which is accepted by P in the FORGE game. We let $n + 1$ be the number of components in $\vec{ct}$. We assume without loss of generality that upd is accepted, is a forgery, and is a non-trivial forgery (otherwise, we know that FORGE game returns 0).

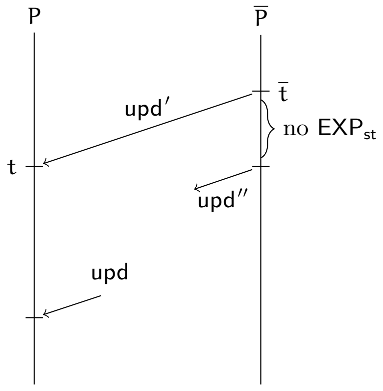


Fig. 8: Forgery

We first assume that P successfully received a message upd' from $\overline{\mathsf{P}}$ before upd. (See Fig. 8.) Due to the definition of a forgery, $\mathsf{received}^{\mathsf{P}}_{\mathsf{msg}}(t) = \mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{msg}}(\overline{t})$ for some time $\overline{t}$ for $\overline{\mathsf{P}}$ when $\overline{\mathsf{P}}$ had a $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send}) \to \mathsf{upd}'$ call. Since the forgery upd is non-trivial, this call starts a ratcheting period for $\overline{\mathsf{P}}$ with no state exposure. This ratcheting period may either never end, or end with some $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send}) \to \mathsf{upd}''$ call.

The $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send}) \to \mathsf{upd}'$ call at time $\overline{t}$ for $\overline{\mathsf{P}}$ defines some value u and some states $\mathsf{st}^{\mathsf{send},u}_{\overline{\mathsf{P}}}$ and $\mathsf{st}^{\mathsf{rec},u}_{\mathsf{P}}$. During this call, $\mathsf{st}^{\mathsf{send},u}_{\overline{\mathsf{P}}}$ is updated with some $\mathsf{st}'_{\mathsf{S}}$ generated by $\mathsf{uni.Send}$. Inside $\mathsf{st}'_{\mathsf{S}}$, there is a $\mathsf{sk}'_{\mathsf{S}}$ generated by $\mathsf{SC.Gen_S}$ in $\mathsf{uni.Send}$. This $\mathsf{sk}'_{\mathsf{S}}$ will be our signing key of interest. It comes with a verifying key $\mathsf{pk}'_{\mathsf{S}}$ of interest. This $\mathsf{pk}'_{\mathsf{S}}$ is put in $\mathsf{st}'_{\mathsf{R}}$ and encrypted in the returned $\vec{ct}$. The $\mathsf{RATCH}(\mathsf{P}, \mathsf{rec}, \mathsf{upd}')$ call at time t for P decrypts it. Due to the matching status and correctness, $\mathsf{st}'_{\mathsf{R}}$ is well decrypted and stored in some $\mathsf{st}^{\mathsf{rec},i'+n'-1}_{\mathsf{P}}$ with $i' + n' - 1 = u$. There could be some sent messages by P between time t and the reception of the forgery upd. This can only add more receive states. In any case, when upd arrives, the first receive state to be used is $\mathsf{st}^{\mathsf{rec},i}_{\mathsf{P}}$ with $i = u$ containing $\mathsf{pk}'_{\mathsf{S}}$.

We observe that the signing key of interest $\mathsf{sk}'_{\mathsf{S}}$ stays stored in $\overline{\mathsf{P}}$ until it is used and erased by the $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send}) \to \mathsf{upd}''$ call. It is unused otherwise. Namely, there is no $\mathsf{EXP_{st}}$ revealing it.

Thanks to the structure of the FORGE game, upd'' (if any) is released before upd is delivered to P. Hence, we observe that since upd is a forgery, we have $\mathsf{upd} \neq \mathsf{upd}''$. Hence, either we have a collision on H or we have a forgery for SC with our key of interest.

Similarly, if P received no upd' before upd, we can do the same with $\overline{t} = 0$ and $u = 1$. We have a key of interest which is generated by $\mathsf{SC.Gen_S}$ in $\mathsf{BARK.Gen}$ for $\overline{\mathsf{P}}$.

To prove FORGE security, we transform the FORGE game into many $\mathsf{EF\text{-}OTCPA}$ games (following Def. 27) with adversaries $\mathcal{A}_i$. For the i^{th} use of $(\mathsf{SC.Gen_S}, \mathsf{SC.Gen_R})$, $\mathcal{A}_i$ simulates the FORGE

game except the i^{th} executions of $\mathsf{SC.Gen_S}$ and $\mathsf{SC.Gen_R}$. Instead, it takes input $\mathsf{sk_R}, \mathsf{pk_S}, \mathsf{pk_R}$ in the EF-OTCPA game. Hence, only $\mathsf{sk_S}$ is missing. The simulation continues until the key $\mathsf{sk_S}$ is needed. It can only be needed by at most one SC.Enc operation, which can be simulated by a one-time chosen plaintext attack in the EF-OTCPA game, or by an $\mathsf{EXP_{st}}$ call. If such a call occurs, $\mathcal{A}_i$ aborts. Similarly, if after the end of the simulation, it appears that the FORGE game returns 0 or the missing key $\mathsf{sk_S}$ is not the signing key of interest, $\mathcal{A}_i$ aborts. What comes out from our previous discussion is that if FORGE returns 1, then there exists one i such that $\mathcal{A}_i$ can output a forgery in the EF-OTCPA game. Since there are at most $2(q+1)$ such $\mathcal{A}_i$, we can conclude. □

Theorem 24 (KIND Security). *For any $\lambda, q, T, \varepsilon$, assuming that* PKC *is* $(\lambda, T', \varepsilon)$-IND-CCA-*secure and* Sym *is* $(\lambda, T', \varepsilon')$-IND-OTCCA-*secure, then* BARK *in Fig. 7 is* $(\lambda, q, T, 2q^2\varepsilon + 2q(q+1)\varepsilon')$-$(\mathsf{C_{leak}} \wedge \mathsf{C^A_{forge}} \wedge \mathsf{C^B_{forge}})$-KIND-*secure. Here,* $T' = T + T_{\mathsf{Init}} + qT_{\mathsf{Send,Receive}}$ *where* T_{Init} *denotes a complexity upper bound of* Setup *and* Initall *and* $T_{\mathsf{Send,Receive}}$ *denotes a complexity upper bound of both* Send *and* Receive.

Due to Th. 16, Th. 23, and Th. 24, we deduce $(\mathsf{C_{leak}} \wedge \mathsf{C^A_{trivial\ forge}} \wedge \mathsf{C^B_{trivial\ forge}})$-KIND-security. The advantage of treating $(\mathsf{C_{leak}} \wedge \mathsf{C^A_{forge}} \wedge \mathsf{C^B_{forge}})$-KIND-security specifically is that we clearly separate the required security assumptions for DSS and PKC.

Due to Th. 19, Th. 21, and Th. 24, we deduce $(\mathsf{C_{leak}} \wedge \mathsf{C_{ratchet}})$-KIND-security.

Proof. It is quite clear that SC is $(\lambda, T', \varepsilon)$-IND-CCA-secure if and only if PKC is $(\lambda, T', \varepsilon)$-IND-CCA-secure.

We take a KIND game which we denote by Γ_b. The idea of the following proof is that we will identify which decryption keys are "critical" for TEST to be indistinguishable, in the sense that the leakage of any of these keys would allow the adversary to decrypt $\mathsf{upd_{test}}$. We will then apply the IND-CCA reduction on those critical keys.

Let assume that P is the sending participant of $\mathsf{upd_{test}}$ in Γ_b and that Γ_b is clean. Let t be the time for P just after sending $\mathsf{upd_{test}}$. Due to cleanness, P is in a matching status at time t (as he received no forgery). We assume that time t for P originates from time $\bar{t}$ for $\overline{\mathsf{P}}$. By looking at the sequence of all RATCH calls with P, we further let c be the number of consecutive $\mathsf{RATCH}(\mathsf{P}, \mathsf{send})$ until sending $\mathsf{upd_{test}}$. If $c = 0$, it means that the previous RATCH call with P before sending $\mathsf{upd_{test}}$ was a reception or that there were no previous RATCH call with P. Fig. 9 represents a case with $c = 2$.

