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

Abstract. Ratcheting communication strengthens privacy, specifically in the presence of internal state exposures or random coin corruptions. This is called post-compromise security. There have been several such secure protocols proposed in the last few years. The strongest level of security comes with a high cost, because of the need for HIBE or at least public-key cryptography.

In this paper, we first design a lightweight protocol called liteARCAD which is solely based on symmetric cryptography, hence only forward secure.

We then present a generic hybrid protocol allowing to compose any two protocols so that the sender can select which of the two protocols to use. When composing liteARCAD and a post-compromise secure protocol, the sender can decide to ratchet or not. For instance, the sender can ratchet once a while, or after letting his device unattended. When doing so with infrequent ratchet, we obtain the strongest security at the price of efficient symmetric cryptography.

We then propose the notion of security awareness. This lets a sender learns, after a while, if his message was safely received (i.e. if it was received and if no adversary can decrypt it, except from trivial attacks) and that no finished active attack occurred (i.e. active attack must continue forever or be detected). We finally propose a generic strengthening to add security awareness to any 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. Moreover, the protocols do not have any control over the time that participants send messages, and, by the same token, 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 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.*

Even though forward secrecy or post-compromise security have been integrated for a while, there have been no formal definitions and protocols provably secure under such notions until recently. After the first formal definitions of ratcheting security given by Bellare et al. [2], many subsequent studies about secure protocols have followed with different security levels and primitives [1, 6–9]. Some of these results are about key-exchange while others study secure messaging. Since secure ratcheted messaging can reduce 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, 10]. The security of Signal was studied by Cohn-Gordon et al. [4]. Unger et al. [11] surveyed many techniques in ratcheting. More recently, Alwen et al. [1] formalized the concept of "double ratcheting" from Signal.

Cohn-Gordon et al. [5] 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) [9] designed a protocol at CRYPTO 2018 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) [7] 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) [6] proposed a protocol called BARK with slightly lower security[3] but relying on neither HIBE nor random oracle. 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) [8] proposed concurrently and independently a protocol with security between optimal security and the security of BARK.[4] 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 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.

We summarize these results in Table 1. The first four rows are based on Durak-Vaudenay [6, Table 1]. The last rows of the table will be discussed shortly.

We are mostly interested in the DV model [6]. 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 a participant can no longer accept messages from his counterpart after he receives a forged message. Actually, even though FORGE security avoids non-trivial 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). This guarantees that the attack is eventually detected by Bob if he is still alive. This property is particularly nice under two assumptions: a lower protocol makes sure that messages come in the right order (as an out of order message is treated as a forgery); an upper protocol can reset a broken communication after the attack is detected.

What makes the DV model simple is that all technicalities are hidden in a *cleanness* notion. That is, the adversary wins only when the attack scenario trace is "clean". The cleanness predicate eliminates trivial attack strategies. 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.

[3] More precisely, the security is called "*sub-optimal*" as detailed later.
[4] We call this security level "*near-optimal*".

Our objectives. In this paper, we study various security notions for the asynchronous ratcheted communication with additional data which we call ARCAD in short. Experience showed that when we want the protocols to be highly secure, we have to give up the efficiency of the protocol and rely on heavy tools. Equivalently, when we want protocols to perform fast, then the security should be lowered to a reasonable level. In real-world applications, the developers do not want to over-complicate or under-perform. At the same time, users seek usability and strong privacy. Therefore, we believe that the confidentiality level of sending messages should be set on demand by the sender or could be tuned by the application itself based on time intervals. For instance, if the users are exchanging hundreds of messages per day, there may not be any real need for ratcheting with strongly secure protocols with healing. Instead, a lighter version of the protocol only with forward secrecy (which is symmetric-key ratcheting) should be enough for security. Alternatively, the sender could ask for healing only when an exposure is likely (e.g. because his device was taken by a third party, remained unattended for a while etc.) or just once a day. Healing may actually scarcely occur in intensive communication. Therefore, we construct a protocol called hybridARCAD that runs a healing ratchet *on demand*.

We also define a security notion by improving RECOVER-security from DV [6]. This security implies that when a participant receives a forgery, he should not be able to receive genuine messages any longer. What is also needed is that the participant who has received a forgery should not be able to *send* messages to his counterpart either. This makes sure that man-in-the-middle attacks are eventually detected.

Another interesting notion is given in Alwen et al. [1] as *immediate decryption.* It allows receiving messages even though some previous ones were not received. Concretely, it is done by keeping all keys in the state of the receiver to decrypt messages until they are needed. Obviously, it has some consequences with regards to security. Namely, when an adversary prevents a message from being delivered, the key remains in the receiver state and this key may be stolen in the future. Hence, even though communication can continue, the participants have no guarantee about the safety status of this message until it is received. For instance, we can imagine that the adversary may collect a few sensitive messages (e.g. all the large ones as they may contain media content) and decrypt them all after exposure of the receiver state. Immediate decryption is nice when the communication network is not reliable and messages may come in a different order at random. However, we believe that this problem can be solved by independent techniques and need not to be addressed by the cryptographic protocol. More precisely, messages can be encapsulated in containers which makes sure that if a message is missing, it can be requested for a second delivery and the received messages can be held until the sequence is reconstructed with no loss. Adding reliability on the communication channel can indeed be solved by a lower-level protocol. Hence, we *do not* provide immediate-decryption security in our constructions. Instead, we focus on a very important aspect of secure messaging protocol which is described as security awareness. To defeat communication interruption due to a message loss or a forgery, we will propose a way for participants to repair it.

Our contributions. We first construct liteARCAD, a weakly secure but efficient protocol which is solely based on symmetric cryptography.

We give an *on-demand ratcheting* protocol by generically composing any strong protocol with any weakly secure but fast one. A concrete instance is built from $\mathsf{ARCAD_{DV}}$ and liteARCAD protocols where the former is the modified version of BARK for ratcheted communication and the latter is a symmetric-cryptography-based variant of BARK. $\mathsf{ARCAD_{DV}}$ is post-compromise secure while liteARCAD is only forward secure. Our on-demand ratcheting protocol is actually a generic hybrid construction (denoted hybrid) which can be based on any two protocols. The sending participant sets which of the two to use in our on-demand ratcheting protocol. The hybrid construction further offers a solution to restore broken communication.

Another important contribution of the present paper is that we implemented PR, JS, $\mathsf{ARCAD_{DV}}$, JMM, ACD, ACD-PK, together with liteARCAD. We observe that liteARCAD is the fastest one. Our goal is to offer a high level of security with the performances of liteARCAD. We reach it with on-demand ratcheting when the participant demands healing scarcely.

Our final contribution is to 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 BARK. 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. We call it a *cleanness extractor*.

We propose a generic strengthening (called blockchain) of protocol to obtain r-RECOVER and s-RECOVER security on the top of any protocol. Applying it to our on-demand ratcheting protocol, we obtain our final protocol ARCAD.

We provide a comparison of all the protocols with r-RECOVER-security, s-RECOVER-security, acknowledgment extractor and cleanness extractor in Table 1.

Table 1: Comparison of Several Protocols with our ARCAD = blockchain(hybrid($\mathsf{ARCAD_{DV}}$, liteARCAD)) from Cor. 33 in Section 5.3: security level; complexity for exchanging n messages; types of coin-leakage security; plain model (i.e. no random oracle); PKC or less (i.e. no HIBE). BARK and $\mathsf{ARCAD_{DV}}$ have identical characteristics. The terms "optimal", "near-optimal", and "sub-optimal" from Durak-Vaudenay [6] are defined on p. 2. "Pragmatic" degrades a bit security to offer on-demand ratcheting. "Id-optimal" is optimal among protocols with immediate decryption.

	PR [9]	JS [7]	JMM [8]	BARK [6]	ACD-PK [1]	ARCAD
Security	optimal	optimal	near-optimal	sub-optimal	id-optimal	pragmatic
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	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

To summarize, our contributions are:

- we design liteARCAD with provable (forward) 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, JMM, ACD, ACD-PS, liteARCAD and $\mathsf{ARCAD_{DV}}$ protocols and compare their performances;
- we define the notion of security awareness, construct a generic protocol strengthening called blockchain, and prove its security.

Notations. 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 Section 2, we revisit the preliminary notions from Durak-Vaudenay [6] and Alwen-Coretti-Dodis [1]. They all are essential to be able to follow our results. In Section 3, we define a new protocol called on-demand ratcheting with better performance than state-of the-art. In Section 4, we present our implementation results with the figures comparing various protocols with our on-demand ratcheting protocol. Finally, in Section 5, we define a new notion named security awareness and build a protocol with regard to the notion.

2 Preliminaries

2.1 ARCAD Definition and Security

In this section, we recall the DV model [6] and we slightly adapt to define asynchronous ratcheted communication with additional data denoted as ARCAD (instead of bidirectional asynchronous ratcheted key agreement BARK). The difference between BARK and ARCAD is the same as the difference between KEM and cryptosystems: pt is input to the Send (instead of output). Additionally, we treat associated data ad to authenticate. Like DV [6][5], 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. All definitions with the reference [6] are unchanged except for possible necessary notation changes. The other definitions in this section are straightforward adaptations to fit ARCAD.

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.*
- $\mathsf{Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_\mathsf{P}, \mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{P}) \to \mathsf{st}_\mathsf{P}$*: This sets up the initial state* st_P *of* P *given his secret key, his role* role_P*, 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 updated state* st'_P*.*
- $\mathsf{Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct}) \to (\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{pt})$*: 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.[6]

For convenience, we define the following initialization procedure for all games. It returns the initial states of A *and* B *as well as some public information* z.

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

None of our security games care 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": we only need to define Initall. We will do so with liteARCAD and prove it a "secure ARCAD" by slight abuse of definition.

In what follows we only consider an ARCAD protocol.

[5] Proceedings version.

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

Definition 2 (Correctness of ARCAD). *Consider the correctness game given on Fig. 1.*[7] *We say that an* ARCAD *protocol is* correct *if for all sequence* sched *of tuples of the form* (P, "send", ad, pt) *or* (P, "rec")*, the game never returns 1. Namely,*

– *at each stage, for each* P*,* $\mathsf{received}^{\mathsf{P}}_{\mathsf{pt}}$ *is prefix of* $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{pt}}$[8]*;*
– *each* RATCH(P, "rec") *call returns* acc = true.

```
Oracle RATCH(P, "rec", ad, ct)
  ct_P ← ct
  ad_P ← ad
  (acc, st'_P, pt_P) ← Receive(st_P, ad_P, ct_P)
  if acc then
      st_P ← st'_P
      append (ad_P, pt_P) to received^P_pt
      append (ad_P, ct_P) to received^P_ct
  end if
  return acc

Oracle RATCH(P, "send", ad, pt)
  pt_P ← pt
  ad_P ← ad
  (st'_P, ct_P) ← Send(st_P, ad_P, pt_P)
  st_P ← st'_P
  append (ad_P, pt_P) to sent^P_pt
  append (ad_P, ct_P) to sent^P_ct
  return ct_P

Game Correctness(sched)
  set all sent^*_* and received^*_* to ∅
  Setup(1^λ) →$ pp
  Initall(1^λ, pp) →$ (st_A, st_B, z)
  initialize two FIFO lists incoming_A, incoming_B ← ∅
  i ← 0
  loop
      i ← i + 1
      if sched_i of form (P, "rec") then
          if incoming_P is empty then return 0
          pull (ad, ct) from incoming_P
          acc ← RATCH(P, "rec", ad, ct)
          if acc = false then return 1
      else
          parse sched_i = (P, "send", ad, pt)
          ct ← RATCH(P, "send", ad, pt)
          push (ad, ct) to incoming_P̄
      end if
      if received^A_pt not prefix of sent^B_pt then return 1
      if received^B_pt not prefix of sent^A_pt then return 1
  end loop
```

Fig. 1: The Correctness Game of ARCAD Protocol.

