Beyond Security and Efficiency: On-Demand Ratcheting with Security Awareness

Andrea Caforio[1(✉)], F. Betül Durak[2], and Serge Vaudenay[1]

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
andrea.caforio@epfl.ch

[2] Robert Bosch LLC - Research and Technology Center, Pittsburgh, PA, USA

Abstract. Secure asynchronous two-party communication applies ratcheting to strengthen privacy, in the presence of internal state exposures. Security with ratcheting is provided in two forms: forward security and post-compromise security. There have been several such secure protocols proposed in the last few years. However, they come with a high cost.

In this paper, we propose two generic constructions with favorable properties. Concretely, our first construction achieves *security awareness*. It allows users to detect non-persistent active attacks, to determine which messages are not safe given a potential leakage pattern, and to acknowledge for deliveries.

In our second construction, we define a hybrid system formed by combining two protocols: typically, a weakly secure "light" protocol and a strongly secure "heavy" protocol. The design goals of our hybrid construction are, first, to let the sender decide which one to use in order to obtain an efficient protocol with *ratchet on demand*; and second, to restore the communication between honest participants in the case of a message loss or an active attack.

We can apply our generic constructions to *any* existing protocol.

1 Introduction

In recent messaging applications, protocols are secured with end-to-end encryption to enable secure communication services for their users. Besides security, there are many other characteristics of communication systems. The nature of two-party protocols is that it is *asynchronous*: the messages should be transmitted regardless of the counterpart being online; the protocols do not have any control over the time that participants send messages; and, the participants change their roles as a *sender* or a *receiver* arbitrarily.

Many deployed systems are built with some sort of security guarantees. However, they often struggle with security vulnerabilities due to the internal state compromises that occur through *exposures* of participants. In order to prevent the attacker from decrypting past communication after an exposure, a state

© International Association for Cryptologic Research 2021
J. A. Garay (Ed.): PKC 2021, LNCS 12711, pp. 649–677, 2021.
https://doi.org/10.1007/978-3-030-75248-4_23

update procedure is applied. Ideally, such updates are done through one-way functions which delete the old states and generate new ones. This guarantees *forward secrecy*. Additionally, to further prevent the attacker from decrypting future communication, *ratcheting* is used. This adds some source of randomness in every state update to obtain what is called *future secrecy*, or *backward secrecy*, or *post-compromise security*, or even *self-healing*.

Formal definitions of ratcheting security given have been recently studied, by Bellare et al. [2], followed by many others subsequent studies [1,7–10]. Some of these schemes are key-exchange protocols while others are secure messaging. Since secure ratcheted messaging boils down to secure key exchange, we consider these works as equivalent.

Previous Work. Early ratcheting protocols were suggested in Off-the-Record (OTR) and then Signal [3,11]. The security of Signal was studied by Cohn-Gordon et al. [5]. Unger et al. [12] surveyed many ratcheting techniques. Alwen et al. [1] formalized the concept of "double ratcheting" from Signal.

Cohn-Gordon et al. [6] proposed a ratcheted protocol at CSF 2016 but requiring synchronized roles. Bellare et al. [2] proposed another protocol at CRYPTO 2017, but unidirectional and without forward secrecy. Poettering and Rösler (PR) [10] designed a protocol with "*optimal*" security (in the sense that we know no better security so far), but using a random oracle, and heavy algorithms such as hierarchical identity-based encryption (HIBE). Yet, their protocol does not guarantee security against compromised random coins. Jaeger and Stepanovs (JS) [8] proposed a similar protocol with security against compromised random coins: with random coin leakage *before* usage. Their protocol also requires HIBE and a random oracle.

Durak and Vaudenay (DV) [7] proposed a protocol with slightly lower security[1] but relying on neither HIBE nor random oracles. They rely on a public-key cryptosystem, a digital signature scheme, a one-time symmetric encryption scheme, and a collision-resistant hash function. They further show that a unidirectional scheme with post-compromise security implies public-key cryptography, which obviates any hope of having a fully secure protocol solely based on symmetric cryptography. At EUROCRYPT 2019, Jost, Maurer, and Mularczyk (JMM) [9] proposed concurrently and independently a protocol with security between optimal security and the security of the DV protocol.[2] They achieve it even with random coin leakage *after* usage. Contrarily to other protocols achieving security with corrupted random coins, in their protocol, random coin leakage does not necessarily imply revealing part of the state of the participant. In the same conference, Alwen, Coretti, and Dodis [1] proposed two other ratcheting protocols denoted as ACD and ACD-PK with security against adversarially *chosen* random coins and *immediate decryption*. Namely, messages can be decrypted even though some previous messages have not been received yet. The ACD-PK protocol offers a good level of security, although having immediate decryption

[1] More precisely, the security is called "*sub-optimal*" [7].

[2] They call this security level "*near-optimal*" [9].

may lower it a bit as it will be discussed shortly. On the other hand, during a phase when the direction of communication does not change, the ACD protocol is fully based on symmetric cryptography, hence has lower security (in particular, no post-compromise security in this period). However, it is much more efficient. Following the authors of ACD, we consider Signal and ACD as equivalent.

We summarize these results in Table 1. The first four rows are based on DV [7, Table 1]. The other rows of the table will be discussed shortly.

Recently, Yan and Vaudenay [13] proposed Encrypt-then-Hash (EtH), a simple, natural, and extremely efficient ratchet protocol based on symmetric cryptography only, which provides forward secrecy but not post-compromise security. In short, it replaces the encryption key by its hash after every encryption or decryption, and needs one key for each direction of communication.

We are mostly interested in the DV model [7]. It gives a simple description of the KIND security and FORGE security. The former deals with key indistinguishability where the generated keys are indistinguishable from random strings and the latter states that update messages for ratcheted key exchange are unforgeable. Additionally, they present the notion of RECOVER-security which guarantees that participants can no longer accept messages from their counterpart after they receive a forged message. Even though FORGE security avoids nontrivial forgeries, there are still (unavoidable) trivial forgeries. They occur when the state of a participant is exposed and the adversary decides to impersonate him. With RECOVER security, when an adversary impersonates someone (say Bob), the impersonated participant is out and can no longer communicate with the counterpart (say Alice). It does not mean to bother participants but rather work for their benefit. Indeed, this security notion guarantees that the attack is eventually detected by Bob if he is still alive. If the protocol has a way to resume secure communication based on an explicit action from the users, this property is particularly appealing.

What makes the DV model simple is that all technicalities are hidden in a *cleanness* notion which eliminates trivial attack strategies. The adversary can only win when the attack scenario trace is "clean". This model makes it easy to consider several cleanness notions, specifically for hybrid protocols. The difficulty is perhaps to provide an exhaustive list of criteria for attacks to be clean.

Our Contributions. We start with formally and explicitly defining a notion of *security awareness* in which the users detect active attacks by realizing they can no longer communicate; users can be confident that nothing in the protocol can compromise the confidentiality of an acknowledged message if it did not leak before; and users can deduce from an incoming message which of the messages they sent have been delivered when the incoming message was formed.

More concretely, we elaborate on the RECOVER security to offer optimal security awareness. We start by defining a new notion called s-RECOVER. We make sure that not only is a receiver of a forgery no longer able to receive genuine messages via r-RECOVER-security but he can no longer send a message to his counterpart either via s-RECOVER-security. The r-RECOVER security is equal to RECOVER security of the DV protocol. Both r-RECOVER and s-RECOVER

notions imply that reception of a genuine message offers a strong guarantee of having no forgery in the past: after an active attack ended, participants realize they can no longer communicate. Our security-awareness notion makes also explicit that the receiver of a message can deduce (in absence of a forgery) which of his messages have been seen by his counterpart (which we call an *acknowledgment extractor*). Hence, each sent message implicitly carries an acknowledgment for all received messages. Finally, what we want from the history of receive/send messages and exposures of a participant is the ability to deduce which message remains private (or "clean"). We call it a *cleanness extractor*.

Then, we give another *generic* construction to compose "any" two protocols with different security levels to allow a sender to select which security level to use. By composing a strongly secure protocol (such as PR, JS, JMM, DV) with a lighter and weakly secure one (such as EtH [13], which is solely based on symmetric cryptography), we obtain the notion of *ratchet on-demand*. When the ratcheting becomes infrequent, we obtain the excellent software performances of EtH as we will show in our implementation results. Hybrid constructions already exist, like Signal/ACD. However, they offer no control on the choice of the protocol to be used. Instead, they ratchet if (and only if) the direction of communication alternates.

We find that there would be an advantage to offer more fine grained flexibility. The decision to ratchet or not could of course be made by the end user or rather be triggered by the application at an upper layer, based on a security policy. For instance, it could make sense to ratchet on a smartphone for every new message following bringing back the app to foreground, or to ratchet no more than once an hour.

Another interesting outcome of our hybrid system is that we can form our hybrid system with *two identical* protocols: an upper one and a lower one. The lower protocol is used to communicate the messages and the upper protocol is used to control the lower protocol: to setup or to reset it. With this hybrid structure with identical protocols, we can repair broken communication in the case of a message loss or active attacks. As far as we observe, the complexity of the hybrid system is the same as the complexity of the underlying protocol. Since our security-aware property breaks communication in the case of an active attack, this repairing construction is a nice additional tool.

Last but not least, we implemented the many existing protocols: PR, JS, DV, JMM, ACD, ACD-PK, together with EtH. We observe that EtH is the fastest one. This is not surprising for all protocols which heavily use public-key cryptography, but it is surprising for ACD. Our goal is to offer a high level of security with the performances of EtH. We reach it with on-demand ratcheting when the participant demands healing scarcely.

Finally, we conclude that security awareness can be added on top of an existing protocol (even a hybrid one) in a generic way to strengthen security. We propose this generic strengthening (called chain) of protocol to obtain r-RECOVER and s-RECOVER security on the top of any protocol. As an example, we apply

it on the ratchet-on-demand hybrid protocol composed with DV and EtH and obtain our final protocol.

We provide a comparison of all the protocols with r-RECOVER-security, s-RECOVER-security, acknowledgment extractor and cleanness extractor in Table 1. Note that this table is made to help both the authors and the readers to have a fair understanding of what specified properties each protocol has or not. We stress that "any" protocol could form a hybrid system to provide ratchet-on-demand and repairing a broken communication in the case of message loss or active attacks. The protocol which is shown in the last column is the case where we chose to use DV and EtH to construct our hybrid system.

Table 1. Comparison of Several Protocols with our protocol chain(hybrid($\mathsf{ARCAD}_{\mathsf{DV}}$, EtH)) from Corollary 29 in Sect. 3.3: security level; worst case complexity for exchanging n messages; types of coin-leakage security; plain model (i.e. no random oracle); PKC or less (i.e. no HIBE). DV and $\mathsf{ARCAD}_{\mathsf{DV}}$ have identical characteristics. $\mathsf{ARCAD}_{\mathsf{DV}}$ is based on DV and described in Appendix B. The terms "optimal", "near-optimal", and "sub-optimal" from Durak-Vaudenay [7] are mentioned on p. 2. "Pragmatic" degrades a bit security to offer on-demand ratcheting. "id-optimal" is optimal among protocols with immediate decryption.

	PR [10]	JS [8]	JMM [9]	DV [7]	ACD-PK [1]	Ours
Security	Optimal	Optimal	Near-optimal	Sub-optimal	Id-optimal	Pragmatic
Worst case complexity	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Coins leakage resilience	No	Pre-send	Post-send	No	Chosen coins	No
Plain model (no ROM)	No	No	No	Yes	Yes	Yes
PKC or less	No	No	Yes	Yes	Yes	Yes
Immediate decryption	No	No	No	No	Yes	No
r-RECOVER security	No	Yes	No	Yes	No	Yes
s-RECOVER security	No	Yes	No	No	No	Yes
ack. extractor	Yes	Yes	Yes	Yes	No	Yes
Cleanness extractor	Yes	Yes	Yes	Yes	Yes	Yes
Category	BARK	ARCAD	ARCAD	BARK	ARCAD	ARCAD

To summarize, our contributions are:

- we formally define the notion of security awareness, construct a generic protocol strengthening called chain, and prove its security;
- we define the notion of on-demand ratcheting, construct a generic hybrid protocol called hybrid, define and prove its security;
- we implement PR, JS, DV, JMM, ACD, ACD-PK, and EtH protocols in order to clearly compare their performances.

Notation. We have two participants named Alice (A) and Bob (B). Whenever we talk about either one of the participants, we represent it as P, then $\overline{\mathsf{P}}$ refers to P's counterpart. We have two roles send and rec for sender and receiver respectively. We define $\overline{\mathsf{send}} = \mathsf{rec}$ and $\overline{\mathsf{rec}} = \mathsf{send}$. When the communication is unidirectional, the participants are called the *sender* S and the *receiver* R.

Structure of the Paper. In Sect. 2, we revisit the preliminary notions from Durak-Vaudenay [7] and Alwen-Coretti-Dodis [1]. They all are essential to be able to follow our results. In Sect. 3, we define a new notion named security awareness and build a protocol with regard to the notion. In Sect. 4, we define a new protocol called on-demand ratcheting with better performance than state-of-the-art. Finally, in Appendix A, we present our implementation results with the figures comparing various protocols. Appendix B presents $\mathsf{ARCAD}_{\mathsf{DV}}$: the DV protocol in a simplified form and in the frame of ARCAD.

2 Preliminaries

2.1 ARCAD Definition and Security

In this section, we recall the DV model [7] and we slightly adapt it to define asynchronous ratcheted communication with additional data denoted as ARCAD. That is, we consider message encryption instead of key agreement (BARK: bidirectional asynchronous ratcheted key agreement). The difference between BARK and ARCAD is the same as the difference between KEM and cryptosystems: pt is input to Send instead of output of Send. Additionally, we treat associated data ad to authenticate. Like DV [7][3], we adopt asymptotic security rather than exact security, for more readability. Adversaries and algorithms are probabilistic polynomially bounded (PPT) in terms of a parameter λ.

As we slightly change our direction from key exchange to encryption, we feel that it is essential to redefine the set of definitions from BARK for ARCAD. In this section, some of the definitions are marked with the reference [7]. It means that these definitions are unchanged except for possible necessary notation changes. The other definitions are straightforward adaptations to fit ARCAD. We try not to overload this section by redefining already existing terminology, hence, we let less essential definitions in the full version [4].