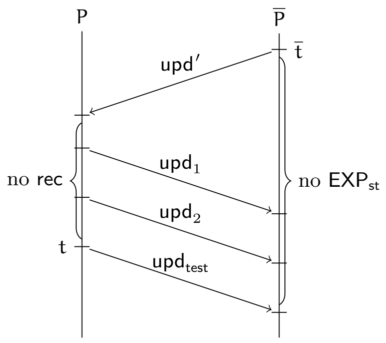


Fig. 9: A clean TEST

Let us first consider the case that $\overline{\mathsf{P}}$ sends a message upd' at time $\bar{t}$. This $\mathsf{RATCH}(\overline{\mathsf{P}}, \mathsf{send}) \to \mathsf{upd}'$ call at time $\bar{t}$ defines a $(\mathsf{st_S}, \mathsf{st_R})$ pair using uni.Init in Step 3 of BARK.Send. The $\mathsf{st_S}$ state is encrypted in upd' and $\mathsf{st_R}$ remains in the state of $\overline{\mathsf{P}}$. It is the new $\mathsf{st}^{\mathsf{rec},\nu}_{\overline{\mathsf{P}}}$ element. This element contains a decryption key which is critical. In the other case, there is no sending at time $\bar{t}$. This means that $\bar{t} = 0$, since time t originates from time $\bar{t}$. Hence, the value of $\mathsf{st}^{\mathsf{rec},\nu}_{\overline{\mathsf{P}}}$ (with $\nu = 1$, actually) is set up at the initialization $\mathsf{InitAll}$ in a similar way and is critical as well. The $\mathsf{st}_{\overline{P}}^{\mathsf{rec},\nu}$ state is enough to simulate $\mathsf{BARK.Receive}$ hence decrypt all messages from P until $\mathsf{upd}_{\mathsf{test}}$.

In both cases, the value st_R of $\mathsf{st}_{\overline{P}}^{\mathsf{rec},\nu}$ at time $\bar{t}$ is associated to a value st_S of $\mathsf{st}_{P}^{\mathsf{send},u}$ at a time just before a series of $c+1$ consecutive $\mathsf{RATCH}(\mathsf{P},\mathsf{send})$ calls ending with $\mathsf{upd}_{\mathsf{test}}$. By definition, if P has no indirect leakage at time t, it means that there is no $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ after time $\bar{t}$ for $\overline{\mathsf{P}}$ and until $\mathsf{upd}_{\mathsf{test}}$ is received by $\overline{\mathsf{P}}$. The reception of any message erases this st_R. Due to cleanness, it is the case that the decryption key in st_R remains local and never leaks.

Next, each of the c $\mathsf{RATCH}(\mathsf{P},\mathsf{send})$ before sending $\mathsf{upd}_{\mathsf{test}}$ defines in $\mathsf{uni.Send}$ a new pair $(\mathsf{st}'_S,\mathsf{st}'_R)$ of associated states. The decryption state st'_R is encrypted with the previously set encryption state and sent encrypted. Thanks to the previous discussion, those st'_R do not leak either.

The idea of the proof below is that we will sequentially apply the $\mathsf{IND\text{-}CCA}$ reduction to each of these keys. Then, we will deduce that the Sym key k selected in $\mathsf{onion.Enc}$ to encrypt $\mathsf{upd}_{\mathsf{test}}$ is indistinguishable from random. Then, we will use the $\mathsf{IND\text{-}OTCCA}$ security of Sym to deduce that k_{test} is indistinguishable from random.

We define games $\Gamma_{b,P,\ell,c}$ with integers ℓ and c and a participant P as follows. Essentially, the parameters (P,ℓ,c) anticipate which RATCH call will send $\mathsf{upd}_{\mathsf{test}}$. The game will output $\perp$ if this is not the case. During the game, a register $\mathsf{critical}$ will be used. Initially, the game sets $\mathsf{critical}=0$. If any $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ occurs while $\mathsf{critical}>0$, the game stops and output $\perp$.

- In a first phase, the game works exactly like in Γ_b, until the ℓ^{th} sending RATCH call by $\overline{\mathsf{P}}$ is made in the $\ell>0$ case. Let upd' denote the sent message in this call $\mathsf{RATCH}(\overline{\mathsf{P}},\mathsf{send})\to\mathsf{upd}'$. We denote by $\bar{t}$ the time for $\overline{\mathsf{P}}$ right after this call. The game sets $\mathsf{critical}\leftarrow c+1$.
  In the $\ell=0$ case, the game just sets $\bar{t}=0$ and $\mathsf{critical}\leftarrow c+1$ and skips the rest of this phase.
- In a second phase, the game continues like in Γ_b until there is a $\mathsf{RATCH}(\mathsf{P},\mathsf{rec},\mathsf{upd}')$ to receive upd' which outputs $\mathsf{acc}=\mathsf{true}$ in the $\ell>0$ case.
  In the $\ell=0$ case, this phase is also skipped.
- In a third phase, the game continues like in Γ_b by observing every RATCH call on P. It counts the number of sending RATCH calls until a receiving RATCH call is made or the game stops. We denote by upd_j the j^{th} message in this sequence of $\mathsf{RATCH}(\mathsf{P},\mathsf{send})\to\mathsf{upd}_j$ calls.
  Every time a message among $\mathsf{upd}_1,\ldots,\mathsf{upd}_{c+1}$ is received and accepted by $\overline{\mathsf{P}}$, the register $\mathsf{critical}$ is decreased by 1.
- In a fourth phase, the game continues like in Γ_b.
- At the end of the game, if upd_{c+1} is not defined or $\mathsf{upd}_{\mathsf{test}}\neq\mathsf{upd}_{c+1}$, the game outputs $\perp$. Otherwise, the game outputs like in Γ_b.

Clearly, assuming that the (P,ℓ,c) parameters correctly "anticipated" which RATCH call sends $\mathsf{upd}_{\mathsf{test}}$, the only active modifications of the game are the ones returning $\perp$ but they all occur in cases which would eventually end up with Γ_b returning $\perp$ as well due to non-cleanness. The changes induce no behavior difference from the point of view of the adversary. Hence, for every set of random coins \$, we have the following property:

- either $\Gamma_b[\$]\to\perp$ and for all (P,ℓ,c) we have $\Gamma_{b,P,\ell,c}[\$]\to\perp$;
- or $\Gamma_b[\$]\to z\in\{0,1\}$, there exists a unique (P,ℓ,c) (anticipating the right $\mathsf{upd}_{\mathsf{test}}$) such that $\Gamma_{b,P,\ell,c}[\$]\to z$, and for all $(\mathsf{P}',\ell',c')\neq(\mathsf{P},\ell,c)$, we have $\Gamma_{b,P',\ell',c'}[\$]\to\perp$.

Hence, $\Pr[\Gamma_b\to 1]=\sum_{P,\ell,c}\Pr[\Gamma_{b,P,\ell,c}\to 1]$. Another property of $\Gamma_{b,P,\ell,c}$ is that no $\mathsf{EXP}_{\mathsf{st}}$ reveals any of the critical decryption keys to the adversary.

Next, we define hybrid games $\Gamma'_{b,P,\ell,c,c'}$ and $\Gamma''_{b,P,\ell,c,c'}$ by changing $\mathsf{onion.Enc}$ and $\mathsf{onion.Dec}$ as in Fig. 10.

Clearly, for $c'=0$, we can see that there is no change between $\Gamma_{b,P,\ell,c}$ and $\Gamma'_{b,P,\ell,c,0}$.