For all global variables ν in the game such as $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$, st_{P}, or ct_{P}, we denote the value of ν at time t by $\nu(t)$. 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.

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. We give a brief description of the DV security notions [6] as follows.

FORGE-security: It makes sure that there is no forgery, except trivial ones.

r-RECOVER-security[9]**:** 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.

PREDICT-security: No message can be received before it was sent.

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

[7] 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".

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

[9] It is called RECOVER-security in [6]. We call it r-RECOVER because we will enrich it with an s-RECOVER notion in Section 5.1.

Definition 3 (Matching status [6]). *We say that* P *is in a* matching status *at time* t *for* P *if*

1. *at any moment of the game before time* t *for* P*,* $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$ *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). The $\mathsf{PREDICT}$-security will become useful to reduce this definition 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 $\bar{\mathsf{t}}$ for $\overline{\mathsf{P}}$.

Definition 4 (Corresponding RATCH calls [6]). *Let* P *be a participant. We consider only the* $\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, ., .)$ *calls by* P *returning* true*. We say that the* i^{th} *call* corresponds *to the* j^{th} $\mathsf{RATCH}(\overline{\mathsf{P}}, \text{“send”}, ., .)$ *call by* $\overline{\mathsf{P}}$ *if* $\mathsf{i} = \mathsf{j}$ *and* P *is in matching status at the time of this* i^{th} *accepting* $\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, ., .)$ *call.*

Definition 5 (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 6 (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*.*

Definition 7 (Direct leakage). *Let* t *be a time and* P *be a participant. We say that* $\mathsf{pt}_{\mathsf{P}}(\mathsf{t})$ *has a* direct leakage *if one of the following conditions is satisfied:*

- *The last* RATCH *call before time* t *is a* $\mathsf{RATCH}(\mathsf{P}, \text{“send”}, \mathsf{ad}, \mathsf{pt})$ *call by the adversary defining* $\mathsf{pt}_{\mathsf{P}}(\mathsf{t}) = \mathsf{pt}$.
- *There is an* $\mathsf{EXP}_{\mathsf{pt}}(\mathsf{P})$ *at a time* t_{e} *such that the last* RATCH *call which is executed by* P *before time* t *and the last* RATCH *call which is executed by* P *before time* t_{e} *are the same.*
- P *is in a matching status and there exists* $\mathsf{t}_0 \leqslant \mathsf{t}_{\mathsf{e}} \leqslant \mathsf{t}_{\mathsf{RATCH}} \leqslant \mathsf{t}$ *and* $\bar{\mathsf{t}}$ *such that time* t *originates from time* $\bar{\mathsf{t}}$*; time* $\bar{\mathsf{t}}$ *originates from time* t_0*; there is one* $\mathsf{EXP}_{\mathsf{st}}(\mathsf{P})$ *at time* t_{e}*; there is one* $\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, ., .)$ *at time* $\mathsf{t}_{\mathsf{RATCH}}$*; and there is no* $\mathsf{RATCH}(\mathsf{P}, ., ., .)$ *between time* $\mathsf{t}_{\mathsf{RATCH}}$ *and time* t*.*

The first condition is specific to ARCAD: Obviously, an adversarial RATCH send call counts as an $\mathsf{EXP}_{\mathsf{pt}}$ call.

Definition 8 (Indirect leakage [6]). *We consider a time* t *and a participant* P*. Let* $\mathsf{t}_{\mathsf{RATCH}}$ *be the time of the last successful* RATCH *call and* role *be its input role. We say that* $\mathsf{pt}_{\mathsf{P}}(\mathsf{t})$ *has an* indirect leakage *if* P *is in matching status at time* t *and one of the following conditions is satisfied*

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

The $\mathsf{IND\text{-}CCA}$ security is relative to a $\mathsf{C}_{\mathsf{clean}}$ predicate. We consider several predicates as defined in the DV model [6]:

$\mathsf{C}_{\mathsf{leak}}$: $\mathsf{pt}_{\mathsf{test}}(\mathsf{t}_{\mathsf{test}})$ has no direct or indirect leakage.
$\mathsf{C}^{\mathsf{S}}_{\mathsf{tforge}}$: (with $\mathsf{t} = \mathsf{trivial}$ or t void, and $\mathsf{S} = \{\mathsf{P}_{\mathsf{test}}\}$ or $\mathsf{S} = \{\mathsf{A}, \mathsf{B}\}$)
no $\mathsf{P} \in \mathsf{S}$ received any t-forgery until having seen $(\mathsf{ad}, \mathsf{ct})_{\mathsf{test}}$.
$\mathsf{C}_{\mathsf{ratchet}}$: $(\mathsf{ad}, \mathsf{ct})_{\mathsf{test}}$ was sent by a participant $\mathsf{P}_{\mathsf{test}}$, then received and accepted by $\overline{\mathsf{P}}_{\mathsf{test}}$, then some $(\mathsf{ad}', \mathsf{ct}')$ were sent by $\overline{\mathsf{P}}_{\mathsf{test}}$, then $(\mathsf{ad}', \mathsf{ct}')$ were received and accepted by $\mathsf{P}_{\mathsf{test}}$.

In Table 1, "*optimal*" security refers to $\mathsf{C_{clean}} = \mathsf{C_{leak}} \wedge \mathsf{C^{P_{test}}_{trivial\ forge}}$ and "*sub-optimal*" security refers to $\mathsf{C_{clean}} = \mathsf{C_{leak}} \wedge \mathsf{C^{A,B}_{trivial\ forge}}$.

Lemma 9 (Trivial attacks [6]). *Assume that* ARCAD *is correct. For any* t *and* P, *if* $\mathsf{pt_P(t)}$ *has a direct or indirect leakage, the adversary can deduce* $\mathsf{pt_P(t)}$.

Game $\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{b},\mathsf{C_{clean}}}(1^\lambda)$

$\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
$\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st_A}, \mathsf{st_B}, z)$
set all sent^*_* and $\mathsf{received}^*_*$ variables to $\emptyset$
set $\mathsf{t_{test}}$ to $\perp$
$\mathsf{b}' \leftarrow \mathcal{A}^{\mathsf{RATCH},\mathsf{EXP_{st}},\mathsf{EXP_{pt}},\mathsf{CHALLENGE}}(z)$
if $\neg\mathsf{C_{clean}}$ **then return** $\perp$
return b'

Oracle $\mathsf{EXP_{st}(P)}$

return $\mathsf{st_P}$

Oracle $\mathsf{CHALLENGE(P, ad, pt)}$

if $\mathsf{t_{test}} \neq \perp$ **then return** $\perp$
if $\mathsf{b} = 0$ **then**
  replace pt by a random string of same length
end if
$\mathsf{ct} \leftarrow \mathsf{RATCH(P, \text{"send"}, ad, pt)}$
$\mathsf{(t, P, ad, pt, ct)_{test}} \leftarrow \mathsf{(time, P, ad, pt, ct)}$
return ct

Oracle $\mathsf{EXP_{pt}(P)}$

return $\mathsf{pt_P}$

Fig. 2: IND-CCA Game.
(Oracles RATCH are defined in Fig. 1)

Definition 10 ($\mathsf{C_{clean}}$-IND-CCA security). *Let* $\mathsf{C_{clean}}$ *be a cleanness predicate. We consider the* $\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{b},\mathsf{C_{clean}}}$ *game of Fig. 2. We say that the ratcheted communication* ARCAD *is* $\mathsf{C_{clean}}$-IND-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* $\mathsf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{b},\mathsf{C_{clean}}}$ *security game is negligible.*

Definition 11 ($\mathsf{C_{clean}}$-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 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 Def. 6).

The purpose of this update in the definition is to allow us to easily define a weaker form of FORGE-security in Th. 17 and in Section 5.3.

Definition 12 (r-RECOVER security [6]). *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 the game be* $\Pr(\mathsf{win} = 1)$. *We say that the ratcheted communication protocol is* r-RECOVER-secure, *if for any PPT adversary, the advantage is negligible.*

Definition 13 (PREDICT security [6]). *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 playing the game be the probability that* 1 *is returned. We say that the ratcheted communication protocol is* PREDICT-secure, *if for any adversary limited to a polynomial number of queries, the advantage is negligible.*

Game $\mathsf{FORGE}^{\mathcal{A}}_{\mathsf{C}_{\mathsf{clean}}}(1^\lambda)$

$\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
$\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
$(\mathsf{P}, \mathsf{ad}, \mathsf{ct}) \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP}_{\mathsf{st}}, \mathsf{EXP}_{\mathsf{pt}}}(z)$
$\mathsf{RATCH}(\mathsf{P}, \text{“rec”}, \mathsf{ad}, \mathsf{ct}) \rightarrow \mathsf{acc}$
if $\mathsf{acc} = \mathsf{false}$ **then return** 0
if $\neg \mathsf{C}_{\mathsf{clean}}$ **then return** 0
if $(\mathsf{ad}, \mathsf{ct})$ is not a forgery for P **then return** 0
return 1

Game $\mathsf{r\text{-}RECOVER}^{\mathcal{A}}(1^\lambda)$

$\mathsf{win} \leftarrow 0$
$\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
$\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
set all sent^*_* and $\mathsf{received}^*_*$ variables to $\emptyset$
$\mathsf{P} \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP}_{\mathsf{st}}, \mathsf{EXP}_{\mathsf{pt}}}(z)$
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** $\mathsf{win} \leftarrow 1$
return win

Game $\mathsf{PREDICT}^{\mathcal{A}}(1^\lambda)$

$\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}$
$\mathsf{Initall}(1^\lambda, \mathsf{pp}) \xrightarrow{\$} (\mathsf{st}_A, \mathsf{st}_B, z)$
$(\mathsf{P}, \mathsf{ad}, \mathsf{pt}) \leftarrow \mathcal{A}^{\mathsf{RATCH}, \mathsf{EXP}_{\mathsf{st}}, \mathsf{EXP}_{\mathsf{pt}}}(z)$
$\mathsf{RATCH}(\mathsf{P}, \text{“send”}, \mathsf{ad}, \mathsf{pt}) \rightarrow \mathsf{ct}$
if $(\mathsf{ad}, \mathsf{ct}) \in \mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ **then return** 1
return 0

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

We give an $\mathsf{ARCAD}_{\mathsf{DV}}$ protocol on Fig. 4 by slightly updating the BARK protocol [6]. It is updated for secure communication instead of key agreement. Also, some part of the protocol ensuring r-RECOVER security is removed. We will re-introduce it generically and in a strengthened manner in Section 5. $\mathsf{ARCAD}_{\mathsf{DV}}$ is based on a hash function H[10], a one-time symmetric cipher Sym[11], a digital signature scheme DSS[12], and a public-key cryptosystem PKC[13]. First, we construct a “naive” signcryption scheme SC which can be of form

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

Then, we extend SC to a multiple-key encryption called onion. Then, we construct a unidirectional scheme uni. Finally, we construct $\mathsf{ARCAD}_{\mathsf{DV}}$ (see Fig.4).

Theorem 14 (Security of $\mathsf{ARCAD}_{\mathsf{DV}}$ [6]). $\mathsf{ARCAD}_{\mathsf{DV}}$ *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* $\mathsf{ARCAD}_{\mathsf{DV}}$ *is* $\mathsf{C}_{\mathsf{trivial}}$-FORGE*-secure,* $(\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{A,B}_{\mathsf{forge}})$-IND-CCA*-secure and* PREDICT*-secure.*[14,15]

[10] H uses a common key hk generated by H.Gen.

