Authenticated Continuous Key Agreement: Active MitM Detection and Prevention

Benjamin Dowling[‡] and Britta Hale[†] ⋆

[‡] University of Sheffield
[†] Naval Postgraduate School

Abstract. Current messaging protocols are incapable of detecting active man-in-the-middle threats. Even common continuous key agreement protocols such as Signal, which offers forward secrecy and post-compromise security, are dependent on the adversary being passive immediately following state compromise, and healing guarantees are lost if the attacker is not. This work offers the first solution for detecting active man-in-the-middle attacks on such protocols by extending authentication beyond the initial public keys and binding it to the entire continuous key agreement. In this, any adversarial fork is identifiable to the protocol participants. We provide a modular construction generic for application with any continuous key agreement protocol, a specific construction for application to Signal, and security analysis. The modularity of our solution enables it to be seamlessly adopted by any continuous key agreement protocol.

Keywords: Authentication · Continuous Key Agreement (CKA) · Signal

1 Introduction

Modern messaging protocols have changed the way we look at security, adversarial compromise, and session establishment. While prior key exchange protocols focus on the 'per-session' establishment of keys, these modern protocols minimize latency by maintaining a long-lived session that continuously develops keying material over multiple epochs, i.e. a Continuous Key Agreement (CKA). Due to its single-session and asynchronous nature, a CKA is particularly attractive for messaging environments where the same communication partners may intermittently reconnect. In this context, the effects of compromise have gained increased attention, with the properties of forward secrecy (FS) and post-compromise security (PCS) [7] being the norm for competitive messaging protocols, as well as appearing as guarantees in analysis of other protocols. In the event of adversarial compromise, FS provides a guarantee of secrecy for past session data, while PCS provides a similar guarantee of future session data – contingent absence of active adversarial injections for one epoch. This PCS assumption results in an inconsistent adversarial model, wherein an adversary that is active for a compromise

⋆ The views expressed in this document are those of the author and do not reflect the official policy or position of the DoD or the U.S. Government.

attack *must* be assumed passive immediately following the compromise in order for security healing guarantees to be applicable.

The reason behind this assumption is that FS and PCS properties are traditionally tied to *key confidentiality* and a failure in entity authenticity of communication partners implies that confidentiality itself may no longer hold. Thus any action on the part of the adversary to impersonate parties through key updates of its own nullifies PCS healing. Prior works note that key compromise followed by an active attacker [1] can be catastrophic to both confidentiality and authentication guarantees in a CKA [20,10]. Hence our question:

Is it possible to detect an attacker actively impersonating parties after a compromise?

If an adversary completely controls the channel between the communicating parties *and* has compromised all secrets, then the answer is clearly negative. But what if there exists another channel between the two parties that the adversary cannot fully control? The remainder of this work shows a generic construction for achieving this channel, answering in the affirmative – even in the case of entire state and long-term key compromise.

1.1 Security After Compromise

In common session-based protocols, such as TLS 1.3 [23] and QUIC [15], both key agreement and entity authentication occur only once per session during session initiation, a.k.a. the handshake phase. Two noteworthy attributes arise from this: **(1) Periodic Re-authentication.** If a state compromise occurs, all current state and keys associated with the given session are assumed to be lost; yet security may again be achieved in a following session if an adversary has not gained access to long-term keys, e.g., signing keys. Entity authentication with secret long-term keys essentially allows for a re-bootstrapping of a secure channel even in the presence of an active attacker. **(2) Shared Memory.** "Memory" between the communicating parties is essentially erased at the end of every session[2], implying no mutual, long-standing history of shared session state to bootstrap from (this will be important later).

In contrast, a CKA protocol constitutes a long, continuous session with an ongoing key agreement evolving throughout. The CKA session may in fact have a multi-year lifespan – with entity authentication occurring only once, at the

[1] To avoid a potential terminology ambiguity around use of *active adversary* (active key compromise attacks or active impersonation attacks through message or key update injection), we will henceforth use *active adversary* strictly to refer to an adversary that is injecting messages or key updates, e.g. an adversarial Diffie-Hellman share. Key compromise will be treated as separate action in an attack, and will be referred to as a *compromise*.

[2] TLS [23] offers a session resumption option using a pre-shared key. As this mode is an option only and refreshment of keying material inside of the mode is a further sub-option in the standard we do not go into a detailed comparison.

very start. Again, we notice two attributes: **(1) Periodic Re-authentication.** There is no periodic entity re-authentication in CKA, unlike in session-based protocols. This implies that a secure channel cannot be re-bootstrapped from long-term keys, even if that adversary has not gained access to them during the compromise. **(2) Shared Memory.** A CKA provides a continuous "memory" between the communicating parties. Thus, to combat the risk of compromise in such a long-lived session, the CKA can periodically refresh its secret values by dividing the entire session into a series of *epochs*, where the state from one epoch is input to the next. If communicating parties authentically communicate at epoch i and one of them is compromised, then the routine process of updating the key would lock out the attacker at epoch $i+1$ (provided entity authentication is not broken by an adversary's injected update). Future impersonation attempts would also be blocked. PCS is predicated on the requirement that the adversary is passive for *one* epoch, else the lack of re-authentication could lead to potentially years of an undetected man-in-the-middle (MitM) attacker. Epochs may be far shorter than a typical TLS session length, which significantly limits data exposure (i.e., FS and PCS guarantees are linked to epochs).

In the first instance, session-based protocols have advantages under (1) but disadvantages under (2), while the reverse is true under a CKA. To illustrate this contrast between session-based and CKA protocols, consider the following case example comparing TLS and CKA security under passive and active attackers. Suppose, for simplicity, that two devices need to communicate on a regular basis for 3 years. The developer can choose between using a CKA, supporting asynchronous communication, or TLS, supporting synchronous communication only. CKA epochs will change over every time a party sends a message, while in the case of TLS sessions will be 1 hour in duration, thrice a day. Now, if an adversary compromises device A approximately one year in, we have the outcomes shown in Table 1, dependent on whether the adversary is passive or active immediately following the compromise (i.e., attempting impersonation along with key updates if necessary). For illustration, we assume that the attack is not detected.

	Protocol	# Session Establishments	Data Compromised / MitM Eavesdrop Capability	Impersonation Capability
Adv. Passive	**CKA**	1	1-2 msg.	1-2 msg.
	TLS	3285	1 hour of data	2 years
Adv. Active	**CKA**	1	2 years of data	2 years
	TLS	3285	1 hour of data	2 years

Table 1: Simple example scenario, comparison under session state compromise.

As seen, active or passive adversarial behavior immediately the time of compromise can have a drastic effect on the attractiveness of using a CKA protocol. Under a passive attacker, PCS in a CKA can allow precise epochal update control and consequent smaller vulnerability windows than in TLS. Meanwhile, if the attacker is active in the epoch immediately following compromise, the lack of

re-authentication in CKA broadens the MitM vulnerability window beyond that of TLS. How can this problem be addressed?

Prior work [9] suggested combining long-term secret keys with continuous authentication [10]. This would essential match CKA security assumptions to TLS, allowing channel security to be bootstrapped from secret long-term keys and changing the *active* adversary CKA row in Table 1 to match the *passive* adversary CKA row (in green) – conditioned on the secrecy of the long-term keys.

If the long-term keys are also compromised, there is nothing that can be done for TLS (under its current use) to improve the vulnerability situation due to the lack of continuous session state. This is where the shared "memory" of a CKA becomes advantageous again: an active adversary in possession of all keys would need to fork the session history between communicating parties in order to perform a MitM attack. This begs the question: *Can such a fork be detected, even under full compromise?* If so, then we also address the prior question on detecting active impersonation. Note that in achieving this, CKA would provide a stronger security property under full compromise than TLS or similar session-based protocols provide.

We model the guarantee of forking detection under active MitM attacks and full compromise as an Authenticated Continuous Key Agreement (ACKA) security model, and provide a construction as a Signal add-on that achieves this security.

1.2 Related Works

Security healing following compromise was first investigated under the term *post-compromise security (PCS)* [8]. The first investigations looked strictly at confidentiality of data and specifically focused on compromise of *session keys*. Loss of signature keys and impersonation were not accounted for, leaving authentication issues and active attacks out of scope. PCS is a key property in both Signal [19] and OTR [6]. *Forward Secrecy (FS)* [14] was another and much earlier topic of research in the security-following-compromise scenario. Continuous Key Agreement, aka. Ratcheted Key Exchange, covers a line research for achieving both PCS and FS [8,21,7,22,2,12,16,5,17,24,3,11,1].

Some CKA messaging protocols such as Signal [19] claim to support on-demand user controlled re-authentication (e.g., through comparison of QR codes or numeric identifiers). However, the keys used in QR code or numeric code generation are not used inside of the CKA protocol evolution, so such action only authenticates to the time of session initiation and not the current protocol state [10]. Note that this also applies to the "trust-on-first-use" model – in case of certificate authority validation, keys are verified during initiation while for trusted-on-first-use they are assumed to be valid. In neither case does authentication extend past the session initiation phase, beyond some exploratory theoretical constructions [9].

Our work leverages the CKA definition of a ratcheting protocol [2], and is composable with any CKA-secure protocol. CKA and its security was introduced as a generalization of the lessons learned from ratcheting protocols such as Signal

[19], which is used in Facebook Messaging, WhatsApp, and Skype as well as variants such as the Proteus protocol used in the Wire messaging application [13]. The CKA protocol description is highly generalized, allowing alignment to many ratcheting-style protocols.

While we specifically focus on CKA as a general framework in this research, other works have also analyzed and provided security experiments for ratcheting key exchange protocols, as mentioned above. These works largely forgo the question of an *active attacker*, as such an attacker is largely viewed to be fatal to the protocol's security. A notable exception is a line of enquiry [16] wherein the authors investigate the tie-in of authenticity to PCS healing. In that work, the authors achieve PCS under the restriction that compromise of secret keys does not include secret authentication (signature) keys, making it more closely aligned to the assumptions of [9]. For the [16] proposed solution, a signature key is generated for the next epoch and committed to in the current epoch. If an adversary then compromises a protocol member and obtains state secrets, it cannot impersonate that member in the following epoch as it does not have access to the authentication secrets. A similar approach is considered in [22].

While these approaches improve on the confidentiality-only PCS assumptions, they still do not consider the case of total compromise wherein the adversary obtains both state and long-term (signature) secrets. Consequently, to date, there is no clear solution for some of the strong-adversary scenarios that originally motivated PCS, namely cases when the adversary has"short-term physical access" or devices are "confiscation at a border crossing" [8]. In all practicality, these cases suggest that compromise of signature keys should also be considered. Our solution covers not only the case of full compromise but also an unrestricted adversary, allowing for active attacks immediately following compromise and providing key misuse detection and prevention.

In real-world applications key misuse detection has gained interest, with industry efforts including Certificate Transparency [4]. While focused on a client-server protocol design, the ideas of [21] are also related to our work, in that a log-based system can be used as a transparency overlay. Our stateful CEA protocol falls in line with the authors' discussion that *contradicting observations* (forks or breaks in a shared "memory") are one of three possible foundational concepts for supporting active attacker detection (a straightforward extension to CEA also supports *acausal observations* where an agent can detect misuse of their own secret). While the authors take a symbolic analysis approach and focus on server-client protocols such as for Cloudflare's Keyless SSL [25], our work provides the first computational analysis model and applies to end-to-end security in continuous key agreement.

Detection of active MitM attacks for CKAs was first covered in [10] via the Mediated Epoch Three-party Authentication (META) protocol. The concepts presented in that research provide another foundational piece to our design and analysis through computation of epoch-specific authentication keys. However, the research was reliant on the pre-existing human construct of users comparing authentication tokens (QR-codes or numeric codes). Naturally, involving the user

implies an unpredictable frequency in code comparison and therefore authentication, as well as the potential for human error. While work has been done towards improving the user experience [26], the ideal case is that authentication is achieved through automated means and without relying on the user's conscious action. Thus, we extend the META concept to be automated.

1.3 Contributions and Outline

Our contributions are as follows:

- The first continuous entity authentication (CEA) protocol for full-compromise and active MitM attacks on continuous key agreement protocols, and accompanying security model for CEA and unlinkability model for CEA.
- The first CKA ($\mathsf{CKA}^{\mathsf{LOG}}$) that allows for active MitM detection and prevention, and accompanying security experiment (ACKA).
- An example CEA construction.
- Security analysis of the composed $\mathsf{CKA}^{\mathsf{LOG}}$ using our CEA construction under ACKA.