For $c'>0$, the game $\Gamma'_{b,P,\ell,c,c'}$ follows $\Gamma''_{b,P,\ell,c,c'-1}$ but for the $\mathsf{RATCH}(\mathsf{P},\mathsf{send})\to\mathsf{upd}_{c'}$ call and the possible $\mathsf{RATCH}(\overline{\mathsf{P}},\mathsf{rec},\mathsf{upd}_{c'})$ call. For the RATCH call sending $\mathsf{upd}_{c'}$, we change the $\mathsf{ct}_i\leftarrow\mathsf{SC.Enc}(\mathsf{st}_S^i,\mathsf{ad}_i,k_i)$ in Step 7 of $\mathsf{onion.Enc}$ as follows:

```
1: **if** $i=n$ [and this execution is to compute $\mathsf{upd}_{c'}$] **then**
2:     pick $k^*\in\{0,1\}^{\mathsf{Sym.kl}(\lambda)}$
```

```
$\mathsf{onion.Enc}(1^\lambda, \mathsf{hk}, \mathsf{st}_S^1, \ldots, \mathsf{st}_S^n, \mathsf{ad}, \mathsf{pt})$
        in $\Gamma'_{b,P,\ell,c,c'}$ (resp. $\Gamma''_{b,P,\ell,c,c'}$)
 1: if sending participant is $\mathsf{P}$ then
 2:     set $j$ such that this execution is to compute $\mathsf{upd}_j$ ($j = \perp$ if none)
 3: else
 4:     $j \leftarrow \perp$
 5: end if
 6: pick $k_1, \ldots, k_n$ in $\{0,1\}^{\mathsf{Sym.kl}(\lambda)}$
 7: $k \leftarrow k_1 \oplus \cdots \oplus k_n$
 8: if $j \neq \perp$ and $j < c'$ (resp. $j \leqslant c'$) then
 9:     pick a random $\mathsf{pt}'$ of same size as $\mathsf{pt}$
10:     $\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt}')$
11:     store $D[j] \leftarrow (\mathsf{ct}_{n+1}, \mathsf{pt})$
12: else
13:     $\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt})$
14: end if
15: $\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
16: for $i = n$ down to 1 do
17:     $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, \mathsf{ct}_{i+1})$
18:     if $i = n$ and $j \leqslant c'$ then
19:         pick $k^* \in \{0,1\}^{\mathsf{Sym.kl}}$
20:         $\mathsf{ct}_i \leftarrow \mathsf{SC.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k^*)$
21:         store $E[j] \leftarrow (\mathsf{ad}_i, \mathsf{ct}_i, k_i)$
22:     else
23:         $\mathsf{ct}_i \leftarrow \mathsf{SC.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k_i)$
24:     end if
25: end for
26: return $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$
```

```
$\mathsf{onion.Dec}(\mathsf{hk}, \mathsf{st}_R^1, \ldots, \mathsf{st}_R^n, \mathsf{ad}, \vec{\mathsf{ct}})$
        in $\Gamma'_{b,P,\ell,c,c'}$ (resp. $\Gamma''_{b,P,\ell,c,c'}$)
 1: if receiving participant is $\overline{\mathsf{P}}$ then
 2:     $j \leftarrow c + 2 - \mathsf{critical}$
 3: else
 4:     $j \leftarrow \perp$
 5: end if
 6: if $|\vec{\mathsf{ct}}| \neq n + 1$ then return $\perp$
 7: parse $\vec{\mathsf{ct}} = (\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$
 8: $\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
 9: for $i = n$ down to 1 do
10:     $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, \mathsf{ct}_{i+1})$
11:     if $i = n$ and $j \neq \perp$ and $j \leqslant c'$ then
12:         parse $E[j] = (\mathsf{ad}, \mathsf{ct}, k)$
13:         if $(\mathsf{ad}_i, \mathsf{ct}_i) = (\mathsf{ad}, \mathsf{ct})$ then
14:             $k_i \leftarrow k$
15:         else
16:             $\mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \rightarrow k_i$
17:         end if
18:     else
19:         $\mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \rightarrow k_i$
20:     end if
21:     if $k_i = \perp$ then return $\perp$
22: end for
23: $k \leftarrow k_1 \oplus \cdots \oplus k_n$
24: if $j \neq \perp$ and $j < c'$ (resp. $j \leqslant c'$) then
25:     parse $D[j] = (\mathsf{ct}, \mathsf{pt}^*)$
26:     if $\mathsf{ct}_{n+1} = \mathsf{ct}$ then
27:         $\mathsf{pt} \leftarrow \mathsf{pt}^*$
28:     else
29:         $\mathsf{pt} \leftarrow \mathsf{Sym.Dec}(k, \mathsf{ct}_{n+1})$
30:     end if
31: else
32:     $\mathsf{pt} \leftarrow \mathsf{Sym.Dec}(k, \mathsf{ct}_{n+1})$
33: end if
34: return $\mathsf{pt}$
```

Fig. 10: New onion in $\Gamma'_{b,P,\ell,c,c'}$ and $\Gamma''_{b,P,\ell,c,c'}$

```
 3:     $\mathsf{ct}_i \leftarrow \mathsf{SC.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k^*)$
 4:     store $E[c'] \leftarrow (\mathsf{ad}_i, \mathsf{ct}_i, k_i)$
 5: else
 6:     $\mathsf{ct}_i \leftarrow \mathsf{SC.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k_i)$
 7: end if
```

It only affects $i = n$, i.e. the $\mathsf{SC.Enc}$ using the last sending state. Similarly, we replace $\mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \to k_i$ in Step 6 of $\mathsf{onion.Dec}$ as follows:

```
 1: if $i = n$ and $c + 2 - \mathsf{critical} = c'$ then
 2:     parse $E[c'] = (\mathsf{ad}, \mathsf{ct}, k)$
 3:     if $(\mathsf{ad}_i, \mathsf{ct}_i) = (\mathsf{ad}, \mathsf{ct})$ then
 4:         $k_i \leftarrow k$
 5:     else
 6:         $\mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \to k_i$
 7:     end if
 8: else
 9:     $\mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \to k_i$
10: end if
```

It only affects $i = n$, i.e. the $\mathsf{SC.Dec}$ using the last receiving state, and $\mathsf{critical} = c + 2 - c'$, i.e. the $\mathsf{onion.Dec}$ using the critical state st_R^n associated to the c'^{th} update. Hence, this only affects the RATCH call receiving $\mathsf{upd}_{c'}$. By this change, the game simply bypasses the encryption of k_n in the transmission but makes the adversary still see a ciphertext which encrypts some junk k^*.

The game $\Gamma''_{b,P,\ell,c,c'}$ follows $\Gamma'_{b,P,\ell,c,c'}$ but makes an additional change in the $\mathsf{RATCH}(P, \mathsf{send}) \to \mathsf{upd}_{c'}$ call and the possible reception of $\mathsf{upd}_{c'}$ by $\overline{P}$. For this call, we change the line $\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt})$ in Step 3 of $\mathsf{onion.Enc}$ as follows:

```
 1: if this execution is to compute $\mathsf{upd}_{c'}$ then
 2:     pick a random $\mathsf{pt}'$ of same size as $\mathsf{pt}$
 3:     $\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt}')$
 4:     store $D[c'] \leftarrow (\mathsf{ct}_{n+1}, \mathsf{pt})$
 5: else
 6:     $\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt})$
 7: end if
```

Similarly, in a $\mathsf{onion.Dec}$ involving a critical state st_R^n associated to the c'^{th} update, we replace $\mathsf{pt} \leftarrow \mathsf{Sym.Dec}(k, \mathsf{ct}_{n+1})$ in Step 10 of $\mathsf{onion.Dec}$ as follows:

```
 1: parse $D[c'] = (\mathsf{ct}, \mathsf{pt}^*)$
 2: if $\mathsf{ct}_{n+1} = \mathsf{ct}$ and $\mathsf{critical} = c + 2 - c'$ then
 3:     $\mathsf{pt} \leftarrow \mathsf{pt}^*$
 4: else
 5:     $\mathsf{pt} \leftarrow \mathsf{Sym.Dec}(k, \mathsf{ct}_{n+1})$
 6: end if
```

By this change, the game simply bypasses the encryption of pt in the transmission but makes the adversary still see a ciphertext which encrypts some junk pt'. The main point is that the game no longer needs to encrypt the c'^{th} critical decryption key as encryption is bypassed.

Because the c'^{th} critical key needs no encryption and does not leak, it is only used to decrypt. Hence, the difference between $\Gamma''_{b,P,\ell,c,c'-1}$ and $\Gamma'_{b,P,\ell,c,c'}$ can be simulated by an $\mathsf{IND\text{-}CCA}$ game on PKC. Similarly, the difference between $\Gamma'_{b,P,\ell,c,c'}$ and $\Gamma''_{b,P,\ell,c,c'}$ can be simulated by an $\mathsf{IND\text{-}OTCCA}$ game on Sym. We have

$$\begin{aligned}\Pr[\Gamma_{b,P,\ell,c} \to 1] - \Pr[\Gamma'_{b,P,\ell,c,0} \to 1] &= 0\\ |\Pr[\Gamma''_{b,P,\ell,c,c'-1} \to 1] - \Pr[\Gamma'_{b,P,\ell,c,c'} \to 1]| &\leqslant \varepsilon\\ |\Pr[\Gamma'_{b,P,\ell,c,c'} \to 1] - \Pr[\Gamma''_{b,P,\ell,c,c'} \to 1]| &\leqslant \varepsilon'\end{aligned}$$

Hence,

$$|\Pr[\Gamma_{b,P,\ell,c} \to 1] - \Pr[\Gamma''_{b,P,\ell,c,c} \to 1]| \leqslant c\varepsilon + (c+1)\varepsilon'$$