[11] 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.

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

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

[14] 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 [6].

[15] Following Durak-Vaudenay [6], for a $\mathsf{C}_{\mathsf{trivial}}$-FORGE-secure scheme, $(\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{A,B}_{\mathsf{forge}})$-IND-CCA security is equivalent to $(\mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{A,B}_{\mathsf{trivial\ forge}})$-IND-CCA security, which corresponds to the “sub-optimal” security in Table 1.

$\mathsf{onion.Enc}(1^\lambda, \mathsf{hk}, \mathsf{st}_S^1, \ldots, \mathsf{st}_S^n, \mathsf{ad}, \mathsf{pt})$

- pick $k_1, \ldots, k_n$ in $\{0,1\}^{\mathsf{Sym.kl}(\lambda)}$
- $k \leftarrow k_1 \oplus \cdots \oplus k_n$
- $\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt})$
- $\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
- **for** $i = n$ down to 1 **do**
  - $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
  - $\mathsf{ct}_i \leftarrow \mathsf{SC.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k_i)$
- **end for**
- **return** $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$

$\mathsf{onion.Dec}(\mathsf{hk}, \mathsf{st}_R^1, \ldots, \mathsf{st}_R^n, \mathsf{ad}, \vec{\mathsf{ct}})$

- **if** $|\vec{\mathsf{ct}}| \neq n+1$ **then return** $\perp$
- parse $\vec{\mathsf{ct}} = (\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$
- $\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
- **for** $i = n$ down to 1 **do**
  - $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
  - $\mathsf{SC.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \rightarrow k_i$
  - **if** $k_i = \perp$ **then return** $\perp$
- **end for**
- $k \leftarrow k_1 \oplus \cdots \oplus k_n$
- $\mathsf{pt} \leftarrow \mathsf{Sym.Dec}(k, \mathsf{ct}_{n+1})$
- **return** pt

$\mathsf{uni.Init}(1^\lambda)$

- $\mathsf{SC.Gen}_S(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_S, \mathsf{pk}_S)$
- $\mathsf{SC.Gen}_R(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_R, \mathsf{pk}_R)$
- $\mathsf{st}_S \leftarrow (\mathsf{sk}_S, \mathsf{pk}_R)$
- $\mathsf{st}_R \leftarrow (\mathsf{sk}_R, \mathsf{pk}_S)$
- **return** $(\mathsf{st}_S, \mathsf{st}_R)$

$\mathsf{uni.Send}(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_S, \mathsf{ad}, \mathsf{pt})$

- $\mathsf{SC.Gen}_S(1^\lambda) \xrightarrow{\$} (\mathsf{sk}'_S, \mathsf{pk}'_S)$
- $\mathsf{SC.Gen}_R(1^\lambda) \xrightarrow{\$} (\mathsf{sk}'_R, \mathsf{pk}'_R)$
- $\mathsf{st}'_S \leftarrow (\mathsf{sk}'_S, \mathsf{pk}'_R)$
- $\mathsf{st}'_R \leftarrow (\mathsf{sk}'_R, \mathsf{pk}'_S)$
- $\mathsf{pt}' \leftarrow (\mathsf{st}'_R, \mathsf{pt})$
- $\mathsf{onion.Enc}(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_S, \mathsf{ad}, \mathsf{pt}') \rightarrow \vec{\mathsf{ct}}$
- **return** $(\mathsf{st}'_S, \vec{\mathsf{ct}})$

$\mathsf{uni.Receive}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}})$

- $\mathsf{onion.Dec}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}}) \rightarrow \mathsf{pt}'$
- **if** $\mathsf{pt}' = \perp$ **then**
  - **return** $(\mathsf{false}, \perp, \perp)$
- **end if**
- parse $\mathsf{pt}' = (\mathsf{st}'_R, \mathsf{pt})$
- **return** $(\mathsf{true}, \mathsf{st}'_R, \mathsf{pt})$

$\mathsf{ARCAD_{DV}.Setup}(1^\lambda)$

- $\mathsf{H.Gen}(1^\lambda) \xrightarrow{\$} \mathsf{hk}$
- **return** hk

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

- $\mathsf{SC.Gen}_S(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_S, \mathsf{pk}_S)$
- $\mathsf{SC.Gen}_R(1^\lambda) \xrightarrow{\$} (\mathsf{sk}_R, \mathsf{pk}_R)$
- $\mathsf{sk} \leftarrow (\mathsf{sk}_S, \mathsf{sk}_R)$
- $\mathsf{pk} \leftarrow (\mathsf{pk}_S, \mathsf{pk}_R)$
- **return** $(\mathsf{sk}, \mathsf{pk})$

$\mathsf{ARCAD_{DV}.Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_P, \mathsf{pk}_{\overline{P}}, P)$

- parse $\mathsf{sk}_P = (\mathsf{sk}_S, \mathsf{sk}_R)$
- parse $\mathsf{pk}_{\overline{P}} = (\mathsf{pk}_S, \mathsf{pk}_R)$
- $\mathsf{st}_P^{\mathsf{send}} \leftarrow (\mathsf{sk}_S, \mathsf{pk}_R)$
- $\mathsf{st}_P^{\mathsf{rec}} \leftarrow (\mathsf{sk}_R, \mathsf{pk}_S)$
- $\mathsf{st}_P \leftarrow (\lambda, \mathsf{hk}, (\mathsf{st}_P^{\mathsf{send}}), (\mathsf{st}_P^{\mathsf{rec}}))$
- **return** st_P

$\mathsf{ARCAD_{DV}.Send}(\mathsf{st}_P, \mathsf{ad}, \mathsf{pt})$

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

$\mathsf{ARCAD_{DV}.Receive}(\mathsf{st}_P, \mathsf{ad}, \mathsf{ct})$

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

Fig. 4: $\mathsf{ARCAD_{DV}}$ Protocol Adapted from BARK [6] without RECOVER-Security.

2.2 The Epoch Notion in Secure Communication

We will define the epochs according to the work done by Alwen et al. [1] but in a different way.[16] Epochs are a set of messages. An epoch is identified by an integer counter e. Each message is

[16] The notion of epoch appeared in Poettering-Rösler [9] before.

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 15 (Epoch). *Epochs are non-intersecting sets of messages which are defined by an integer. 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, there is not last message. Therefore we define* $\mathsf{e}^{\mathsf{P}}_{\mathsf{send}}$ *and* $\mathsf{e}^{\mathsf{P}}_{\mathsf{rec}}$ *differently. 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 depict a sample of a bidirectional communication in Fig. 5. The figure shows the epoch number assignments based on our definitions.

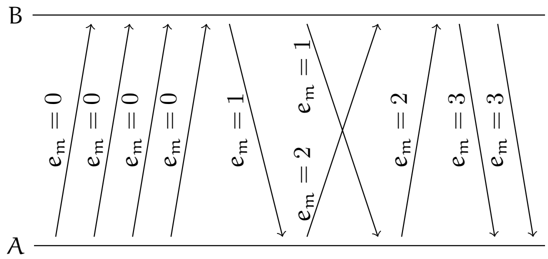


Fig. 5: Bidirectional Exchanges between A and B with Epoch Numbers.

Property 16. From the epoch definition, we have the following properties.

1. At all times, $|\mathsf{e}^{\mathsf{P}}_{\mathsf{send}} - \mathsf{e}^{\mathsf{P}}_{\mathsf{rec}}| \leqslant 1$.
2. The epoch numbers for a unidirectional stream of messages are even if the sender is the participant A and it is odd if the sender is B.
3. A new epoch for a participant P always starts with a $\mathsf{RATCH}(\mathsf{P}, \text{“}\mathsf{send}\text{”})$ calls and ends with $\mathsf{RATCH}(\mathsf{P}, \text{“}\mathsf{rec}\text{”})$ calls.
4. If a participant P accepts a message corresponding to an epoch number $\mathsf{e_m}$, then $\mathsf{e}^{\mathsf{P}}_{\mathsf{send}} \geqslant \mathsf{e_m} + 1$.

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 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 primitive. 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 the post-compromise security, we secure part of the communication only with the forward secure protocol. While the forward secrecy can be built along with symmetric key primitives, the post-compromise security cannot be achieved without public-key cryptosystems as it is shown in [6]. Therefore, we use two subroutines in the 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 $\mathsf{liteARCAD}$ protocol given in Section 3.1. The flag is set in the ad input and it is denoted as $\mathsf{ad.flag}$. We call the strong protocol as $\mathsf{ARCAD_{main}}$ and the weak one as $\mathsf{ARCAD_{sub}}$. The first message which a participant sees (either in sending or receiving) forces the flag to indicate $\mathsf{ARCAD_{main}}$ as we have no initial $\mathsf{ARCAD_{sub}}$ state. Ideally, the time to set the flag for specific security can be decided during the deployment of the application using the protocol. This choice also may 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 $\mathsf{ARCAD_{main}}$ without $\mathsf{ARCAD_{sub}}$. If we do it for no message exchange, then we obtain $\mathsf{ARCAD_{sub}}$. The details are explained shortly in the following sections.

We start by designing $\mathsf{liteARCAD}$.

3.1 liteARCAD: a Light Protocol without Post-Compromise Security

In deployed communication protocols such as Signal, apart from the ratcheting with public-key cryptosystem, there are lighter cryptographic primitives to enable ratcheting with less security but more efficiency. We define a generic version of the lighter layer of "double ratcheting" with the $\mathsf{liteARCAD}$ protocol.

On Fig. 6, we adapt $\mathsf{ARCAD_{DV}}$ of Fig. 4 by replacing the signcryption SC by a symmetric one-time authenticated encryption (OTAE) scheme.[17] We obtain a lightweight ARCAD which achieves most of the security properties except post-compromise security. In fact, it is known that a secure and a correct unidirectional ARCAD implies public-key encryption [6]. Therefore, we do not expect full security from this symmetric-only protocol.

When there is a state exposure, it allows simulating every subsequent reception of messages. Additionally, it also allows to decrypt what is sent and to simulate a new state exposure. Therefore, there is no possible healing after a state exposure. To formalize our $\mathsf{IND\text{-}CCA}$-security, we prune out post-compromise security but leave forward secrecy by using the following cleanness predicate.

$\mathsf{C_{noexp}}$: neither A nor B had an $\mathsf{EXP_{st}}$ before seeing $(\mathsf{ad}, \mathsf{ct})_{\mathsf{test}}$.

When $\mathsf{C_{noexp}}$ holds, the notion of direct and indirect leakage boils down to the cases based on $\mathsf{EXP_{pt}}$ leakages. Hence, $\mathsf{C_{leak}} \wedge \mathsf{C_{noexp}} = \mathsf{C_{sym}}$ can be defined by

$\mathsf{C_{sym}}$: the following conditions are all satisfied
- there is no $\mathsf{EXP_{pt}}(\mathsf{P_{test}})$ after time $\mathsf{t_{test}}$ until there is a $\mathsf{RATCH}(\mathsf{P_{test}}, .)$;
- if $\mathsf{P_{test}}$ is in a matching status at time $\mathsf{t_{test}}$ and its last $\mathsf{RATCH}(\mathsf{P_{test}}, .)$ call corresponds to some $\mathsf{RATCH}(\overline{\mathsf{P}}_{\mathsf{test}}, .)$ at some time $\bar{\mathsf{t}}$, then there is no $\mathsf{EXP_{pt}}(\overline{\mathsf{P}}_{\mathsf{test}})$ after time $\bar{\mathsf{t}}$ until there is another $\mathsf{RATCH}(\overline{\mathsf{P}}_{\mathsf{test}}, .)$ call;
- neither A nor B had an $\mathsf{EXP_{st}}$ before seeing $(\mathsf{ad}, \mathsf{ct})_{\mathsf{test}}$.