Definition 1 (ARCAD). *An* asynchronous ratcheted communication with additional data (ARCAD) *consists of the following PPT algorithms:*

- $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$*: This defines the common public parameters* pp*.*
- $\mathsf{Gen}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{sk}, \mathsf{pk})$*: This generates the secret key* sk *and the public key* pk *of a participant.*
- $\mathsf{Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_\mathsf{P}, \mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{P}) \rightarrow \mathsf{st}_\mathsf{P}$*: This sets up the initial state* st_P *of* P *given his secret key, and the public key of his counterpart.*
- $\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt}) \xrightarrow{\$} (\mathsf{st}'_\mathsf{P}, \mathsf{ct})$*: it takes as input a plaintext* pt *and some associated data* ad *and produces a ciphertext* ct *along with an updated state* st'_P*.*
- $\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct}) \rightarrow (\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{pt})$*: it takes as input a ciphertext* ct *and some associated data* ad *and produces a plaintext* pt *with an updated state* st'_P *together with a flag* acc*.*[4]

[3] Proceedings version.

[4] In our work, we assume that $\mathsf{acc} = \mathsf{false}$ implies that $\mathsf{st}'_\mathsf{P} = \mathsf{st}_\mathsf{P}$ and $\mathsf{pt} = \perp$, i.e. the state is not updated when the reception fails. Other authors assume that $\mathsf{st}'_\mathsf{P} = \mathsf{pt} = \perp$, i.e. no further reception can be done.

An additional $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \rightarrow (\mathsf{st}_A, \mathsf{st}_B, z)$ *algorithm, which returns the initial states of* A *and* B *as well as public information* z*, is defined as follows:*

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

Initall is defined for convenience as an initialization procedure for all games. None of our security games actually cares about how Initall is made from Gen and Init. This is nice because there is little to change to define a notion of "symmetric-cryptography-based ARCAD" with a slight abuse of definition: we only need to define Initall. This approach was already adopted for EtH [13] which was proven as a "secure ARCAD" in this way.

For all global variables v in the game such as $\mathsf{received}^P_{\mathsf{ct}}$, st_P, or ct_P (which appear in Fig. 1 and Fig. 2, for instance), we denote the value of v at time t by $v(t)$. The notion of *time* is participant-specific. It refers to the number of elementary operations he has done. We assume neither synchronization nor central clock. Time for two different participants can only be compared when they are run non-concurrently by an adversary in a game.

Definition 2 (Correctness of ARCAD). *Consider the correctness game given on Fig. 1.*[5] *We say that an* ARCAD *protocol is* correct *if for all sequence* sched *of tuples of the form* $(P, \text{"send"}, \mathsf{ad}, \mathsf{pt})$ *or* $(P, \text{"rec"})$*, the game never returns 1. Namely,*

- *at each stage, for each* P*,* $\mathsf{received}^P_{\mathsf{pt}}$ *is prefix of* $\mathsf{sent}^{\overline{P}}_{\mathsf{pt}}$[6]*;*
- *each* $\mathsf{RATCH}(P, \text{"rec"})$ *call returns* $\mathsf{acc} = \mathsf{true}$.

We note that $\mathsf{RATCH}(P, \text{"rec"}, \mathsf{ad}, \mathsf{ct})$ ignores messages when decryption fails. For this reason, when we say that a participant P "receives" a message, we may implicitly mean that the message was accepted. More precisely, it means that decryption succeeded and RATCH returned $\mathsf{acc} = \mathsf{true}$.

In addition to the RATCH oracle (in Fig. 1) which is used to ratchet (either to send or to receive), we define several other oracles (in Fig. 2): $\mathsf{EXP}_{\mathsf{st}}$ to obtain the state of a participant; $\mathsf{EXP}_{\mathsf{pt}}$ to obtain the last received message pt; CHALLENGE to send either the plaintext or a random string. All those oracles are used without change throughout all security notions in this paper.

Definition 3 (Matching status [7]). *We say that* P *is in a* matching status *at time* t *for* P *if*

[5] We use the programming technique of "function overloading" to define the RATCH oracle: there are two definitions depending on whether the second input is "rec" or "send".

[6] By saying that $\mathsf{received}^P_{\mathsf{pt}}$ is prefix of $\mathsf{sent}^{\overline{P}}_{\mathsf{pt}}$, we mean that $\mathsf{sent}^{\overline{P}}_{\mathsf{pt}}$ is the concatenation of $\mathsf{received}^P_{\mathsf{pt}}$ with a (possible empty) list of $(\mathsf{ad}, \mathsf{pt})$ pairs.

Oracle $\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, \mathsf{ad}, \mathsf{ct})$
1: $\mathsf{ct}_\mathsf{P} \leftarrow \mathsf{ct}$
2: $\mathsf{ad}_\mathsf{P} \leftarrow \mathsf{ad}$
3: $(\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{pt}_\mathsf{P}) \leftarrow \mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}_\mathsf{P}, \mathsf{ct}_\mathsf{P})$
4: **if** acc **then**
5: $\quad \mathsf{st}_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
6: $\quad$ append $(\mathsf{ad}_\mathsf{P}, \mathsf{pt}_\mathsf{P})$ to $\mathsf{received}^\mathsf{P}_\mathsf{pt}$
7: $\quad$ append $(\mathsf{ad}_\mathsf{P}, \mathsf{ct}_\mathsf{P})$ to $\mathsf{received}^\mathsf{P}_\mathsf{ct}$
8: **end if**
9: **return** acc

Oracle $\mathsf{RATCH}(\mathsf{P}, \text{“send”}, \mathsf{ad}, \mathsf{pt})$
10: $\mathsf{pt}_\mathsf{P} \leftarrow \mathsf{pt}$
11: $\mathsf{ad}_\mathsf{P} \leftarrow \mathsf{ad}$
12: $(\mathsf{st}'_\mathsf{P}, \mathsf{ct}_\mathsf{P}) \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}_\mathsf{P}, \mathsf{pt}_\mathsf{P})$
13: $\mathsf{st}_\mathsf{P} \leftarrow \mathsf{st}'_\mathsf{P}$
14: append $(\mathsf{ad}_\mathsf{P}, \mathsf{pt}_\mathsf{P})$ to $\mathsf{sent}^\mathsf{P}_\mathsf{pt}$
15: append $(\mathsf{ad}_\mathsf{P}, \mathsf{ct}_\mathsf{P})$ to $\mathsf{sent}^\mathsf{P}_\mathsf{ct}$
16: **return** ct_P

Game Correctness(sched)
1: set all sent^*_* and $\mathsf{received}^*_*$ 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 $\mathsf{incoming}_A, \mathsf{incoming}_B \leftarrow \emptyset$
5: $i \leftarrow 0$
6: **loop**
7: $\quad i \leftarrow i + 1$
8: $\quad$ **if** sched_i of form $(\mathsf{P}, \text{“rec”})$ **then**
9: $\quad\quad$ **if** $\mathsf{incoming}_\mathsf{P}$ is empty **then return** 0
10: $\quad\quad$ pull $(\mathsf{ad}, \mathsf{ct})$ from $\mathsf{incoming}_\mathsf{P}$
11: $\quad\quad \mathsf{acc} \leftarrow \mathsf{RATCH}(\mathsf{P}, \text{“rec”}, \mathsf{ad}, \mathsf{ct})$
12: $\quad\quad$ **if** $\mathsf{acc} = \mathsf{false}$ **then return** 1
13: $\quad$ **else**
14: $\quad\quad$ parse $\mathsf{sched}_i = (\mathsf{P}, \text{“send”}, \mathsf{ad}, \mathsf{pt})$
15: $\quad\quad \mathsf{ct} \leftarrow \mathsf{RATCH}(\mathsf{P}, \text{“send”}, \mathsf{ad}, \mathsf{pt})$
16: $\quad\quad$ push $(\mathsf{ad}, \mathsf{ct})$ to $\mathsf{incoming}_{\overline{\mathsf{P}}}$
17: $\quad$ **end if**
18: $\quad$ **if** $\mathsf{received}^A_\mathsf{pt}$ not prefix of $\mathsf{sent}^B_\mathsf{pt}$ **then return** 1
19: $\quad$ **if** $\mathsf{received}^B_\mathsf{pt}$ not prefix of $\mathsf{sent}^A_\mathsf{pt}$ **then return** 1
20: **end loop**

Fig. 1. The Correctness Game of ARCAD Protocol.

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

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

Intuitively, P is in a matching status at a given time if his state is not influenced by an active attack (i.e. message injection/modification/erasure/replay).

Definition 4 (Forgery). *Given a participant* P *in a game, we say that* $(\mathsf{ad}, \mathsf{ct}) \in \mathsf{received}^\mathsf{P}_\mathsf{ct}$ *is a forgery if at the moment of the game just before* P *received* $(\mathsf{ad}, \mathsf{ct})$, P *was in a matching status, but no longer after receiving* $(\mathsf{ad}, \mathsf{ct})$.

Definition 5 (Trivial forgery). *Let* $(\mathsf{ad}, \mathsf{ct})$ *be a forgery received by* P. *At the time* t *just before the* $\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, \mathsf{ad}, \mathsf{ct})$ *call,* P *was in a matching status. We assume that time* t *for* P *originates from time* $\bar{\mathsf{t}}$ *for* $\overline{\mathsf{P}}$. *If there is an* $\mathsf{EXP}_\mathsf{st}(\overline{\mathsf{P}})$ *call between time* $\bar{\mathsf{t}}$ *for* $\overline{\mathsf{P}}$ *and the next* $\mathsf{RATCH}(\overline{\mathsf{P}}, \text{“send”}, ., .)$ *call (or just after time* $\bar{\mathsf{t}}$ *is there is no further* $\mathsf{RATCH}(\overline{\mathsf{P}}, \text{“send”}, ., .)$ *call), we say that* $(\mathsf{ad}, \mathsf{ct})$ *is a* trivial forgery.

We give a brief description of the DV security notions [7] as follows.
FORGE-security: It makes sure that there is no forgery, except trivial ones.

$\mathsf{r\text{-}RECOVER}$**-security**[7]**:** If an adversary manages to forge (trivially or not) a message to one of the participants, then this participant can no longer accept genuine messages from his counterpart.

$\mathsf{PREDICT}$**-security:** The adversary cannot guess the value ct which will be output from the Send algorithm.

KIND**-security:** We omit this security notion which is specific to key exchange. Instead, we consider $\mathsf{IND\text{-}CCA}$**-security** in a real-or-random style.

We define the ratcheting security with $\mathsf{IND\text{-}CCA}$ notion. Before defining it, we like to introduce a predicate called $\mathsf{C_{clean}}$ as $\mathsf{IND\text{-}CCA}$ is relative to this predicate. The purpose of $\mathsf{C_{clean}}$ is to discard trivial attacks. Somehow, the technicality of the security notion is hidden in this cleanness notion. An "optimal" cleanness predicate discards only trivial attacks but other predicates may discard more and allow to have more efficient protocols [7].

More precisely, for "clean" cases, a security property must be guaranteed. A "trivial" attack (i.e. an attack that no protocol can avoid) implies a non-clean case. If the cleanness notion is tight, this is an equivalence.

In the full version [4] we recall the most useful cleanness predicates. In short, $\mathsf{C_{leak}} \wedge \mathsf{C^{A,B}_{trivial\ forge}}$ corresponds to the DV-cleanness notion for post-compromise security ("sub-optimal") and $\mathsf{C_{sym}}$ is the weaker cleanness notion for forward secrecy only which is adapted to symmetric cryptographic schemes.

Game $\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{b,\mathsf{C_{clean}}}(1^\lambda)$	Oracle $\mathsf{CHALLENGE}(\mathsf{P},\mathsf{ad},\mathsf{pt})$
1: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$	1: **if** $t_{\mathsf{test}} \neq \perp$ **then return** $\perp$
2: $\mathsf{Initall}(1^\lambda,\mathsf{pp}) \xrightarrow{\$} (\mathsf{st_A},\mathsf{st_B},z)$	2: **if** $b = 0$ **then**
3: set all sent^*_* and $\mathsf{received}^*_*$ variables to $\emptyset$	3: replace pt by a random string of same length
4: set t_{test} to $\perp$	4: **end if**
5: $b' \leftarrow \mathcal{A}^{\mathsf{RATCH},\mathsf{EXP_{st}},\mathsf{EXP_{pt}},\mathsf{CHALLENGE}}(z)$	5: $\mathsf{ct} \leftarrow \mathsf{RATCH}(\mathsf{P},\text{"send"},\mathsf{ad},\mathsf{pt})$
6: **if** $\neg\mathsf{C_{clean}}$ **then return** $\perp$	6: $(t,\mathsf{P},\mathsf{ad},\mathsf{pt},\mathsf{ct})_{\mathsf{test}} \leftarrow (\mathsf{time_P},\mathsf{P},\mathsf{ad},\mathsf{pt},\mathsf{ct})$
7: **return** b'	7: **return** ct
Oracle $\mathsf{EXP_{st}}(\mathsf{P})$	Oracle $\mathsf{EXP_{pt}}(\mathsf{P})$
1: **return** $\mathsf{st_P}$	1: **return** $\mathsf{pt_P}$

Fig. 2. $\mathsf{IND\text{-}CCA}$ Game. (Oracle RATCH is defined in Fig. 1)

Definition 6 **($\mathsf{C_{clean}}$-$\mathsf{IND\text{-}CCA}$ security).** *Let $\mathsf{C_{clean}}$ be a cleanness predicate. We consider the $\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{b,\mathsf{C_{clean}}}$ game of Fig. 2. We say that the* ARCAD *is* $\mathsf{C_{clean}}$-$\mathsf{IND\text{-}CCA}$-secure *if for any PPT adversary, the advantage*

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

of $\mathcal{A}$ in $\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{b,\mathsf{C_{clean}}}$ security game is negligible.

[7] It is called $\mathsf{RECOVER}$-security in DV [7]. We call it $\mathsf{r\text{-}RECOVER}$ because we will enrich it with an $\mathsf{s\text{-}RECOVER}$ notion in Sect. 3.1.