In $\Gamma''_{b,P,\ell,c,c}$, the value of k_{test} is also replaced in the encryption. Hence, if it does not leak from $\mathsf{EXP}_{\mathsf{key}}$, it is actually never used. $\Gamma''_{1,P,\ell,c,c}$ reveals in TEST this never used value. $\Gamma''_{0,P,\ell,c,c}$ reveals in TEST another never used value. We deduce

$$\Pr[\Gamma''_{0,\mathsf{P},\ell,\mathsf{c},\mathsf{c}} \to 1] = \Pr[\Gamma''_{1,\mathsf{P},\ell,\mathsf{c},\mathsf{c}} \to 1]$$

Hence,

$$|\Pr[\Gamma_{0,\mathsf{P},\ell,\mathsf{c}} \to 1] - \Pr[\Gamma_{1,\mathsf{P},\ell,\mathsf{c}} \to 1]| \leqslant 2\mathsf{c}\varepsilon + 2(\mathsf{c}+1)\varepsilon'$$

We then use $\Pr[\Gamma_b \to 1] = \sum_{\mathsf{P},\ell,\mathsf{c}} \Pr[\Gamma_{b,\mathsf{P},\ell,\mathsf{c}} \to 1]$. Given P, each (ℓ, c) pair maps to a unique $\mathsf{upd}_{\mathsf{test}}$ sent by P. Hence, the number of $(\mathsf{P}, \ell, \mathsf{c})$ triplets is bounded by q. Since $\mathsf{c} \leqslant \mathsf{q}$, we obtain

$$|\Pr[\Gamma_0 \to 1] = \Pr[\Gamma_1 \to 1]| \leqslant 2\mathsf{q}^2\varepsilon + 2\mathsf{q}(\mathsf{q}+1)\varepsilon'$$

□

6 Conclusion

We studied the BARK protocol and its security. For security, we marked three important security objectives: the BARK protocol should be KIND-secure; the BARK protocol should be resistant to forgery attacks (FORGE-security), and the BARK protocol should not self-heal after impersonation (RECOVER-security). By relaxing the cleanness notion in KIND-security, we designed a protocol based on an IND-CCA-secure cryptosystem and a one-time signature scheme. We used neither random oracle nor key-update primitives.

Acknowledgements. We thank Joseph Jaeger for his valuable comments to the first version of this paper. We thank Paul Rösler for insightful discussions and comments. We also owe to Andrea Caforio whose implementation results contributed to support our design.

References

1. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *Advances in Cryptology – EUROCRYPT 2019 (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2019. Full version: `https://eprint.iacr.org/2018/1037.pdf`.
2. Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *Advances in Cryptology – CRYPTO 2017*, pages 619–650. Springer International Publishing, 2017.
3. Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.
4. Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, pages 453–474, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
5. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466, April 2017.
6. Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, June 2016.
7. David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 425–455, Cham, 2018. Springer International Publishing.
8. Yevgeniy Dodis, Michael J. Freedman, Stanislaw Jarecki, and Shabsi Walfish. Optimal signcryption from any trapdoor permutation. Available at: `https://eprint.iacr.org/2004/020.pdf`.
9. F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In *Advances in information and Computer Security – IWSEC 2019*, volume 11689 of *Lecture Notes in Computer Science*, pages 343–362. Springer, 2019. Full version: `https://eprint.iacr.org/2018/889.pdf`.

10. Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 519–548, Cham, 2017. Springer International Publishing.
11. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In *Advances in Cryptology – CRYPTO 2018 (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 33–62. Springer International Publishing, 2018. Full version: `https://eprint.iacr.org/2018/553.pdf`.
12. Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In *Advances in Cryptology – EUROCRYPT 2019 (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 159–188. Springer, 2019. Full version: `https://eprint.iacr.org/2018/954.pdf`.
13. Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Provable Security*, pages 1–16, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
14. Miyako Ohkubo, Koutarou Suzuki, and Shingo Kinoshita. Cryptographic approach to "privacy-friendly" tags. In *RFID Privacy Workshop*, 2003.
15. Miyako Ohkubo, Koutarou Suzuki, and Shingo Kinoshita. Efficient hash-chain based RFID privacy protection scheme. In *International Conference on Ubiquitous Computing (Ubicomp), Workshop Privacy: Current Status and Future Directions*, 2004.
16. Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In *Advances in Cryptology – CRYPTO 2018 (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 3–32. Springer International Publishing, 2018. Full version: `https://eprint.iacr.org/2018/296.pdf`.
17. Open Whisper Systems. Signal protocol library for Java/Android. GitHub repository `https://github.com/WhisperSystems/libsignal-protocol-java`, 2017.
18. Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, May 2015.
19. Serge Vaudenay. Adversarial correctness favors laziness. Presented at the CRYPTO 2018 Rump Session.
20. WhatsApp. Whatsapp encryption overview. Technical white paper, available at: `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`, 2016.

A Used Definitions

Function families and collision-resistant hash functions. A function family H defines a polynomially bounded algorithm $\mathsf{H.Gen}(1^\lambda)$ which generates a key hk (we may denote its length as $\mathsf{H.kl}$) and a deterministic algorithm $\mathsf{H.Eval}(\mathsf{hk}, m)$ which takes a key hk and a message m to produce a digest of fixed length (we may denote it by $\mathsf{H.ln}$). We will need a collision-resistant hash function H. It should be intractable, given a honestly generated hashing key hk, to find two different messages m and m' such that $\mathsf{H.Eval}(\mathsf{hk}, m) = \mathsf{H.Eval}(\mathsf{hk}, m')$.

Definition 25 (Collision-resistant hash function). *We say that a function family* H *is* $(\lambda, \mathsf{T}, \varepsilon)$*-collision resistant if for any adversary* $\mathcal{A}$ *limited to time complexity* T*, the probability to win is bounded by* ε*.*

```
1: $\mathsf{H.Gen}(1^\lambda) \xrightarrow{\$} \mathsf{hk}$
2: $\mathcal{A}(\mathsf{hk}) \xrightarrow{\$} (m_1, m_2)$
3: if $\mathsf{H.Eval}(\mathsf{hk}, m_1) = \mathsf{H.Eval}(\mathsf{hk}, m_2)$ and $m_1 \neq m_2$ then win
```

Signcryption. Our construction is based on signcryption. Actually, we do not use a strong signcryption scheme as defined by Dodis et al. [8] but rather a naive combination of signature and encryption. We only want that it encrypts and authenticates at the same time. We take the following definition for our naive signcryption scheme.

Definition 26 (Signcryption scheme). *A* signcryption scheme SC *consists of four polynomially bounded algorithms: two key generation algorithms* $\mathsf{Gen_S}(1^\lambda) \xrightarrow{\$} (\mathsf{sk_S}, \mathsf{pk_S})$*; and* $\mathsf{Gen_R}(1^\lambda) \xrightarrow{\$}$

$(\mathsf{sk_R}, \mathsf{pk_R})$*; an encryption algorithm* $\mathsf{Enc}(\mathsf{sk_S}, \mathsf{pk_R}, \mathsf{ad}, \mathsf{pt}) \xrightarrow{\$} \mathsf{ct}$*; a deterministic decryption algorithm* $\mathsf{Dec}(\mathsf{sk_R}, \mathsf{pk_S}, \mathsf{ad}, \mathsf{ct}) \to \mathsf{pt}$ *returning a plaintext or* $\perp$*. The correctness property is that for all* pt *and* ad*,*

$$\Pr[\mathsf{Dec}(\mathsf{sk_R}, \mathsf{pk_S}, \mathsf{ad}, \mathsf{Enc}(\mathsf{sk_S}, \mathsf{pk_R}, \mathsf{ad}, \mathsf{pt})) = \mathsf{pt}] = 1$$

when the keys are generated with Gen*.*

This notion comes with two security notions.