Similarly, the notion of trivial forgery changes as the exposure of the state of P now allows to forge for P as well, due to the symmetric key. (Before, it was only allowing to forge for $\overline{\mathsf{P}}$ as keys were asymmetric.) Thus, a forgery becomes trivial when an $\mathsf{EXP_{st}}$ occurs. Hence, the FORGE game cannot allow any state exposure at all. We formalize the security by using the $\mathsf{C_{noexp}}$ cleanness predicate in FORGE-security. There is no $(\mathsf{ad}, \mathsf{ct})_{\mathsf{test}}$ message in FORGE-security, thus $\mathsf{C_{noexp}}$ means no $\mathsf{EXP_{st}}$ as all.

[17] OTAE uses a key in space $\mathsf{OTAE}.\mathcal{K}_\lambda$ and algorithms $\mathsf{OTAE.Enc}$ and $\mathsf{OTAE.Dec}$.

$\mathsf{liteARCAD.Setup} = \mathsf{H.Gen}$

$\mathsf{liteARCAD.InitAll}(1^\lambda, \mathsf{hk})$

pick $\mathsf{sk}_1, \mathsf{sk}_2$ in $\mathsf{OTAE}.\mathcal{K}_\lambda$
$\mathsf{st}_A^{\mathsf{send}} \leftarrow (\lambda, \mathsf{hk}, (\mathsf{sk}_1), (\mathsf{sk}_2))$
$\mathsf{st}_B^{\mathsf{send}} \leftarrow (\lambda, \mathsf{hk}, (\mathsf{sk}_2), (\mathsf{sk}_1))$
return $(\mathsf{st}_A, \mathsf{st}_B, \perp)$

$\mathsf{onion.Send}(1^\lambda, \mathsf{hk}, \mathsf{st}_S^1, \ldots, \mathsf{st}_S^n, \mathsf{ad}, \mathsf{pt})$

pick $k_1, \ldots, k_n$ in $\{0,1\}^{\mathsf{Sym.kl}(\lambda)}$
$k \leftarrow k_1 \oplus \cdots \oplus k_n$
$\mathsf{ct}_{n+1} \leftarrow \mathsf{Sym.Enc}(k, \mathsf{pt})$
$\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
for $i = n$ down to 1 **do**
  $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
  $\mathsf{ct}_i \leftarrow \mathsf{OTAE.Enc}(\mathsf{st}_S^i, \mathsf{ad}_i, k_i)$
end for
return $(\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$

$\mathsf{onion.Receive}(\mathsf{hk}, \mathsf{st}_R^1, \ldots, \mathsf{st}_R^n, \mathsf{ad}, \vec{\mathsf{ct}})$

parse $\vec{\mathsf{ct}} = (\mathsf{ct}_1, \ldots, \mathsf{ct}_{n+1})$
$\mathsf{ad}_{n+1} \leftarrow \mathsf{ad}$
for $i = n$ down to 1 **do**
  $\mathsf{ad}_i \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{ad}_{i+1}, n, \mathsf{ct}_{i+1})$
  $\mathsf{OTAE.Dec}(\mathsf{st}_R^i, \mathsf{ad}_i, \mathsf{ct}_i) \rightarrow k_i$
  if $k_i = \perp$ **then return** $\perp$
end for
$k \leftarrow k_1 \oplus \cdots \oplus k_n$
$\mathsf{pt} \leftarrow \mathsf{Sym.Dec}(k, \mathsf{ct}_0)$
return pt

$\mathsf{uni.Init}(1^\lambda)$

pick sk in $\mathsf{OTAE}.\mathcal{K}_\lambda$
$\mathsf{st}_S \leftarrow \mathsf{sk}$
$\mathsf{st}_R \leftarrow \mathsf{sk}$
return $(\mathsf{st}_S, \mathsf{st}_R)$

$\mathsf{uni.Send}(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_S, \mathsf{ad}, \mathsf{pt})$

pick sk in $\mathsf{OTAE}.\mathcal{K}_\lambda$
$\mathsf{pt}' \leftarrow (\mathsf{sk}, \mathsf{pt})$
$\mathsf{onion.Enc}(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_S, \mathsf{ad}, \mathsf{pt}') \rightarrow \vec{\mathsf{ct}}$
return $(\mathsf{sk}, \vec{\mathsf{ct}})$

$\mathsf{uni.Receive}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}})$

$\mathsf{onion.Dec}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}}) \rightarrow \mathsf{pt}'$
if $\mathsf{pt}' = \perp$ **then**
  return $(\mathsf{false}, \perp, \perp)$
end if
parse $\mathsf{pt}' = (\mathsf{sk}, \mathsf{pt})$
return $(\mathsf{true}, \mathsf{sk}, \mathsf{pt})$

$\mathsf{liteARCAD.Send}(\mathsf{st}_P, \mathsf{ad}, \mathsf{pt})$

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

$\mathsf{liteARCAD.Receive}(\mathsf{st}_P, \mathsf{ad}, \mathsf{ct})$

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

Fig. 6: liteARCAD Protocol (Adapted from $\mathsf{ARCAD_{DV}}$ in Fig. 4).

Theorem 17 (Security of liteARCAD). *Let* liteARCAD *be the* ARCAD *scheme on Fig. 6. It is correct. If* $\mathsf{Sym.kl} = \Omega(\lambda)$, liteARCAD *is* PREDICT*-secure. If* OTAE *is* SEF-OTCMA *and* IND-OTCCA*-secure,* Sym *is* IND-OTCCA*-secure, and* H *is collision-resistant, then* liteARCAD *is* $\mathsf{C}_{\mathsf{noexp}}$-FORGE*-secure and* $\mathsf{C}_{\mathsf{sym}}$-IND-CCA*-secure.*

Proof. We start from an initial game Γ which has a "special message" $(\mathsf{ad}, \mathsf{ct})$. We denote by Q the participant who sends the special message. This special message could be a challenge message in the IND-CCA game. In the game Γ, we define the event E that no participant P has an $\mathsf{EXP}_{\mathsf{st}}(\mathsf{P})$ query before having seen the special message. We assume that the game Γ has the property that whenever E does not occur, then Γ never returns 1. We define below for every (Q, i, j) hybrids $\Gamma_{\mathsf{Q},i,j}$ and $\Gamma'_{\mathsf{Q},i,j}$ which essentially guess Q and how many messages are sent and received by Q before sending the special message.

First of all, we extend the data structure of an OTAE key sk by adding a flag. By default, the flag of sk is down. When the flag is up, we say that the key sk is marked. Like this, we want to mark keys which never leak. The hybrids $\Gamma_{\mathsf{Q},i,j}$ and $\Gamma'_{\mathsf{Q},i,j}$ essentially mark the keys in the i first messages by Q and the j first messages by $\overline{\mathsf{Q}}$. The game maintains counters m_{P} for the number of messages sent by P and counters m'_{P} for the number of messages received by P, for every P.

The liteARCAD.InitAll code is modified by marking the initial keys sk_1 and sk_2 as follows:

liteARCAD.InitAll$(1^\lambda, \mathsf{hk})$

pick $\mathsf{sk}_1, \mathsf{sk}_2$ in $\mathsf{OTAE}.\mathcal{K}_\lambda$
mark sk_1 and sk_2
$\mathsf{st}_{\mathsf{A}}^{\mathsf{send}} \leftarrow (\lambda, \mathsf{hk}, (\mathsf{sk}_1), (\mathsf{sk}_2))$
$\mathsf{st}_{\mathsf{B}}^{\mathsf{send}} \leftarrow (\lambda, \mathsf{hk}, (\mathsf{sk}_2), (\mathsf{sk}_1))$
initialize $m_{\mathsf{A}}, m_{\mathsf{B}}, m'_{\mathsf{A}}, m'_{\mathsf{B}} \leftarrow 0$
return $(\mathsf{st}_{\mathsf{A}}, \mathsf{st}_{\mathsf{B}}, \perp)$

If an $\mathsf{EXP}_{\mathsf{st}}$ reveals a marked key, the game aborts:

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

if $\mathsf{st}_{\mathsf{P}}^{\mathsf{send}}$ or $\mathsf{st}_{\mathsf{P}}^{\mathsf{rec}}$ in st_{P} contain a marked key **then**
    abort the game
end if
return st_{P}

If uni.Send on participant P is invoked for the m_{P}-th time for $\mathsf{P} = \mathsf{Q}$ (resp. for $\mathsf{P} = \overline{\mathsf{Q}}$), if all the keys on $\vec{\mathsf{st}}_{\mathsf{S}}$ are marked, and if $m_{\mathsf{P}} \leqslant i$ (resp. $m_{\mathsf{P}} \leqslant j$), the plaintext pt' is stored with index $(\mathsf{P}, m_{\mathsf{P}})$ together with $(\mathsf{ad}, \mathsf{ct})$, and the keys sk and $\mathsf{st}_{\mathsf{P}}^{\mathsf{rec},v+1}$ are marked. The game $\Gamma'_{\mathsf{Q},i,j}$ additionally replaces $\mathsf{pt}' = (\mathsf{sk}, \mathsf{st}_{\mathsf{Snew}}, \mathsf{pt}_0)$ by $\mathsf{pt}' = (\mathsf{sk}', \mathsf{st}'_{\mathsf{Snew}}, \mathsf{pt}_0)$ with random sk' and $\mathsf{st}'_{\mathsf{Snew}}$. The change also checks the guesses Q, i, and j when the special message is sent.

uni.Send$(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_{\mathsf{S}}, \mathsf{ad}, \mathsf{pt})$

$m_{\mathsf{P}} \leftarrow m_{\mathsf{P}} + 1$
pick sk in $\mathsf{OTAE}.\mathcal{K}_\lambda$
$\mathsf{pt}' \leftarrow (\mathsf{sk}, \mathsf{pt})$
$\mathsf{pt}_{\mathsf{S}} \leftarrow \mathsf{pt}'$
if all the keys on $\vec{\mathsf{st}}_{\mathsf{S}}$ are marked **then**
    if $(\mathsf{P} = \mathsf{Q}$ and $m_{\mathsf{P}} \leqslant i)$ or $(\mathsf{P} = \overline{\mathsf{Q}}$ and $m_{\mathsf{P}} \leqslant j)$ **then**
        parse $\mathsf{pt} = (\mathsf{st}_{\mathsf{Snew}}, \mathsf{pt}_0)$
        mark sk (to be stored in $\mathsf{st}_{\mathsf{P}}^{\mathsf{send},u}$) and $\mathsf{st}_{\mathsf{P}}^{\mathsf{rec},v+1}$
        ($\Gamma'_{i,j}$ only) pick sk' and $\mathsf{st}'_{\mathsf{Snew}}$ at random
        ($\Gamma'_{i,j}$ only) $\mathsf{pt}' \leftarrow (\mathsf{sk}', \mathsf{st}'_{\mathsf{Snew}}, \mathsf{pt}_0)$
    end if
end if
$\mathsf{onion.Enc}(1^\lambda, \mathsf{hk}, \vec{\mathsf{st}}_{\mathsf{S}}, \mathsf{ad}, \mathsf{pt}') \rightarrow \vec{\mathsf{ct}}$
$\mathsf{S}[\mathsf{P}, m_{\mathsf{P}}] \leftarrow (\mathsf{pt}_{\mathsf{S}}, \mathsf{ad}, \vec{\mathsf{ct}})$
if this is the special message and either $\mathsf{Q} \neq \mathsf{P}$, $m_{\mathsf{P}} \neq i+1$, or $m'_{\mathsf{P}} \neq j$ **then**
    abort the game

end if
return $(\mathsf{sk}, \vec{\mathsf{ct}})$

If $\mathsf{uni.Receive}$ is invoked for the m'_P-th time for $P = Q$ (resp. for $P = \overline{Q}$), if all the keys on $\vec{\mathsf{st}}_R$ are marked, if $m'_P \leqslant j$ (resp. $m'_P \leqslant i$), then the keys in pt' are marked. For $\Gamma'_{Q,i,j}$, $\mathsf{pt}' = (\mathsf{sk}', \mathsf{st}'_{S\mathsf{new}}, \mathsf{pt})$ is replaced by $\mathsf{pt}' = (\mathsf{sk}, \mathsf{st}_{S\mathsf{new}}, \mathsf{pt})$ before marking and the game checks that it is not a forgery (the game aborts otherwise).