The remainder of this paper is organized in the following way. In Section 2.1 we provide background on CKA and the existing security model. In Section 3.1 we introduce CEA, associated security experiments, and a CEA construction. Section 4 introduces an expanded CKA construction based on CEA and CKA subcomponents, and a combined security experiment for CKA and CEA. Section 6 provides security analyses for the constructions.

2 High-Level Intuition and Preliminaries

We begin with some intuition as to how the generic ACKA protocol is built.

In our setting, Alice and Bob continually establish secrets via a CKA protocol. As standard in CKA literature, Alice and Bob already share a pre-shared secret value *pss* (to be updated and maintained as part of the secret state *sk*). Critically, we require this to be shared and globally unique among honest pairwise sessions – but not secret. If Alice and Bob have ever held an agreed upon state *pss*[3], then *pss* can be used to bootstrap authentication. Our adversary is allowed to compromise this initial shared state. In the context of long-lived messaging protocols, this setting is equivalent to a correctly-assumed trust-on-first-use scenario or even that the parties authenticated initial keying materials.

We begin with a Continuous Key Agreement protocol, where users exchange public keying material and output new keys per epoch. The core issue is that once the attacker knows the secret shared state, they can inject public keying material themselves and fork the user's view. We use a *Continuous Entity Authentication* (CEA) scheme to generate so-called *fingerprints* of the public keying material of the communicating party: when exchanged reliably, they can confirm that the keying material received by the opposite party was indeed generated honestly.

[3] The pre-shared state *pss* can be modelled as an initial state, without loss of generality.

Of course, if the adversary controls the communication channel, reliable exchange of this fingerprint cannot be assumed and it achieves nothing. Thus, the parties require a communication channel that the adversary cannot manipulate or use to fork user's views: here we use a *Secure Logging* scheme. Senders will upload their locally-generated fingerprints to the log – which they can themselves verify correct posting to – thus preventing the adversary from dropping their fingerprint in transit. This allows the users to exchange and verify their fingerprints – failure of the adversary to similarly post fingerprints if impersonating a user raises an alert flag for the receiver, while doing so also forces the adversary to advertise their active attacks (detectable by the impersonated user).

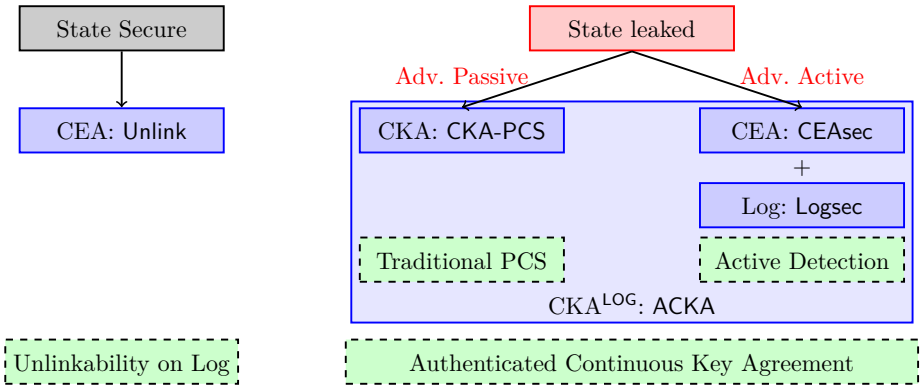

Fig. 1: Case Scenarios around Compromise. Following a compromise, the protocol could heal through traditional PCS via the underlying CKA if the adversary is passive (CKA-PCS). If the adversary is not passive, then the CEA protocol allows for detection and prevention of a MitM, assuming CEAsec and immutability of the log (Logsec). The combined $\mathsf{CKA}^{\mathsf{LOG}}$ protocol provides Authenticated Continuous Key Agreement (PCS + active detection). In absence of a compromise, we require that the CEA-generated fingerprints do not increase linkability of the conversation between epochs (Unlink).

Such a logging action requires the users (and the adversary) to notate epoch fingerprints so that they cannot be confused for fingerprints in other sessions/epochs, and cannot be forked by a key compromising adversary within the same epoch. Thus, our CEA also uses unkeyed cryptographic primitives (specifically collision-resistant hash functions) to generate generate a *label* for a given epoch based on the previous epoch state. This provides a deductive step – a previous shared state is used to generate labels, which in turn aid in identifying forks in fingerprints. As long as a shared state exists at some point (even if known to the adversary), a fork is detectable.

Specifically, an adversary will collide on a fingerprint label with the honest party that they are trying to impersonate, and the presence of two fingerprints for a single label will alert all communication parties to a impersonation attack;

or an adversary will impersonate a party, introducing a fingerprint that the impersonated user can identify as dishonest.

Thus, an honest execution of the protocol will follow this pattern: the communicating parties share some initial pre-shared state (*pss*) that is input to the CEA. Whenever a sender generates some new keying material from their CKA session, they also generate a label (based on prior state), and a fingerprint pair (based on the current key update) via the CEA protocol, and upload this to the secure logging scheme. (The sender will also check if the label has already been uploaded to the log with a fingerprint, which would indicate an impersonation, instigating the sender to initiate appropriate – likely out-of-band – action on detection of the attack.) Afterwards, the internal sender CEA state is updated with the CKA key, and the internal sender CKA state is also updated with the CKA key per its normal functionality, i.e., generating an independent key for users to use in some symmetric key ratchet using a key derivation function.

Whenever the receiver gets new public keying material from their communicating partner, the receiver uses the keying material as input to the CEA to generate a local label and fingerprint, and looks up the label in the log – if two such fingerprints of the same label exist, or a different fingerprint exists at the label then they have locally generated, then an active (key compromising) impersonation attack is taking place, and the receiver will initiate appropriate action. Thus, past agreement on shared state (via the computed label) is used to bootstrap agreement on the next ratchet state (matching fingerprints). A MitM attack would fork the fingerprints, and be identifiable as there is a history prior to that fork. The ACKA security model captures this ability – which we demonstrate is feasible based on the security of the log (Logsec) and CEA (CEAsec).

Naturally, an adversary may be either active or passive following a compromise. Ergo we define not only security for detection of an active MitM adversary, but also design for the holistic after compromise solution space. Depending on adversarial behavior, the security guarantees diverge as follows (shown in Fig. 1): If the adversary is passive for an epoch following compromise, the protocol security follows the route of traditional PCS. If the adversary is active in impersonating parties as a MitM and injects their own update to form a forked view of the conversation between Alice and Bob, our protocol detects the adversarial activity. Furthermore, we require that any introduced fingerprinting from CEA for detecting an active adversary do not impact conversation tracking (linkability) for the underlying CKA in the case of a passive adversary, i.e., that the use of fingerprints do not introduce a linkability advantage.

2.1 Preliminaries

We follow standard assumptions by requiring a KDF to be a PRF [18].

Definition 1 (Key Derivation Function (KDF)). *A* key derivation function*,* $\mathrm{KDF}(sk, \mathtt{id}) \to k$*, is a pseudorandom function* $\mathcal{K} \times \mathcal{N} \to \mathcal{E}$ *that takes as input original keying material* sk *and a key identifier* $\mathtt{id}$ *and outputs a key* k*.*

2.2 Continuous Key Agreement

The continuous key agreement (CKA) definition and its security experiment [2] are provided below. On a high-level, a CKA protocol is a multi-stage key exchange protocol, where protocol participants continually exchange public keyshares and output (after each message) a new symmetric key.

Definition 2 (Continuous Key Agreement). *A continuous-key-agreement (CKA) scheme is quadruple of algorithms CKA = (CKA-Init-A, CKA-Init-B, CKA-S, CKA-R) where*

- *CKA-Init-A (and similarly CKA-Init-B) is an algorithm that takes a key ik and produces an initial state $\gamma^A \leftarrow$ CKA-Init-A(ik) (γ^B resp.),*
- *CKA-S is a potentially probabilistic algorithm takes a state γ, and produces a new state, message, and key $(\gamma', T, I) \leftarrow\$$ CKA-S(γ), and*
- *CKA-R takes a state γ and a message T, and produces a new state and key $(\gamma', I) \leftarrow$ CKA-R(γ, T)*

The space of initialization keys ik is denoted $\mathcal{IK}$ and the space of CKA keys is denoted $\mathcal{I}$.

$\underline{\mathsf{Exp}^{\mathsf{CKA\text{-}PCS},t^*,\Delta_{\mathrm{CKA}}}_{\mathrm{CKA},\mathcal{A}}(\lambda)}$:
$ik \leftarrow\$ \mathcal{IK}$
$\gamma^A \leftarrow$ CKA-Init-A(ik)
$\gamma^B \leftarrow$ CKA-Init-B(ik)
$t_A, t_B \leftarrow 0$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow \mathcal{A}^{\textbf{corr-A},\textbf{send-A},\textbf{send-A}'(\cdot),\textbf{receive-A},\textbf{chall-A}}$
return $b = b'$

corr-A:
req allow-corr **or** finished$_A$
return γ^A

send-A:
$t_A{+}{+}$
$(\gamma, T_{t_A}, I_{t_A}) \leftarrow\$$ CKA-S(γ)
return (T_{t_A}, I_{t_A})

send-A$'(r)$:
$t_A{+}{+}$
req allow-corr
$(\gamma, T_{t_A}, I_{t_A}) \leftarrow\$$ CKA-S$(\gamma; r)$
return (T_{t_A}, I_{t_A})

receive-A:
$t_A{+}{+}$
$(\gamma^A, *) \leftarrow$ CKA-R(γ^A, T_{t_A})

chall-A:
$t_A{+}{+}$
req $t_A = t^*$*
$(\gamma, T_{t_A}, I_{t_A}) \leftarrow\$$ CKA-S(γ)
if $b = 0$ **then**
  return (T_{t_A}, I_{t_A})
else
  $I \leftarrow\$ \mathcal{I}$
  return (T_{t_A}, I)
end if

Fig. 2: The queries for party A in the CKA-PCS security experiment; queries for party B are defined analogously.

CKA Security Intuitively, CKA security could best be described as "*having seen a transcript $T_1, T_2, \ldots$, the keys $I_1, I_2, \ldots$ look uniformly random and independent.* The adversary is given the power to control the random coins used by the sender

and leak the current state of either party, but may not modify the messages T_i. Since the adversary may leak the state and control randomness, the keys produced under such circumstances need not be secure. CKA security, initially described in [2], is shown in Figure 2. Next, we formalise what it means for a CKA to be secure.

Definition 3 (CKA **Security).** *Let* CKA *be a continuous key agreement protocol, and let* $t^* \in \mathbb{N}$ *be an epoch index and* $\Delta_{CKA} \in \mathbb{N}$ *be the number of epochs until an epoch no longer contains secret information pertaining to a challenge. For a particular pair of predicates allow-corr* $_P :\Longleftrightarrow \max(t_A, t_B) \leq t^* - 2$*, and finished*$_P :\Longleftrightarrow t_P \geq t^* + \Delta_{CKA}$*, and a PPT algorithm* $\mathcal{A}$*, we define the advantage of* $\mathcal{A}$ *in the* CKA-PCS *security experiment to be:* $\mathsf{Adv}^{\mathsf{CKA\text{-}PCS},t^*,\Delta_{CKA}}_{\mathrm{CKA},\mathcal{A}}(\lambda) = |\Pr[\mathsf{Exp}^{\mathsf{CKA\text{-}PCS},t^*,\Delta_{CKA}}_{\mathrm{CKA},\mathcal{A}}(\lambda) = 1] - \frac{1}{2}|$*. We say that* CKA *is* CKA-PCS*-secure if, for all* $\mathcal{A}$*,* $\mathsf{Adv}^{\mathsf{CKA\text{-}PCS},t^*,\Delta_{CKA}}_{\mathrm{CKA},\mathcal{A}}(\lambda)$ *is negligible in the security parameter* λ*.*

2.3 Secure logging schemes

To support the CKA in detection of adversarial action, we leverage an append-only and immutable logging scheme. In practice, this may be realized by blockchain or other distributed ledger protocols; to provide maximum flexibility in choice, we define the logging scheme here generically as well as its required security. One additional functionality that our logging scheme requires is the ability to query labels to return values stored in the log.

When formalized in literature, logging schemes usually separate algorithms to one for promising to add an entry to the log, and one update-style algorithm that actually adds the entries to the log. We combine these into Append to simplify our definitions. In addition, we require that the logging scheme is able to prove that two views of the log (from different users) are *consistent*, i.e. that the log has not been retroactively modified, but simply appended to. This is captured in ProveAppend. Logged values, *val*, are indexed by *labels*.