Definition 27 (EF-OTCPA)**.** *A signcryption scheme* $(\lambda, \mathsf{T}, \varepsilon)$*-resists to* existential forgeries under one-time chosen plaintext attacks (EF-OTCPA) *if for any adversary* $\mathcal{A}$ *limited to time complexity* T *playing the following game, the probability to win is bounded by* ε*.*

```
1: $\mathsf{Gen_S}(1^\lambda) \xrightarrow{\$} (\mathsf{sk_S}, \mathsf{pk_S})$
2: $\mathsf{Gen_R}(1^\lambda) \xrightarrow{\$} (\mathsf{sk_R}, \mathsf{pk_R})$
3: $\mathcal{A}(\mathsf{sk_R}, \mathsf{pk_S}, \mathsf{pk_R}) \xrightarrow{\$} (\mathsf{st}, \mathsf{ad}, \mathsf{pt})$
4: $\mathsf{Enc}(\mathsf{sk_S}, \mathsf{pk_R}, \mathsf{ad}, \mathsf{pt}) \xrightarrow{\$} \mathsf{ct}$
5: $\mathcal{A}(\mathsf{st}, \mathsf{ct}) \xrightarrow{\$} (\mathsf{ad}', \mathsf{ct}')$
6: if $(\mathsf{ad}, \mathsf{ct}) = (\mathsf{ad}', \mathsf{ct}')$ then abort
7: $\mathsf{Dec}(\mathsf{sk_R}, \mathsf{pk_S}, \mathsf{ad}', \mathsf{ct}') \to \mathsf{pt}'$
8: if $\mathsf{pt}' = \perp$ then abort
9: the adversary wins
```

Definition 28 (IND-CCA)**.** *A signcryption scheme is* $(\lambda, \mathsf{q}, \mathsf{T}, \varepsilon)$-IND-CCA-secure *if for any adversary* $\mathcal{A}$ *limited to* q *queries and time complexity* T*, playing the following game, the advantage* $\Pr[\mathsf{IND\text{-}CCA}_0^{\mathcal{A}} \xrightarrow{\$} 1] - \Pr[\mathsf{IND\text{-}CCA}_1^{\mathcal{A}} \xrightarrow{\$} 1]$ *is bounded by* ε*.*

```
Game $\mathsf{IND\text{-}CCA}_b^{\mathcal{A}}$
  1: $\mathsf{challenge} = \perp$
  2: $\mathsf{Gen_S}(1^\lambda) \xrightarrow{\$} (\mathsf{sk_S}, \mathsf{pk_S})$
  3: $\mathsf{Gen_R}(1^\lambda) \xrightarrow{\$} (\mathsf{sk_R}, \mathsf{pk_R})$
  4: $\mathcal{A}^{\mathsf{Ch},\mathsf{Dec}}(\mathsf{sk_S}, \mathsf{pk_S}, \mathsf{pk_R}) \xrightarrow{\$} \mathsf{b}'$
  5: return $\mathsf{b}'$

Oracle $\mathsf{Dec}(\mathsf{ad}, \mathsf{ct})$
  6: if $(\mathsf{ad}, \mathsf{ct}) = \mathsf{challenge}$ then abort
  7: $\mathsf{Dec}(\mathsf{sk_R}, \mathsf{pk_S}, \mathsf{ad}, \mathsf{ct}) \to \mathsf{pt}$
  8: return $\mathsf{pt}$

Oracle $\mathsf{Ch}(\mathsf{ad}, \mathsf{pt})$
  1: if $\mathsf{challenge} \neq \perp$ then abort
  2: if $\mathsf{b} = 0$ then replace $\mathsf{pt}$ by a random message
     of same length
  3: $\mathsf{Enc}(\mathsf{sk_S}, \mathsf{pk_R}, \mathsf{ad}, \mathsf{pt}) \xrightarrow{\$} \mathsf{ct}$
  4: $\mathsf{challenge} \leftarrow (\mathsf{ad}, \mathsf{ct})$
  5: return $\mathsf{ct}$
```

Public-key cryptosystem. We define a PKC.

Definition 29 (PKC scheme)**.** *A* public-key cryptosystem scheme PKC *consists of three polynomially bounded algorithms: one key generation algorithm* $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$*; an encryption algorithm* $\mathsf{Enc}(\mathsf{pk}, \mathsf{pt}) \xrightarrow{\$} \mathsf{ct}$*; a deterministic decryption algorithm* $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{pt}$ *returning a plaintext or* $\perp$*. The correctness property is that for all* pt*,*

$$\Pr[\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, \mathsf{pt})) = \mathsf{pt}] = 1$$

when the keys are generated with Gen*.*

Definition 30 (IND-CCA)**.** *A* PKC *is* $(\lambda, \mathsf{q}, \mathsf{T}, \varepsilon)$-IND-CCA-secure *if for any adversary* $\mathcal{A}$ *limited to* q *queries and time complexity* T*, playing the following game, the advantage* $\Pr[\mathsf{IND\text{-}CCA}_0^{\mathcal{A}} \xrightarrow{\$} 1] - \Pr[\mathsf{IND\text{-}CCA}_1^{\mathcal{A}} \xrightarrow{\$} 1]$ *is bounded by* ε*.*

```
Game $\mathsf{IND\text{-}CCA}_b^{\mathcal{A}}$
  1: $\mathsf{challenge} = \perp$
  2: $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$
  3: $\mathcal{A}^{\mathsf{Ch},\mathsf{Dec}}(\mathsf{pk}) \xrightarrow{\$} \mathsf{b}'$
  4: return $\mathsf{b}'$

Oracle $\mathsf{Dec}(\mathsf{ct})$
  5: if $\mathsf{ct} = \mathsf{challenge}$ then abort
  6: $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{pt}$
  7: return $\mathsf{pt}$

Oracle $\mathsf{Ch}(\mathsf{pt})$
  1: if $\mathsf{challenge} \neq \perp$ then abort
  2: if $\mathsf{b} = 0$ then replace $\mathsf{pt}$ by a random message
     of same length
  3: $\mathsf{Enc}(\mathsf{pk}, \mathsf{pt}) \xrightarrow{\$} \mathsf{ct}$
  4: $\mathsf{challenge} \leftarrow \mathsf{ct}$
  5: return $\mathsf{ct}$
```

Digital signature scheme. We define a DSS.

Definition 31 **(DSS scheme).** *A* digital signature scheme DSS *consists of three polynomially bounded algorithms: one key generation algorithm* $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$ *which generates a key pair; a signing algorithm* $\mathsf{Sign}(\mathsf{sk}, x) \xrightarrow{\$} \sigma$*; a deterministic verification algorithm* $\mathsf{Verify}(\mathsf{pk}, x, \sigma)$ *returning a bit. The correctness property is that for all* x*,*

$$\Pr[\mathsf{Verify}(\mathsf{pk}, x, \mathsf{Sign}(\mathsf{sk}, x)) = 1] = 1$$

when the keys are generated with Gen*.*

Definition 32 **($\mathsf{SEF\text{-}OTCMA}$).** *A* DSS $(\lambda, \mathsf{T}, \varepsilon)$*-resists to* strong existential forgeries under one-time chosen message attacks ($\mathsf{SEF\text{-}OTCMA}$) *if for any adversary $\mathcal{A}$ limited to time complexity* T *playing the following game, the probability to win is bounded by ε.*

```
1: $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$
2: $\mathcal{A}(\mathsf{pk}) \xrightarrow{\$} (\mathsf{st}, x)$
3: $\mathsf{Sign}(\mathsf{sk}, x) \xrightarrow{\$} \sigma$
4: $\mathcal{A}(\mathsf{st}, \sigma) \xrightarrow{\$} x', \sigma'$
5: if $(x, \sigma) = (x', \sigma')$ then abort
6: $\mathsf{Verify}(\mathsf{pk}, x', \sigma') \to b$
7: if $b = 0$ then abort
8: the adversary wins
```

One-time symmetric encryption. We use a symmetric encryption scheme Sym which defines a key length $\mathsf{Sym.kl}(\lambda)$, a plaintext domain $\mathsf{Sym}.\mathcal{D}(\lambda)$, and two polynomially bounded deterministic algorithms $\mathsf{Sym.Enc}$ and $\mathsf{Sym.Dec}$ satisfying

$$\mathsf{Sym.Dec}(\mathsf{k}, \mathsf{Sym.Enc}(\mathsf{k}, \mathsf{pt})) = \mathsf{pt}$$

for any key k in $\{0,1\}^{\mathsf{Sym.kl}}$ and any bitstring pt in $\mathsf{Sym}.\mathcal{D} \subseteq \{0,1\}^*$. It must satisfy one-time security.