$\mathsf{uni.Receive}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}})$
$m'_P \leftarrow m'_P + 1$
$\mathsf{onion.Dec}(\mathsf{hk}, \vec{\mathsf{st}}_R, \mathsf{ad}, \vec{\mathsf{ct}}) \to \mathsf{pt}'$
if $\mathsf{pt}' = \perp$ **then**
    return $(\mathsf{false}, \perp, \perp)$
end if
if all the keys on $\vec{\mathsf{st}}_R$ are marked and $S[\overline{P}, m'_P]$ is defined **then**
    if $(P = Q$ and $m'_P \leqslant j)$ or $(P = \overline{Q}$ and $m'_P \leqslant i)$ **then**
        parse $S[\overline{P}, m'_P] = (\mathsf{pt}_S, \mathsf{ad}_S, \mathsf{ct}_S)$
        ($\Gamma'_{i,j}$ only) **if** $\mathsf{ad} \neq \mathsf{ad}_S$ or $\vec{\mathsf{ct}} \neq \mathsf{ct}$ **then** abort the game
        ($\Gamma'_{i,j}$ only) $\mathsf{pt}' \leftarrow \mathsf{pt}_S$
        parse $\mathsf{pt}' = (\mathsf{sk}, \mathsf{st}_{S\mathsf{new}}, \mathsf{pt}_0)$ and mark sk and $\mathsf{st}_{S\mathsf{new}}$ in pt'
    end if
end if
parse $\mathsf{pt}' = (\mathsf{sk}, \mathsf{pt})$
return $(\mathsf{true}, \mathsf{sk}, \mathsf{pt})$

Clearly, the only behavior difference between Γ and $\Gamma_{Q,i,j}$ is that $\Gamma_{Q,i,j}$ may abort if a marked key is requested to be revealed or (Q, i, j) is a wrong guess. Because of the property of Γ, we know that the abort case imply a wrong guess for (Q, i, j) or Γ not returning 1. Hence, we have

$$\Pr[\Gamma \to 1] = \sum_{Q,i,j} \Pr[\Gamma_{Q,i,j} \to 1]$$

For every i and j, we prove by induction that $\Pr[\Gamma_{Q,i,j} \to 1] - \Pr[\Gamma'_{Q,i,j} \to 1]$ is negligible. Actually, none of the marked key is ever used for anything but encryption or decryption. Each key is used to encrypt only one message. If $\mathsf{uni.Send}$ encrypts a message with $\vec{\mathsf{st}}_S$, it can only be decrypted with the same keys. Hence, we can use the $\mathsf{IND\text{-}OTCCA}$ security of OTAE to replace the encrypted k_i, then the $\mathsf{IND\text{-}OTCCA}$ security of Sym to show that $\Pr[\Gamma_{Q,i,j} \to 1] - \Pr[\Gamma'_{Q,i,j} \to 1]$ is negligible. Similarly, $\mathsf{SEF\text{-}OTCMA}$ security implies no forgery on OTAE messages $(\mathsf{ad}_i, \mathsf{ct}_i)$, $i = 1, \ldots, n$ for marked keys. Due to the way ad_i is computed (namely: n and ct_{n+1} are hashed into ad_n and the message pieces are chained by hashing ad_{i+1} and ct_{i+1} into ad_i), since H is collision-resistant, this implies no forgery on n and ct_{n+1} either. Hence, we have no forgery on $(\mathsf{ad}, \vec{\mathsf{ct}})$. This implies that $\Gamma'_{Q,i,j}$ does not abort due to receiving a forgery.

We deduce that the difference between $\Pr[\Gamma \to 1]$ and $\sum_{Q,i,j} \Pr[\Gamma'_{Q,i,j} \to 1]$ is negligible.

FORGE-*security.* In the FORGE game, we make an extra send query to P which is the special message. The property of Γ is satisfied: there is no $\mathsf{EXP}_{\mathsf{st}}$. The game $\Gamma'_{Q,i,j}$ returns 1 if P receives a forgery, which cannot be the case as it occurs before the special message and the game would abort otherwise. Hence, $\Pr[\Gamma'_{Q,i,j} \to 1]$ is negligible.

IND-CCA-*security.* In the IND-CCA game, the special message is the one of the CHALLENGE query. Again, the property of Γ is satisfied: no participant has a $\mathsf{EXP}_{\mathsf{st}}$ before seeing the special message. Hence, we can use the IND-OTCCA security of Sym and OTAE to replace pt for the difference in $\Gamma'_{i,j}$ between $b = 0$ and $b = 1$.

PREDICT-*security.* Like in DV [6], due to the correctness of OTAE, guessing ct before it is produced by RATCH implies guessing the k_n key which RATCH will select on onion.Send. Hence, we obtain PREDICT-security. □

3.2 Our Hybrid On-Demand ARCAD Protocol

We give our on-demand ARCAD protocol on Fig. 7. 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. 4). The latter is typically a weaker-but-faster protocol like liteARCAD (Fig. 6). The use of one or the other is based on a flag that can be turn on and off in ad (it is checked with 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 $\mathsf{hybridARCAD} = \mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{main}}, \mathsf{ARCAD}_{\mathsf{sub}})$.

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 a communication which was broken due to a message loss. Normal communication works in the lower ARCAD session, hence with a flag down. That is, normal communication uses $\mathsf{ARCAD}_{\mathsf{sub}}$ but 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. Of course, we can also make nested hybrid protocols with more than two levels of protocols inside. It may increase the state sizes but the performance should be nearly the same.

3.3 Security Definitions

We modify the predicates and the notion of FORGE-security from Section 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). 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}, $\mathcal{A}'$ instead encrypts a random string of the same length. When it comes to decrypt the obtained ciphertext, the random plaintext is ignored and the RATCH calls with st_{R} are simulated with the RATCH calls for the $\mathsf{ARCAD}_{\mathsf{sub}}$ game. The simulation is no longer perfect but it does not add an $\mathsf{EXP}_{\mathsf{st}}(\overline{\mathsf{P}})$ call.

Hybrid cleanness. We assume two cleanness predicates $\mathsf{C}_{\mathsf{clean}}$ and $\mathsf{C}_{\mathsf{main}}$ (which could be the same) for $\mathsf{ARCAD}_{\mathsf{main}}$ and one cleanness predicate $\mathsf{C}_{\mathsf{sub}}$ for $\mathsf{ARCAD}_{\mathsf{sub}}$. We define a hybrid predicate $\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$ as follows. Let Γ be an IND-CCA game played by an adversary $\mathcal{A}$ against hybridARCAD.

```
hybridARCAD.Setup(1^λ)
  pp_main ← ARCAD_main.Setup(1^λ)
  pp_sub ← ARCAD_sub.Setup(1^λ)
  return (pp_main, pp_sub)

hybridARCAD.Gen(1^λ, pp_main, pp_sub)
  return ARCAD_main.Gen(1^λ, pp_main)
```

```
hybridARCAD.Init(1^λ, (pp_main, pp_sub), sk_P, pk_P̄, P)
  ARCAD_main.Init(1^λ, pp_main, sk_P, pk_P̄, P) → st_main
  initialize array st_sub[] to empty
  if P = A then (e_send, e_rec) ← (0, −1)
  else (e_send, e_rec) ← (−1, 0)
  end if
  initialize array ctr with ctr[0] = 0
  st_P ← (λ, pp_sub, st_main, st_sub[], e_send, e_rec, ctr[], true)
  return st_P
```

```
hybridARCAD.Send(st_P, ad, pt)
  parse st_P as (λ, pp_sub, st_main, st_sub[], e_send, e_rec, ctr[], init)
  if ad.flag or init then
      if e_send < e_rec then e ← e_rec + 1; c ← 0
      else e ← e_send ; c ← ctr[e] + 1
      end if
      ARCAD_sub.InitAll(1^λ, pp_sub) →$ (st_S, st_R, z)          ▷ create a new sub-state.
      st_sub[e, c] ← st_S
      pt′ ← (st_R, pt); ad′ ← (ad, 1, e, c)
      ARCAD_main.Send(st_main, ad′, pt′) →$ (st_main, ct′)      ▷ send using the main state.
      ct ← (ct′, e, c)
      e_send ← e ; ctr[e_send] ← c
  else
      e ← max(e_send, e_rec); c ← ctr[e]
      ad′ ← (ad, 0, e, c)
      ARCAD_sub.Send(st_sub[e, c], ad′, pt) →$ (st_sub[e, c], ct′)   ▷ send using the sub-state.
      ct ← (ct′, e, c)
  end if
  clean-up: erase st_sub[e, c] for all (e, c) such that (e, c) < (e_send, ctr[e_send]) and (e, c) < (e_rec, ctr[e_rec])
  clean-up: erase ctr[e] for all e such that e < e_send and e < e_rec
  st_P ← (λ, pp_sub, st_main, st_sub[], e_send, e_rec, ctr[], false)
  return (st_P, ct)

hybridARCAD.Receive(st_P, ad, ct)
  parse st_P as (λ, pp_sub, st_main, st_sub[], e_send, e_rec, ctr[], init)
  parse ct as (ct′, e, c)
  if (e, c) < (e_rec, ctr[e_rec]) then return (false, st_P, ⊥)   ▷ (e, c) must increase
  if ad.flag or init then
      ad′ ← (ad, 1, e, c)
      ARCAD_main.Receive(st_main, ad′, ct′) → (acc, st_main, pt′)
      parse pt′ as (st_R, pt)
      if acc then
          st_sub[e, c] ← st_R
          e_rec ← e; ctr[e] ← c
      end if
  else
      ad′ ← (ad, 0, e, c)
      if st_sub[e, c] undefined then return (false, st_P, ⊥)
      ARCAD_sub.Receive(st_sub[e, c], ad′, ct′) → (acc, st_sub[e, c], pt)
  end if
  clean-up: erase st_sub[e, c] for all (e, c) such that (e, c) < (e_send, ctr[e_send]) and (e, c) < (e_rec, ctr[e_rec])
  clean-up: erase ctr[e] for all e such that e < e_send and e < e_rec
  st_P ← (λ, pp_sub, st_main, st_sub[], e_send, e_rec, ctr[], false)
  return (acc, st_P, pt)
```

Fig. 7: On-Demand hybridARCAD = hybrid($\mathsf{ARCAD_{main}}, \mathsf{ARCAD_{sub}}$) Protocol.