Definition 4 (Logging Algorithm). *A logging scheme* Log *consists of the following algorithms, some of which are run by the log itself, and some run by a verifier. We define* Log *as a tuple of algorithms:* Log.{Setup, Append, ProveAppend, Query, AppendVerify, ConVfy}. Setup() *is used by a logger to initialize its log:*

- Setup() $\rightarrow_{\$}$ (st, pk_{Log}) *is a probabilistic algorithm which outputs some initial secret state st for the log, and a logging public key pk. The corresponding secret key is stored with other secret state values.*

  The following are used by a logger to prove various properties to verifiers:

- Append$(st, label, val) \rightarrow_{\$} (st', c, V)$ *is a potentially probabilistic algorithm which takes as input the log state st, a label for val to be "appended at", and outputs an updated state* st'*, membership commitment proof c, and a view of the log* V*.*

– $\text{ProveAppend}(st, V_0, V_1) \rightarrow_\$ C$ *is a potentially probabilistic algorithm which takes as input the log state* st*, two views of the log* V_0, V_1*, and outputs a consistency proof* C*, or an error symbol* $\perp$*.*
– $\mathsf{Query}(st, label) \rightarrow (\boldsymbol{val}, \boldsymbol{c})$ *is a deterministic algorithm which takes as input the log state* st*, a label, and outputs all values* $\boldsymbol{val}$ *corresponding to label in the log, as well as their corresponding membership proofs* $\boldsymbol{c}$*.*

The following algorithms are used by verifiers to verify membership proofs for val, and to check the consistency (append-only-ness) of the log.

– $\text{AppendVerify}(val, c, V, pk_{\text{Log}}) \rightarrow b$ *is a deterministic algorithm which takes as input a value* val*, membership proof* c*, a view of the log* V*, and the log public key* pk_{Log}*. It outputs a bit* b *indicating success or failure of verification.*
– $\text{ConVfy}(V_0, V_1, C, pk_{\text{Log}}) \rightarrow b$ *is a deterministic algorithm which takes as input two views* V_0, V_1*, a consistency proof* C*, and the log public key* pk_{Log}*, and outputs a bit* b *indicating success or failure of verification.*

Definition 5 (Logging Algorithm Correctness). *We say that a logging scheme* Log *is correct if*

– *for all* (st, pk_{Log}) *such that* $\text{Setup}() \rightarrow_\$ (st, pk_{\text{Log}})$*,*
– *all* $n \in \mathbb{N}$*, all ordered lists of label, value pairs* $(l_0, val_0), \ldots, (l_n, val_n) \in \mathbb{L} \times \mathcal{V}$*, and*
– *all membership proof and view pairs* $(c_0, V_0), \ldots, (c_n, V_n)$ *such that*

$$\text{Append}(st, l_0, val_0) \rightarrow_\$ (st_0, c_0, V_0), \ldots, \text{Append}(st_{n-1}, l_n, val_n) \rightarrow_\$ (st_n, c_n, V_n)$$

– $\forall i, j, k \in \mathbb{N}$ *such that* $i, j \leq k$*,* $\text{ProveAppend}(st_k, V_i, V_j) \rightarrow_\$ C_{i,j,k}$ *and*
– $\forall m, p \in \mathbb{N}$ *such that* $m \leq p$*,* $\mathsf{Query}(st_p, l_m) \rightarrow (\boldsymbol{val_m}, \boldsymbol{c_m})$

the following holds $\forall q, r \in \mathbb{N}$*:*

1. *If* $l_q = label_m$*, then* $val_q \in \boldsymbol{val_m}$*,* $c_q \in \boldsymbol{c_m}$
2. $\text{AppendVerify}(val_q, c_q, V_r, pk_{\text{Log}}) \rightarrow 1 \iff r > q$
3. $\text{ConVfy}(V_i, V_j, C_{i,j,k}, pk_{\text{Log}}) \rightarrow 1 \iff i, j \leq k$

Definition 6 (Secure Log). *We say that a logging scheme tuple* $\text{Log}.\{\text{Setup}, \text{Append}, \text{ProveAppend}, \mathsf{Query}, \text{AppendVerify}, \text{ConVfy}\}$ *is secure if* $\Pr[(\mathsf{Exp}^{\mathit{Logsec}\text{-}\mathsf{Consistency}}_{\text{Log},\mathcal{A}}(\lambda) = 1) \vee (\mathsf{Exp}^{\mathit{Logsec}\text{-}\mathsf{Exclude}}_{\text{Log},\mathcal{A}}(\lambda) = 1)] \leq \mathsf{negl}$ *for some negligible function* negl*.*

Logging security is separated into two sub-experiments. The first experiment, Logsec-Exclude captures the adversary's ability to produce a commitment that verifies for different sets of committed values. The second experiment, Logsec-Consistency captures the adversary's ability to fork the view of the log, which would break immutability and append-only properties.

$\mathsf{Exp}^{\text{Logsec-Exclude}}_{\text{Log},\mathcal{A}}(\lambda)$:

1: Setup() $\rightarrow_{\$}$ (st, pk_{Log})
2: $(label_0, val_0, \sigma_0), \ldots, (label_N, val_N, \sigma_I), label_I, \boldsymbol{val_I}, \boldsymbol{c_I} \leftarrow_{\$} \mathcal{A}(1^{\lambda}, st, pk_{Log})$
3: $\forall i = 0, \ldots, N$: LOG$[label_i] \leftarrow val_i$
4: $\forall j$ s.t. $label_j = label_I$: $\boldsymbol{val'} \xleftarrow{\cup}$ LOG$[label_j]$
5: Append$(st, label_0' val_0; \sigma_0) \rightarrow (st_0, c_0, V_0)$
6: $\forall i = 1, \ldots, N$: Append$(st_{i-1}, label_i, val_i; \sigma_i) \rightarrow (st_i, c_i, V_i)$
7: **if** (AppendVerify$(\boldsymbol{val_I}, \boldsymbol{c_I}, label_I, V_N, pk_{Log}) = 1) \wedge (\boldsymbol{val} \neq \boldsymbol{val'})$ **then**
8: **return** 1
9: **end if**
10: **return** 0

$\mathsf{Exp}^{\text{Logsec-Consistency}}_{\text{Log},\mathcal{A}}(\lambda)$:

1: Setup() $\rightarrow_{\$}$ (st, pk_{Log})
2: $(label_0, val_0, \sigma_0), \ldots, (label_N, val_N, \sigma_N), (label_0', val_0', \sigma_0'), \ldots, (label_N', val_N', \sigma_N'),$ $val_I, c_I, V_I, val_J, c_J, V_J, C_{I,J} \leftarrow_{\$} \mathcal{A}(1^{\lambda}, st, pk_{Log})$
3: $\forall i = 0, \ldots, N$: LOG$[label_i] \leftarrow val_i$
4: $\forall i = 0, \ldots, N$: LOG$'[label_i'] \leftarrow val_i'$
5: Append$(st, label_0, val_0; \sigma_0) \rightarrow (st_0, c_0, V_0)$
6: $\forall i = 1, \ldots, N$: Append$(st_{i-1}, label_i, val_i; \sigma_i) \rightarrow (st_i, c_i, V_i)$
7: Append$(st, label_0', val_0'; \sigma_0') \rightarrow (st_0', c_0', V_0')$
8: $\forall i = 1, \ldots, N$: Append$(st_{i-1}', label_i', val_i'; \sigma_i') \rightarrow (st_i', c_i', V_i')$
9: **if** (LOG$\neq$LOG$'$) $\wedge$ (AppendVerify$(\boldsymbol{val_I}, \boldsymbol{c_I}, V_I) = 1) \wedge$ (AppendVerify$(\boldsymbol{val_J}, \boldsymbol{c_J}, V_J) = 1) \wedge$ (ConVfy$(V_I, V_J, C_{I,J}, pk_{\text{Log}}) = 1)$ **then**
10: **return** 1
11: **end if**
12: **return** 0

Fig. 3: Immutable and Append-Only Logging Security Experiments. Note that Append is a potentially probabilistic algorithm – in the event that it is instantiated probabilistically, the notation Append$(st, label_i, val_i; \sigma_i)$ captures the that the adversary has access to the appender's random coins.

3 Continuous Entity Authentication

In this section we motivate the formalism and security for our first contribution, a continuous entity authentication (CEA) protocol. Roughly, CEA protocols can be viewed at a high level as analogous to Message Authentication Codes (MAC),[4] where a key from one epoch is used to authenticate update information for the next. We term the analogous "MAC tag" output the *fingerprint* of the epoch. However, CEA protocols do not only generate fingerprints; they also enable users to update their shared secret state (analogous to updating a MAC "authentication key") and additionally generate public identification labels. To maintain generality, we define CEA as its own independent primitive as opposed to strengthening CKAs with additional functionalities.

[4] We will discuss in the constructions in Section 5 why a MAC is an analogy vs. a practical instantiation of the CEA fingerprinting sub-primitive.

Recall that our motivation is to detect when a key-compromising attacker injects CKA updates T between communicating partners. Our CEA protocol computes digests (or fingerprints) of these updates; comparing these fingerprints ensures the absence of an active attacker. These fingerprints will (eventually) be logged by a logging scheme, and at some point a user will query the log to recover fingerprints for comparison, requiring a public label to query.[5] Finally, since a CEA protocol is intended to be composed with a CKA, the output keys from a CKA can be used to update the secret state of the CEA protocol.

Notation Since we intend to compose CEA with a CKA, our notation, maintained state and outputs align. To maintain generality, we define separate notation for CEA, as CEA also does not need to share full state with the CKA. This highlights its use as an independent primitive and allows flexibility over what is being authenticated through CEA (e.g. whether the users wish to authenticate *info* that equates to an update value, T, or to a ciphertext). Thus, Table 2 describes *terminology* in gray (left column) and provides a loose alignment of notation and real-world examples (right columns) for intuition.

Consider, for example, the Signal protocol. In Signal, a shared root key rk is maintained and asymmetrically "ratcheted" forward via asymmetric ratchet keys $ratchetPK$. This results not only in a next epoch root state for forward chaining but also an epoch-specific state ck (which can be used to derive application data protection keys). In the CKA definition, CKA-S will output a message T and a new epoch secret I at the same time. Consequently, when considered as a CKA, Signal's ratchet update is the update message ($T = ratchetPK$) and the new epochal chain key is the epoch secret ($I = ck$).

Signal currently derives so-called fingerprints *fprint* on demand; however, the onus is on the user to compare these out-of-band (e.g. via QR codes). Additionally, Signal uses no secret state to generate these fingerprints (simply computed as an iterative hash over the public keys and identifiers of the communicating parties).

To give intuition to the composition of CKA with CEA, we establish the following order: the fingerprint is first computed over the new update message T and current CEA state. Afterwards I is used to update the CEA state and the fingerprint is sent to the log. Before CKA-R is executed on the receiver side, the fingerprint is retrieved from the log and verified. The output of CKA-R (the epoch secret I) is used to update the CEA state.

We now turn to formalising a CEA protocol and capturing its security.

3.1 CEA Definition

The formal definition and security experiment for CEA are as follows.

Definition 7. *A Continuous Entity Authentication Protocol* CEA *is a tuple of algorithms* CEA.{Setup, Update, Fprint, Lprint, Verify}*. Associated with* CEA *are respective input and output spaces $\mathcal{ID}$, $\mathcal{S}$, $\mathcal{E}$, $\mathcal{F}$, $\mathcal{L}$. We describe the algorithms below:*

[5] In absence of a label, a user can only identify a matching (potentially forged) fingerprint, but no instances of duplicate postings or mis-matching fingerprints.

Variable description	**CKA**	**CEA**	**Signal**
Preshared secret state	$ik \in \mathcal{K}$	$pss \in \mathcal{S}$	$rk \in \mathcal{K}$
Protocol state	$\gamma \in \Gamma$	$st \in \mathcal{S}$	$a, b \in \mathbb{Z}_p$, $rk \in \mathcal{K}$
Per-epoch keys (output / input)	$I_t \in \mathcal{I}$	$epk \in \mathcal{E}$	$ck \in \mathcal{K}$
Messages	$T \in \mathcal{M}$	$info \in \{0,1\}^*$	$ratchetPK = g^a$ (resp. g^b)
Fingerprints An index for a given *fprint*		*fprint* *label*	*fprint*

Table 2: Intuition for rough notation alignment for CKA schemes and CEA schemes. Signal is included as a real-world example.