Definition 33 **(One-time $\mathsf{IND\text{-}OTCCA}$).** *A symmetric encryption scheme is* $(\lambda, \mathsf{T}, \varepsilon)$*-*$\mathsf{IND\text{-}OTCCA}$*-*secure *if for any adversary $\mathcal{A}$ limited to time complexity* T*, playing the following game, the advantage* $\Pr[\mathsf{IND\text{-}OTCCA}_0^{\mathcal{A}} \xrightarrow{\$} 1] - \Pr[\mathsf{IND\text{-}OTCCA}_1^{\mathcal{A}} \xrightarrow{\$} 1]$ *is bounded by ε.*

```
Game $\mathsf{IND\text{-}OTCCA}_b^{\mathcal{A}}$
1: $\mathsf{challenge} = \perp$
2: pick $\mathsf{k}$ in $\{0,1\}^{\mathsf{Sym.kl}}$
3: $\mathcal{A}^{\mathsf{Ch},\mathsf{Dec}}() \xrightarrow{\$} b'$
4: return $b'$

Oracle $\mathsf{Dec}(\mathsf{ct})$
5: if $\mathsf{ct} = \mathsf{challenge}$ then abort
6: return $\mathsf{Sym.Dec}(\mathsf{k}, \mathsf{ct})$

Oracle $\mathsf{Ch}(\mathsf{pt})$
1: if $\mathsf{challenge} \neq \perp$ then abort
2: if $b = 0$ then replace $\mathsf{pt}$ by a random message
   of same length
3: $\mathsf{Sym.Enc}(\mathsf{k}, \mathsf{pt}) \to \mathsf{ct}$
4: $\mathsf{challenge} \leftarrow \mathsf{ct}$
5: return $\mathsf{ct}$
```

KEM. We finally define KEM.

Definition 34 **(KEM scheme).** *A* KEM scheme *consists of three polynomially bounded algorithms: a key pair generation* $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$*, an encapsulation algorithm* $\mathsf{Enc}(\mathsf{pk}) \xrightarrow{\$} (\mathsf{k}, \mathsf{ct})$*, and a decapsulation algorithm* $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{k}$*. It is correct if* $\Pr[\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = \mathsf{k}] = 1$ *when the keys are generated with* Gen *and* $\mathsf{Enc}(\mathsf{pk}) \to (\mathsf{k}, \mathsf{ct})$*.*

This notion comes with a security notion.

Definition 35 **($\mathsf{IND\text{-}CPA}$ security for KEM).** *A* KEM *scheme is* $(\lambda, \mathsf{T}, \varepsilon)$*-*$\mathsf{IND\text{-}CPA}$ *secure if for any adversary $\mathcal{A}$ limited to time complexity* T*, playing the following game, the advantage* $\Pr[\mathsf{IND\text{-}CPA}_0^{\mathcal{A}} \xrightarrow{\$} 1] - \Pr[\mathsf{IND\text{-}CPA}_1^{\mathcal{A}} \xrightarrow{\$} 1]$ *is bounded by ε.*
Game $\mathsf{IND\text{-}CPA}_b^{\mathcal{A}}$

```
1: $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$
2: $\mathsf{Enc}(\mathsf{pk}) \xrightarrow{\$} (\mathsf{k}, \mathsf{ct})$
3: if $b = 0$ then replace $\mathsf{k}$ by a random key of same length
4: $\mathcal{A}(\mathsf{pk}, \mathsf{ct}, \mathsf{k}) \xrightarrow{\$} b'$
5: return $b'$
```

B $\mathsf{C}_{\mathsf{forge}}^{\mathsf{P}_{\mathsf{test}}}$ Forbids More Than Necessary

Let us consider $\mathsf{SC.Enc}(\mathsf{sk_S}, \mathsf{pk_R}, \mathsf{pt}) = \mathsf{PKC.Enc}(\mathsf{pk_R}, \mathsf{pt})$ (which does not use $\mathsf{sk_S}/\mathsf{pk_S}$), where PKC is an IND-CCA-secure cryptosystem without the plaintext aware (PA) security. Hence, there exists an algorithm $\mathsf{C}(\mathsf{pk_R}; r) = \mathsf{ct}$ such that $(\mathsf{pk_R}, r, \mathsf{PKC.Dec}(\mathsf{sk_R}, \mathsf{ct}))$ and $(\mathsf{pk_R}, r, \mathsf{random})$ are indistinguishable.[7] We can show that the uniARK obtained from the onion of Fig. 7 has $(\mathsf{C_{leak}} \wedge \mathsf{C}_{\mathsf{forge}}^{\mathsf{P}_{\mathsf{test}}})$-KIND security. We can consider the following adversary:

```
1: $\mathsf{EXP_{st}}(S) \to \mathsf{pk_R}$
2: pick $r$; $\mathsf{C}(\mathsf{pk_R}; r) \to \mathsf{ct}$
3: $\mathsf{RATCH}(R, \mathsf{rec}, \mathsf{ct}) \to \mathsf{true}$
4: $\mathsf{TEST}(R) \to \mathsf{K}^*$
```

Due to the non-PA security, we do have privacy for the tested key. However, this adversary is ruled out by $\mathsf{C}_{\mathsf{forge}}^{\mathsf{P}_{\mathsf{test}}}$. Hence, this cleanness predicate does forbid more than necessary: we have KIND security for more attacks than allowed.

C Comparison with Other Protocols

C.1 Comparison with Bellare et al. [2]

Bellare et al. [2] consider uniARK (unidirectional BARK). They consider the KIND security defined by the game in Fig. 11 (with slightly adapted notations). This game has a single exposure oracle revealing the state st, the key k, and also the last used coins, but for the sender only. It also allows multiple TEST queries.

In the KIND game, the restricted flag is set when there is a trivial forgery. (It could be unset by receiving a genuine upd but we can ignore it for schemes with RECOVER security.) We can easily see that the cleanness notion required by the TEST queries corresponds to $\mathsf{C_{leak}} \wedge \mathsf{C}_{\mathsf{trivial\ forge}}^{\mathsf{P}_{\mathsf{test}}} \wedge \mathsf{C}_{\mathsf{noEXP}(R)}$.

```
Game $\mathsf{KIND}_b^{\mathcal{A}}$
 1: $i_s \leftarrow 0$; $i_r \leftarrow 0$
 2: $\mathsf{Init}(1^\lambda) \xrightarrow{\$} (\mathsf{st_S}, \mathsf{st_R}, z)$
 3: pick $k$
 4: $k_s \leftarrow k$ ; $k_R \leftarrow k$
 5: $b' \overset{\$}{\leftarrow} \mathcal{A}^{\mathsf{RATSEND,RATREC,EXP,CHSEND,CHREC}}(z)$
 6: return $b'$

Oracle EXP
 1: if $\mathsf{op}[i_s] =$ "ch" then return $\perp$
 2: $\mathsf{op}[i_s] =$ "exp"
 3: return $(r, \mathsf{st_S}, k_S)$

Oracle RATSEND
 1: pick $r$; $(\mathsf{st}'_S, \mathsf{upd_S}, k_S) \leftarrow \mathsf{Send}(\mathsf{st_S}; r)$
 2: $\mathsf{auth}[i_s] \leftarrow \mathsf{upd}$; $i_s \leftarrow i_s + 1$
 3: return $\mathsf{upd}$

Oracle RATREC(upd)
 1: $(\mathsf{acc}, \mathsf{st_R}, k_R) \leftarrow \mathsf{Receive}(\mathsf{st_R}, \mathsf{upd})$
 2: if not acc then return false
 3: if $\mathsf{op}[i_r] =$ "exp" then restricted $\leftarrow$ true
 4: if $\mathsf{upd} = \mathsf{auth}[i_r]$ then restricted $\leftarrow$ false
 5: $i_r \leftarrow i_r + 1$; return true

Oracle CHSEND
 1: if $\mathsf{op}[i_s] =$ "exp" then return $\perp$
 2: $\mathsf{op}[i_s] \leftarrow$ "ch"
 3: if $\mathsf{rkey}[i_s] = \perp$ then $\mathsf{rkey}[i_s] \overset{\$}{\leftarrow} \{0,1\}^{\mathsf{kl}}$
 4: if $b = 1$ then return $k_s$ else return $\mathsf{rkey}[i_s]$

Oracle CHREC
 1: if restricted then return $k_R$
 2: if $\mathsf{op}[i_r] =$ "exp" then return $\perp$
 3: $\mathsf{op}[i_r] \leftarrow$ "ch"
 4: if $\mathsf{rkey}[i_r] = \perp$ then $\mathsf{rkey}[i_r] \overset{\$}{\leftarrow} \{0,1\}^{\mathsf{kl}}$
 5: if $b = 1$ then return $k_R$ else return $\mathsf{rkey}[i_r]$
```

Fig. 11: The security game in Bellare et al. [2].