We let $(\mathsf{ad},\mathsf{ct})$ be the challenge message $(\mathsf{ad}_{\mathsf{test}},\mathsf{ct}_{\mathsf{test}})$ 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

$$\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}(\Gamma) = \begin{cases} \mathsf{C}_{\mathsf{main}}(\mathsf{main}(\Gamma)) \text{ if } (\mathsf{ad},\mathsf{ct}) \text{ belongs to } \mathsf{ARCAD}_{\mathsf{main}} \\ \left.\begin{array}{l} \mathsf{C}_{\mathsf{sub}}(\mathsf{sub}_{e,c}(\Gamma)) \text{ if } \mathsf{C}^{e,c}_{\mathsf{clean}}(\mathsf{main}(\Gamma)) \\ \mathsf{C}_{\mathsf{sub}}(\mathsf{sub}'_{e,c}(\Gamma)) \text{ if } \neg\mathsf{C}^{e,c}_{\mathsf{clean}}(\mathsf{main}(\Gamma)) \end{array}\right\} \text{otherwise} \end{cases}$$

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

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

We easily obtain the following result.

Lemma 18. *If* $\mathsf{ARCAD}_{\mathsf{main}}$ *is* $\mathsf{C}_{\mathsf{main}}$-IND-CCA-*secure and* $\mathsf{ARCAD}_{\mathsf{sub}}$ *is* $\mathsf{C}_{\mathsf{sub}}$-IND-CCA-*secure, then* hybridARCAD *is* $\mathsf{C}_{\mathsf{clean}}$-IND-CCA *with* $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}^{\mathsf{C}_{\mathsf{main}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$.

Proof. Let Γ be an IND-CCA game for hybridARCAD. Let us assume that Γ is clean with our new cleanness notion $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}^{\mathsf{C}_{\mathsf{main}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$.

Let $(\mathsf{ad},\mathsf{ct})$ be the challenge message. If there is no challenge message in Γ, we let $(\mathsf{ad},\mathsf{ct})$ be the last message sent by any participant in Γ. The $(\mathsf{ad},\mathsf{ct})$ message belongs to either $\mathsf{ARCAD}_{\mathsf{main}}$ or $\mathsf{ARCAD}_{\mathsf{sub}}$. It depends on ad.flag and on whether this is the very first message of the participant or not (because we force to use $\mathsf{ARCAD}_{\mathsf{main}}$ in this case).

We define the following non-overlapping events/cases:

- $\mathsf{C}_{\mathsf{main}}$: $(\mathsf{ad},\mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{main}}$;
- $\mathsf{C}^{e,c}_{\mathsf{true}}$: $(\mathsf{ad},\mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{sub}}$, has number (e,c), and $\mathsf{C}^{e,c}_{\mathsf{C}_{\mathsf{main}}}(\mathsf{main}(\Gamma))$ is true;
- $\mathsf{C}^{e,c}_{\mathsf{false}}$: $(\mathsf{ad},\mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{sub}}$, has number (e,c), and $\mathsf{C}^{e,c}_{\mathsf{C}_{\mathsf{main}}}(\mathsf{main}(\Gamma))$ is false.

We know that Γ is clean following $\mathsf{C}_{\mathsf{clean}}$. In the $\mathsf{C}_{\mathsf{main}}$ case ($(\mathsf{ad},\mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{main}}$), by definition of $\mathsf{C}_{\mathsf{clean}}$, we deduce that $\mathsf{main}(\Gamma)$ is $\mathsf{C}_{\mathsf{main}}$-clean. The outcome of $\mathsf{main}(\Gamma)$ and Γ is obviously the same. So is the advantage. Due to the $\mathsf{C}_{\mathsf{main}}$-IND-CCA security of $\mathsf{ARCAD}_{\mathsf{main}}$, the advantage in Γ conditioned to $\mathsf{C}_{\mathsf{main}}$ is negligible.

In what follows, we consider that $(\mathsf{ad},\mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{sub}}$.

$\mathsf{C}^{e,c}_{\mathsf{C}_{\mathsf{main}}}$ indicates if the $\mathsf{ARCAD}_{\mathsf{main}}$ message of pair (e,c) can be replaced by the encryption of something random to produce the same result, but with negligible probability: If $\mathsf{C}^{e,c}_{\mathsf{C}_{\mathsf{main}}}$ is true, $\mathsf{sub}_{e,c}(\Gamma)$ produces the same outcome as Γ. So, the advantages of Γ and $\mathsf{sub}_{e,c}(\Gamma)$ have a negligible difference when $\mathsf{C}^{e,c}_{\mathsf{true}}$ holds. By definition of $\mathsf{C}_{\mathsf{clean}}$, $\mathsf{sub}_{e,c}(\Gamma)$ must be $\mathsf{C}_{\mathsf{sub}}$-clean. Due to the $\mathsf{C}_{\mathsf{sub}}$-IND-CCA security of $\mathsf{ARCAD}_{\mathsf{sub}}$, the advantage in $\mathsf{sub}_{e,c}(\Gamma)$ is negligible. Hence, the advantage in Γ conditioned to $\mathsf{C}^{e,c}_{\mathsf{true}}$ is negligible.

Similarly, if $\mathsf{C}^{e,c}_{\mathsf{C}_{\mathsf{main}}}(\Gamma)$ does not hold, $\mathsf{C}_{\mathsf{clean}}$ implies that $\mathsf{sub}'_{e,c}(\Gamma)$ is clean. This game produces exactly the same outcome as Γ when $\mathsf{C}^{e,c}_{\mathsf{false}}$ holds. So is the advantage. Due to the $\mathsf{C}_{\mathsf{sub}}$-IND-CCA security of $\mathsf{ARCAD}_{\mathsf{sub}}$, the advantage in Γ conditioned to $\mathsf{C}^{e,c}_{\mathsf{false}}$ is negligible.

In all cases, the advantage in Γ is negligible. As the number of cases is polynomially bounded, the advantage in Γ is negligible. □

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 $\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$ predicate defines a new FORGE notion for $\mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{DV}},\mathsf{liteARCAD})$. 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. 8: 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 $\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)$. 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. 8. Only Steps 4 and 8 are new. Later in Section 5, we will strengthen the protocols so that it becomes fully FORGE-secure. We easily prove the following result.

Lemma 19. *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* $\mathsf{C}_{\mathsf{hybrid}} = \mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$*-*$\mathsf{FORGE}^*$*.*

Proof. We proceed like in the proof of Lemma 18. Let Γ be a FORGE^* game for hybridARCAD. Let $(\mathsf{P}, \mathsf{ad}, \mathsf{ct})$ be the output of the adversary. The $(\mathsf{ad}, \mathsf{ct})$ message belongs to either $\mathsf{ARCAD}_{\mathsf{main}}$ or $\mathsf{ARCAD}_{\mathsf{sub}}$. We show below that

$$\mathsf{Adv}(\Gamma) \leqslant \mathsf{Adv}(\mathsf{main}(\Gamma)) + \sum_{\mathrm{e},\mathrm{c}} \mathsf{Adv}(\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)) + \sum_{\mathrm{e},\mathrm{c}} \mathsf{Adv}(\mathsf{sub}'_{\mathsf{e},\mathsf{c}}(\Gamma)) + \mathsf{negl}$$

Applying FORGE^* security for the three terms, $\mathsf{Adv}(\Gamma)$ is negligible. To prove the above inequality, we show that when Γ returns 1, then at least one of the three other games return 1, with negligible exceptions.

We first assume that $(\mathsf{ad}, \mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{main}}$ and $\Gamma = \mathsf{FORGE}^*$ succeeds to return 1. Since Γ returns 1, both participants are in a matching status before we deliver the forgery to P. Hence, both participants are in a matching status in $\mathsf{main}(\Gamma)$ too. Similarly, since $(\mathsf{ad}, \mathsf{ct})$ is accepted by $\mathsf{RATCH}(\mathsf{P}, .)$ in Γ and it belongs to $\mathsf{main}(\Gamma)$, it is accepted by $\mathsf{RATCH}(\mathsf{P}, .)$ in $\mathsf{main}(\Gamma)$ too. Let seq_1 be the value of $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$ in Γ before receiving $(\mathsf{ad}, \mathsf{ct})$. Since both participants were in a matching status, we know that $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ starts with seq_1 in Γ. As Γ returns 1, we know that $(\mathsf{ad}, \mathsf{ct})$ does not appear anywhere in $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ after seq_1. In $\mathsf{main}(\Gamma)$, the values of $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$ and $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ are subsequences of the values in Γ. By the same reasoning, we have $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}} = (\mathsf{seq}'_1, (\mathsf{ad}, \mathsf{ct}))$ in $\mathsf{main}(\Gamma)$ and $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ starts with seq'_1. But seq'_1 must be a sub-sequence of seq_1 so $(\mathsf{ad}, \mathsf{ct})$ cannot appear after it in $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$. Finally, since $\mathsf{C}_{\mathsf{hybrid}}$ holds and $(\mathsf{ad}, \mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{main}}$, $\mathsf{C}_{\mathsf{main}}(\mathsf{main}(\Gamma))$ holds, by definition of $\mathsf{C}_{\mathsf{hybrid}}$. This means that $\mathsf{main}(\Gamma)$ is $\mathsf{C}_{\mathsf{main}}$-clean. We deduce that $\mathsf{main}(\Gamma)$ succeeds to return 1 as well.

Similarly, if $(\mathsf{ad}, \mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{sub}}$ and Γ returns 1, we treat two cases depending on whether $\mathsf{C}^{\mathsf{e},\mathsf{c}}_{\mathsf{clean}}(\Gamma)$ holds or not. Let Γ' be the game in which ct is replaced by the encryption of a random string. If $\mathsf{C}^{\mathsf{e},\mathsf{c}}_{\mathsf{clean}}(\Gamma)$ is true, thanks to $\mathsf{C}_{\mathsf{clean}}$-IND-CCA security, Γ and Γ' produce the same output, but with negligible probability. Hence, Γ' outputs 1, except in negligible cases. Like in the previous case, we deduce that $\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)$ outputs 1:

- RATCH accepts in Γ' implies that RATCH accepts in $\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)$;
- $(\mathsf{ad}, \mathsf{ct})$ appears in $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$ in neither Γ' nor $\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)$);
- $\mathsf{sub}_{\mathsf{e},\mathsf{c}}(\Gamma)$ is $\mathsf{C}_{\mathsf{sub}}$-clean because $(\mathsf{ad}, \mathsf{ct})$ belongs to $\mathsf{ARCAD}_{\mathsf{sub}}$ and $\mathsf{C}^{\mathsf{e},\mathsf{c}}_{\mathsf{clean}}(\Gamma)$ is true.

Finally, if $\mathsf{C}^{e,c}_{\mathsf{clean}}(\Gamma)$ is false, we apply the same reasoning with $\mathsf{sub}'_{e,c}(\Gamma)$. □

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 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 Section 5, we will enrich the protocol with r-RECOVER security which will fix both problems. Hence, we will obtain FORGE-security.

4 Implementations/Comparisons with Existing Protocols

We compare the performances of $\mathsf{ARCAD}_{\mathsf{DV}}$ and liteARCAD to other ratcheted messaging and key agreement protocols that have surfaced throughout 2018. In particular, we implemented five other schemes from the literature. They are the bidirectional asynchronous key-agreement protocol BRKE by PR [9], the similar secure messaging protocol by JS [7], the secure messaging protocol by JMM [8] and a modularized version of two protocols by ACD [1]. In ACD [1], the given protocols are both with symmetric key cryptography and public-key cryptography. We did not implement the BARK protocol [6], as $\mathsf{ARCAD}_{\mathsf{DV}}$ is a slightly modified version of BARK.

All the protocols were implemented in Go [18] and measured with its built-in benchmarking suite [19] 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 exists, 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 AEAD scheme, we used AES-GCM. For public key cryptosystem, we used 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. 9—10.