- $\text{Setup}(\lambda, pss, id_A, id_B) \to_\$ (st_A^0), (st_B^0)$*: a probabilistic initialization algorithm takes as input a security parameter* λ *and some preshared secret state* $pss \in \mathcal{S}$*, and party identities* $id_A, id_B \in \mathcal{ID}$*, and outputs some initial state* st_U^0 *for each party* $U \in \{A, B\}$*. We assume that there exists some unique mapping between the preshared secret state pss and the identities* $\{A, B\}$ *sharing this state, and also that some global ordering of identities exists.*
- $\text{Update}(st_U^t, epk) \to (st_U^{t+1})$*: a deterministic algorithm that takes as input the local state* st_U^t *for a party* $U \in \{A, B\}$ *in epoch* t*, and some secret epoch value* $epk \in \mathcal{E}$*, and outputs an updated state* st_U^{t+1} *for the party* U *in the next epoch* $t + 1$*.*
- $\text{Fprint}(st_U^t, info) \to_\$ fprint$*: a potentially probabilistic algorithm that takes as input the local state* st_U^t *for a party* $U \in \{A, B\}$ *in epoch* t*, and an arbitrary-length bit string* $info \in \{0,1\}^*$ *to be authenticated. It outputs a fingerprint* $fprint \in \mathcal{F}$*.*
- $\text{Lprint}(st_U^t, context) \to label$*: a deterministic algorithm that takes as input the local state* st_U^t *for a party* $U \in \{A, B\}$ *in epoch* t*, and an arbitrary-length bit string* $context \in \{0,1\}^*$*. It outputs a label* $label \in \mathcal{L}$ *for the fingerprint.*
- $\text{Verify}(st_U^t, info, context, fprint, label) \to \{\texttt{accept}, \texttt{reject}\}$*: a deterministic algorithm that takes as input the local state* st_U^t *of a party* U *in epoch* t*, an arbitrary-length bit string* $info \in \{0,1\}^*$ *to be authenticated, optional context information* $context \in \{0,1\}^* \cup \{\perp\}$*, the fingerprint fprint, and the label label to be verified, and outputs a flag indicating whether verification was accepted or rejected.*

Definition 8 (Correctness of CEA**).** *For all* $pss \in \mathcal{S}$*, let* $\text{CEA.Setup}(\lambda, pss, id_A, id_B) \to_\$ (st_A^0), (st_B^0)$ *and let* $(st_A^0), \ldots, (st_A^t), (st_B^0), \ldots, (st_B^t)$ *be a series of outputs such that:*

$$\text{CEA.Update}(st_A^0, ek^0) \to_\$ st_A^1, \ldots, \text{CEA.Update}(st_A^{t-1}, ek^{t-1}) \to_\$ (st_A^t)$$
$$\text{CEA.Update}(st_B^0, ek^0) \to_\$ st_B^1, \ldots, \text{CEA.Update}(st_B^{t-1}, ek^{t-1}) \to_\$ (st_B^t)$$

for all $t \in \mathbb{N}$ *where* $ek^0, \ldots, ek^t \in \mathcal{U}$*. Then for all epochs* $t \in \mathbb{N}$*:*

$$\text{CEA.Verify}(st_A^t, info_B^{t+1}, context_B^t, fprint_B^{t+1}, label_B^{t+1}) = \texttt{accept} \text{ and}$$
$$\text{CEA.Verify}(st_B^t, info_A^{t+1}, context_A^t, fprint_A^{t+1}, label_A^{t+1}) = \texttt{accept}$$

where

$\text{CEA.Fprint}(st_B^t, info_B^{t+1}) \to_\$ fprint_B^{t+1}, \text{CEA.Lprint}(st_B^t, context_B^t) \to_\$ label_B^{t+1}$,
$\text{CEA.Fprint}(st_A^t, info_A^{t+1}) \to fprint_A^{t+1}$*, and* $\text{CEA.Lprint}(st_A^t, context_A^t) \to label_A^{t+1}$.

The motivation behind separating the Fprint and Update algorithms may not be immediately clear – especially in the context of the Signal protocol, where there is only a single fingerprint generated per epoch. The first reason is that this separation makes it easier to extend the formalism to capture constructions that wish to output multiple fingerprints per epoch. Imagine generating a fingerprint for each *message* exchanged between the two communicating parties, instead of each *update*, and thereby authenticating the data sent, vs. the key schedule state. The second reason is that it clearly separates the purposes between the two crucial aspects of CEA protocols, where Update performs the ratcheting and Fprint provides the fingerprint mechanism for authentication.

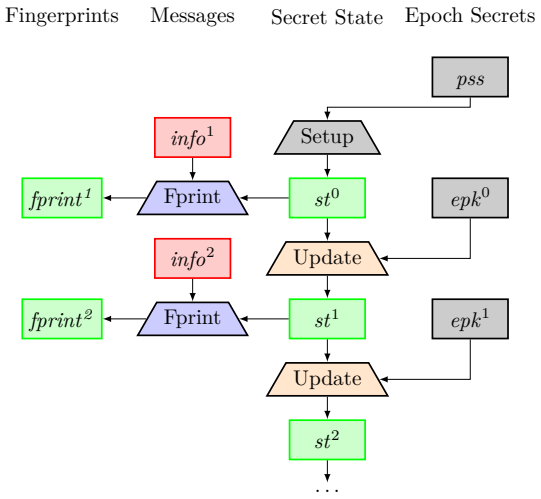

Fig. 4: CEA Key Schedule Diagram

Lprint and Fprint represent the label and fingerprint generation algorithms for authentication. Labels act as indices based on prior agreed and authenticated values, while fingerprints cover the new information to be authentication. Such information would normally include update values (e.g. public Diffie-Hellman ratcheting updates). Lprint also takes as input an optional context field. Fig. 4 illustrates the CEA key schedule.

3.2 CEA Security

Here we present the security model for a continuous entity authentication protocol. We distinguish between the adversary's ability to track conversations (given no knowledge of secrets), and to break authentication (given full compromise of secrets), and separate out CEA security for these two cases.

Unlinkability The first guarantee is considered under adversarial ability to guess whether or not a fingerprint corresponds to a given conversation (i.e. session). In more detail, even if it does not imply breaking authentication, it a weakness if the adversary is able to identify messaging or update frequency within a conversation. This also applies to the compromise scenario; it would be a strict reduction in security if the adversary was able to link session traffic even after post-compromise healing. Thus the unlinkability guarantee is quite strong, and we require that the adversary is not able to identify which session a fingerprint is tied to (i.e. the Alice and Bob pair that are communicating). The unlinkability experiment appears in Fig. 5.

$\mathsf{Exp}^{\mathsf{Unlink},\mathcal{IDS},\mathcal{E},\mathcal{F},\mathcal{L}}_{\mathrm{CEA},\mathcal{A},U}(\lambda)$:

1: $pss \leftarrow_{\$} \mathcal{S}$
2: $\mathcal{A} \rightarrow id_A, id_B$
3: **if** $id_A, id_B \notin \mathcal{ID}$ **then**
4: **return** $\bot$
5: **end if**
6: $t \leftarrow 0$
7: $b \leftarrow_{\$} \{0,1\}$
8: $FC \leftarrow$ **false**
9: $(st_A^0), (st_B^0)$ $\leftarrow_{\$}$ CEA.Setup$(\lambda, pss, id_A, id_B)$
10: $\mathcal{A}^{\mathsf{Update}(\cdot),\mathsf{Corrupt}(),\mathsf{Fprint}(\cdot)}(\lambda) \rightarrow_{\$} b'$
11: **return** $b' = b$

Update(epk):

1: $t \leftarrow t + 1$
2: Corrupt$[t] \leftarrow$ Corrupt$[t-1]$
3: **if** $epk = \bot$ **then**
4: $epk \leftarrow_{\$} \mathcal{E}$
5: Corrupt$[t] \leftarrow$ **false**
6: **end if**
7: $st_A^t \leftarrow$ CEA.Update(st_A^{t-1}, epk)
8: $st_B^t \leftarrow$ CEA.Update(st_B^{t-1}, epk)

Corrupt():

1: Corrupt$[t] \leftarrow$ **true**
2: **if** $FC =$ **true then**
3: **return** $\bot$
4: **end if**
5: **return** st_A^t, st_B^t

Fprint($info$, $context$):

1: **if** $t \bmod 2 = 0$ **then**
2: $fprint^t \leftarrow_{\$}$ CEA.Fprint$(st_A^t, info)$
3: $label^t \leftarrow$ CEA.Lprint$(st_A^t, context)$
4: **else**
5: $fprint^t \leftarrow_{\$}$ CEA.Fprint$(st_B^t, info)$
6: $label^t \leftarrow$ CEA.Lprint$(st_B^t, context)$
7: **end if**
8: **if** (Corrupt$[t-1] =$ **false**$) \wedge (b)$ **then**
9: $fprint^t \leftarrow_{\$} \mathcal{F}$
10: **end if**
11: **if** (Corrupt$[t] =$ **false**$) \wedge (b)$ **then**
12: $label^t \leftarrow_{\$} \mathcal{L}$
13: **end if**
14: $FC \leftarrow$ **true**
15: **return** $fprint^t, label^t$

Fig. 5: Unlinkability Experiment for CEA. Note that the lines in blue (Fprint lines 11 and 12) only occur in the label unlinkability game.

Definition 9 (Unlinkability Security). *We say that a* CEA *protocol* CEA.{Setup, Update, Fprint, Lprint, Verify} *is Unlink-secure if*

$$\Pr[\mathsf{Exp}^{\mathit{Unlink},\mathcal{IDS},\mathcal{E},\mathcal{F},\mathcal{L}}_{\mathrm{CEA},\mathcal{A},U}(\lambda) - 1/2] \leq \mathsf{negl}$$

for some negligible function negl.

Authentication Intuitively this property says that if Alice and Bob start in a consistent state (e.g. derived from the same pss) and receive consistent updates

from the CKA, then the Bob (the receiver) will only accept an update message sent by Alice, and vice versa. To simplify the formalization, we allow the adversary more power than he is likely to have in the real world: the adversary will be able to choose Alice and Bob's shared CEA key *pss*. (We assume Alice is sender and Bob is receiver for this round for clarity of discussion.) It may also choose any epoch to play the experiment in and has access to the secret keying material for all epochs (i.e. allowed access to an Update oracle and control of the update material *epk*).

Eventually the adversary selects some update material $info_U = info_A$ for Alice and generates Alice's $fprint_U = fprint_A$ and $label_U = label_A$ based on this choice. In this way security is based on the consistency of Alice's update material with that which Bob received, and not its secrecy. We also allow the adversary to choose the other label/fingerprint pairs stored in the log.

Bob proceeds to verify the received $info_{U'} = info_B$ against his state and the real $fprint_A$ and $label_A$. The adversary wins the game if: 1) the $label_B$ was not the valid label generated by Alice (i.e. the adversary has succeeded in desynchronizing the states of Alice and Bob) or 2) $info_{sent} \neq info_{received}$. Further details are shown in Fig. 6.

$\underline{\mathsf{Exp}^{\mathsf{CEAsec}}_{\mathrm{CEA},\mathcal{A}}(\lambda)}$:

$\mathcal{A} \rightarrow_{\$} (pss, id_A, id_B)$
if $id_A, id_B \notin \mathcal{ID}$ **then**
  return $\perp$
end if
$(st_A^0), (st_B^0) \leftarrow_{\$} \mathrm{CEA.Setup}(\lambda, pss, id_A, id_B)$
$\mathcal{A}^{\mathsf{Update}(\cdot)}(st_A^0, st_B^0) \rightarrow_{\$} (U, info_U, context_U)$
if $U = A$ **then**
  $U' \leftarrow B$
else
  $U' \leftarrow A$
end if
$fprint_U \leftarrow_{\$} \mathrm{CEA.Fprint}(st_U^t, info_U)$
$label_U \leftarrow \mathrm{CEA.Lprint}(st_U^t, context_U)$
$\mathcal{A}(fprint_U, label_U) \rightarrow_{\$} info_{U'}$
if $info_U = info_{U'}$ **then**
  return 0
end if
$fprint_{U'} \leftarrow_{\$} \mathrm{CEA.Fprint}(st_{U'}^t, info_{U'})$
$label_{U'} \leftarrow \mathrm{CEA.Lprint}(st_{U'}^t, context_U)$
$\mathsf{vfy} \leftarrow \mathrm{CEA.Verify}(st_{U'}^t, info_{U'}, context_U, fprint_U, label_U))$
if $(label_{U'} \neq label_U) \vee \big((label_{U'} = label_U) \wedge (\mathsf{vfy} = 1)\big)$ **then**
  return 1
end if
return 0

$\underline{\mathsf{Update}(epk)}$:

$st_A^{t+1} \leftarrow \mathrm{CEA.Update}(st_A^t, epk)$
$st_B^{t+1} \leftarrow \mathrm{CEA.Update}(st_B^t, epk)$
return st_A^{t+1}, st_B^{t+1}

Fig. 6: Security experiment for CEA algorithm and adversary $\mathcal{A}$.