[7] As an example, we can start from an IND-CCA-secure $\mathsf{PKC_0}$ and add a ciphertext in the public key to define PKC. PKC.Gen: $\mathsf{PKC_0.Gen} \to (\mathsf{sk}, \mathsf{pk_0})$; pick x; $\mathsf{PKC_0.Enc}(\mathsf{pk}, x) \to y$; $\mathsf{pk} \leftarrow (\mathsf{pk_0}, y)$. Set Enc and Dec the same in $\mathsf{PKC_0}$ and PKC. Then $\mathsf{C}(\mathsf{pk}; r) = y$. PKC is also IND-CCA-secure and C has the required property.

C.2 Comparison with Poettering-Rösler [16]

Poettering and Rösler [16] have a different way to define correctness. Unfortunately, their definition is not complete as it takes schemes doing nothing as correct [19]. Indeed, the trivial scheme letting all states equal to $\perp$ and doing nothing is correct (and obviously secure).

The Poettering-Rösler construction allows to generate keys while treating "associated data" ad at the same time. However, their security notion does not seem to imply authentication of ad although their proposed protocol does. Like ours, this construction method starts from unidirectional, but their unidirectional scheme is not FORGE-secure as the state of the receiver allows to forge messages. Another important difference is that their scheme erases the state of the receiver as soon as the reception of an upd fails, instead of just rejecting it and waiting for a correct one. This makes their scheme vulnerable to denial-of-services attack.

The scheme construction uses no encryption. It also accumulates many keys in states, but instead of using an onion encryption, it does many parallel KEM and combines all generated keys as input to a random oracle. They feed the random oracle with the local history of communication as well (instead of using a collision-resistant hash function). It uses a KEM with a special additional property which could be realized with a hierarchical identity-based encryption (HIBE). Instead, we use a signcryption scheme. Finally, it uses the output of the random oracle to generate a new sk/pk pair. One of the participants erases sk and keeps pk while the other keeps sk. In our construction, one participant generates the pair, sends sk to the other, and erases it.

```
Game $\mathsf{KIND}_b^{\mathcal{A}}$
 1: for $P \in \{A, B\}$ do
 2:     $s_P, r_P \leftarrow 0$
 3:                                   ▷ number of sent and received messages
 4:     $e_P \leftarrow 0$
 5:                                   ▷ $e_P$: number of in-sync received messages
 6:     $\mathsf{EP}_P[\cdot] \leftarrow \perp$          ▷ $\mathsf{EP}_P[s]$: value of $e_P$ at the $s^{\text{th}}$ send
 7:     $E_P^{\vdash}, E_P^{\dashv} \leftarrow 0$
 8:                                   ▷ $E_P^{\vdash}$: number of in-sync sent acked by $\overline{P}$
 9:                                   ▷ $E_P^{\dashv} \leftarrow 0$: number of in-sync sent messages
10:     $\mathsf{adc}_P[\cdot] \leftarrow \perp$         ▷ list of sent $(\mathsf{ad}, \mathsf{upd})$
11:     $\mathsf{is}_P \leftarrow \mathsf{true}$          ▷ $\mathsf{is}_P$ says if $P$ is in-sync
12:     $k_P[\cdot] \leftarrow \perp$, $\mathsf{XP}_P \leftarrow \emptyset$   ▷ list of $s$ during $\mathsf{EXP}_{\mathsf{st}}(P)$
13:     $\mathsf{TR}_P \leftarrow \emptyset$             ▷ list of forbidden $\mathsf{TEST}(P, \ldots)$
14:     $\mathsf{CH}_P \leftarrow \emptyset$             ▷ list of $\mathsf{TEST}(P, \ldots)$ made
15: end for
16: $\mathsf{Init}(1^\lambda) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B)$
17: $b' \leftarrow \mathcal{A}^{\mathsf{RATSEND}, \mathsf{RATREC}, \mathsf{EXPst}, \mathsf{EXTkey}, \mathsf{TEST}}()$
18: if $\mathsf{TR}_A \cap \mathsf{CH}_A \neq \emptyset$ or $\mathsf{TR}_B \cap \mathsf{CH}_B \neq \emptyset$ then abort
19: if $\mathsf{TR}_B \cap \mathsf{CH}_B \neq \emptyset$ or $\mathsf{TR}_B \cap \mathsf{CH}_B \neq \emptyset$ then abort
20: return $b'$

Oracle $\mathsf{RATSEND}(P, \mathsf{ad})$
 1: if $S_P = \perp$ then abort
 2: $(\mathsf{st}_P, k, \mathsf{upd}) \leftarrow \mathsf{Send}(\mathsf{st}_P, \mathsf{ad})$
 3: if $\mathsf{is}_P$ then
 4:     $\mathsf{adc}_P[s_P] \leftarrow (\mathsf{ad}, \mathsf{upd})$
 5:     $\mathsf{EP}_P[s_P] \leftarrow e_P$
 6:     $E_P^{\dashv} \leftarrow E_P^{\dashv} + 1$
 7: end if
 8: $k_P[P, e_P, s_P] \leftarrow k$
 9: $s_P \leftarrow s_P + 1$
10: return $\mathsf{upd}$

Oracle $\mathsf{EXP}_{\mathsf{key}}(P, \mathsf{role}, e, s)$
 1: if $k_P[\mathsf{role}, e, s] \in \{\perp, \diamond\}$ then abort  ▷ not allowed if $k_P$ is not defined or
    is available from $k_{\overline{P}}$
 2: $k \leftarrow k_P[\mathsf{role}, e, s]$
 3: $k_P[\mathsf{role}, e, s] \leftarrow \diamond$
 4: return $k$
```

```
Oracle $\mathsf{RATREC}(P, \mathsf{ad}, \mathsf{upd})$
 1: if $S_P = \perp$ then abort
 2: if $\mathsf{is}_P \wedge \mathsf{adc}_{\overline{P}}[r_P] \neq (\mathsf{ad}, \mathsf{upd})$ then        ▷ first forgery
 3:     $\mathsf{is}_P \leftarrow \mathsf{false}$
 4:     if $r_P \in \mathsf{XP}_{\overline{P}}$ then                          ▷ trivial forgery
 5:         $\mathsf{TR}_P \leftarrow \mathsf{TR}_P \cup \{\mathsf{send}\} \times \{0, 1, \ldots\} \times \{s_P, s_P + 1, \ldots\}$
 6:         $\mathsf{TR}_P \leftarrow \mathsf{TR}_P \cup \{\mathsf{rec}\} \times \{0, 1, \ldots\} \times \{r_P, r_P + 1, \ldots\}$
 7:     end if
 8: end if
 9: if $\mathsf{is}_P$ then
10:     $E_P^{\vdash} \leftarrow \mathsf{EP}_{\overline{P}}[r_P]$
11:     $e_P \leftarrow e_P + 1$
12: end if
13: $(\mathsf{st}_P, k) \leftarrow \mathsf{Receive}(\mathsf{st}_P, \mathsf{ad}, \mathsf{upd})$
14: if $\mathsf{st}_P = \perp$ then return $\perp$
15: if $\mathsf{is}_P$ then $k \leftarrow \diamond$                 ▷ k is already available on $\overline{P}$
16: $k_P[\mathsf{rec}, E_P^{\vdash}, r_P] \leftarrow k$
17: $r_P \leftarrow r_P + 1$
18: return

Oracle $\mathsf{EXP}_{\mathsf{st}}(P)$
 1: $\mathsf{TR}_P \leftarrow \mathsf{TR}_P \cup \{\mathsf{rec}\} \times \{E_P^{\vdash}, \ldots, E_P^{\dashv}\} \times \{r_P, r_P + 1, \ldots\}$
 2: if $\mathsf{is}_P$ then
 3:     $\mathsf{XP}_P \leftarrow \mathsf{XP}_P \cup \{s_P\}$
 4:     $\mathsf{TR}_{\overline{P}} \leftarrow \mathsf{TR}_{\overline{P}} \cup \{\mathsf{send}\} \times \{E_P^{\vdash}, \ldots, E_P^{\dashv}\} \times \{r_P, r_P + 1, \ldots\}$
 5: end if
 6: return $\mathsf{st}_P$

Oracle $\mathsf{TEST}(P, \mathsf{role}, e, s)$
 1: if $k_P[\mathsf{role}, e, s] \in \{\perp, \diamond\}$ then abort
 2: $k \leftarrow k_P[\mathsf{role}, e, s]$
 3: if $b = 0$ then $k \leftarrow$ random
 4: $k_P[\mathsf{role}, e, s] \leftarrow \diamond$
 5: $\mathsf{CH}_P \leftarrow \mathsf{CH}_P \cup \{(\mathsf{role}, e, s)\}$
 6: return $k$
```

Fig. 12: The KIND game of Poettering-Rösler [16].

