A preliminary version of this paper appears in the proceedings of the 46th Annual International Cryptology Conference (Crypto 2026). This is the full version.

On the State-Compromise Security of End-to-End Real-Time Group Communication

Mang Zhao[0009−0002−4720−6551] and Qian Wang ✉[0000−0002−8967−8525]

School of Cyber Science and Engineering, Wuhan University, Wuhan, P.R.China
{mang.zhao, qianwang}@whu.edu.cn

Version 1.1, June 15, 2026

Abstract. Real-time group communication protocols, such as Zoom and Microsoft Teams, aim to provide end-to-end security for audio and video conferences even in the presence of a malicious server. Despite their widespread deployment, particularly since the COVID-19 pandemic, their intended security guarantees lack comprehensive formalization. Prior work largely focuses on Zoom, and analyzes its security in models that rely on restrictive assumptions, such as the existence of a trusted server at certain points in time or a long-lived leader that never leaves the group. Moreover, existing analyses assume that group-specific session states of group members are secure and incorruptible, leaving the impact of potential full state compromise on group security unexplored.
In this work, we propose a set of essential security guarantees for real-time group communication with state-compromise resilience against fully malicious servers and provide the first construction that provably satisfies all of these guarantees. To formally prove that our design achieves its goal, we formalize a novel *continuous group key distribution* protocol and introduce an associated security model that captures all the intended guarantees. We propose a generic construction that is provably secure in this model and suggest both classical and post-quantum secure instantiations.
Besides these main design goals, we introduce a novel *multi-recipient authenticated key encapsulation mechanism*, which serves as a building block for our generic construction. We define two core security notions for maKEM, propose both concrete and generic constructions, and prove their security in the random oracle model and the standard model, respectively.

1 Introduction

Real-time group communication such as video and audio conferencing allows a group of people to synchronously meet in a virtual room. Since COVID-19 pandemic, real-time group communication and its applications such as Zoom,

Cisco Webex, Google Meeting, and Microsoft Teams, have become essential in the daily lives of hundreds of millions of users. While most of these applications claim to provide some form of end-to-end security, i.e., group meetings are secure even in the presence of malicious servers, formal studies on their provable security remain surprisingly limited.

To the best of our knowledge, all existing works [25,24] that formally analyze the security of real-time group communication protocols focus exclusively on Zoom, and their security models still rely on relatively strong trust assumptions about the server. For example, Dodis, Jost, Kesavan, and Marcedone [25] provide a pioneer security analysis for Zoom client version 5.12 and prove its security in a model that considers a "powerful attacker that controls the group evolution and fully controls the network and Zoom's server infrastructure". However, certain trust assumptions about the Zoom server remain necessary in their model. These include: (i) a trusted public key infrastructure (PKI) that honestly distributes group members' public keys; (ii) an honest access-control mechanism (ACM) that allows only authorized parties to join their intended groups; and (iii) an honest group roster distribution mechanism that provides the latest roster to a newly appointed leader when the previous one leaves. These assumptions imply that the security model in [25] captures only partial end-to-end security.

Cremers, Ronen, and Zhao [24] conduct a concurrent security analysis for Zoom white-paper version 4.0. They prove the security of Zoom in a weaker model assuming a trusted PKI and present a transformation that provably renders Zoom secure in the stronger model that "makes no assumptions on the server". However, the stronger model in [24] assumes that the every group has a long-lived leader who never leaves. This impractical assumption, on one hand, prevents a leader from unexpectedly leaving the group, e.g., due to network loss, and on the other hand, prevents attackers from freely injecting malicious leave notifications on behalf the leader, thereby restricting the resulting security guarantees to a partially malicious server setting.

Moreover, both works assume that the long-term states of group members may be corrupted, while their group-specific session states remain secure due to the "short-liveness" of real-time group communication. We argue, however, that this assumption does not fully capture real-world deployment scenarios. Currently, most video conferencing applications, such as Zoom, Cisco Webex, Google Meet, and Microsoft Teams, support meetings lasting up to 24 hours, which significantly extends the exposure window of group-specific states. This suggests that the "short-liveness" assumption may no longer accurately reflect modern deployment scenarios, thereby increasing the likelihood of session-state compromise, for example via malware on compromised devices. Thus, the risks of session-state compromise, and even full state compromise, cannot be neglected.

These observations highlight the limitations of existing security formalizations for fully malicious servers and naturally raise the following questions:

Is it possible to formalize a security model for real-time group communication against fully malicious servers with potential full state compromise capabilities? If so, how can we design a protocol that is provably secure in such a mode?