Remark 1. In the above experiments we allow $\mathcal{A}$ to choose the *context*. For our particular CEA construction, the context is not needed; however, we wished to capture strong adversarial capabilities. Namely – similarly to how we require that security is not predicated on secrecy of keys – we do not predicate security on the authenticity of the context data.

Remark 2. For the unlinkability experiment, $\mathcal{A}$'s control of *context* means that even if $\mathcal{A}$ knows or influences the choice of context information, $\mathcal{A}$ should not be able to link the resultant fingerprints. This has parallels to IND-CPA security where control of input plaintext does not provide $\mathcal{A}$ with a guessing advantage. If a verification oracle was added to the unlinkability experiment (e.g. analogous to IND-CCA) then it would be a further requirement that the any context data previously provided during an Fprint query must be used during the a corresponding verification query.

Definition 10 (CEA **Security).** *We say that a* CEA *protocol* CEA.$\{\text{Setup}, \text{Update}, \text{Fprint}, \text{Lprint}, \text{Verify}\}$ *is* CEAsec*-secure if*

$$\Pr[\mathsf{Exp}^{\mathsf{CEAsec}}_{\text{CEA},\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}$$

for some negligible function negl.

4 Composed Protocol

In this second we introduce our second contribution by building an authenticated continuous key agreement protocol from the composition of a CKA, CEA and a logging scheme. If the underlying CKA satisfies $\mathsf{CKA\text{-}PCS}$, and CEA satisfies the CEAsec experiment, then the new combined CKA protocol should satisfy an extended security notion – Authenticated and Continuous Key Agreement (ACKA).

4.1 Authenticated Continuous Key Agreement (ACKA) protocol

We begin by formalising the notion of an Authenticated Continuous Key Agreement (ACKA) protocol. On a high-level, an ACKA protocol is an extension of CKA protocols that achieve security even in the face of an adversary that is capable of compromising secret state and injecting messages between communicating parties – as a result, it is a stronger notion of security than previous notions of CKA security.

Definition 11 (Authenticated Continuous Key Agreement). *An authenticated continuous key-agreement (*ACKA*) scheme is a tuple of algorithms* ACKA(ACKA-Init-A, ACKA-Init-A, ACKA-S, ACKA-R) *where*

- ACKA-Init-A *(and similarly* ACKA-Init-A*) is an algorithm that takes a preshared key pss and produces an initial state* $\gamma^A \leftarrow$ ACKA-Init-A(pss) *(and* γ^B*),*

– ACKA-S *is a potentially probabilistic algorithm takes a state* γ*, and produces a new state, message, and key* $(\gamma', T, I) \leftarrow\$$ ACKA-S(γ)*, and*
– ACKA-R *takes a state* γ *and a message* T*, and produces a new state and key* $(\gamma', I) \leftarrow$ ACKA-R(γ, T)*.*

The space of preshared keys pss is denote $\mathcal{P}$ *and the space of* ACKA *keys is denoted* $\mathcal{I}$*.*

$\mathsf{Exp}^{\mathsf{ACKA}}_{\mathrm{ACKA},\mathcal{A}}(\lambda)$:
1: $pss \leftarrow_\$ \mathcal{A}$
2: allow-chal$_A$, allow-chal$_B \leftarrow$ **false**
3: $r_{t_A} \leftarrow_\$$ **false**, $r_{t_B} \leftarrow_\$$ **false**
4: **if** $pss = \perp$ **then**
5: $pss \leftarrow_\$ \mathcal{S}$
6: allow-chal$_A$, allow-chal$_B \leftarrow$ **true**
7: $r_{t_A} \leftarrow_\$$ **true**, $r_{t_B} \leftarrow_\$$ **true**
8: **end if**
9: $t_A \leftarrow 0$, $t_B \leftarrow 0$, $Tr_{t_A} \leftarrow \emptyset$, $Tr_{t_B} \leftarrow \emptyset$
10: ACKA-$\gamma_A^{t_A} \leftarrow_\$$ ACKA-Init-A(pss)
11: ACKA-$\gamma_B^{t_B} \leftarrow_\$$ ACKA-Init-A(pss)
12: $b \leftarrow_\$ \{0,1\}$, **win** $\leftarrow 0$
13: $b' \leftarrow_\$ \mathcal{A}^{queries}$
14: **return** $(b' = b) \vee$ **win**

corr-A:
1: allow-chal$_A \leftarrow$ **false**
2: allow-chal$_B \leftarrow$ **false**
3: **return** CKA-γ_A^t

send-A:
1: $t_A{+}{+}$
2: (ACKA-$\gamma_A^{t_A}, T_{t_A}, I_{t_A}$)
$\leftarrow_\$$ ACKA-S(CKA-$\gamma_A^{t_A-1}$)
3: **if** $r_{t_A-i} =$ **true** $\forall i \in [1, \lceil R/2 \rceil - 1]$ **then**
4: **if** $r_{t_B-i} =$ **true** $\forall i \in [1, \lceil R/2 \rceil]$ **then**
5: allow-chal$_A =$ **true**
6: **end if**
7: **end if**
8: $Tr_{t_A} \leftarrow T_{t_A}$
9: $r_{t_A} =$ **true**
10: **return** (T_{t_A}, I_{t_A})

send-A′(r):
1: $t_A{+}{+}$
2: $r_{t_A} \leftarrow$ **false**
3: (ACKA-$\gamma_A^{t_A}, T_{t_A}, I_{t_A}$)
$\leftarrow_\$$ ACKA-S(CKA-$\gamma_A^{t_A-1}; r$)
4: $Tr_{t_A} \leftarrow T_{t_A}$
5: **return** (T_{t_A}, I_{t_A})

receive-A:
1: $t_A{+}{+}$
2: (ACKA-$\gamma_A^{t_A}, I_{t_A}$) $\leftarrow$
ACKA-R(CKA-$\gamma_A^{t_A-1}, T_{t_B}$)
3: **if** $r_{t_A-i} =$ **true** $\forall i \in [1, \lceil R/2 \rceil - 1]$ **then**
4: **if** $r_{t_B-i} =$ **true** $\forall i \in [1, \lceil R/2 \rceil]$ **then**
5: allow-chal$_A =$ **true**
6: **end if**
7: **end if**
8: **return** I_{t_A}

inject-a(T'):
1: (ACKA-$\gamma_A^{t_A+1}, I_{t_A+1}$)
$\leftarrow$ ACKA-R(CKA-$\gamma_A^{t_A}, T'$)
2: **if** $T' \neq Tr_{t_B} \wedge I_{t_A+1} \neq \perp$ **then**
3: **win** $\leftarrow 1$
4: **end if**
5: **return** I_{t_A+1}

chall-A:
1: $t_A{+}{+}$
2: (ACKA-$\gamma_A^{t_A}, T_{t_A}, I_{t_A}$)
$\leftarrow$ ACKA-S(CKA-$\gamma_A^{t_A-1}$)
3: **if** $b = 1 \wedge$ allow-chal$_A$ **then**
4: $I_{t_A} \leftarrow\$ \mathcal{K}$
5: **end if**
6: **return** (T_{t_A}, I_{t_A})

Fig. 7: Full ACKA Security Experiment Under Active $\mathcal{A}$.

ACKA Security Intuitively, ACKA security could best be described as modifying the original CKA security definition in the following way: "*any attacker attempting for forge messages will be rejected,* ***and*** *having seen a transcript* $T_1, T_2, \ldots$*, the keys* $I_1, I_2, \ldots$ *look uniformly random and independent.* The adversary is given the

power to control the random coins used by the sender and leak the current state of either party, and may also modify the messages T_i. Since the adversary may leak the state and control randomness, the keys produced under such circumstances need not be secure. ACKA security is shown in Fig. 7. Next, we formalise what it means for an ACKA protocol to be secure.

Definition 12 (ACKA **Security).** *Let* ACKA *be a continuous key agreement protocol and $R \in \mathbb{N}$ be the number of epochs until an epoch no longer contains secret information pertaining to a challenge. For a PPT algorithm $\mathcal{A}$, we define the advantage of $\mathcal{A}$ in the* ACKA *security experiment to be:*

$$\mathsf{Adv}^{\mathsf{ACKA},R}_{\mathrm{ACKA},\mathcal{A}}(\lambda) = \left| \Pr[\mathsf{Exp}^{\mathsf{ACKA},R}_{\mathrm{ACKA},\mathcal{A}}(\lambda) = 1] - \frac{1}{2} \right| .$$

We say that ACKA *is* ACKA*-secure if, for all $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{ACKA},R}_{\mathrm{ACKA},\mathcal{A}}(\lambda)$ is negligible in the security parameter λ.*

The combined Authenticated Continuous Key Agreement security experiment is depicted in Fig. 7. This experiment starts by allowing the adversary $\mathcal{A}$ to sample the pre-shared secret *pss*, capturing the scenario where the adversary has knowledge of the state involved in the protocol and may even have influenced the randomness involved, e.g. through backdooring a random number generator. The only restriction that is enforced is that at initialization the state is the same (i.e. *pss* is used to initialize both A and B). However, the alternative is also possible; the session may be established without the adversary's intervention, leading to the case of honest and secret state generation $pss \leftarrow_{\$} \mathcal{S}$. At this point, the ACKA algorithm state for parties A and B is initialized.

In addition to the option of $\mathcal{A}$ controlling the pre-shared secret, we also allow the ability for the adversary to corrupt at any point (**corr-A**).

The model captures two cases for message sending (**send-A** and **send-A**$'(r)$). Either query may be made at any epoch. In the former case, the protocol operates as normal, with A (resp. B) providing updates; furthermore, as long as the last pair of updates from each entity was honestly generated, we also set a flag to allow the adversary to challenge the given run of the protocol (i.e. guessing b'). In parallel, allowing a challenge when the two most recent updates are honest on the receiver side is captured with **receive-A**.

In the alternative case, **send-A**$'(r)$, the adversary is allowed to provide malicious ratchet randomness of its choice to be processed by the sender (including after corruption). Likewise, although in a different form of malicious action, on the receiver side we allow the adversary to inject an update message its choice to the receiver (**inject-a**(T')). However, if the receiver correctly processes an adversarially injected update message without failure, then we allow a win to the adversary.

Finally, we permit the adversary to challenge ciphertext as normal.

4.2 CKA$^{\mathsf{LOG}}$ Protocol

We provide an generalized ACKA construction, CKA$^{\mathsf{LOG}}$, that leverages any CKA protocol in composition with any CEA protocol and any Log scheme satisfying

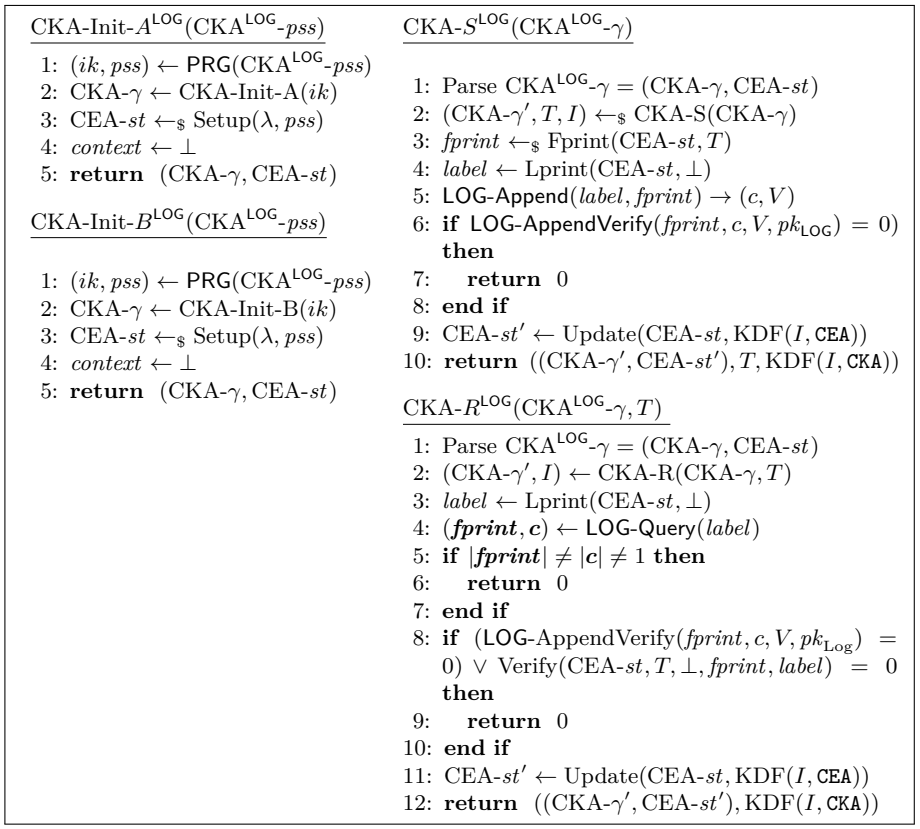

Fig. 8: Composed CKA$^{\mathsf{LOG}}$ construction for an ACKA protocol. ***fprint*** represents a vector of fingerprint values stored at the provided label in the log.