We recall the KIND game of Poettering-Rösler [16] in Fig. 12 (with slightly adapted notations). The adversary can make several TEST queries. Furthermore, TEST(P) queries are not necessarily on the last active k_P but can be on any previously generated k_P value. For this reason, TEST takes as input the index (a triplet (role, e, s)) of the tested key. This does not change the security notion.

The KIND game keeps a flag is_P stating if P is "in-sync". It means that P did not receive any forgery. This is a bit weaker than our matching status. However, assuming that a protocol is such that participants who received a forgery are no longer able to send valid messages to their counterparts, in-sync is equivalent to the matching status. As we can see, a key k_P produced during a reception is erased if P is in-sync, because it is available on the $\overline{\mathsf{P}}$ side from where it could be tested. This is one way to rule out some trivial attacks.

The other way is to mark a TEST as forbidden in a TR list. We can see in the KIND game (Step 2–8 in RATREC) that if P receives a trivial forgery (this is deduced by $r_\mathsf{P} \in \mathsf{XP}_{\overline{\mathsf{P}}}$), then no further TEST(P) is allowed. This means that $\mathsf{C}^{\mathsf{P}_\mathsf{test}}_{\mathsf{trivial\ forge}}$ is included in the cleanness predicate of this KIND game.

We can easily check that C_leak is included in the cleanness predicate. Hence, this KIND game looks equivalent to ours with cleanness predicate $\mathsf{C}_\mathsf{leak} \wedge \mathsf{C}^{\mathsf{P}_\mathsf{test}}_{\mathsf{trivial\ forge}}$.

This security notion does not seem to imply FORGE security.

C.3 Comparison with Jaeger-Stepanovs [11]

We recall the AEAC game of Jaeger-Stepanovs [11] in Fig. 13 (with slightly adapted notations). The RATSEND oracle implements the left-or-right challenge at the same time. Hence, the adversary can make several challenges. Additionally, the RATREC oracle implements a decrypt-or-silent oracle which leaks b in the case of a non-trivial forgery. (The oracle always decrypts after a trivial forgery and never decrypts if no forgery. Its behavior changes only in the presence of a non-trivial forgery and with no previous trivial forgery.) Hence, FORGE security is implied by AEAC security. A novelty here is that the adversary can get the *next* random coins to be used: z_P for sending or η_P for receiving. (Bellare et al. [2] allowed to expose the *last* coins.) This is managed by all instructions in gray in Fig. 13. Extracting these coins must be followed by the appropriate oracle query (enforced by the nextop state).

We cannot challenge P after P received a trivial forgery (due to the $\mathsf{restricted}_\mathsf{P}$ flag). Hence, we have some kind of $\mathsf{C}^{\mathsf{P}_\mathsf{test}}_{\mathsf{trivial\ forge}}$ condition for cleanness. Since C_leak is necessary, we can say that this model includes the $\mathsf{C}_\mathsf{leak} \wedge \mathsf{C}^{\mathsf{P}_\mathsf{test}}_{\mathsf{trivial\ forge}}$ predicate.

```
Game $\mathsf{AEAC}_b^{\mathcal{A}}$
1: for $P \in \{A, B\}$ do
2:     $s_P, r_P \leftarrow 0$
3:     $\mathsf{restricted}_P \leftarrow \mathsf{false}$                    ▷ P received a trivial forgery
4:     $\mathsf{forge}_P[\cdot] \leftarrow \mathsf{nontrivial}$  ▷ $\mathsf{forge}_P[r]$ says if $r^{th}$ reception could be a trivial
   forgery
5:     $\mathcal{X}_P \leftarrow 0$      ▷ challenge forbidden if $r_P < \mathcal{X}_P$ because some $\mathsf{EXP}_{\mathsf{st}}(\overline{P})$
   occurred
6:     pick $z_P, \eta_P$
7: end for
8: $(\mathsf{st}_A, \mathsf{st}_B) \leftarrow \mathsf{Init}(1^\lambda)$
9: $b' \leftarrow \mathcal{A}^{\mathsf{RATSEND},\mathsf{RATREC},\mathsf{EXPst}}()$
10: return $b'$

Oracle $\mathsf{RATSEND}(P, \mathsf{pt}_0, \mathsf{pt}_1, \mathsf{ad})$
1: if $\mathsf{nextop} \notin \{(P, \mathsf{send}), \perp\}$ then return $\perp$
2: if $|\mathsf{pt}_0| \neq |\mathsf{pt}_1|$ then return $\perp$
3: if $(r_P < \mathcal{X}_P \vee \mathsf{restricted}_P \vee \mathsf{ch}_P[s_P + 1] = \mathsf{forbidden}) \wedge \mathsf{pt}_0 \neq \mathsf{pt}_1$ then
   return $\perp$
4: $(\mathsf{st}_P, \mathsf{ct}) \leftarrow \mathsf{Send}(\mathsf{st}_P, \mathsf{ad}, \mathsf{pt}_b; z_P)$
5: $\mathsf{nextop} \leftarrow \perp$, $s_P \leftarrow s_P + 1$, pick $z_P$
6: if $\neg\mathsf{restricted}_P$ then $\mathsf{ctable}_{\overline{P}}[s_P] \leftarrow (\mathsf{ct}, \mathsf{ad})$
7:                                   ▷ register ct if P had no trivial forgery
8: if $\mathsf{pt}_0 \neq \mathsf{pt}_1$ then $\mathsf{ch}_P[s_P] \leftarrow \mathsf{done}$
9:                                    ▷ challenge was done for the $s^{th}$ send
10: return ct

Oracle $\mathsf{RATREC}(P, \mathsf{ct}, \mathsf{ad})$
1: if $\mathsf{nextop} \notin \{(P, \mathsf{rec}), \perp\}$ then return $\perp$
2: $(\mathsf{st}_P, \mathsf{pt}) \leftarrow \mathsf{Receive}(\mathsf{st}_P, \mathsf{ad}, \mathsf{ct}; \eta_P)$
3: $\mathsf{nextop} \leftarrow \perp$, pick $\eta_P$
4: if $\mathsf{pt} = \perp$ then return $\perp$
5: $r_P \leftarrow r_P + 1$
6: if $\mathsf{forge}_P[r_P] = \mathsf{trivial} \wedge (\mathsf{ct}, \mathsf{ad}) \neq \mathsf{ctable}_P[r_P]$ then $\mathsf{restricted}_P \leftarrow \mathsf{true}$     ▷
   trivial forgery
7: if $\mathsf{restricted}_P \vee (b = 0 \wedge (\mathsf{ct}, \mathsf{ad}) \neq \mathsf{ctable}_P[r_P])$ then return pt   ▷ return
   pt only after trivial forgeries
8:                           ▷ (b = 0 case) return pt for a non-trivial forgery
9: return $\perp$

Oracle $\mathsf{EXP}_{\mathsf{st}}(P, \mathsf{coins})$
1: if $\mathsf{nextop} \neq \perp$ then return $\perp$
2: if $\mathsf{restricted}_P$ then return $(\mathsf{st}_P, z_P, \eta_P)$
3: if $\exists i : r_P < i \leqslant s_{\overline{P}} \wedge \mathsf{ch}_{\overline{P}}[i] = \mathsf{done}$ then return $\perp$
4:                        ▷ challenge from $\overline{P}$ was done but not received yet
5: $\mathsf{forge}_{\overline{P}}[s_P + 1] \leftarrow \mathsf{trivial}$, $z, \eta \leftarrow \perp$, $\mathcal{X}_{\overline{P}} \leftarrow s_P + 1$
6: if $\mathsf{coins} = \mathsf{send}$ then
7:     $\mathsf{nextop} \leftarrow (P, \mathsf{send})$, $z \leftarrow z_P$, $\mathcal{X}_{\overline{P}} \leftarrow s_P + 2$
8:     $\mathsf{forge}_{\overline{P}}[s_P + 1] \leftarrow \mathsf{trivial}$, $\mathsf{ch}_P[s_P + 2] \leftarrow \mathsf{forbidden}$
9: else if $\mathsf{coins} = \mathsf{rec}$ then
10:     $\mathsf{nextop} \leftarrow (P, \mathsf{rec})$, $\eta \leftarrow \eta_P$
11: end if
12: return $(\mathsf{st}_P, z, \eta)$
```

Fig. 13: The AEAC game of Jaeger-Stepanovs [11].