(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 liteARCAD, followed by the two ACD protocols, then $\mathsf{ARCAD}_{\mathsf{DV}}$, then the JMM protocol, and lastly the strongest protocols PR and JS. $\mathsf{ARCAD}_{\mathsf{DV}}$ and JMM may be comparable except for deferred unidirectional communication.

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

Clearly, $\mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{DV}}, \mathsf{liteARCAD})$ has performances which are weighted averages of the ones of $\mathsf{ARCAD}_{\mathsf{DV}}$ and liteARCAD, depending on the frequency of raising the flag ad.flag.

[18] `https://golang.org/`
[19] `https://golang.org/pkg/testing/`

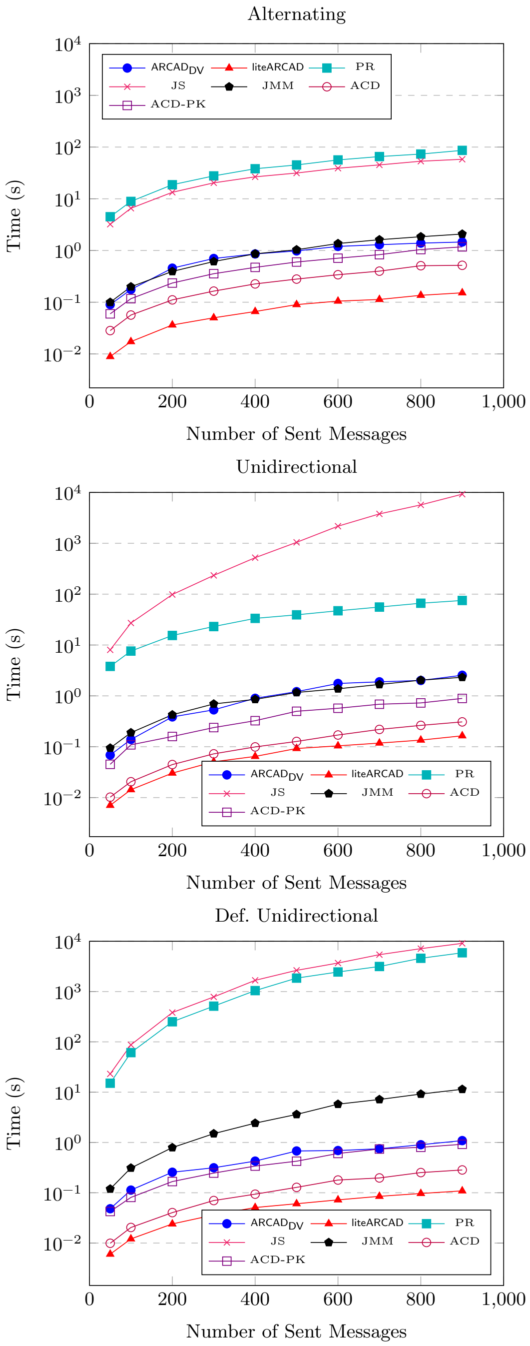
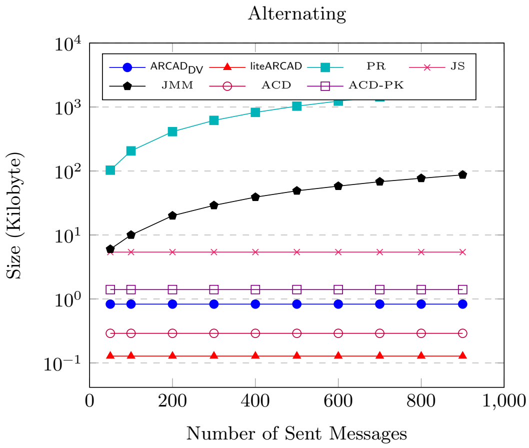


Fig. 9: Runtime Benchmarks
The protocol in [9] is represented with PR; [7] with JS; [8] with JMM; and [1] with ACD and ACD-PK. ACD-PK is the public-key version with stronger security.



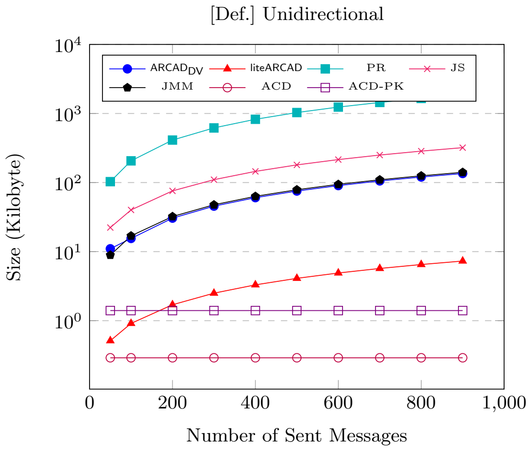


Fig. 10: State Size Benchmarks
Due to the equivalent state sizes in unidirectional and deferred unidirectional traffic, one figure is omitted

5 Security Awareness

5.1 s-RECOVER Security

We gave the DV r-RECOVER security definition [6] in Def. 12. 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 an another definition called s-RECOVER.

Definition 20 (s-RECOVER security). *Consider the* s-RECOVER$^{\mathcal{A}}$ *game in Fig. 11 associated to the adversary $\mathcal{A}$. Let the advantage of $\mathcal{A}$ in succeeding the game be* $\Pr(\mathsf{win} = 1)$. *We say that the ratcheted communication protocol is* s-RECOVER-secure*, if for any PPT adversary, the advantage is negligible.*

```
Game s-RECOVER^A(1^λ)
 win ← 0
 Setup(1^λ) →$ pp
 Initall(1^λ, pp) →$ (st_A, st_B, z)
 set all sent*_* and received*_* variables to ∅
 P ← A^{RATCH,EXP_st,EXP_pt}(z)
 if received^P_ct is a prefix of sent^P̄_ct then
     set t̄ to the time when P̄ sent the last message in received^P_ct
     if received^P̄_ct(t̄) is not a prefix of sent^P_ct then win ← 1
 end if
 return win
```

Fig. 11: 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 it cuts the normal exchange between the participants. Hence, if a participant eventually receives a genuine message, 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 suggested in Section 3.2.

We directly obtain the following useful result:

Lemma 21. *If* 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, except with negligible probability.*

Proof. Let Γ be a game. Let $(\mathsf{ad}, \mathsf{ct})$ be a message which was sent by $\overline{\mathsf{P}}$ then received and accepted by P.

We consider an r-RECOVER adversary $\mathcal{A}$ which simulates Γ until P receives $(\mathsf{ad}, \mathsf{ct})$, and output P. We can parse $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}} = (\mathsf{seq}_1, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_2)$ with seq_2 empty and $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}} = (\mathsf{seq}_3, (\mathsf{ad}, \mathsf{ct}), \mathsf{seq}_4)$. Due to r-RECOVER security, we have $\mathsf{seq}_1 = \mathsf{seq}_3$, but with negligible cases. Hence, $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$ is prefix of $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$, except with negligible probability.

We now let $\mathcal{A}$ play the s-RECOVER security game. Due to s-RECOVER security, since $\mathsf{received}^{\mathsf{P}}_{\mathsf{ct}}$ is prefix of $\mathsf{sent}^{\overline{\mathsf{P}}}_{\mathsf{ct}}$, then $\mathsf{received}^{\overline{\mathsf{P}}}_{\mathsf{ct}}(\bar{t})$ is a prefix of $\mathsf{sent}^{\mathsf{P}}_{\mathsf{ct}}$, but with negligible probability. Due to PREDICT-security, no message arrives before it is sent. Hence, P is in a matching status, except with negligible probability. □

Our notion of $\mathsf{RECOVER}$-security and forgery is quite strong in the sense that it focuses on the ciphertext. Some protocols such as JMM [8] 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 could be seen as not so important because it must sign the same content. However, the signature has a 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 $\mathsf{RECOVER}$ security as we defined it. Contrarily, PR [9] hashes $(\mathsf{ad},\mathsf{ct})$ but does not use it in the next ad or to compute the next ct. So, PR has no $\mathsf{RECOVER}$ security, either.[20]

5.2 Security Awareness

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 22 (Transcript). *In a game, for a participant* P*, we define the* transcript *of* P *as the chronological sequence* $\mathsf{T_P}$ *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* $\mathsf{RATCH}(\mathsf{P},\text{“}\mathsf{rec}\text{”},.,.)$*,* $\mathsf{RATCH}(\mathsf{P},\text{“}\mathsf{send}\text{”},.,.)$*,* $\mathsf{EXP}_{\mathsf{pt}}(\mathsf{P})$*,* $\mathsf{EXP}_{\mathsf{st}}(\mathsf{P})$*, or* $\mathsf{CHALLENGE}(\mathsf{P})$*), except the unsuccessful* RATCH *calls which are omitted. For each pair with a* RATCH *or* $\mathsf{CHALLENGE}$ *oracle,* extra *specifies the role (*“send” *or* “rec”*) and the message* $(\mathsf{ad},\mathsf{ct})$ *of the oracle call. For other pairs,* $\mathsf{extra}=\perp$*.*

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

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

Definition 23 (Acknowledgment extractor). *We consider a game* Γ *and 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* $\mathsf{f}(\mathsf{T}_{\mathsf{P}}^{\mathsf{RATCH}}(\mathsf{t}))=(\mathsf{ad}',\mathsf{ct}')$ *when* P *is in a matching status.*

Given this extractor, P can reconstruct the sub-sequence of $\mathsf{T}_{\overline{\mathsf{P}}}^{\mathsf{RATCH}}(\bar{\mathsf{t}})$ iteratively. Typically, the genuine $(\mathsf{ad},\mathsf{ct})$ implies no forgery before. Thus, our extractor is not required to detect forgeries.

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

Definition 24 (Cleanness extractor). *We consider a game* Γ *and a participant* P*. Let* t *be a time for* P *and* $\bar{\mathsf{t}}$ *be a time for* $\overline{\mathsf{P}}$*. Let* $\mathsf{T_P}(\mathsf{t})$ *and* $\mathsf{T}_{\overline{\mathsf{P}}}(\bar{\mathsf{t}})$ *be the partial transcripts at those time. We say that there is a* cleanness extractor for $\mathsf{C}_{\mathsf{clean}}$ *if there is an efficient function* g *such that* $\mathsf{g}(\mathsf{T_P}(\mathsf{t}),\mathsf{T}_{\overline{\mathsf{P}}}(\bar{\mathsf{t}}))$ *has the following properties: if there is one* $\mathsf{CHALLENGE}$ *in the transcript and, either* P *received* $(\mathsf{ad}_{\mathsf{test}},\mathsf{ct}_{\mathsf{test}})$ *or there is a round trip* $\mathsf{P}\to\overline{\mathsf{P}}\to\mathsf{P}$ *starting with* P *sending* $(\mathsf{ad}_{\mathsf{test}},\mathsf{ct}_{\mathsf{test}})$ *to* $\overline{\mathsf{P}}$*, then* $\mathsf{g}(\mathsf{T_P}(\mathsf{t}),\mathsf{T}_{\overline{\mathsf{P}}}(\bar{\mathsf{t}}))=\mathsf{C}_{\mathsf{clean}}(\Gamma)$*. Otherwise,* $\mathsf{g}(\mathsf{T_P}(\mathsf{t}),\mathsf{T}_{\overline{\mathsf{P}}}(\bar{\mathsf{t}}))=\perp$*.*

[20] 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 [**?**, Fig.3, eprint version]. Therefore, ct'' may be accepted by B so PR is not r-$\mathsf{RECOVER}$ secure. The same holds for s-$\mathsf{RECOVER}$ security.