Logsec to achieve ACKA security. The CKA$^{\mathsf{LOG}}$ protocol construction shown in Fig. 8 is based on one-sided man-in-the-middle detection (the receive detects forgery). Other constructions for various detection combinations are also possible.

The construction proceeds with the normal CKA update information generation, which is then also used to generate a corresponding fingerprint. We align the output space of the KDF, fingerprint space $\mathcal{E}$, and the second component of the input space of Update.

The sender then posts the label and fingerprint for the epoch on the log, receiving a confirmation, and updates the CEA state. On receive of CKA update information, the receiver proceeds with the normal CKA key computation and state update, but also generates the epochal label. Note that the label is deterministic and generated off of the last shared state, being independent of the current update information. The receiver then queries the Log on the label and receives back all fingerprints.

For both the sender and receiver, the underlying CKA epochal key I is used to derive two separate keys – one for the authentication CEA state and one for

the key exchange CKA state. This is done via a KDF over text differentiators `CKA` and `CEA`, and is to ensure key separation. Fig. 9 provides a high-level illustration of an ACKA composed protocol.

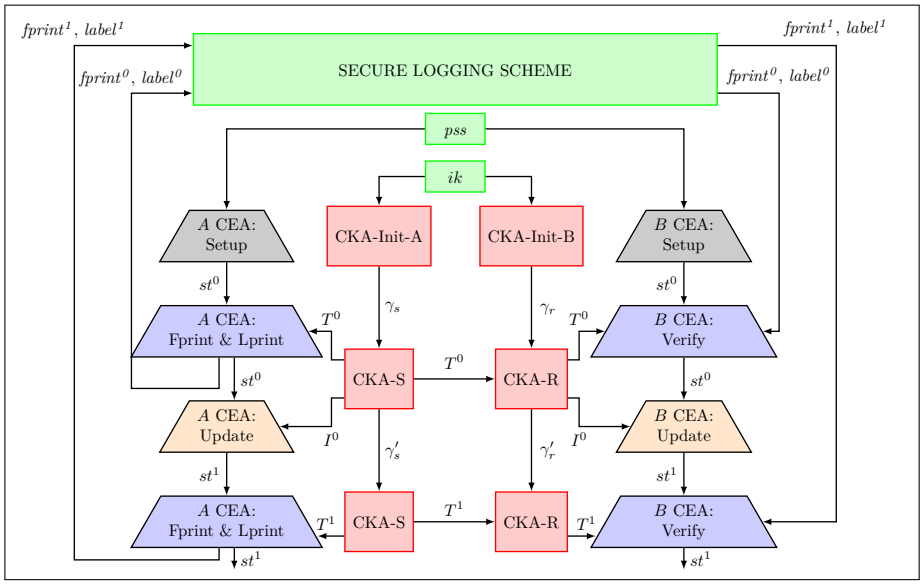

Fig. 9: Detailed Composition Diagram. For cleanness of presentation, some details are omitted. In addition, in this figure A produces fingerprints and B verifies them – in our formalism both parties have the ability for these actions.

The composed protocol is designed to meet the ACKA security experiment description (Fig. 7), based on its ability to satisfy CKA security (Fig. 2), CEA authentication security (Fig. 6), CEA unlinkability security (Fig. 5), and Log security (Fig. 3).

Remark 3. While Fig. 8 is a CKA-based construction (i.e. following the CKA definition), it is modified to include oracle access to a log, LOG, with corresponding Append and AppendVerify queries. These correspond to the logging algorithm in Definition 4 with the exception of secret state, i.e. they represent how an external party, Alice, may query the log to process the Append (resp. AppendVerify) algorithm on the provided inputs.

5 Constructions

We now give a specific CEA construction as a proof of concept.

5.1 CEA Construction

Fig. 10 shows a CEA protocol construction. There are other potential constructions that satisfy the CEA definition. The choices made in this construction are for simplicity of the protocol as well as its security analysis.

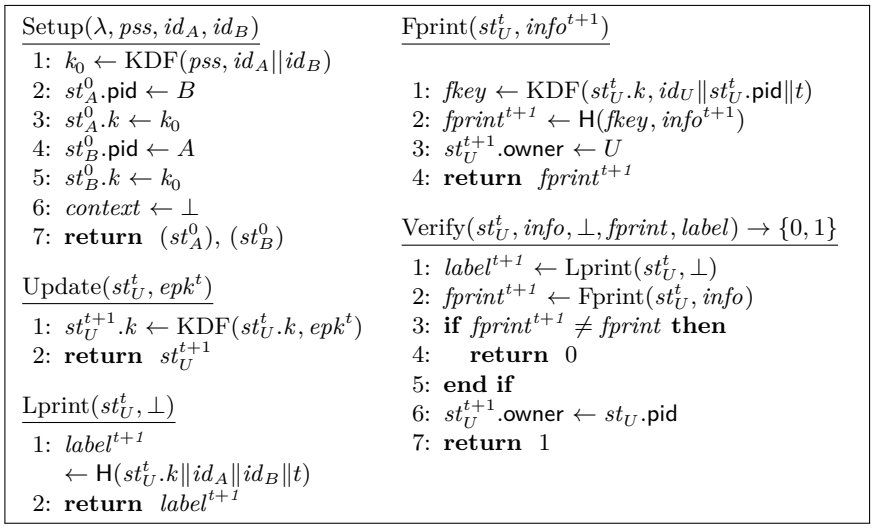

Fig. 10: An instantiation of a Continuous Entity Authentication Protocol.

Lprint computes the label over the current state and partner identities, i.e. already agreed values. Then Fprint computes the fingerprint over the intended update info. For key separation in the proof, Fprint calculates *fkey* based on the current state, before the hash over the update information.

It is of note that, despite the analogous view mentioned in Section 3 of *fprint* as a MAC tag, we do not actually use a MAC algorithm to generate it in this construction. Instead, we rely on a hash algorithm. This is due to the collision resistance that is required vs. key randomness properties (see Section 6).

When collision resistance is assessed for a MAC, it is about collision over the function inputs exclusive of the MAC key. In normal MAC use this is a natural requirement; an adversary without access to the key should not be able to find two inputs that result in collision on the MAC tag. However, under CEA security we assume that a compromise has taken place and the adversary therefore has access to all keying material – thus collision resistance must also cover keying material, making the security assumption different from typical MAC collision resistance. Since collision resistance must be across all inputs, we are led to a natural choice of a hash algorithm in Line 2 of *fprint* generation.

Remark 4. This construction and the ensuing security analyses in Section 6 provide insight into non-standard assumptions for cryptographic security. Cryptographic security is usually bootstrapped from secret keying material of some variety (symmetric or asymmetric) which is inaccessible to the adversary during a time window of interest. Naturally, this leads to assumptions on pseudorandomness and occasional reliance on the Random Oracle Model for a KDF. This work highlights that a form of cryptographic security for a protocol is still possible even if the adversary has access to all secret keys – both session keys and long-term authentication secrets, with the only constraint being consistency at some point

in the past (Alice and Bob holding the same view). Perhaps unsurprisingly, this protocol authentication property is bootstrapped from a related algorithm-level property; collision resistance of a hash function be the analogous security definition enforcing a matching view of prior inputs.

While we do not exploit the context field for our construction in Fig. 10, due to selection of one-way detection (only the receive side detects a MitM), it can be used to generate separate labels for different parties, thus allowing for bi-directional detection.

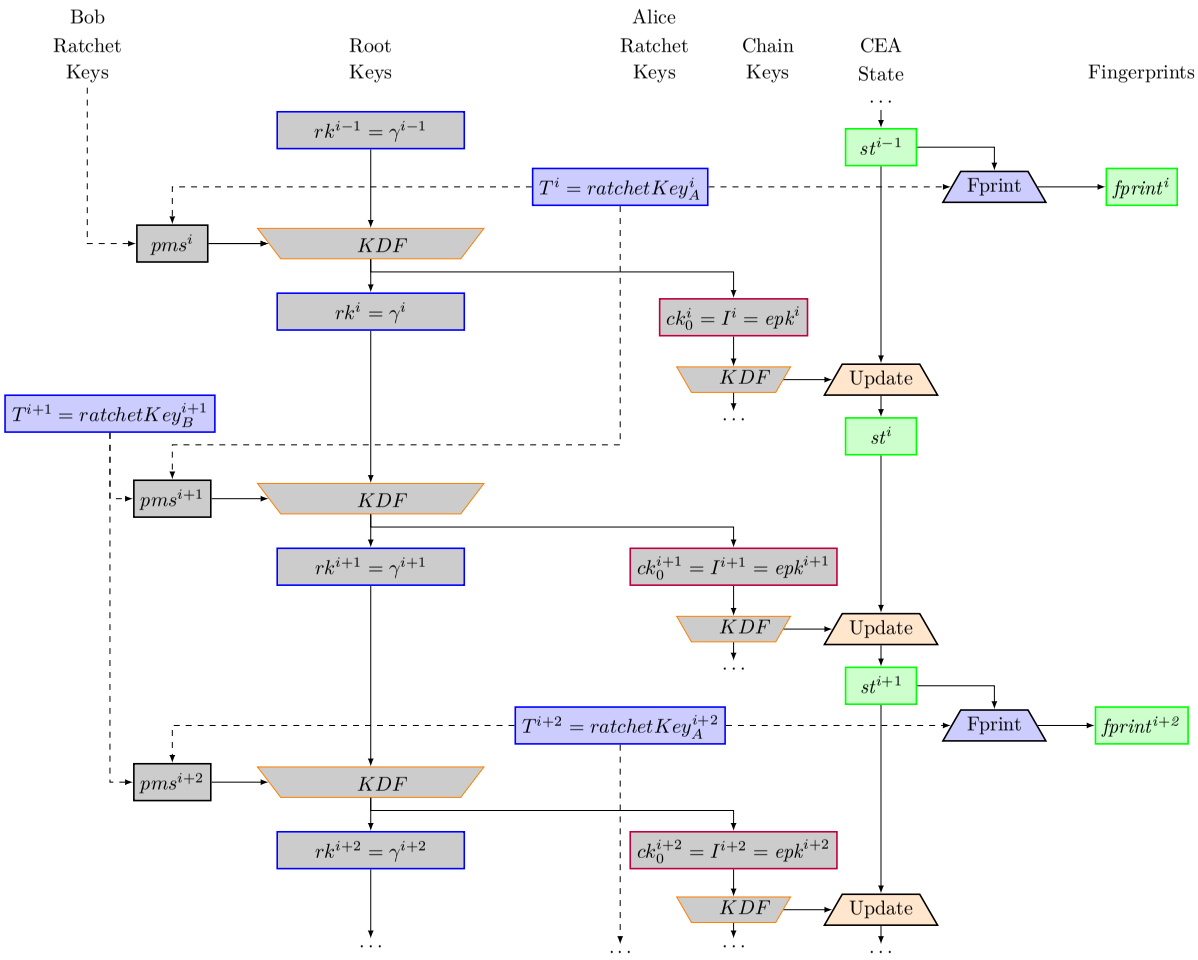

Fig. 11: The modified key schedule from the Signal Double Ratchet protocol based on the construction.