Game $\mathsf{FORGE}^{\mathcal{A}}_{\mathsf{C_{clean}}}(1^\lambda)$	Game $\mathsf{r\text{-}RECOVER}^{\mathcal{A}}(1^\lambda)$
1: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$ 2: $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$ 3: $(\mathsf{P}, \mathsf{ad}, \mathsf{ct}) \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP_{st}}, \mathsf{EXP_{pt}}}(z)$ 4: $\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, \mathsf{ad}, \mathsf{ct}) \to \mathsf{acc}$ 5: **if** $\mathsf{acc} = \mathsf{false}$ **then return** 0 6: **if** $\neg\mathsf{C_{clean}}$ **then return** 0 7: **if** $(\mathsf{ad}, \mathsf{ct})$ is not a forgery (Def. 4) for P **then return** 0 8: **return** 1	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 sent^*_* and $\mathsf{received}^*_*$ variables to $\emptyset$ 5: $\mathsf{P} \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP_{st}}, \mathsf{EXP_{pt}}}(z)$ 6: **if** we can parse $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}} = (\mathsf{seq}_1, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_2)$ and $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}} = (\mathsf{seq}_3, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_4)$ with $\mathsf{seq}_1 \neq \mathsf{seq}_3$ (where $(\mathsf{ad}, \mathsf{ct})$ is a single message and all seq_i are finite sequences of single messages) **then** $\mathsf{win} \leftarrow 1$ 7: **return** win
Game $\mathsf{PREDICT}^{\mathcal{A}}(1^\lambda)$ 1: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$ 2: $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$ 3: $(\mathsf{P}, \mathsf{ad}, \mathsf{pt}) \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP_{st}}, \mathsf{EXP_{pt}}}(z)$	4: $\mathsf{RATCH}(\mathsf{P}, \text{“send”}, \mathsf{ad}, \mathsf{pt}) \to \mathsf{ct}$ 5: **if** $(\mathsf{ad}, \mathsf{ct}) \in \mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ **then return** 1 6: **return** 0

Fig. 3. FORGE, r-RECOVER, and PREDICT Games. (Oracles RATCH, $\mathsf{EXP_{st}}$, $\mathsf{EXP_{pt}}$ are defined in Fig. 1 and Fig. 2.)

Definition 7 ($\mathsf{C_{clean}}$-FORGE security). *Given a cleanness predicate* $\mathsf{C_{clean}}$*, consider* $\mathsf{FORGE}^{\mathcal{A}}_{\mathsf{C_{clean}}}$ *game in Fig. 3 associated to the adversary* $\mathcal{A}$*. Let the advantage of* $\mathcal{A}$ *be the probability that the game outputs 1. We say that* ARCAD *is* $\mathsf{C_{clean}}$-FORGE-secure *if, for any PPT adversary, the advantage is negligible.*

In this definition, we added the notion of cleanness which determines if an attack is trivial or not. The original notion of FORGE security [7] is equivalent to using the following $\mathsf{C_{trivial}}$ predicate $\mathsf{C_{clean}}$:

$\mathsf{C_{trivial}}$: the last $(\mathsf{ad}, \mathsf{ct})$ message is not a trivial forgery (following Definitions 5).

The purpose of this update in the definition is to allow us to easily define a weaker form of FORGE-security for symmetric protocols and in Sect. 3.3.

Definition 8 (r-RECOVER security [7]). *Consider the* $\mathsf{r\text{-}RECOVER}^{\mathcal{A}}$ *game in Fig. 3 associated to the adversary* $\mathcal{A}$*. Let the advantage of* $\mathcal{A}$ *in succeeding in the game be* $\Pr(\mathsf{win} = 1)$*. We say that the* ARCAD *is* r-RECOVER-secure*, if for any PPT adversary, the advantage is negligible.*

Definition 9 (PREDICT security [7]). *Consider* $\mathsf{PREDICT}^{\mathcal{A}}(1^\lambda)$ *game in Fig. 3 associated to the adversary* $\mathcal{A}$*. Let the advantage of* $\mathcal{A}$ *in succeeding in the game be the probability that* 1 *is returned. We say that the* ARCAD *is* PREDICT-secure*, if for any PPT adversary, the advantage is negligible.*

PREDICT-security is useful to reduce the notion of matching status to the two conditions that $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$ is a prefix of $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ at time t for P and $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ is a prefix of $\mathsf{sent}^{\mathsf{P}}_{\mathsf{ct}}$ at time $\overline{\mathsf{t}}$ for $\overline{\mathsf{P}}$.

2.2 The Epoch Notion in Secure Communication

We define the epochs in an equivalent way to the work done by Alwen et al. [1].[8] Epochs are useful to designate the sequence of messages, as both participants may not see exactly the same. We will use epoch numbers in the design of our hybrid scheme for on-demand ratcheting in Sect. 4.1.

Epochs are a set of consecutive messages going in the same direction. An epoch is identified by an integer counter e. Each message is assigned one epoch counter $\mathsf{e_m}$. Hence, the epochs are non-intersecting. For convenience, each participant P keeps the epoch value $\mathsf{e}^{\mathsf{P}}_{\mathsf{send}}$ of the last sent message and the epoch value $\mathsf{e}^{\mathsf{P}}_{\mathsf{rec}}$ of the last received message. They are used to assign an epoch to a message to be sent.

Definition 10 (Epoch). *Epochs are non-intersecting sets of messages which are defined by an integer. During the game, we let $\mathsf{e}^{\mathsf{P}}_{\mathsf{rec}}$ (resp. $\mathsf{e}^{\mathsf{P}}_{\mathsf{send}}$) be the epoch of the last received (resp. sent) message by P. At the very beginning of the protocol, we define $\mathsf{e}^{\mathsf{P}}_{\mathsf{send}}$ and $\mathsf{e}^{\mathsf{P}}_{\mathsf{rec}}$ specifically. For the participant A, $\mathsf{e}^{\mathsf{A}}_{\mathsf{rec}} = -1$ and $\mathsf{e}^{\mathsf{A}}_{\mathsf{send}} = 0$. For the participant B, $\mathsf{e}^{\mathsf{B}}_{\mathsf{send}} = -1$ and $\mathsf{e}^{\mathsf{B}}_{\mathsf{rec}} = 0$. The procedure to assign an epoch $\mathsf{e_m}$ to a new sent message follows the rule described next:*
If $\mathsf{e}^{\mathsf{P}}_{\mathsf{rec}} < \mathsf{e}^{\mathsf{P}}_{\mathsf{send}}$, then the message is put in the epoch $\mathsf{e_m} = \mathsf{e}^{\mathsf{P}}_{\mathsf{send}}$. Otherwise, it is put in epoch $\mathsf{e_m} = \mathsf{e}^{\mathsf{P}}_{\mathsf{rec}} + 1$.

Let $\mathsf{e_P} = \max\{\mathsf{e}^{\mathsf{P}}_{\mathsf{rec}}, \mathsf{e}^{\mathsf{P}}_{\mathsf{send}}\}$. Let $\mathsf{b_A} = 0$ and $\mathsf{b_B} = 1$. We have

$$\mathsf{e}^{\mathsf{P}}_{\mathsf{send}} = \begin{cases} \mathsf{e_P} & \text{if } \mathsf{e_P} \bmod 2 = \mathsf{b_P} \\ \mathsf{e_P} - 1 & \text{otherwise} \end{cases} \qquad \mathsf{e}^{\mathsf{P}}_{\mathsf{rec}} = \begin{cases} \mathsf{e_P} & \text{if } \mathsf{e_P} \bmod 2 \neq \mathsf{b_P} \\ \mathsf{e_P} - 1 & \text{otherwise} \end{cases}$$

Therefore, it is equivalent to maintain $(\mathsf{e}^{\mathsf{P}}_{\mathsf{rec}}, \mathsf{e}^{\mathsf{P}}_{\mathsf{send}})$ or $\mathsf{e_P}$. The procedure to manage $\mathsf{e_P}$ and $\mathsf{e_m}$ is described by Alwen et al. [1].

We will use a counter c for each epoch e. We will use the order on (e, c) pairs defined by

$$(\mathsf{e}, \mathsf{c}) < (\mathsf{e}', \mathsf{c}') \iff (\mathsf{e} < \mathsf{e}' \vee (\mathsf{e} = \mathsf{e}' \wedge \mathsf{c} < \mathsf{c}'))$$

3 Security Awareness

3.1 s-RECOVER Security

We gave the DV r-RECOVER security definition [7] in Definition 8. It is an important notion to capture that P cannot accept a genuine ct from $\overline{\mathsf{P}}$ after P receives a forgery. However, r-RECOVER-security does not capture the fact that when it is $\overline{\mathsf{P}}$ who receives a forgery, P could still accept messages which come from $\overline{\mathsf{P}}$. We strengthen r-RECOVER security with another definition called s-RECOVER.

[8] The notion of epoch appeared in Poettering-Rösler [10] before.

Definition 11 (s-RECOVER **security).** *In the* s-RECOVER$^{\mathcal{A}}$ *game in Fig. 4 with the adversary* $\mathcal{A}$*, we let the advantage of* $\mathcal{A}$ *in succeeding in the game be* $\Pr(\mathsf{win} = 1)$*. We say that the* ARCAD *is* s-RECOVER-secure*, if for any PPT adversary, the advantage is negligible.*

Game $\mathsf{s\text{-}RECOVER}^{\mathcal{A}}(1^\lambda)$
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 sent_*^* and $\mathsf{received}_*^*$ variables to $\emptyset$
5: $\mathsf{P} \leftarrow \mathcal{A}^{\mathsf{RATCH},\mathsf{EXP}_{\mathsf{st}},\mathsf{EXP}_{\mathsf{pt}}}(z)$
6: **if** $\mathsf{received}_{\mathsf{ct}}^{\mathsf{P}}$ is a prefix of $\mathsf{sent}_{\mathsf{ct}}^{\overline{\mathsf{P}}}$ **then**
7: set $\bar{t}$ to the time when $\overline{\mathsf{P}}$ sent the last message in $\mathsf{received}_{\mathsf{ct}}^{\mathsf{P}}$
8: **if** $\mathsf{received}_{\mathsf{ct}}^{\overline{\mathsf{P}}}(\bar{t})$ is not a prefix of $\mathsf{sent}_{\mathsf{ct}}^{\mathsf{P}}$ **then** $\mathsf{win} \leftarrow 1$
9: **end if**
10: **return** win

Fig. 4. s-RECOVER Security Game. (RATCH and EXP oracles are defined in Fig. 1 and Fig. 2.)

Ideally, what we want from the protocol is that participants can detect forgeries by realizing that they are no longer able to communicate to each other. We cannot prevent impersonation to happen after a state exposure but we want to make sure that the normal exchange between the participants is cut. Hence, if a participant eventually receives a genuine message (e.g. because it was authenticated after meeting in person), he should feel safe that no forgeries happened. Contrarily, detecting a communication cut requires an action from the participants, such as restoring communication using a super hybrid structure, as we will suggest in Sect. 4.1.

We directly obtain the following useful result:[9]

Lemma 12. *If an* ARCAD *is* r-RECOVER, s-RECOVER, *and* PREDICT *secure, whenever* P *receives a genuine message from* $\overline{\mathsf{P}}$ *(i.e., an* $(\mathsf{ad}, \mathsf{ct})$ *pair sent by* $\overline{\mathsf{P}}$ *is accepted by* P*),* P *is in a matching status (following Definition 3), except with negligible probability.*

Our notion of RECOVER-security and forgery is quite strong in the sense that it focuses on the ciphertext. Some protocols such as JMM [9] focus on the plaintext. In JMM, ct includes some encrypted data and some signature but only the encrypted data is hashed. Hence, an adversary can replace the signature by another signature after exposure of the signing key. It can be seen as not so important because it must sign the same content. However, the signature has a

[9] The proof is provided in the full version [4].

key update and the adversary can make the receiver update to any verifying key to desynchronize, then re-synchronize at will. Consequently, *the* JMM *protocol does not offer* RECOVER *security* as we defined it. Contrarily, PR [10] hashes $(\mathsf{ad}, \mathsf{ct})$ but does not use it in the next ad or to compute the next ct. Thus, PR *has no* RECOVER *security* either.[10] One may think that it is easy to fix this by hashing all messages but this is not as simple. We propose in Sect. 3.3 the chain transformation which can fix any protocol, thanks to Lemma 18.

3.2 Security Awareness

To have a security-awareness notion, we want r-RECOVER, s-RECOVER, and PREDICT security[11], we want to have an acknowledgment extractor (to be aware of message delivery), and we want to have a cleanness extractor (to be aware of the cleanness of every message, if not subject to trivial exposure). The last two notions are defined below. This means that on the one hand, impersonations are eventually discovered, and on the other hand, by assuming that no impersonation occurs and assuming that exposures are known, a participant P knows exactly which messages are safe, at least after one round-trip occurred.

Definition 13 (Security-awareness). *A protocol is* $\mathsf{C}_{\mathsf{clean}}$-security-aware *if*

- *it is* r-RECOVER, s-RECOVER, *and* PREDICT*-secure;*
- *there is an acknowledgment extractor (Definition 15);*
- *there is a cleanness extractor for* $\mathsf{C}_{\mathsf{clean}}$ *(Definition 16).*

To make participants aware of the security status of any (challenge) message, they need to know the history of exposures, they need to be able to reconstruct the history of RATCH calls from their own view, and they need to be able to evaluate the $\mathsf{C}_{\mathsf{clean}}$ predicate. Thankfully, the $\mathsf{C}_{\mathsf{clean}}$ predicates that we consider only depend on these histories. We first formally define the notion of transcript.

Definition 14 (Transcript). *In a game, for a participant* P*, we define the* transcript *of* P *as the chronological sequence* T_P *of all* $(\mathsf{oracle}, \mathsf{extra})$ *pairs involving* P *where each pair represents an oracle call to* oracle *with* P *as input (i.e. either* RATCH(P, "rec", ., .)*,* RATCH(P, "send", ., .)*,* $\mathsf{EXP}_{\mathsf{pt}}(\mathsf{P})$*,* $\mathsf{EXP}_{\mathsf{st}}(\mathsf{P})$*, or* CHALLENGE(P)*), except the unsuccessful* RATCH *calls which are omitted. For each pair with a* RATCH *or* CHALLENGE *oracle,* extra *specifies the role (*"send" *or* "rec"*) and the message* $(\mathsf{ad}, \mathsf{ct})$ *of the oracle call. For other pairs,* $\mathsf{extra} = \perp$*.*

---

[10] More precisely, in PR, if A is exposed then issues a message ct, the adversary can actually forge a ciphertext ct' transporting the same pk and vfk and deliver it to B in a way which makes B accept. If A issues a new message ct'', delivering ct'' to B will pass the signature verification. The decryption following-up may fail, except if the kuKEM encryption scheme taking care of encryption does not check consistency, which is the case in the proposed one [10, Fig. 3, eprint version]. Therefore, ct'' may be accepted by B so PR is not r-RECOVER secure. The same holds for s-RECOVER security.