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 $(\mathsf{ad}_{\mathsf{test}}, \mathsf{ct}_{\mathsf{test}})$ 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.

To have a security-awareness notion, we want r-RECOVER and s-RECOVER security, we want to have an acknowledgment extractor, and we want to have a cleanness extractor. 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 25 (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;*
- *there is a cleanness extractor for* $\mathsf{C}_{\mathsf{clean}}$*.*

5.3 Strongly Secure ARCAD with Security Awareness

In this section, we will take a secure ARCAD such as the one defined in Section 3, which we denote by ARCAD_0 and transform it into another secure ARCAD which we denote by $\mathsf{ARCAD} = \mathsf{blockchain}(\mathsf{ARCAD}_0)$, 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, Asent, Areceived) and store it as 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.

Areceived is a counter of received messages which need to be reported when we run the next Send operation. For each Send operation, we may attach to ct the last Hreceived to acknowledge for received messages and reset Areceived to 0.

Asent is a *list of the hash* 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 Asent, the counterpart $\overline{\mathsf{P}}$ keeps a counter Areceived 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 Asent.

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. 12. The security of the protocol is proved with the following lemmas.

Lemma 26. *If* H *is collision-resistant,* ARCAD *is* s-RECOVER *and* r-RECOVER*-secure.*

Proof. All (ad, 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 (ad, 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 (ad, ct) message includes the

$\mathsf{ARCAD.Setup}(1^\lambda)$ $\mathsf{ARCAD}_0.\mathsf{Setup}(1^\lambda) \xrightarrow{\$} \mathsf{pp}_0$ $\mathsf{H.Gen}(1^\lambda) \xrightarrow{\$} \mathsf{hk}$ $\mathsf{pp} \leftarrow (\mathsf{hk}, \mathsf{pp}_0)$ **return** pp $\mathsf{ARCAD.Gen} = \mathsf{ARCAD}_0.\mathsf{Gen}$	$\mathsf{ARCAD.Init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_\mathsf{P}, \mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{P})$ parse $\mathsf{pp} = (\mathsf{hk}, \mathsf{pp}_0)$ $\mathsf{ARCAD}_0.\mathsf{Init}(1^\lambda, \mathsf{pp}_0, \mathsf{sk}_\mathsf{P}, \mathsf{pk}_{\overline{\mathsf{P}}}, \mathsf{P}) \xrightarrow{\$} \mathsf{st}'_\mathsf{P}$ $\mathsf{Hsent}, \mathsf{Hreceived} \leftarrow \perp$ $\mathsf{Asent} \leftarrow [], \mathsf{Areceived} \leftarrow 0$ $\mathsf{st}_\mathsf{P} \leftarrow (\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{Asent}, \mathsf{Areceived})$ **return** st_P

$\mathsf{ARCAD.Send}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{pt})$
parse st_P as $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{Asent}, \mathsf{Areceived})$
if $\mathsf{Areceived} = 0$ **then** $\mathsf{ack} \leftarrow \perp$ **else** $\mathsf{ack} \leftarrow \mathsf{Hreceived}$
$\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{Hsent}, \mathsf{ack})$
$\mathsf{ARCAD}_0.\mathsf{Send}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{pt}) \xrightarrow{\$} (\mathsf{st}'_\mathsf{P}, \mathsf{ct}')$
$\mathsf{ct} \leftarrow (\mathsf{ct}', \mathsf{Hsent}, \mathsf{ack})$
$\mathsf{Areceived} \leftarrow 0$
$\mathsf{Hsent} \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{Hsent}, \mathsf{ad}, \mathsf{ct})$
$\mathsf{Asent} \leftarrow (\mathsf{Asent}, \mathsf{Hsent})$
$\mathsf{st}_\mathsf{P} \leftarrow (\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{Asent}, \mathsf{Areceived})$
return $(\mathsf{st}_\mathsf{P}, \mathsf{ct})$
	$\mathsf{ARCAD.Receive}(\mathsf{st}_\mathsf{P}, \mathsf{ad}, \mathsf{ct})$
	parse st_P as $(\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{Asent}, \mathsf{Areceived})$
	parse ct as $(\mathsf{ct}', \mathsf{h}, \mathsf{ack})$
	if $\mathsf{h} \neq \mathsf{Hreceived}$ or $\mathsf{ack} \notin \{\perp\} \cup \mathsf{Asent}$ **then**
	    return $(\mathsf{false}, \mathsf{st}_\mathsf{P}, \perp)$
	end if
	$\mathsf{ad}' \leftarrow (\mathsf{ad}, \mathsf{h}, \mathsf{ack})$
	$\mathsf{ARCAD}_0.\mathsf{Receive}(\mathsf{st}'_\mathsf{P}, \mathsf{ad}', \mathsf{ct}') \rightarrow (\mathsf{acc}, \mathsf{st}'_\mathsf{P}, \mathsf{pt}')$
	if acc **then**
	    $\mathsf{Hreceived} \leftarrow \mathsf{H.Eval}(\mathsf{hk}, \mathsf{Hreceived}, \mathsf{ad}, \mathsf{ct})$
	    $\mathsf{Areceived} \leftarrow \mathsf{Areceived} + 1$
	    if $\mathsf{ack} \neq \perp$ **then** remove in Asent all elements of Asent until ack (included)
	    $\mathsf{st}_\mathsf{P} \leftarrow (\mathsf{st}'_\mathsf{P}, \mathsf{hk}, \mathsf{Hsent}, \mathsf{Hreceived}, \mathsf{Asent}, \mathsf{Areceived})$
	end if
	return $(\mathsf{acc}, \mathsf{st}_\mathsf{P}, \mathsf{pt}')$

Fig. 12: Our security-aware $\mathsf{ARCAD} = \mathsf{blockchain}(\mathsf{ARCAD}_0)$ Protocol.

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 be approved by P, thus it must match the list of hash 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 27. ARCAD *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 ARCAD 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 Asent. Both $(\mathsf{ad}', \mathsf{ct}')$ and this hash can be computed from $\mathsf{T}^{\mathsf{RATCH}}_{\mathsf{P}}(t)$. Hence, ARCAD has an extractor. □

Lemma 28. ARCAD *has a cleanness extractor for* $\mathsf{C}_{\mathsf{leak}}$*,* $\mathsf{C}^{\mathsf{S}}_{\mathsf{tforge}}$ *(*$\mathsf{t} = \mathsf{trivial}$ *or void,* $\mathsf{S} = \mathsf{P}_{\mathsf{test}}$ *or* $\mathsf{S} = \{\mathsf{A}, \mathsf{B}\}$*),* $\mathsf{C}_{\mathsf{ratchet}}$*, and* $\mathsf{C}_{\mathsf{noexp}}$*.*

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

Proof. This is quite trivial for $\mathsf{C}_{\mathsf{noexp}}$ and $\mathsf{C}_{\mathsf{ratchet}}$. For $\mathsf{C}^{\mathsf{S}}_{\mathsf{tforge}}$, we can directly see from transcripts where the forgeries are and we can determine if they are trivial or not. For $\mathsf{C}_{\mathsf{leak}}$, we can easily inspect all cases of direct and indirect leakage and see they they can be deduced from the available transcripts. □

Consequently, ARCAD is security-aware. We additionally show that ARCAD is as secure as ARCAD_0 in the next results.

Lemma 29. *Let* $\mathsf{C}_{\mathsf{clean}} \in \{\mathsf{C}_{\mathsf{trivial}}, \mathsf{C}_{\mathsf{noexp}}\}$ *and* $\mathsf{ARCAD} = \mathsf{blockchain}(\mathsf{ARCAD}_0)$*. If* ARCAD_0 *is* $\mathsf{C}_{\mathsf{clean}}$*-*$\mathsf{FORGE}$*-secure (resp.* $\mathsf{C}_{\mathsf{clean}}$*-*$\mathsf{FORGE}^*$*-secure), then* ARCAD *is* $\mathsf{C}_{\mathsf{clean}}$*-*$\mathsf{FORGE}$*-secure (resp.* $\mathsf{C}_{\mathsf{clean}}$*-*$\mathsf{FORGE}^*$*-secure).*

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

We easily extend this result to hybrid constructions.

Lemma 30. *Given* $\mathsf{ARCAD}_{\mathsf{main}}$ *and* $\mathsf{ARCAD}_{\mathsf{sub}}$*, let*

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

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

Proof. Due to Lemma 19, $\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$-$\mathsf{FORGE}^*$-security works like in the previous result. To extend to $\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{main}},\mathsf{C}_{\mathsf{sub}}}$-$\mathsf{FORGE}$-security, we just observe that ARCAD is $\mathsf{r\text{-}RECOVER}$-secure due to Lemma 26. 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, but with negligible probability. □

Lemma 31. *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* $\mathsf{C}_{\mathsf{clean}}$*-*$\mathsf{IND\text{-}CCA}$*-secure, then* ARCAD *is* $\mathsf{C}_{\mathsf{clean}}$*-*$\mathsf{IND\text{-}CCA}$*-secure.*

Proof. We reduce an adversary playing the $\mathsf{IND\text{-}CCA}$ game with ARCAD to an adversary playing the $\mathsf{IND\text{-}CCA}$ 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 32. *Given* $\mathsf{ARCAD}_{\mathsf{main}}$ *and* $\mathsf{ARCAD}_{\mathsf{sub}}$*, let*

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

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

Corollary 33. *Let* $\mathsf{ARCAD} = \mathsf{blockchain}(\mathsf{hybrid}(\mathsf{ARCAD}_{\mathsf{DV}}, \mathsf{liteARCAD}))$*. With the assumptions from Th. 14 and Th. 17, if* H *is collision-resistant,* ARCAD *is* $\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{trivial}},\mathsf{C}_{\mathsf{noexp}}}$*-*$\mathsf{FORGE}$*-secure,* $\mathsf{C}^{\mathsf{C}_{\mathsf{clean}}}_{\mathsf{C}_{\mathsf{clean}},\mathsf{C}_{\mathsf{noexp}}}$*-*$\mathsf{IND\text{-}CCA}$*-secure, and with security-awareness, with* $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{leak}} \wedge \mathsf{C}^{\mathsf{A},\mathsf{B}}_{\mathsf{forge}}$*.*

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.

6 Conclusion

We revisited the security of ARCAD protocols. We proposed an additional lite protocol liteARCAD. We compared the performance of existing protocols with liteARCAD. Based on the good results of liteARCAD, we proposed an hybrid construction which would mostly use liteARCAD 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.

References

1. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *Advances in Cryptology – EUROCRYPT 2019 (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2019. Full version: `https://eprint.iacr.org/2018/1037.pdf`.
2. Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *Advances in Cryptology – CRYPTO 2017*, pages 619–650. Springer International Publishing, 2017.
3. Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.
4. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466, April 2017.
5. Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, June 2016.
6. F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In *Advances in information and Computer Security – IWSEC 2019*, volume 11689 of *Lecture Notes in Computer Science*, pages 343–362. Springer, 2019. Full version: `https://eprint.iacr.org/2018/889.pdf`.
7. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 33–62. Springer International Publishing, 2018.
8. Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In *Advances in Cryptology – EUROCRYPT 2019 (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 159–188. Springer, 2019. Full version: `https://eprint.iacr.org/2018/954.pdf`.
9. Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 3–32. Springer International Publishing, 2018.
10. Open Whisper Systems. Signal protocol library for Java/Android. GitHub repository `https://github.com/WhisperSystems/libsignal-protocol-java`, 2017.
11. Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, May 2015.