5.2 ACKA Construction

Fig. 11 provides a high-level illustration of an ACKA composed protocol, using the Signal Double Ratchet protocol as an example underlying CKA (shown in gray) and the specific CEA construction as Fig. 10 (shown in color). Note that by the generalized construction in Fig. 8, other CKA variants could also replace Signal – this is a construction example only.

6 Security Analysis

In this section, we prove that our instantiation of a continuous entity authentication protocol (described in Figure 10) achieves CEAsec and Unlink security. Then we turn to proving that our generalized, composed $\text{CKA}^{\mathsf{LOG}}$ construction achieves active CKA (or ACKA) security.

CEAsec-security of the CEA instantiation. We begin by analysing the CEAsec-security of the CEA instantiation.

Theorem 1. *The continuous entity authentication protocol* CEA *described in Figure 10 is* CEAsec*-secure under the collision-resistance of the hash function* H*. That is, for any PPT algorithm* $\mathcal{A}$ *against the* CEAsec *security game (described in Figure 6)* $\mathsf{Adv}^{\mathsf{CEAsec}}_{\text{CEA},\mathcal{A}}(\lambda)$ *is negligible.*

Proof. Recall that by the description of the CEAsec security game in Figure 6, that $\mathcal{A}$ can win in two ways:

1. $\mathcal{A}$ outputs a pair of strings $(info_U, info_{U'})$ such that:
   $(fprint_U, label_U) \leftarrow \text{CEA.Fprint}(U.st, info_U)$
   $(fprint_{U'}, label_{U'}) \leftarrow \text{CEA.Fprint}(U'.st, info_{U'})$
   but $label_U \neq label_{U'}$
2. $\mathcal{A}$ outputs a pair of strings $(info_U, info_{U'})$ such that:
   $(fprint_U, label_U) \leftarrow \text{CEA.Fprint}(U.st, info_U)$
   $1 \leftarrow \text{CEA.Verify}(U'.st, info_{U'}, fprint_U, label_U)$.

In either case, the adversary $\mathcal{A}$ has full control of the update and preshared state inputs, there exists no secret state unknown nor uncontrolled by $\mathcal{A}$. In what follows, we divide the proof into the two cases above, and bound $\mathcal{A}$'s advantage in the CEAsec game by the sum of their advantages in either case.

Case 1 We begin by analyzing the case where $\mathcal{A}$ has caused the challenger to output two distinct labels after the same update values. We note that for our construction, the label is computed as $label_t = \mathsf{H}(st^t_U.k \| id_A \| id_B \| t)$ in epoch t.

We note that the epoch index t is incremented during each Update query sent by $\mathcal{A}$, which updates *both* parties epoch index identically. In addition, id_A and id_B are public, static values. Thus for the computation of $label_U$ and $label_{U'}$, these values are equal. Thus, since H is a deterministic hash function, all we must do is demonstrate that for both parties, $st^t_U.k = st^t_{U'}.k$ in each epoch.

Both parties will compute the secret state as $st_U^t.k \leftarrow \text{KDF}(st_U^t.k, ek^t)$. In addition, when $\mathcal{A}$ calls the Update query, both parties receive the same ek^t. Thus, if $st_U^t.k = st_{U'}^t.k$ and KDF is a deterministic function, then when $\mathcal{A}$ calls Update, both parties compute $st_U^{t+1}.k = st_{U'}^{t+1}.k$. We note that when the experiment begins, both parties compute the first epoch secret state as $st_U^0.k = \text{KDF}(pss, id_A \| id_B)$. Since both parties are given the same value pss as input, all other values are public and static, and KDF is a deterministic function, then the advantage of $\mathcal{A}$ in causing the challenger to output two distinct labels $label_U$, $label_{U'}$ is 0.

Case 2 We now analyze the case where $\mathcal{A}$ outputs a pair of strings $(info_U, info_{U'})$ such that $(fprint_U, label_U) \leftarrow \text{CEA.Fprint}(U.st, info_U)$ but $1 \leftarrow \text{CEA.Verify}(U'.st, info_{U'}, fprint_U, label_U)$.

We note that for our construction, $fprint' \leftarrow \mathsf{H}(fkey^t, info)$, and Verify simply computes the $fprint$ value locally and accepts (and thus outputs 1 if $fprint_U = fprint_{U'}$. Note that $fkey^t$ is computed as $fkey^t \leftarrow \text{KDF}(st_U^{t-1}.k, id_U \| id_{U'} \| t)$. By the same argument as *Case 1*, since KDF is a deterministic function and $st_U^{t-1}.k$ are computed identically for both parties, thus $fkey^t$ is similarly equal for both parties.

Thus, it must be that for the challenger to output 1, $\mathsf{H}(fkey^t, info_U) = \mathsf{H}(fkey^t, info_{U'})$. We bound the advantage of $\mathcal{A}$ in winning in this case by the collision-resistance of the hash function H. Specifically, the challenger aborts if any two hash evaluations compute the same hash value for different inputs. We can trivially break the collision-resistance of H when this occurs by outputting the two distinct input values to the collision-resistance challenger of H.

Combining our results we find that $\mathsf{Adv}^{\mathsf{CEAsec}}_{\text{CEA},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H}}(\lambda)$.

Unlink-security of the CEA instantiation.

We now analyse the Unlink-security of the CEA instantiation. This will later allow us to argue that displaying the fingerprints and labels generated by the communicating parties does not negatively impact the user traceability of the overall CKA protocol.

Theorem 2. *The continuous entity authentication protocol* CEA *described in Figure 10 is Unlink-secure under the PRF and dual-PRF security of* KDF *and modelling* H *as a random oracle. That is, for any PPT algorithm $\mathcal{A}$ against the Unlink security game (described in Figure 5)* $\mathsf{Adv}^{\mathit{Unlink}}_{\text{CEA},n_S,\mathcal{A}}(\lambda)$ *is negligible.*

Proof. Recall that in the Unlink security game in Figure 5, $\mathcal{A}$ can win by guessing the bit b sampled uniformly at random by the challenger at the beginning of the game. We note that the behaviour of the experiment differs depending on the value of b when $\mathcal{A}$ has not compromised the state of the parties at an epoch t, and then calls Fprint. In this case, then the fingerprint $fprint$ and the label $label$ is not computed normally but instead sampled uniformly at random from the corresponding fingerprint and label spaces $\mathcal{F}$, $\mathcal{L}$. We prove that $\mathcal{A}$'s advantage in distinguishing this change is negligible in the following game hops.

Game 0. This is the unlinkability game described in Figure 5. Thus we have:

$$\mathsf{Adv}^{\mathsf{Unlink}}_{\mathrm{CEA},n_S,\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathtt{Game0}}_{\mathcal{A}}(\lambda) \ .$$

Game 1. In this game, we replace the secret state computed by both parties $st_U^{t+1}.k \leftarrow \mathrm{KDF}(st_U^t.k, epk_t)$ with a uniformly random value $\widetilde{st_U^{t+1}.k}$ when $\mathcal{A}$ calls $\mathsf{Update}(\perp)$. Note that when this occurs, epk_t is sampled uniformly at random from the epoch space $\mathcal{E}$ instead of being provided by $\mathcal{A}$. Note that for the behaviour of the Unlink game to change based on the bit b, such an Update call must be made.

We make this replacement by calling a dual-PRF challenger PRF with input $st_U^t.k$, and replace $st_U^{t+1}.k$ with the output from the dual-PRF challenger. Since epk^i is uniformly at random and unknown to $\mathcal{A}$, this replacement is sound. We note that by the security of the dual-PRF assumption, $\mathcal{A}$'s advantage in distinguishing this replacement is negligible. Since there are a maximum of n_S epochs, the difference in $\mathcal{A}$'s advantage is bound by:

$$\mathsf{Adv}^{\mathtt{Game0}}_{\mathcal{A}}(\lambda) \leq n_S \cdot \mathsf{Adv}^{\mathbf{dualPRF}}_{\mathrm{KDF},\mathcal{B}}(\lambda) + \mathsf{Adv}^{\mathtt{Game1}}_{\mathcal{A}}(\lambda) \ .$$

Game 2. In this game, we replace the secret state computed by both parties $st_U^{t+1}.k \leftarrow \mathrm{KDF}(\widetilde{st_U^t.k}, epk_t)$ with a uniformly random value $\widetilde{st_U^{t+1}.k}$ when $\mathcal{A}$ calls $\mathsf{Update}(epk_t)$ but epoch t is uncorrupted.

We make this replacement by calling a PRF challenger with input epk_t, and replace $st_U^{t+1}.k$ with the output from the PRF challenger. Since $\widetilde{st_U^t.k}$ is uniformly at random and unknown to $\mathcal{A}$, this replacement is sound. We note that by the security of the PRF assumption, $\mathcal{A}$'s advantage in distinguishing this replacement is negligible. Since there are a maximum of n_S epochs, the difference in $\mathcal{A}$'s advantage is bound by:

$$\mathsf{Adv}^{\mathtt{Game1}}_{\mathcal{A}}(\lambda) \leq n_S \cdot \mathsf{Adv}^{\mathbf{PRF}}_{\mathrm{KDF},\mathcal{B}}(\lambda) + \mathsf{Adv}^{\mathtt{Game2}}_{\mathcal{A}}(\lambda) \ .$$

Game 3. In this game, we replace the fingerprint key *fkey* computed by both parties $\mathit{fkey} \leftarrow \mathrm{KDF}(\widetilde{st_U^{t-1}.k}, id_U \| st_U^t.\mathsf{pid} \| t)$ with uniformly random values $\widetilde{\mathit{fkey}}$, $\widetilde{\mathsf{aux}}$ when $\mathcal{A}$ calls $\mathrm{Fprint}(\mathit{info})$ but epoch $t-1$ is uncorrupted.

We make this replacement by calling a PRF challenger with input $id_U \| st_U^t.\mathsf{pid} \| t$, and replace *fkey* with the output from the KDF challenger. Since $\widetilde{st_U^{t-1}.k}$ is uniformly at random (by Game 1 and Game 2) and unknown to $\mathcal{A}$ (since $\mathsf{Corrupt}[t-1] \neq \mathbf{true}$), this replacement is sound. We note that by the security of the PRF assumption, $\mathcal{A}$'s advantage in distinguishing this replacement is negligible. Since there are a maximum of n_S epochs, the difference in $\mathcal{A}$'s advantage is bound by:

$$\mathsf{Adv}^{\mathtt{Game2}}_{\mathcal{A}}(\lambda) \leq n_S \cdot \mathsf{Adv}^{\mathbf{PRF}}_{\mathrm{KDF},\mathcal{B}}(\lambda) + \mathsf{Adv}^{\mathtt{Game3}}_{\mathcal{A}}(\lambda) \ .$$

Game 4. In this game, we replace the fingerprint computed by both parties $\mathit{fprint} \leftarrow \mathsf{H}(\widetilde{\mathit{fkey}}, \mathit{info})$ with a uniformly random value $\widetilde{\mathit{fprint}}$ when $\mathcal{A}$ calls $\mathrm{Fprint}(\mathit{info})$ but epoch $t-1$ is uncorrupted. In addition, we replace the label *label* computed by both parties $\mathit{label} \leftarrow \mathsf{H}(\widetilde{st_U^t.k} \| id_U \| st_U^t.\mathsf{pid} \| t)$ with a uniformly random value $\widetilde{\mathit{label}}$ when $\mathcal{A}$ calls $\mathrm{Fprint}(st_U^t, \mathit{info})$ but epoch t is uncorrupted.

Recall that we model hash functions as random oracles. Thus, this game is merely a syntatic change, as the fingerprint and label are already sampled uniformly at random by the random oracle. Since $\widetilde{st_U^t.k}$ is uniformly at random (by Game 1 and Game 2) and unknown to $\mathcal{A}$ (since $\mathsf{Corrupt}[t-1] \neq \mathbf{true}$), and $\widetilde{\mathit{fkey}}$ is uniformly at random (by Game 3), this replacement is sound. Thus both *label* and *fprint* are computed uniformly-at-random, and thus the behaviour of the challenger when $b = 0$ and $b = 1$ is identical. Thus we have: $\mathsf{Adv}^{\mathtt{Game}3}_{\mathcal{A}}(\lambda) = 0$.