[11] We want it to be able to apply Lemma 12 and be aware of matching status.

The partial transcript *of* P *up to time* t *is the prefix* $\mathsf{T_P(t)}$ *of* $\mathsf{T_P}$ *of all oracle calls until time* t*. The* RATCH-transcript *of* P *is the list* $\mathsf{T_P^{RATCH}}$ *of all* extra *elements in* $\mathsf{T_P}$ *which are not* $\perp$ *(i.e. it only includes* $\mathsf{RATCH/CHALLENGE}$ *calls). Similarly, the* partial RATCH-transcript *of* P *up to time* t *is the list* $\mathsf{T_P^{RATCH}(t)}$ *of* extra *elements in* $\mathsf{T_P(t)}$ *which are not* $\perp$*.*

Next, we formalize that a participant can be aware of which of his messages were received by his counterpart.

Definition 15 (Acknowledgment extractor). *We consider a game* Γ *where the transcript* $\mathsf{T_P}$ *is formed for a participant* P*. Given a message* $(\mathsf{ad}, \mathsf{ct})$ *successfully received by* P *at time* t *and which was sent by* $\overline{\mathsf{P}}$ *at time* $\bar{\mathsf{t}}$*, we let* $(\mathsf{ad}', \mathsf{ct}')$ *be the last message successfully received by* $\overline{\mathsf{P}}$ *before time* $\bar{\mathsf{t}}$*. (If there is no such message, we set it to* $\perp$*.)*

An acknowledgment extractor *is an efficient function* f *such that* $\mathsf{f}(\mathsf{T_P^{RATCH}(t)}) = (\mathsf{ad}', \mathsf{ct}')$ *for any time* t *when* P *is in a matching status (Definition 3).*

Given this extractor, P can iteratively reconstruct the entire flow of messages, and which messages crossed each other during transmission.

We formalize awareness of a participant for the safety of each message.

Definition 16 (Cleanness extractor). *We consider a game* Γ *where the transcript* $\mathsf{T_P}$ *is formed for a participant* P*. Let* t *be a time for* P *and* $\bar{\mathsf{t}}$ *be a time for* $\overline{\mathsf{P}}$*. Let* $\mathsf{T_P(t)}$ *and* $\mathsf{T_{\overline{P}}(\bar{t})}$ *be the partial transcripts at those time. We say that there is a* cleanness extractor for $\mathsf{C_{clean}}$ *if there is an efficient function* g *such that* $\mathsf{g}(\mathsf{T_P(t)}, \mathsf{T_{\overline{P}}(\bar{t})})$ *has the following properties: if there is one* $\mathsf{CHALLENGE}$ *in the* $\mathsf{T_P(t)}$ *transcript and, either* P *received* $(\mathsf{ad_{test}}, \mathsf{ct_{test}})$ *or there is a round trip* $\mathsf{P} \to \overline{\mathsf{P}} \to \mathsf{P}$ *starting with* P *sending* $(\mathsf{ad_{test}}, \mathsf{ct_{test}})$ *to* $\overline{\mathsf{P}}$*, then* $\mathsf{g}(\mathsf{T_P(t)}, \mathsf{T_{\overline{P}}(\bar{t})}) = \mathsf{C_{clean}}(\Gamma)$*. Otherwise,* $\mathsf{g}(\mathsf{T_P(t)}, \mathsf{T_{\overline{P}}(\bar{t})}) = \perp$*.*

The function g is able to predict whether the game is "clean" for any challenge message. The case with an incomplete round trip $\mathsf{P} \to \overline{\mathsf{P}} \to \mathsf{P}$ starting with P sending $(\mathsf{ad_{test}}, \mathsf{ct_{test}})$ to $\overline{\mathsf{P}}$ is when the tested message was sent but somehow never acknowledged for the reception. If the message never arrived, we cannot say for sure if the game is clean because the counterpart may later either receive it and make the game clean or have a state exposure and make the game not clean. In other cases, the cleanness can be determined for sure.

3.3 Strongly Secure ARCAD with Security Awareness

In this section, we take a secure ARCAD (it could be $\mathsf{ARCAD_{DV}}$, in the full version [4], or the hybrid one defined in Sect. 4) which we denote by ARCAD_0 and we transform it into another secure ARCAD which we denote by $\mathsf{ARCAD}_1 = \mathsf{chain}(\mathsf{ARCAD}_0)$, that is *security aware*. We achieve security awareness by keeping some hashes in the states of participants. The intuitive way to build it is to make chains of hash of ciphertexts (like a blockchain) which will be sent and received and to associate each message to the digest of the chain. This enables a

participant P to acknowledge its counterpart about received messages whenever P sends a new message.

We define a tuple (Hsent, Hreceived, snt_noack, rec_toack) and store it in the state of a participant. Hsent is the hash of all sent ciphertexts. It is computed by the sender and delivered to the counterpart along with ct. It is updated with hashing key hk and the old Hsent every time a new Send operation is called. Likewise, Hreceived is the hash of all received ciphertexts. It is computed with hk and the last stored Hreceived by the receiver upon receiving a message. It is updated every time a new Receive operation is run.

Using Hsent and Hreceived alone is sufficient for r-RECOVER security but not for s-RECOVER security.

rec_toack is a counter of received messages which need to be reported when the next Send operation is run. For each Send operation, the protocol attaches to ct the last Hreceived to acknowledge for received messages and reset rec_toack to 0. rec_toack is incremented by each Receive.

snt_noack is a *list of the hashes* of sent ciphertexts which are waiting for an acknowledgment. Basically, it is initialized to an empty array in the beginning and whenever a new Hsent is computed, it is accumulated in this array. The purpose of such a list is to keep track of the sent messages for which the sender expects an acknowledgment. More precisely, when the participant P keeps its list of sent ciphertexts in snt_noack, the counterpart $\overline{\mathsf{P}}$ keeps a counter rec_toack telling that an acknowledgment is needed. Remember that $\overline{\mathsf{P}}$ sends Hreceived back to the participant P to acknowledge him about received messages. As soon as $\overline{\mathsf{P}}$ acknowledges, P deletes the hash of the acknowledged ciphertexts from snt_noack.

The principle of our construction is that if an adversary starts to impersonate a participant after exposure, there is a fork in the list of message chains which is viewed by both participants and those chains can never merge again without making a collision.

We give our security aware protocol on Fig. 5. The security of the protocol is proved with the following lemmas.

Theorem 17. *If* ARCAD_0 *is correct, then* $\mathsf{chain}(\mathsf{ARCAD}_0)$ *is correct.*

The proof is straightforward.

Lemma 18. *If* H *is collision-resistant,* $\mathsf{chain}(\mathsf{ARCAD}_0)$ *is* RECOVER*-secure (for both* s-RECOVER *and* r-RECOVER *security).*

Proof. All $(\mathsf{ad}, \mathsf{ct})$ messages seen by one participant P in one direction (send or receive) are chained by hashing. Hence, if $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}} = (\mathsf{seq}_1, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_2)$, the $(\mathsf{ad}, \mathsf{ct})$ message includes (in the second field of ct) the hash h of seq_1. If $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}} = (\mathsf{seq}_3, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_4)$, the $(\mathsf{ad}, \mathsf{ct})$ message includes the hash h of seq_3. If H is collision-resistant, then $\mathsf{seq}_1 \neq \mathsf{seq}_3$ with negligible probability. Hence, we have r-RECOVER security.

Additionally, all genuine $(\mathsf{ad}, \mathsf{ct})$ messages include (in the third field of ct) the hash ack of messages which are received by the counterpart. This list must

$\mathsf{ARCAD}_1.\mathsf{Setup}(1^\lambda)$
1: $\mathsf{ARCAD}_0.\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}_0$
2: $\mathsf{H.Gen}(1^\lambda) \xrightarrow{\$} \mathsf{hk}$
3: $\mathsf{pp} \leftarrow (\mathsf{hk}, \mathsf{pp}_0)$
4: **return** pp

$\mathsf{ARCAD}_1.\mathsf{Gen} = \mathsf{ARCAD}_0.\mathsf{Gen}$

$\mathsf{ARCAD}_1.\mathsf{Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_\mathsf{P}, \mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{P})$
1: parse $\mathsf{pp} = (\mathsf{hk}, \mathsf{pp}_0)$
2: $\mathsf{ARCAD}_0.\mathsf{Init}(1^\lambda, \mathsf{pp}_0, \mathsf{sk}_\mathsf{P}, \mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{P}) \xrightarrow{\$} \mathsf{st}'_\mathsf{P}$
3: $\mathsf{Hsent}, \mathsf{Hreceived} \leftarrow \perp$
4: $\mathsf{snt_noack} \leftarrow []$, $\mathsf{rec_toack} \leftarrow 0$
5: $\mathsf{st}_\mathsf{P} \leftarrow (\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{snt_noack}, \mathsf{rec_toack})$
6: **return** st_P

$\mathsf{ARCAD}_1.\mathsf{Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$
1: parse st_P as $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{snt_noack}, \mathsf{rec_toack})$
2: **if** $\mathsf{rec_toack} = 0$ **then** $\mathsf{ack} \leftarrow \perp$ **else** $\mathsf{ack} \leftarrow \mathsf{Hreceived}$
3: $\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{Hsent}, \mathsf{ack})$
4: $\mathsf{ARCAD}_0.\mathsf{Send}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{pt}) \xrightarrow{\$} (\mathsf{st}'_\mathsf{P}, \mathsf{ct}')$
5: $\mathsf{ct} \leftarrow (\mathsf{ct}', \mathsf{Hsent}, \mathsf{ack})$
6: $\mathsf{rec_toack} \leftarrow 0$
7: $\mathsf{Hsent} \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{Hsent}, \mathsf{ad}, \mathsf{ct})$
8: $\mathsf{snt_noack} \leftarrow (\mathsf{snt_noack}, \mathsf{Hsent})$
9: $\mathsf{st}_\mathsf{P} \leftarrow (\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{snt_noack}, \mathsf{rec_toack})$
10: **return** $(\mathsf{st}_\mathsf{P}, \mathsf{ct})$

$\mathsf{ARCAD}_1.\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$
1: parse st_P as $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{snt_noack}, \mathsf{rec_toack})$
2: parse ct as $(\mathsf{ct}', \mathsf{h}, \mathsf{ack})$
3: **if** $\mathsf{h} \neq \mathsf{Hreceived}$ or $\mathsf{ack} \notin \{\perp\} \cup \mathsf{snt_noack}$ **then**
4: **return** $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \perp)$
5: **end if**
6: $\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{h}, \mathsf{ack})$
7: $\mathsf{ARCAD}_0.\mathsf{Receive}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{ct}') \rightarrow (\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{pt}')$
8: **if** acc **then**
9: $\mathsf{Hreceived} \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{Hreceived}, \mathsf{ad}, \mathsf{ct})$
10: $\mathsf{rec_toack} \leftarrow \mathsf{rec_toack} + 1$
11: **if** $\mathsf{ack} \neq \perp$ **then** remove in $\mathsf{snt_noack}$ all elements of $\mathsf{snt_noack}$ until ack (included)
12: $\mathsf{st}_\mathsf{P} \leftarrow (\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{snt_noack}, \mathsf{rec_toack})$
13: **end if**
14: **return** $(\mathsf{acc}, \mathsf{st}_\mathsf{P}, \mathsf{pt}')$

Fig. 5. Our Security-Aware $\mathsf{ARCAD}_1 = \mathsf{chain}(\mathsf{ARCAD}_0)$ Protocol.

be approved by P, thus it must match the list of hashes of messages that P sent. Hence, if $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$ is prefix of $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ and $\bar{t}$ is the time when $\overline{\mathsf{P}}$ sent the last message in $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$, then this message includes the hash of $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{ct}}(\bar{t})$ which must be a hash of a prefix of $\mathsf{sent}^{\mathsf{P}}_{\mathsf{ct}}$. Thus, unless there is a collision in the hash function, $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{ct}}(\bar{t})$ is a prefix of $\mathsf{sent}^{\mathsf{P}}_{\mathsf{ct}}$ and we have s-RECOVER security. □

Lemma 19. $\mathsf{chain}(\mathsf{ARCAD}_0)$ *has an acknowledgment extractor.*

Proof. Let $(\mathsf{ad}, \mathsf{ct})$ be a message sent by $\overline{\mathsf{P}}$ to P in a matching status. Let $(\mathsf{ad}', \mathsf{ct}')$ be the last message received by $\overline{\mathsf{P}}$ before sending $(\mathsf{ad}, \mathsf{ct})$. Due to the protocol, ct includes the value of $\mathsf{Hreceived}$ after receiving $(\mathsf{ad}', \mathsf{ct}')$. Since this message is from P, P recognizes this hash $\mathsf{Hreceived} = \mathsf{Hsent}$ from $\mathsf{snt_noack}$. Both $(\mathsf{ad}', \mathsf{ct}')$ and this hash can be computed from $\mathsf{T}^{\mathsf{RATCH}}_{\mathsf{P}}(t)$. Hence, $\mathsf{chain}(\mathsf{ARCAD}_0)$ has an extractor. □

Lemma 20. $\mathsf{chain}(\mathsf{ARCAD}_0)$ *has a cleanness extractor for the following predicates:*

$$\mathsf{C}_{\mathsf{leak}}, \mathsf{C}^{\mathsf{P}_{\mathsf{test}}}_{\mathsf{trivialforge}}, \mathsf{C}^{A,B}_{\mathsf{trivialforge}}, \mathsf{C}^{\mathsf{P}_{\mathsf{test}}}_{\mathsf{forge}}, \mathsf{C}^{A,B}_{\mathsf{forge}}, \mathsf{C}_{\mathsf{ratchet}}, \mathsf{C}_{\mathsf{noexp}}$$

Hence, there is an extractor for all cleanness predicates which we considered.[12]

The following result is trivial.

Lemma 21. *If* ARCAD_0 *is* $\mathsf{PREDICT}$*-secure, then* $\mathsf{chain}(\mathsf{ARCAD}_0)$ *is* $\mathsf{PREDICT}$*-secure.*

Consequently, if ARCAD_0 is $\mathsf{PREDICT}$-secure, $\mathsf{chain}(\mathsf{ARCAD}_0)$ is security-aware.

[12] The proof is given in the full version [4].

4 On-Demand Ratcheting

In this section, we define a bidirectional secure communication messaging protocol with *hybrid on-demand* ratcheting. The aim is to design such a protocol to integrate two ratcheting protocols with different security levels: a strongly secure protocol using public-key cryptography and a weaker but much more efficient protocol with symmetric key primitives. The core of the protocol is to use the weak protocol with frequent exchanges and to use the strong one on demand by the sending participant. Hence, we build a more efficient protocol with on-demand ratcheting. Yet, it comes with a security drawback. Even though the security for the former is to provide post-compromise security, we secure part of the communication only with the forward secure protocol.

The sender uses a flag to tell which level of security the communication will have and apply ratcheting with public-key cryptography or the lighter primitives such as the EtH protocol [13]. The flag is set in the ad input and it is denoted as $\mathsf{ad.flag}$. We call the strong protocol as $\mathsf{ARCAD}_{\mathsf{main}}$ and the weak one as $\mathsf{ARCAD}_{\mathsf{sub}}$. Ideally, the time to set the flag for specific security can be decided during the deployment of the application using the protocol. This choice may also be left to the users who can decide based on the confidentiality-level of their communication. The more often the protocol turns the flag on, the more secure is the hybrid on-demand protocol. If we do it for every message exchange, then we obtain $\mathsf{ARCAD}_{\mathsf{main}}$ without $\mathsf{ARCAD}_{\mathsf{sub}}$. If we do it for no message exchange, then we obtain $\mathsf{ARCAD}_{\mathsf{sub}}$.

4.1 Our Hybrid On-Demand ARCAD Protocol

We give our on-demand ARCAD protocol on Fig. 6. It uses two sub-protocols called $\mathsf{ARCAD}_{\mathsf{main}}$ and $\mathsf{ARCAD}_{\mathsf{sub}}$. The former is to represent a strong-but-slow protocol such as $\mathsf{ARCAD}_{\mathsf{DV}}$ (Fig. 11). The latter is typically a weaker-but-faster protocol like EtH [13]. The use of one or the other is based on a flag that can be turned on and off in ad (it is checked with $\mathsf{ad.flag}$ operation in the protocol). To have the flag on lets the protocol run $\mathsf{ARCAD}_{\mathsf{main}}$ while setting the flag off means to run $\mathsf{ARCAD}_{\mathsf{sub}}$. Assuming that $\mathsf{ARCAD}_{\mathsf{main}}$ is ratcheting (i.e. post-compromise secure) and $\mathsf{ARCAD}_{\mathsf{sub}}$ is not, this defines on-demand ratcheting. We denote our hybrid protocol as $\mathsf{hybridARCAD} = \mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{main}}, \mathsf{ARCAD}_{\mathsf{sub}})$.

We use as a reference the (e, c) number of messages in the $\mathsf{ARCAD}_{\mathsf{main}}$ thread. Every $\mathsf{ARCAD}_{\mathsf{main}}$ message creates a new $\mathsf{ARCAD}_{\mathsf{sub}}$ send/receive state pair. The sending participant keeps the generated send state in a $\mathsf{sub}[e, c]$ register under the (e, c) number of the message and sends the generated receive state together with his message. The very first message which a participant sees (either in sending or receiving) forces the flag to indicate $\mathsf{ARCAD}_{\mathsf{main}}$ as we have no initial $\mathsf{ARCAD}_{\mathsf{sub}}$ state. The (e, c) number if authenticated and also explicitly added in the ciphertext. The receiving participant checks that (e, c) increases and uses the $\mathsf{sub}[e, c]$ register state to receive the message.

Theorem 22. *If the protocols* $\mathsf{ARCAD}_{\mathsf{main}}$ *and* $\mathsf{ARCAD}_{\mathsf{sub}}$ *are both correct, then the protocol* $\mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{main}}, \mathsf{ARCAD}_{\mathsf{sub}})$ *is correct.*

The proof is provided in the full version [4].

4.2 Application: Super-Scheme to (Re)set a Protocol

Our hybrid construction finds another application than on-demand ratcheting: defense against message loss or active attacks. Indeed, by using $\mathsf{ARCAD}_{\mathsf{main}} = \mathsf{ARCAD}_{\mathsf{sub}}$, we can set ad.flag to restore an $\mathsf{ARCAD}_{\mathsf{sub}}$ communication which was broken due to a message loss. Normal communication works in the $\mathsf{ARCAD}_{\mathsf{sub}}$ session, hence with a flag down. However, we may use $\mathsf{ARCAD}_{\mathsf{main}}$ to start a new $\mathsf{ARCAD}_{\mathsf{sub}}$ session. If $\mathsf{ARCAD}_{\mathsf{sub}}$ gets broken due to a message loss or an active attack on it, $\mathsf{ARCAD}_{\mathsf{main}}$ can be used to restart a new $\mathsf{ARCAD}_{\mathsf{sub}}$ session. We cannot resume if the $\mathsf{ARCAD}_{\mathsf{main}}$ session is broken. However, we can also make nested hybrid protocols with more than two levels of protocols inside for safety. It may increase the state sizes but the performance should be nearly the same. Then, only persistent message drop attacks would succeed to make a denial of service.

4.3 Security Definitions

We modify the predicates and the notion of FORGE-security from Sect. 2. In our hybrid protocol, each message $(\mathsf{ad}, \mathsf{ct})$ has a clearly defined (e, c) pair. A ct which is input or output from RATCH comes with an ad which has a clearly defined ad.flag bit.

Sub-games. Given a game Γ for the hybridARCAD scheme with an adversary $\mathcal{A}$, we define a game $\mathsf{main}(\Gamma)$ for $\mathsf{ARCAD}_{\mathsf{main}}$ with an adversary $\mathcal{A}'$ which simulates everything but the $\mathsf{ARCAD}_{\mathsf{main}}$ calls in Γ. Namely, $\mathcal{A}'$ simulates the enrichment of the states and all $\mathsf{ARCAD}_{\mathsf{sub}}$ management together with $\mathcal{A}$.

Given a game Γ_{main} for $\mathsf{ARCAD}_{\mathsf{main}}$ using no CHALLENGE oracle and an (e, c) pair, we denote by $\mathsf{main}_{\mathsf{e},\mathsf{c}}(\Gamma_{\mathsf{main}})$ the variant of Γ_{main} in which the RATCH Send call making the message $(\mathsf{ad}, \mathsf{ct})$ with pair (e, c) is replaced by a CHALLENGE query with $\mathsf{b} = 1$. This perfectly simulates Γ_{main} and produces the same value, and we can evaluate a predicate $\mathsf{C}_{\mathsf{clean}}$ relative to this challenge message. We define $\mathsf{C}^{\mathsf{e},\mathsf{c}}_{\mathsf{clean}}(\Gamma_{\mathsf{main}}) = \mathsf{C}_{\mathsf{clean}}(\mathsf{main}_{\mathsf{e},\mathsf{c}}(\Gamma_{\mathsf{main}}))$. Intuitively, $\mathsf{C}^{\mathsf{e},\mathsf{c}}_{\mathsf{clean}}(\Gamma_{\mathsf{main}})$ means that the message of pair (e, c) was safely encrypted and should be considered as private because no trivial attack leaks it.

We also define $\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)$ and $\mathsf{sub}'_{\mathsf{e},\mathsf{c}}(\Gamma)$. We let P be the sending participant of the $\mathsf{ARCAD}_{\mathsf{main}}$ message of pair (e, c). In $\mathsf{sub}'_{\mathsf{e},\mathsf{c}}(\Gamma)$, the adversary $\mathcal{A}'$ simulates everything but the $\mathsf{ARCAD}_{\mathsf{sub}}$ calls involving messages with pair (e, c). The initial states of P and $\overline{\mathsf{P}}$ are also set by the game $\mathsf{sub}'_{\mathsf{e},\mathsf{c}}(\Gamma)$. However, it makes an $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ call at the beginning of the protocol to get the initial state st_{R} for $\mathsf{ARCAD}_{\mathsf{sub}}$. With this state, $\mathcal{A}'$ can simulate the encryption of st_{R} with $\mathsf{ARCAD}_{\mathsf{main}}$ and all the rest. Clearly, the simulation is perfect but it adds an initial $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ call.

The $\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)$ game is a variant of $\mathsf{sub}'_{\mathsf{e},\mathsf{c}}(\Gamma)$ without the additional $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$. To simulate the encryption of st_{R}, $\mathcal{A}'$ encrypts a random string instead. When it

$\mathsf{hybridARCAD.Setup}(1^\lambda)$
1: $\mathsf{pp_{main}} \leftarrow \mathsf{ARCAD_{main}.Setup}(1^\lambda)$
2: $\mathsf{pp_{sub}} \leftarrow \mathsf{ARCAD_{sub}.Setup}(1^\lambda)$
3: **return** $(\mathsf{pp_{main}}, \mathsf{pp_{sub}})$

$\mathsf{hybridARCAD.Gen}(1^\lambda, \mathsf{pp_{main}}, \mathsf{pp_{sub}})$
4: **return** $\mathsf{ARCAD_{main}.Gen}(1^\lambda, \mathsf{pp_{main}})$

$\mathsf{hybridARCAD.Init}(1^\lambda, (\mathsf{pp_{main}}, \mathsf{pp_{sub}}), \mathsf{sk_P}, \mathsf{pk_{\overline{P}}}, \mathsf{P})$
1: $\mathsf{ARCAD_{main}.Init}(1^\lambda, \mathsf{pp_{main}}, \mathsf{sk_P}, \mathsf{pk_{\overline{P}}}, \mathsf{P}) \rightarrow \mathsf{st_{main}}$
2: initialize array $\mathsf{st_{sub}}[]$ to empty
3: **if** $\mathsf{P} = A$ **then** $(\mathsf{e_{send}}, \mathsf{e_{rec}}) \leftarrow (0, -1)$
4: **else** $(\mathsf{e_{send}}, \mathsf{e_{rec}}) \leftarrow (-1, 0)$
5: **end if**
6: initialize array ctr with $\mathsf{ctr}[0] = -1$
7: $\mathsf{st_P} \leftarrow (\lambda, \mathsf{pp_{sub}}, \mathsf{st_{main}}, \mathsf{st_{sub}}[], \mathsf{e_{send}}, \mathsf{e_{rec}}, \mathsf{ctr}[])$
8: **return** $\mathsf{st_P}$

$\mathsf{hybridARCAD.Send}(\mathsf{st_P}, \mathsf{ad}, \mathsf{pt})$
1: parse $\mathsf{st_P}$ as $(\lambda, \mathsf{pp_{sub}}, \mathsf{st_{main}}, \mathsf{st_{sub}}[], \mathsf{e_{send}}, \mathsf{e_{rec}}, \mathsf{ctr}[])$
2: $e \leftarrow \max(\mathsf{e_{send}}, \mathsf{e_{rec}})$; $\mathsf{c} \leftarrow \mathsf{ctr}[e]$ ▷ current epoch
3: **if** $\mathsf{ad.flag}$ or $\mathsf{c} = -1$ **then**
4: **if** $\mathsf{e_{send}} < \mathsf{e_{rec}}$ **then** $e \leftarrow \mathsf{e_{rec}} + 1$; $\mathsf{c} \leftarrow 0$
5: **else** $e \leftarrow \mathsf{e_{send}}$; $\mathsf{c} \leftarrow \mathsf{ctr}[e] + 1$
6: **end if**
7: $\mathsf{ARCAD_{sub}.InitAll}(1^\lambda, \mathsf{pp_{sub}}) \xrightarrow{\$} (\mathsf{st_S}, \mathsf{st_R}, z)$ ▷ create a new sub-state.
8: $\mathsf{st_{sub}}[e, \mathsf{c}] \leftarrow \mathsf{st_S}$
9: $\mathsf{pt}' \leftarrow (\mathsf{st_R}, \mathsf{pt})$; $\mathsf{ad}' \leftarrow (\mathsf{ad}, 1, e, \mathsf{c})$
10: $\mathsf{ARCAD_{main}.Send}(\mathsf{st_{main}}, \mathsf{ad}', \mathsf{pt}') \xrightarrow{\$} (\mathsf{st_{main}}, \mathsf{ct}')$ ▷ send using the main state.
11: $\mathsf{ct} \leftarrow (\mathsf{ct}', e, \mathsf{c})$
12: $\mathsf{e_{send}} \leftarrow e$; $\mathsf{ctr}[\mathsf{e_{send}}] \leftarrow \mathsf{c}$
13: **else**
14: $\mathsf{ad}' \leftarrow (\mathsf{ad}, 0, e, \mathsf{c})$
15: $\mathsf{ARCAD_{sub}.Send}(\mathsf{st_{sub}}[e, \mathsf{c}], \mathsf{ad}', \mathsf{pt}) \xrightarrow{\$} (\mathsf{st_{sub}}[e, \mathsf{c}], \mathsf{ct}')$ ▷ send using the sub-state.
16: $\mathsf{ct} \leftarrow (\mathsf{ct}', e, \mathsf{c})$
17: **end if**
18: clean-up: erase $\mathsf{st_{sub}}[e, \mathsf{c}]$ for all (e, c) such that $(e, \mathsf{c}) < (\mathsf{e_{send}}, \mathsf{ctr}[\mathsf{e_{send}}])$ and $(e, \mathsf{c}) < (\mathsf{e_{rec}}, \mathsf{ctr}[\mathsf{e_{rec}}])$
19: clean-up: erase $\mathsf{ctr}[e]$ for all e such that $e < \mathsf{e_{send}}$ and $e < \mathsf{e_{rec}}$
20: $\mathsf{st_P} \leftarrow (\lambda, \mathsf{pp_{sub}}, \mathsf{st_{main}}, \mathsf{st_{sub}}[], \mathsf{e_{send}}, \mathsf{e_{rec}}, \mathsf{ctr}[])$
21: **return** $(\mathsf{st_P}, \mathsf{ct})$