Summing our advantages we arrive at our result:

$$\mathsf{Adv}^{\mathsf{Unlink}}_{\mathrm{CEA},n_S,\mathcal{A}}(\lambda) \leq n_S \cdot \mathsf{Adv}^{\mathbf{dualPRF}}_{\mathrm{KDF},\mathcal{B}}(\lambda) + 2n_S \cdot \mathsf{Adv}^{\mathbf{PRF}}_{\mathrm{KDF},\mathcal{B}}(\lambda) \ .$$

ACKA-security of the $\mathrm{CKA}^{\mathsf{LOG}}$ composed protocol.

With all pieces in place, we can now analyse the ACKA-security of the $\mathrm{CKA}^{\mathsf{LOG}}$ composed protocol.

Theorem 3. *The composed* $\mathrm{CKA}^{\mathsf{LOG}}$ *protocol* CEA *described in Figure 8 is* ACKA*-secure under the* CKA-PCS *security of the continuous key agreement protocol* CKA*, the* CEAsec*-security of the continuous entity authentication protocol* CEA *and the* Logsec-Exclude *security of the logging scheme* LOG*. That is, for any PPT algorithm* $\mathcal{A}$ *against the* ACKA *security experiment (described in Figure 7),* $\mathsf{Adv}^{\mathrm{ACKA}}_{\mathrm{CKA}^{\mathsf{LOG}},\mathcal{A}}(\lambda)$ *is negligible.*

Proof. Recall that by the description of the ACKA security game (in Figure 7) that $\mathcal{A}$ can win if it guesses the bit b sampled by the challenger at the beginning of the experiment. We note that bit b only influences the behaviour of the security game when sampling the I_{t_A} value created during a **chall-A** (resp. I_{t_B} during a **chall-b**) query. In what follows, we divide the proof into two cases:

1. $\mathcal{A}$ never queried **inject-a**(T'), and;
2. $\mathcal{A}$ has queried **inject-a**(T') at least once.

We focus on the second case first, and show that any adversary that queries **inject-a**(T') will cause the game to be aborted when processing **inject-a**(T'). As a result, we need only prove that an attacker that has never called **inject-a**(T') cannot distinguish between the cases where $b = 0$ and $b = 1$.

Case 1: $\mathcal{A}$ has queried inject-a(T') at least once.

Game 0. This is the ACKA game described in Figure 7. Thus we have:

$$\mathsf{Adv}^{\mathrm{ACKA}}_{\mathrm{CKA}^{\mathsf{LOG}},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathtt{Game}0}_{\mathcal{A}}(\lambda) \ .$$

Game 1. In this game, we abort the game if $\mathcal{A}$ causes A and B to produce two different labels from the underlying CEA protocol.

Specifically, when $\mathcal{A}$ queries **inject-a**(T'), the challenger aborts if A computes a different label $label_B{}'$ to B's $label_B$. To do so, whenever $\mathcal{A}$ queries **send-A**, the challenger calls $\mathsf{Update}(I_{t_A})$ to the CEAsec challenger. The first time $\mathcal{A}$ queries **inject-a**T', then the challenger returns (B, T_{t_B}) to the CEAsec challenger (where T_{t_B} is the last public keyshare output by B). The CEAsec challenger will output $fprint_B$, $label_B$, which will then be uploaded to the LOG, and T' is returned to the CEAsec challenger.

We note that if $T' \neq T_{t_B}$ and $\text{CEA.Lprint}(st_t^A, \perp) \neq label_B$, then $\mathcal{A}$ has won the CEAsec security experiment. Thus we have

$$\mathsf{Adv}_{\mathcal{A}}^{\mathtt{Game0}}(\lambda) \leq \mathsf{Adv}_{\text{CEA},\mathcal{B}}^{\mathsf{CEAsec}}(\lambda) + \mathsf{Adv}_{\mathcal{A}}^{\mathtt{Game1}}(\lambda) \ .$$

Game 2. In this game, we abort if the adversary $\mathcal{A}$ injects T' to A (in some epoch t), but A and B compute the same fingerprint in epoch t, despite $T' \neq T_{t_B}$. We proceed identically as in Game 2, and note that if $\mathcal{A}$ causes this to occur, then $\mathcal{A}$ can be used to win the CEAsec game (specifically, the two fingerprints $fprint^A$, $fprint^B$ generated by A and B are returned to the CEAsec challenger).

Since we made no additional changes to the behaviour of the challenger, this introduces no additional advantage for $\mathcal{A}$. Thus we have

$$\mathsf{Adv}_{\mathcal{A}}^{\mathtt{Game1}}(\lambda) \leq \mathsf{Adv}_{\mathcal{A}}^{\mathtt{Game2}}(\lambda) \ .$$

Game 3. In this game, we abort if the adversary $\mathcal{A}$ causes a view V_A of the log LOG such that $\text{AppendVerify}(fprint_A, c', V, pk_{\text{Log}}) = 1$. We do so by interacting with a $\mathsf{Logsec\text{-}Exclude}$ challenger. Each time $\mathcal{A}$ calls **send-A** or **send-b**, the challenger saves $(label, T_{t'}, r)$ (where r is some sampled randomness for LOG). When $\mathcal{A}$ calls **inject-a**(T') at some epoch t, then the challenger returns $((label^0, fprint^0, r_0), \ldots, (label^{t-1}, fprint^{t-1}, r_{t-1}), (label^t, fprint, r)), label, fprint$ to the $\mathsf{Logsec\text{-}Exclude}$ challenger. If this sequence of (label, fingerprint) pairs would allow $\mathcal{A}$ to create a view of the log that would verify $\text{AppendVerify}(fprint_A, c', V, pk_{\text{Log}}) = 1$ (thus creating a view of the log that "excludes" B's fingerprint $fprint^{t-1}$ by verifying without it), then the challenger would win against the $\mathsf{Logsec\text{-}Exclude}$ security game. Thus we have

$$\mathsf{Adv}_{\mathcal{A}}^{\mathtt{Game2}}(\lambda) \leq \mathsf{Adv}_{\mathsf{LOG},\mathcal{B}}^{\mathsf{Logsec\text{-}Exclude},}(\lambda) + \mathsf{Adv}_{\mathcal{A}}^{\mathtt{Game3}}(\lambda) \ .$$

At the end of Game 3 we note that if $\mathcal{A}$ calls **inject-a**(T') in epoch t such that $T_{t-1} \neq T'$, then:

- If only one fingerprint $fprint$ corresponding to A's label in epoch t exists in the log, then this must be B's fingerprint. Not that A would not generate the same fingerprint as B, and thus the A would reject the new epoch when comparing B's fingerprint to her own.

– If more than one fingerprint *fprint* corresponding to A's label in epoch t exists in the log, then this is both B's and $\mathcal{A}$'s fingerprint, and AppendVerify would fail to verify correctly, since A will only be verifying the fingerprint that results from $\mathcal{A}$'s injected keyshare, and thus A would reject the new epoch.

Thus, in Game 3 the advantage $\mathcal{A}$ has in causing A to process an **inject-a**(T') is negligible. Symmetrically, this argument can be applied to B at the cost of an additional factor of 2 to $\mathcal{A}$'s advantage, and thus:

$$\mathsf{Adv}^{\mathtt{Game3}}_{\mathcal{A}}(\lambda) = 0 \ .$$

We now turn to the case where $\mathcal{A}$ never queries **inject-a**(T').

Case 2: $\mathcal{A}$ has never queried inject-a(T').

Game 0. This is the ACKA game described in Figure 7. Thus we have:

$$\mathsf{Adv}^{\mathrm{ACKA}}_{\mathrm{CKA}^{\mathrm{LOG}},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathtt{Game0}}_{\mathcal{A}}(\lambda) \ .$$

Game 1. In this game, we replace the secrets output by **chall-A** with uniformly random values $\widetilde{I} \leftarrow_{\$} \mathcal{I}$. We do so by replacing all computations for the underlying CKA scheme with queries to a CKA-PCS challenger. We note that the CKA-PCS security experiment almost identically matches the ACKA security game, with the exception that in the ACKA security game $\mathcal{A}$ is able to inject keyshares. However, we are in **Case 2**, where $\mathcal{A}$ never calls **inject-a**. As a result, this exactly matches the CKA-PCS game, and thus whenever $\mathcal{A}$ is successful in the ACKA security experiment, we can construct an adversary $\mathcal{A}'$ against the CKA-PCS of the underlying CKA. Thus we have

$$\mathsf{Adv}^{\mathrm{ACKA}}_{\mathrm{CKA}^{\mathrm{LOG}},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathsf{CKA\text{-}PCS}}_{\mathrm{CKA},\mathcal{B}}(\lambda) \ .$$

References

1. The complexities of healing in secure group messaging: Why cross-group effects matter pp. 1847–1864 (2021)
2. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the signal protocol. In: EUROCRYPT (2019)
3. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) Theory of Cryptography. pp. 261–290. Springer International Publishing (2020)
4. B. Laurie, A. Langley, and E. Kasper: Certificate transparency. Tech. rep. (2013), `http://www.rfc-editor.org/rfc/rfc6962`
5. Blazy, O., Bossuat, A., Bultel, X., Fouque, P.A., Onete, C., Pagnin, E.: SAID: Reshaping Signal into an Identity-Based Asynchronous Messaging Protocol with Authenticated Ratcheting. IACR Cryptology ePrint Archive (2019)
6. Borisov, N., Goldberg, I., Brewer, E.: Off-the-record communication, or, why not to use pgp. In: Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society. p. 77–84. WPES '04 (2004)
7. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy, (Euro S&P). pp. 451–466 (2017)
8. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On Post-compromise Security. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016. pp. 164–178 (2016). https://doi.org/10.1109/CSF.2016.19
9. Dowling, B., Günther, F., Poirrier, A.: Continuous authentication in secure messaging. In: Computer Security – ESORICS 2022. pp. 361–381. Springer Nature Switzerland (2022)
10. Dowling, B., Hale, B.: Secure Messaging Authentication against Active Man-in-the-Middle Attacks. 2021 IEEE European Symposium on Security and Privacy, (Euro S&P) (2021)
11. Dowling, B., Hale, B.: Secure messaging authentication against active man-in-the-middle attacks. In: 2021 IEEE European Symposium on Security and Privacy (EuroS P). pp. 54–70 (2021)
12. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement without key-update primitives. IACR Cryptol. ePrint Arch. **2018**, 889 (2018)
13. GmbH, W.S.: Wire Security Whitepaper. Tech. rep. (October 2020), `https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf`
14. Günther, C.G.: An identity-based key-exchange protocol. In: Quisquater, J.J., Vandewalle, J. (eds.) Advances in Cryptology — EUROCRYPT '89. pp. 29–37. Springer Berlin Heidelberg (1990)
15. Iyengar, J., Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport, RFC 9000. `https://datatracker.ietf.org/doc/rfc9000/` (Feb 2022)
16. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Advances in Cryptology - EUROCRYPT 2019. Lecture Notes in Computer Science, vol. 11476, pp. 159–188. Springer (2019)
17. Jost, D., Maurer, U., Mularczyk, M.: A unified and composable take on ratcheting. IACR Cryptol. ePrint Arch. **2019**, 694 (2019)
18. Krawczyk, H.: Cryptographic extraction and key derivation: The hkdf scheme. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. pp. 631–648. Springer Berlin Heidelberg (2010)

19. Marlinspike, M., Perrin, T.: The Signal Protocol. Tech. rep. (November 2016), https://signal.org/docs/specifications/x3dh/
20. Marlinspike, M., Perrin, T.: The Signal Protocol: Key Compromise. Tech. rep. (November 2016), https://signal.org/docs/specifications/x3dh/#key-compromise
21. Milner, K., Cremers, C., Yu, J., Ryan, M.: Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Applications. In: 2017 IEEE 30th Computer Security Foundations Symposium (CSF). pp. 203–216 (2017)
22. Poettering, B., Rösler, P.: Asynchronous ratcheted key exchange. CRYPTO, '18 (2018)
23. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. https://tools.ietf.org/html/rfc8446 (Aug 2018)
24. Rösler, P., Mainka, C., Schwenk, J.: More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In: 2018 IEEE European Symposium on Security and Privacy, (Euro S&P). pp. 415–429 (2018)
25. Sullivan, N.: Keyless SSL: The Nitty Gritty Technical Details. Tech. rep. (2014), https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/
26. Tan, J., Bauer, L., Bonneau, J., Cranor, L.F., Thomas, J., Ur, B.: Can unicorns help users compare crypto key fingerprints? In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. pp. 3787–3798. CHI '17, ACM (2017)