$\mathsf{hybridARCAD.Receive}(\mathsf{st_P}, \mathsf{ad}, \mathsf{ct})$
22: parse $\mathsf{st_P}$ as $(\lambda, \mathsf{pp_{sub}}, \mathsf{st_{main}}, \mathsf{st_{sub}}[], \mathsf{e_{send}}, \mathsf{e_{rec}}, \mathsf{ctr}[])$
23: parse ct as $(\mathsf{ct}', e, \mathsf{c})$
24: **if** $(e, \mathsf{c}) < (\mathsf{e_{rec}}, \mathsf{ctr}[\mathsf{e_{rec}}])$ **then return** $(\mathsf{false}, \mathsf{st_P}, \perp)$ ▷ (e, c) must increase
25: **if** $\mathsf{ad.flag}$ or $(e = 0$ and $\mathsf{ctr}[0] = -1)$ **then**
26: $\mathsf{ad}' \leftarrow (\mathsf{ad}, 1, e, \mathsf{c})$
27: $\mathsf{ARCAD_{main}.Receive}(\mathsf{st_{main}}, \mathsf{ad}', \mathsf{ct}') \rightarrow (\mathsf{acc}, \mathsf{st_{main}}, \mathsf{pt}')$
28: parse pt' as $(\mathsf{st_R}, \mathsf{pt})$
29: **if** acc **then**
30: $\mathsf{st_{sub}}[e, \mathsf{c}] \leftarrow \mathsf{st_R}$
31: $\mathsf{e_{rec}} \leftarrow e$; $\mathsf{ctr}[e] \leftarrow \mathsf{c}$
32: **end if**
33: **else**
34: $\mathsf{ad}' \leftarrow (\mathsf{ad}, 0, e, \mathsf{c})$
35: **if** $\mathsf{st_{sub}}[e, \mathsf{c}]$ undefined **then return** $(\mathsf{false}, \mathsf{st_P}, \perp)$
36: $\mathsf{ARCAD_{sub}.Receive}(\mathsf{st_{sub}}[e, \mathsf{c}], \mathsf{ad}', \mathsf{ct}') \rightarrow (\mathsf{acc}, \mathsf{st_{sub}}[e, \mathsf{c}], \mathsf{pt})$
37: **end if**
38: clean-up: erase $\mathsf{st_{sub}}[e, \mathsf{c}]$ for all (e, c) such that $(e, \mathsf{c}) < (\mathsf{e_{send}}, \mathsf{ctr}[\mathsf{e_{send}}])$ and $(e, \mathsf{c}) < (\mathsf{e_{rec}}, \mathsf{ctr}[\mathsf{e_{rec}}])$
39: clean-up: erase $\mathsf{ctr}[e]$ for all e such that $e < \mathsf{e_{send}}$ and $e < \mathsf{e_{rec}}$
40: $\mathsf{st_P} \leftarrow (\lambda, \mathsf{pp_{sub}}, \mathsf{st_{main}}, \mathsf{st_{sub}}[], \mathsf{e_{send}}, \mathsf{e_{rec}}, \mathsf{ctr}[])$
41: **return** $(\mathsf{acc}, \mathsf{st_P}, \mathsf{pt})$

Fig. 6. On-Demand $\mathsf{hybridARCAD} = \mathsf{hybrid}(\mathsf{ARCAD_{main}}, \mathsf{ARCAD_{sub}})$ Protocol.

comes to decrypt the obtained ciphertext, the random plaintext is ignored and the RATCH calls with $\mathsf{st_R}$ are simulated with the RATCH calls for the $\mathsf{ARCAD_{sub}}$ game. The simulation is no longer perfect but it does not add an $\mathsf{EXP_{st}}(\overline{\mathsf{P}})$ call.

Hybrid Cleanness. We assume two cleanness predicates $\mathsf{C_{clean}}$ and $\mathsf{C_{main}}$ (which could be the same) for $\mathsf{ARCAD_{main}}$ and one cleanness predicate $\mathsf{C_{sub}}$ for $\mathsf{ARCAD_{sub}}$. We define a hybrid predicate $\mathsf{C}^{\mathsf{C_{clean}}}_{\mathsf{C_{main}},\mathsf{C_{sub}}}$ as follows. By abuse of notation, we write $\mathsf{C^{clean}_{main,sub}}$ instead, for more readability. Let Γ be a game played by an adversary $\mathcal{A}$ against $\mathsf{hybridARCAD}$.

We let $(\mathsf{ad},\mathsf{ct})$ be the challenge message $(\mathsf{ad_{test}},\mathsf{ct_{test}})$ if it exists. Otherwise, $(\mathsf{ad},\mathsf{ct})$ is the last message in Γ. We let (e,c) be the number of $(\mathsf{ad},\mathsf{ct})$. We let

$$\mathsf{C^{clean}_{main,sub}}(\Gamma) = \begin{cases} \text{if } (\mathsf{ad},\mathsf{ct}) \text{ belongs to } \mathsf{ARCAD_{main}} : \mathsf{C_{main}}(\mathsf{main}(\Gamma)) \\ \text{else} : \begin{cases} \text{if } \mathsf{C}^{e,\mathsf{c}}_{\mathsf{clean}}(\mathsf{main}(\Gamma)) : & \mathsf{C_{sub}}(\mathsf{sub}_{e,\mathsf{c}}(\Gamma)) \\ \text{else} : & \mathsf{C_{sub}}(\mathsf{sub}'_{e,\mathsf{c}}(\Gamma)) \end{cases} \end{cases}$$

This means that if the challenge holds on an $\mathsf{ARCAD_{main}}$ message, we only care for $\mathsf{main}(\Gamma)$ to be $\mathsf{C_{main}}$-clean. Otherwise, either the $\mathsf{ARCAD_{main}}$ message initiating the relevant $\mathsf{ARCAD_{sub}}$ session is $\mathsf{C_{clean}}$ or not. If it is clean, we can replace it and consider $\mathsf{C_{sub}}$-cleanness for $\mathsf{sub}_{e,\mathsf{c}}(\Gamma)$. Otherwise, the initial $\mathsf{ARCAD_{sub}}$ state $\mathsf{st_R}$ trivially leaked (or was exposed, equivalently) and we consider $\mathsf{C_{sub}}$-cleanness for $\mathsf{sub}'_{e,\mathsf{c}}(\Gamma)$. The role of $\mathsf{C_{clean}}$ is to control which of the two games to use. $\mathsf{C_{clean}}$ must be a privacy cleanness notion for main. Contrarily, $\mathsf{C_{main}}$ and $\mathsf{C_{sub}}$ could be either privacy or authenticity notions.

Note that $\mathsf{C_{sub}}(\mathsf{sub}'_{e,\mathsf{c}}(\Gamma)) = \mathsf{false}$ for $\mathsf{C_{sub}} = \mathsf{C_{noexp}}$, due to the $\mathsf{EXP_{st}}$ call.

We easily obtain the following result.

Lemma 23. *If* $\mathsf{ARCAD_{main}}$ *is* $\mathsf{C_{main}}$-$\mathsf{IND\text{-}CCA}$-*secure and* $\mathsf{ARCAD_{sub}}$ *is* $\mathsf{C_{sub}}$-$\mathsf{IND\text{-}CCA}$-*secure, then* $\mathsf{hybridARCAD}$ *is* $\mathsf{C_{clean}}$-$\mathsf{IND\text{-}CCA}$ *with* $\mathsf{C_{clean}} = \mathsf{C^{main}_{main,sub}}$.

Proof (sketch).[13] Let us assume that Γ is clean in the sense of $\mathsf{C_{clean}}$.

Let $(\mathsf{ad},\mathsf{ct})$ be the challenge (or last) message. If $(\mathsf{ad},\mathsf{ct})$ belongs to $\mathsf{ARCAD_{main}}$, then $\mathsf{main}(\Gamma)$ is $\mathsf{C_{main}}$-clean. The outcome of $\mathsf{main}(\Gamma)$ and Γ is the same. Due to the $\mathsf{C_{main}}$-$\mathsf{IND\text{-}CCA}$ security of $\mathsf{ARCAD_{main}}$, the advantage in Γ is negligible. Let us now assume that $(\mathsf{ad},\mathsf{ct})$ belongs to $\mathsf{ARCAD_{sub}}$.

$\mathsf{C}^{e,\mathsf{c}}_{\mathsf{C_{main}}}$ indicates if the $\mathsf{ARCAD_{main}}$ message of pair (e,c) can be replaced by the encryption of something random to produce the same result, except with negligible probability. In this case, $\mathsf{sub}_{e,\mathsf{c}}(\Gamma)$ produces the same outcome as Γ and $\mathsf{C_{clean}}$ implies that it must be $\mathsf{C_{sub}}$-clean. Due to the $\mathsf{C_{sub}}$-$\mathsf{IND\text{-}CCA}$ security of $\mathsf{ARCAD_{sub}}$, the advantage in Γ is negligible.

Similarly, if $\mathsf{C}^{e,\mathsf{c}}_{\mathsf{C_{main}}}(\Gamma)$ does not hold, $\mathsf{C_{clean}}$ implies that $\mathsf{sub}'_{e,\mathsf{c}}(\Gamma)$ is clean. It produces the same outcome as Γ. Due to the $\mathsf{C_{sub}}$-$\mathsf{IND\text{-}CCA}$ security of $\mathsf{ARCAD_{sub}}$, the advantage in Γ is negligible. □

[13] More details are provided in the full version [4].

In the FORGE game, we replace the $\mathsf{C}_{\mathsf{trivial}}$ predicate. Typically, by taking $\mathsf{C}_{\mathsf{main}}$ as the predicate that tests if the last $(\mathsf{ad}, \mathsf{ct})$ message is a trivial forgery and by taking $\mathsf{C}_{\mathsf{sub}}$ as the predicate that additionally tests if no $\mathsf{EXP}_{\mathsf{st}}$ occurred, the $\mathsf{C}^{\mathsf{clean}}_{\mathsf{main,sub}}$ predicate defines a new FORGE notion for $\mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{DV}}, \mathsf{EtH})$. More generally, if $\mathsf{ARCAD}_{\mathsf{main}}$ is $\mathsf{C}_{\mathsf{main}}$-FORGE-secure and $\mathsf{ARCAD}_{\mathsf{sub}}$ is $\mathsf{C}_{\mathsf{sub}}$-FORGE-secure, we would like to have $\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$-FORGE-security.

Game $\mathsf{FORGE}^{*\mathcal{A}}_{\mathsf{C}_{\mathsf{clean}}}(1^\lambda)$
1: $\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
2: $\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
3: $(\mathsf{P}, \mathsf{ad}, \mathsf{ct}) \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP}_{\mathsf{st}}, \mathsf{EXP}_{\mathsf{pt}}}(z)$
4: **if** one participant (or both) is NOT in a matching status **then return** 0
5: $\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, \mathsf{ad}, \mathsf{ct}) \to \mathsf{acc}$
6: **if** $\mathsf{acc} = \mathsf{false}$ **then return** 0
7: **if** $\neg\mathsf{C}_{\mathsf{clean}}$ **then return** 0
8: **if** we can parse $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}} = (\mathsf{seq}_1, (\mathsf{ad}, \mathsf{ct}))$ and $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}} = (\mathsf{seq}_1, \mathsf{seq}_2, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_3)$ **then return** 0
9: **return** 1

Fig. 7. Relaxed FORGE Security.

We almost have the reduction but there is something missing. Namely, a forgery for hybridARCAD in Γ may not be a forgery for neither $\mathsf{ARCAD}_{\mathsf{main}}$ in $\mathsf{main}(\Gamma)$ nor $\mathsf{ARCAD}_{\mathsf{sub}}$ in $\mathsf{sub}_{e,c}(\Gamma)$. This happens if the adversary in Γ drops the delivery of the last messages in a sub scheme. We relax FORGE-security using the FORGE^* game in Fig. 7. Only Steps 4 and 8 are new. Our chain strengthening in Sect. 3 can later make the protocols fully FORGE-secure. We easily prove the following result.

Lemma 24. *If* $\mathsf{ARCAD}_{\mathsf{main}}$ *is* $\mathsf{C}_{\mathsf{clean}}$-IND-CCA-*secure and* $\mathsf{C}_{\mathsf{main}}$-$\mathsf{FORGE}^*$-*secure and if* $\mathsf{ARCAD}_{\mathsf{sub}}$ *is* $\mathsf{C}_{\mathsf{sub}}$-$\mathsf{FORGE}^*$-*secure, then* hybridARCAD *is* $\mathsf{C}_{\mathsf{hybrid}}$-$\mathsf{FORGE}^*$*, where* $\mathsf{C}_{\mathsf{hybrid}} = \mathsf{C}^{\mathsf{clean}}_{\mathsf{main,sub}}$.

Proof (sketch).[14] If $(\mathsf{ad}, \mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{main}}$ and $\Gamma = \mathsf{FORGE}^*$ succeeds to return 1, then $\mathsf{C}_{\mathsf{main}}(\mathsf{main}(\Gamma))$ holds and $\mathsf{main}(\Gamma)$ succeeds to return 1 as well. Similarly, if $(\mathsf{ad}, \mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{sub}}$ and Γ returns 1, then, depending on $\mathsf{C}^{e,c}_{\mathsf{clean}}(\Gamma)$, either $\mathsf{C}_{\mathsf{sub}}(\mathsf{sub}_{e,c}(\Gamma))$ or $\mathsf{C}_{\mathsf{sub}}(\mathsf{sub}'_{e,c}(\Gamma))$ holds, and either game succeeds to return 1 (thanks to IND-CCA security in the latter case). Applying FORGE^* security of those protocols, this occurs with negligible probability. □

What FORGE^* security does not guarantee is that some forgeries in a sub-scheme may occur in the far future, due to state exposure. Fortunately, our protocol mitigates this problem by making sure that old sub-protocols become obsolete. Indeed, our protocol makes sure that sent messages always have an increasing sequence of (e, c) pairs, and the same for received messages. Hence, we

[14] More details are provided in the full version [4].

cannot have a forgery with an old (e, c) pair. Another problem which is explicit in Step 8 of the game is that the adversary may prevent P from receiving a sequence seq_2 sent from $\overline{\mathsf{P}}$ (namely in a sub-protocol). In Sect. 3, making the protocol r-RECOVER-secure fixes both problems. (See Lemma 26.) Hence, we will obtain FORGE-security.

4.4 Security-Aware Hybrid Construction

In this section, we apply our results from Sect. 3.3 to our hybrid constructions.

Lemma 25. *Let* $\mathsf{C}_{\mathsf{clean}} \in \{\mathsf{C}_{\mathsf{trivial}}, \mathsf{C}_{\mathsf{noexp}}\}$ *and* $\mathsf{ARCAD}_1 = \mathsf{chain}(\mathsf{ARCAD}_0)$*. If* ARCAD_0 *is* $\mathsf{C}_{\mathsf{clean}}$*-FORGE-secure (resp.* $\mathsf{C}_{\mathsf{clean}}$*-FORGE***-secure), then* ARCAD_1 *is* $\mathsf{C}_{\mathsf{clean}}$*-FORGE-secure (resp.* $\mathsf{C}_{\mathsf{clean}}$*-FORGE***-secure).*

Proof. We reduce an adversary playing the FORGE game with ARCAD_1 to an adversary playing the FORGE game with ARCAD_0 by simulating the hashings. ARCAD_1 is an extension of ARCAD_0 such that an ARCAD_1 message $(\mathsf{ad}, (\mathsf{ct}', h, \mathsf{ack}))$ is equivalent to an ARCAD_0 message $((\mathsf{ad}, h, \mathsf{ack}), \mathsf{ct}')$. It is just reordering $(\mathsf{ad}, \mathsf{ct})$. Hence, a forgery for ARCAD_1 must be a forgery for ARCAD_0. FORGE*-security works the same. □

Lemma 26. *Given* $\mathsf{ARCAD}_{\mathsf{main}}$ *and* $\mathsf{ARCAD}_{\mathsf{sub}}$*, let*

$$\mathsf{ARCAD}_0 = \mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{main}}, \mathsf{ARCAD}_{\mathsf{sub}}) \ , \ \mathsf{ARCAD}_1 = \mathsf{chain}(\mathsf{ARCAD}_0)$$

If $\mathsf{ARCAD}_{\mathsf{main}}$ *is* $\mathsf{C}_{\mathsf{clean}}$*-IND-CCA-secure and* $\mathsf{C}_{\mathsf{main}}$*-FORGE***-secure and* $\mathsf{ARCAD}_{\mathsf{sub}}$ *is* $\mathsf{C}_{\mathsf{sub}}$*-FORGE***-secure, then* ARCAD_1 *is* $\mathsf{C}^{\mathsf{clean}}_{\mathsf{main},\mathsf{sub}}$*-FORGE***-secure. If* H *is additionally collision-resistant, then* ARCAD_1 *is* $\mathsf{C}^{\mathsf{clean}}_{\mathsf{main},\mathsf{sub}}$*-FORGE-secure.*

Proof. Due to Lemma 24, $\mathsf{C}^{\mathsf{clean}}_{\mathsf{main},\mathsf{sub}}$-FORGE*-security works like in the previous result. To extend to $\mathsf{C}^{\mathsf{clean}}_{\mathsf{main},\mathsf{sub}}$-FORGE-security, we just observe that ARCAD_1 is r-RECOVER-secure due to Lemma 18. We thus deduce $\mathsf{seq}_2 = \perp$ from having $\mathsf{receive}^{\mathsf{P}}_{\mathsf{ct}} = (\mathsf{seq}_1, (\mathsf{ad}, \mathsf{ct}))$ and $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}} = (\mathsf{seq}_1, \mathsf{seq}_2, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_3)$. Hence, we have a full forgery, except with negligible probability. □

Lemma 27. *Let* $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}}$*,* $\mathsf{C}_{\mathsf{ratchet}}$*,* $\mathsf{C}_{\mathsf{noexp}}$*, or* $\mathsf{C}^{\mathsf{S}}_{\mathsf{tforge}}$ *(*$\mathsf{t} = \mathsf{trivial}$ *or* $\perp\}$*,* $\mathsf{S} = \mathsf{P}_{\mathsf{test}}$ *or* $\{\mathsf{A}, \mathsf{B}\}$*), If* ARCAD_0 *is* $\mathsf{C}_{\mathsf{clean}}$*-IND-CCA-secure, then* ARCAD_1 *is* $\mathsf{C}_{\mathsf{clean}}$*-IND-CCA-secure.*

Proof. We reduce an adversary playing the IND-CCA game with ARCAD_1 to an adversary playing the IND-CCA game with ARCAD_0 by simulating the hashings. We easily see that the cleanness is the same and that the simulation is perfect. □

We easily extend this result to hybrid constructions. We conclude with our final result.

Theorem 28. *Given* $\mathsf{ARCAD_{main}}$ *and* $\mathsf{ARCAD_{sub}}$*, let*

$$\mathsf{ARCAD}_0 = \mathsf{hybrid}(\mathsf{ARCAD_{main}}, \mathsf{ARCAD_{sub}}) \ , \ \mathsf{ARCAD}_1 = \mathsf{chain}(\mathsf{ARCAD}_0)$$

We assume that 1. H *is collision-resistant; 2.* $\mathsf{ARCAD_{main}}$ *is* $\mathsf{C_{clean}}$-IND-CCA-*secure and* $\mathsf{C_{main}}$-FORGE*-*secure; 3.* $\mathsf{ARCAD_{sub}}$ *is* $\mathsf{C_{sub}}$-FORGE*-*secure and* $\mathsf{C'_{clean}}$-IND-CCA-*secure. Then,* ARCAD_1 *is 1.* r-RECOVER-*secure, 2.* s-RECOVER-*secure, 3.* $\mathsf{C^{clean}_{main,sub}}$-FORGE-*secure, 4.* $\mathsf{C^{clean}_{clean,clean'}}$-IND-CCA-*secure, 5. with acknowledgement extractor.*

Corollary 29. *Let* $\mathsf{ARCAD}_1 = \mathsf{chain}(\mathsf{hybrid}(\mathsf{ARCAD_{DV}}, \mathsf{EtH}))$ *(where* $\mathsf{ARCAD_{DV}}$ *is defined on Fig. 11) and let* $\mathsf{C_{clean}} = \mathsf{C_{leak}} \wedge \mathsf{C^{A,B}_{forge}}$. *With the assumptions from Theorem 30 and the* EtH *result [13, Th.2], if* H *is collision-resistant,* ARCAD_1 *is* $\mathsf{C^{clean}_{trivial,noexp}}$-FORGE-*secure,* $\mathsf{C^{clean}_{clean,sym}}$-IND-CCA-*secure, and with security-awareness.*

In particular, when a sender deduces an acknowledgment for his message m from a received message m', if he can make sure that m' is genuine and that no trivial exposure for m happened, then he can be sure that his message m is private, no matter what happened before or what will happen next.

5 Conclusion

We revisited the DV security model. We proposed an hybrid construction which would mostly use EtH and occasionally a stronger protocol, upon the choice of the sender, thus achieving on-demand ratcheting. Finally, we proposed the notion of security awareness to enable participants to have a better idea on the safety of their communication. We achieved what we think is the optimal awareness. Concretely, a participant is aware of which of his messages arrived to his counterpart when he sent the last received one. We make sure that any forgery (possibly due to exposure) would fork the chain of messages which is seen by both participants and result in making them unable to continue communication. We also make sure that assuming that the exposure history is known, participants can deduce which messages leaked.

A Implementations/Comparisons with Existing Protocols

We compare the performances of $\mathsf{ARCAD_{DV}}$ and EtH to other ratcheted messaging and key agreement protocols that have surfaced since 2018. In particular, we implemented five other schemes from the literature[15]. Namely, the bidirectional asynchronous key-agreement protocol BRKE by PR [10], the similar secure messaging protocol by JS [8], the secure messaging protocol by JMM [9] and a modularized version of two protocols by ACD [1].In ACD [1], the given protocols

[15] Our code is available at https://github.com/qantik/ratcheted.

are both with symmetric key cryptography ACD and public-key cryptography ACD-PK. We did not implement the DV protocol [7], as $\mathsf{ARCAD_{DV}}$ is a slightly modified version of DV, hence has identical performances.

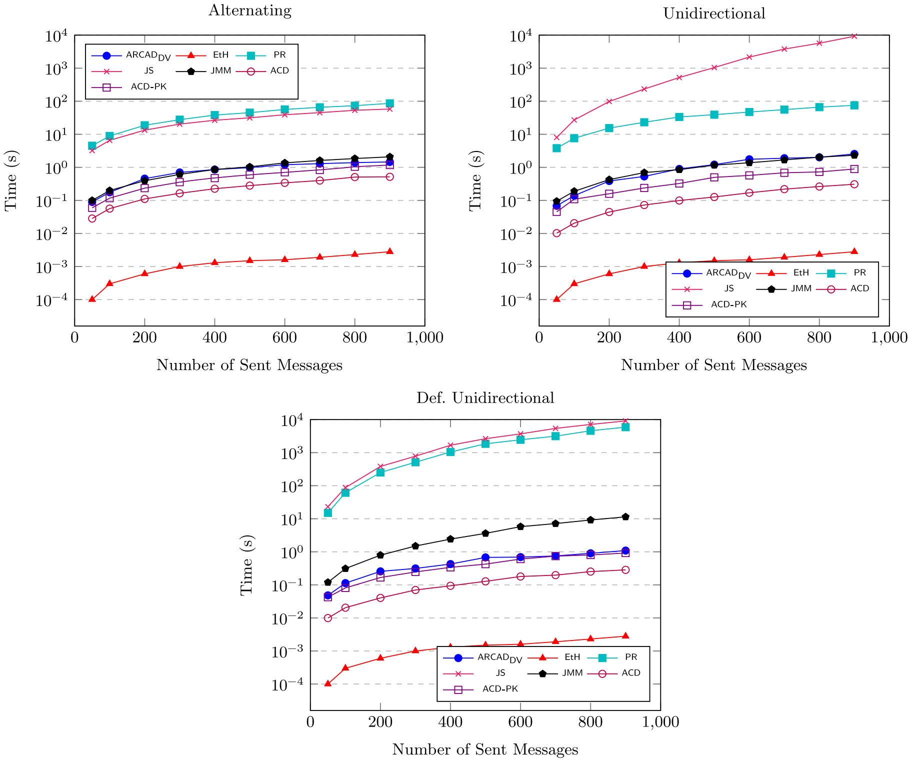

Fig. 8. Runtime Benchmarks The protocol in [10] is represented with PR; [8] with JS; [9] with JMM; and [1] with ACD and ACD-PK. ACD-PK is the public-key version with stronger security.

All the protocols were implemented in Go[16] and measured with its built-in benchmarking suite[17] on a regular fifth generation Intel Core i5 processor. In order to mitigate potential overheads garbage collection has been disabled for all runs. Go is comparable in speed to C/C++ though further performance gains are within reach when the protocols are re-implemented in the latter two. Additionally, some protocols deploy primitives for which no standard implementations exist, which is, for example, the case for the HIBE constructions used in the PR and JS protocols, making custom implementations necessary that can certainly be improved upon. For the deployed primitives, when we needed an

[16] https://golang.org/.
[17] https://golang.org/pkg/testing/.

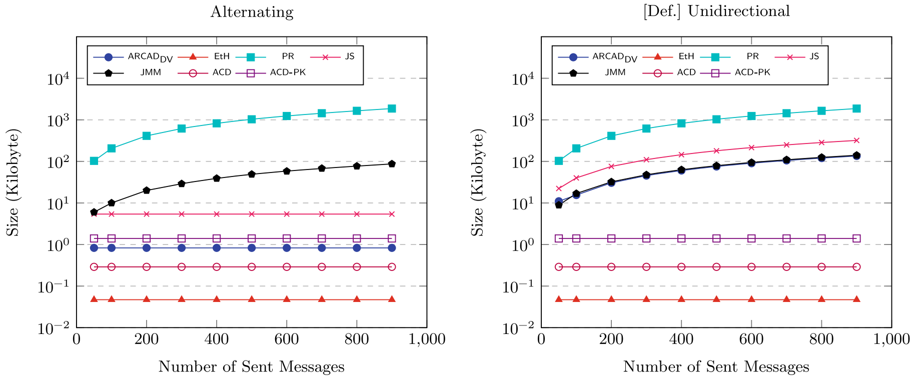

Fig. 9. State Size Benchmarks Due to the equivalent state sizes in unidirectional and deferred unidirectional traffic, one figure is omitted

AEAD scheme, we used AES-GCM. For public key cryptosystem, we used the elliptic curve version of ElGamal (ECIES); for the signature scheme, we used ECDSA. And, finally for the PRF-PRNG in [1] protocol, we used HKDF with SHA-256. Lastly, the protocols themselves may offer some room for performance tweaks.

The benchmarks can be categorized into two types as depicted in Fig. 8–9.

(a) Runtime designates the total required time to exchange n messages, ignoring potential latency that normally occurs in a network.
(b) State size shows the maximal size of a user state throughout the exchange of n messages.

A state is all the data that is kept in memory by a user. Each type itself is run on three canonical ways traffic can be shaped when two participants are communicating. In alternating traffic the parties are synchronized, i.e. take turns sending messages. In unidirectional traffic one participant first sends $\frac{n}{2}$ messages which are received by the partner who then sends the other half. Finally, in deferred unidirectional traffic both participants send $\frac{n}{2}$ messages before they start receiving. ACD-PK adds some public-key primitives to the double ratchet by ACD [1] to plug some post-compromise security gaps. These two variations serve as baselines to see how the metrics of a protocol can change when some of its internals are replaced or extended. Also note that due to the equivalent state sizes in unidirectional and deferred unidirectional traffic one figure is omitted.

As we can see, overall, the fastest protocol is EtH, followed by the two ACD protocols, then ARCAD$_{DV}$, then the JMM protocol, and lastly the strongest protocols PR and JS. ARCAD$_{DV}$ and JMM may be comparable except for deferred unidirectional communication.

The smallest state size is obtained with EtH. ARCAD$_{DV}$ performs well in terms of state size.

Clearly, $\mathsf{hybrid}(\mathsf{ARCAD_{DV}}, \mathsf{EtH})$ has performances which are weighted averages of the ones of $\mathsf{ARCAD_{DV}}$ and EtH, depending on the frequency of on-demand ratcheting.

B $\mathsf{ARCAD_{DV}}$ Formal Protocol

With slight modifications, we transform the DV protocol [7] into an ARCAD that we call $\mathsf{ARCAD_{DV}}$.

$\mathsf{ARCAD_{DV}}$ is based on a hash function H[18], a one-time symmetric cipher Sym[19], a digital signature scheme DSS[20], and a public-key cryptosystem PKC[21].

$\mathsf{ARCAD_{DV}}$, just as DV, consists of many modules which are built on top of each other. The "smallest" module is a "naive" signcryption scheme SC which can be of the form

$$\mathsf{SC.Enc}(\overbrace{\mathsf{sk_S}, \mathsf{pk_R}}^{\mathsf{st_S}}, \mathsf{ad}, \mathsf{pt}) = \mathsf{PKC.Enc}(\mathsf{pk_R}, (\mathsf{pt}, \mathsf{DSS.Sign}(\mathsf{sk_S}, (\mathsf{ad}, \mathsf{pt}))))$$

$$\mathsf{SC.Dec}(\underbrace{\mathsf{sk_R}, \mathsf{pk_S}}_{\mathsf{st_R}}, \mathsf{ad}, \mathsf{ct}) = \begin{bmatrix} (\mathsf{pt}, \sigma) \leftarrow \mathsf{PKC.Dec}(\mathsf{sk_R}, \mathsf{ct}) ; \\ \mathsf{DSS.Verify}(\mathsf{pk_S}, (\mathsf{ad}, \mathsf{pt}), \sigma) \ ? \ \mathsf{pt} \ : \ \bot \end{bmatrix}$$

SC extends to a multiple-state (and multiple-key) encryption called onion. It handles the the case where the states get accumulated during a sequential send or receive operation during the communication. It generates a secret key to encrypt a plaintext. This secret key is, then, secret shared and encrypted under different states so that if a state is exposed, its shares would still remain confidential. onion leads to a unidirectional scheme called uni where participants have fixed roles as either senders or receivers. The underlying idea of unidirectional communication is to let the sender generate the next send/receive states for the future exchange during the current send operation and transmit the next receive state to the receiver. These future states are shown as $\mathsf{st'_S}$ and $\mathsf{st'_R}$ in the second row of Fig. 10. After each uni.Send and uni.Rec operations, the states are completely flushed to ensure security.

Finally, unidirectional communication allow us to construct the bidirectional $\mathsf{ARCAD_{DV}}$ as shown in the last row of Fig. 10. Since the communication become bidirectional, the participant P also keeps states for receiving. More specifically, the sender generates a pair of fresh states and transmits the send state to the counterpart so that s/he can use it to send a reply to back to the sender with this states.

$\mathsf{ARCAD_{DV}}$ is depicted on Fig. 11.

---

[18] H uses a common key hk generated by H.Gen and an algorithm H.Eval.

[19] Sym uses a key of length Sym.kl, encrypts over the domain $\mathsf{Sym}.\mathcal{D}$ with algorithm Sym.Enc and decrypts with Sym.Dec.

[20] DSS uses a key generation DSS.Gen, a signing algorithm DSS.Sign, and a verification algorithm DSS.Verify.

[21] PKC uses a key generation PKC.Gen, an encryption algorithm PKC.Enc, and a decryption algorithm PKC.Dec.

Note that we removed some parts of the protocol which ensure r-RECOVER security. This is because the generic transformation in Sect. 3 which we apply on $\mathsf{ARCAD_{DV}}$ will restore it in a stronger and generic way.

We recall the security results.

Theorem 30 (Security of $\mathsf{ARCAD_{DV}}$ [7]). $\mathsf{ARCAD_{DV}}$ *is correct. If* $\mathsf{Sym.kl}(\lambda) = \Omega(\lambda)$*,* H *is collision-resistant,* DSS *is* SEF-OTCMA*,* PKC *is* IND-CCA*-secure, and* Sym *is* IND-OTCCA*-secure, then* $\mathsf{ARCAD_{DV}}$ *is* $\mathsf{C_{trivial}}$-FORGE*-secure,* $(\mathsf{C_{leak}} \wedge \mathsf{C^{A,B}_{forge}})$-IND-CCA*-secure and* PREDICT*-secure.*[22,23]

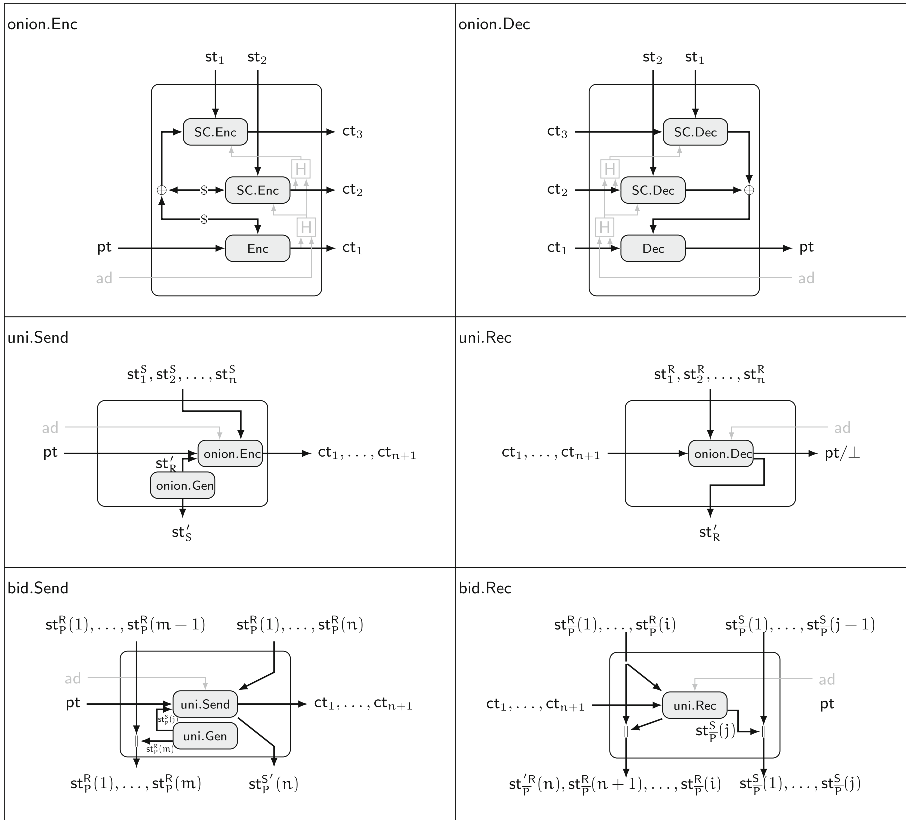

Fig. 10. High-level overview of the protocol described in Fig. 11

[22] SEF-OTCMA is the strong existential one-time chosen message attack. IND-OTCCA is the real-or-random indistinguishability under one-time chosen plaintext and chosen ciphertext attack. Their definitions are given in [7].

[23] Following Durak-Vaudenay [7], for a $\mathsf{C_{trivial}}$-FORGE-secure scheme, $(\mathsf{C_{leak}} \wedge \mathsf{C^{A,B}_{forge}})$-IND-CCA security is equivalent to $(\mathsf{C_{leak}} \wedge \mathsf{C^{A,B}_{trivial\ forge}})$-IND-CCA security, which corresponds to the "sub-optimal" security in Table 1.

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: $\quad \mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
7: $\quad \mathsf{ct}_i \leftarrow \mathsf{SC.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k_i)$
8: **end for**
9: **return** $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$

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: $\quad \mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
6: $\quad \mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \rightarrow k_i$
7: $\quad$ **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** pt

uni.Init(1^λ)
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** $(\mathsf{st}_S, \mathsf{st}_R)$

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** $(\mathsf{st}'_S, \vec{\mathsf{ct}})$

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**
3: $\quad$ **return** $(\mathsf{false}, \perp, \perp)$
4: **end if**
5: parse $\mathsf{pt}' = (\mathsf{st}'_R, \mathsf{pt})$
6: **return** $(\mathsf{true}, \mathsf{st}'_R, \mathsf{pt})$

$\mathsf{ARCAD_{DV}.Setup}(1^\lambda)$
1: $\mathsf{H.Gen}(1^\lambda) \xrightarrow{\$} \mathsf{hk}$
2: **return** hk

$\mathsf{ARCAD_{DV}.Gen}(1^\lambda, \mathsf{hk})$
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** $(\mathsf{sk}, \mathsf{pk})$

$\mathsf{ARCAD_{DV}.Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_P, \mathsf{pk}_{\overline{P}}, P)$
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: $\mathsf{st}_P \leftarrow (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send}}), (\mathsf{st}_P^{\mathsf{rec}}))$
6: **return** st_P

$\mathsf{ARCAD_{DV}.Send}(\mathsf{st}_P, \mathsf{ad}, \mathsf{pt})$
1: parse $\mathsf{st}_P = (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v}))$
2: $\mathsf{uni.Init}(1^\lambda) \xrightarrow{\$} (\mathsf{st}_{S\mathsf{new}}, \mathsf{st}_P^{\mathsf{rec},v+1})$ ▷ append a new receive state to the $\mathsf{st}_P^{\mathsf{rec}}$ list
3: $\mathsf{pt}' \leftarrow (\mathsf{st}_{S\mathsf{new}}, \mathsf{pt})$ ▷ then, $\mathsf{st}_{S\mathsf{new}}$ is erased to avoid leaking
4: 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
5: $\mathsf{uni.Send}(1^\lambda, \mathsf{hk}, \mathsf{st}_P^{\mathsf{send},i}, \ldots, \mathsf{st}_P^{\mathsf{send},u}, \mathsf{ad}, \mathsf{pt}') \xrightarrow{\$} (\mathsf{st}_P^{\mathsf{send},u}, \mathsf{ct})$ ▷ update $\mathsf{st}_P^{\mathsf{send},u}$
6: $\mathsf{st}_P^{\mathsf{send},i}, \ldots, \mathsf{st}_P^{\mathsf{send},u-1} \leftarrow \perp$ ▷ flush the send state list: only $\mathsf{st}_P^{\mathsf{send},u}$ remains
7: $\mathsf{st}'_P \leftarrow (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v+1}))$
8: **return** $(\mathsf{st}'_P, \mathsf{ct})$

$\mathsf{ARCAD_{DV}.Receive}(\mathsf{st}_P, \mathsf{ad}, \mathsf{ct})$
9: parse $\mathsf{st}_P = (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v}))$
10: set $n+1$ to the number of components in ct ▷ the onion has n layers
11: set i to the smallest index such that $\mathsf{st}_P^{\mathsf{rec},i} \neq \perp$
12: **if** $i + n - 1 > v$ **then return** $(\mathsf{false}, \mathsf{st}_P, \perp)$
13: $\mathsf{uni.Receive}(\mathsf{hk}, \mathsf{st}_P^{\mathsf{rec},i}, \ldots, \mathsf{st}_P^{\mathsf{rec},i+n-1}, \mathsf{ad}, \mathsf{ct}) \rightarrow (\mathsf{acc}, \mathsf{st}_P'^{\mathsf{rec},i+n-1}, \mathsf{pt}')$
14: **if** $\mathsf{acc} = \mathsf{false}$ **then return** $(\mathsf{false}, \mathsf{st}_P, \perp)$
15: parse $\mathsf{pt}' = (\mathsf{st}_P^{\mathsf{send},u+1}, \mathsf{pt})$ ▷ a new send state is added in the list
16: $\mathsf{st}_P^{\mathsf{rec},i}, \ldots, \mathsf{st}_P^{\mathsf{rec},i+n-2} \leftarrow \perp$ ▷ update stage 1: $n-1$ entries of $\mathsf{st}_P^{\mathsf{rec}}$ were erased
17: $\mathsf{st}_P^{\mathsf{rec},i+n-1} \leftarrow \mathsf{st}_P'^{\mathsf{rec},i+n-1}$ ▷ update stage 2: update $\mathsf{st}_P^{\mathsf{rec},i+n-1}$
18: $\mathsf{st}'_P \leftarrow (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send},1}, \ldots, \mathsf{st}_P^{\mathsf{send},u+1}), (\mathsf{st}_P^{\mathsf{rec},1}, \ldots, \mathsf{st}_P^{\mathsf{rec},v}))$
19: **return** $(\mathsf{acc}, \mathsf{st}'_P, \mathsf{pt})$

Fig. 11. $\mathsf{ARCAD_{DV}}$ Protocol Adapted from DV [7] without RECOVER-Security.

References

1. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: security notions, proofs, and modularization for the signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 129–158. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_5

2. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: the security of messaging. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10403, pp. 619–650. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63697-9_21
3. Borisov, N., Goldberg, I., Brewer, E.: Off-the-record communication, or, why not to use PGP. In: Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, New York, NY, USA, pp. 77–84. ACM (2004)
4. Caforio, A., Betül Durak, F., Vaudenay, S.: On-demand ratcheting with security awareness. IACR Eprint 2019/965. https://eprint.iacr.org/2019/965.pdf
5. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 451–466, April 2017
6. Cohn-Gordon, K., Cremers, C., Garratt, L.: On post-compromise security. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), pp. 164–178, June 2016
7. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. In: Attrapadung, N., Yagi, T. (eds.) IWSEC 2019. LNCS, vol. 11689, pp. 343–362. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26834-3_20
8. Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: the safety of messaging. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10991, pp. 33–62. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_2
9. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 159–188. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_6
10. Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10991, pp. 3–32. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_1
11. Open Whisper Systems. Signal protocol library for Java/Android. GitHub repository (2017). https://github.com/WhisperSystems/libsignal-protocol-java
12. Unger, N., et al.: SoK: secure messaging. In: 2015 IEEE Symposium on Security and Privacy, pp. 232–249, May 2015
13. Yan, H., Vaudenay, S.: Symmetric asynchronous ratcheted communication with associated data. In: Aoki, K., Kanaoka, A. (eds.) IWSEC 2020. LNCS, vol. 12231, pp. 184–204. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58208-1_11