	Assumptions		Security Guarantees						
	M-PKI	**M-ACM**	**(S)KA**	**(S)RA**	**(PC)MA**	**KS**	**PFS-aS**	**PFS-wS**	**PCS**
LL-CGKA [25]	✘	✘	✔̸	✔̸	✘	✔	✔	✘	✘
mGKD [24]	✔	✔	✔̸	✘	✔̸	✔	✔	✘	✘
This Work	✔	✔	✔	✔	✔	✔	✔	✔	✔

Table 1. Comparison of security guarantees captured by our model with related works. **M-PKI** = Malicious public key infrastructure; **M-ACM** = Malicious access control mechanism; **(S)KA** = Group key authentication and its stronger variant; **(S)RA** = Group roster authentication and its stronger variant; **(PC)MA** = Group member authorization and its post-compromise variant; **PFS-aS** = Perfect forward secrecy across sessions; **KS** = Group Key Secrecy; **PFS-wS** = Perfect forward secrecy within sessions; **PCS** = Post-compromise security. ✔ denotes captured; ✘ denotes uncaptured; ✔̸ ddenotes that only the weaker variant, which does not consider session-state compromise, is captured.

Contributions. This work aims to solve the above questions. Our main contributions are as follows:

1. We formally define a novel continuous group key distribution (CGKD) protocol for real-time group communication that allows authorized members to freely join and leave, and to refresh their states during the group communication, thereby enabling resilience against state compromise.
2. We propose a set of core security guarantees for real-time group communication against fully malicious servers and state compromise, including perfect forward secrecy (within sessions), post-compromise security, and post-compromise authorization. We present an associated security model for our CGKD protocols that, for the first time, captures all of these core security guarantees for real-time group communication. We compare the security guarantees captured by our model with those in [25,24] in Table 1.
3. We propose our generic CGKD design and prove it secure in our security model. Our proof is provided in the standard model, and we suggest both classical and post-quantum secure instantiations, making our design the first provably post-quantum secure real-time group communication protocol.
4. As a side contribution, we introduce a novel cryptographic primitive called multi-recipient authenticated key encapsulation mechanism (maKEM), which serves as a building block for our generic CGKD design and may be of independent research interest. We define two core security notions for maKEM: IND-CCA security for confidentiality and CTI-CCA security for authenticity. Furthermore, we propose a concrete construction that provably satisfies these notions in the random oracle model and another construction that provably satisfies them in the standard model.

Outline. We recall the background on real-world group communication protocols and multi-recipient cryptography, and introduce our notations in Section 2. We

present the maKEM syntax, its security models, and constructions in Section 3. We define our protocol syntax in Section 4, propose its core security guarantees and associated security model in Section 5, and present our generic construction in Section 6. We provide a summary of this paper in Section 7. We provide all preliminaries, detailed correctness model, detailed protocol description, and all proofs in the supplementary material in the appendix.

2 Background

2.1 Real-Time Group Communication

As introduced in Section 1, although real-time group communication protocols and its applications hold considerable weight in our modern life, their formal security analyses are however limited and, to the best of our knowledge, all focuses exclusively on Zoom.

Isobe and Ito [29] provide possibly the first study that examines the security of real-time group communication protocols. They identify several feasible active and passive impersonation, tampering, and denial of service attacks against Zoom white-paper version 2.3.1, and propose corresponding countermeasures.

Dodis, Jost, Kesavan, and Marcedone [25] formalize Zoom client version 5.12 as a leader-based continuous group key agreement with liveness (LL-CGKA) protocol and propose several core security guarantees, such as authenticity (aka. group consistency), confidentiality, and strong liveness, and prove that Zoom is secure in this model. Briefly speaking, for a given honest leader and all its honest participants, the authenticity ensures that the group members must share the same group key and roster, the confidentiality ensures that their group keys are random from any attacker's view; and the strong liveness ensures that (1) an attacker should not be able to arbitrarily delay communication, and (2) an attacker must not be able to delay or prevent any roster modification made by the leaders from taking effect. Dodis, Jost, Kesavan, and Marcedone prove that Zoom satisfies all of these core security guarantees under a partially malicious server model, e.g., assuming the existence of a trusted PKI and a trusted ACM. In addition, they also disclose Zoom's limitation on liveness, and provably improve it by leveraging additional interaction among participants or clock synchronicity.

Cremers, Ronen, and Zhao [24] formalize Zoom white-paper version 4.0 as a multi-stage group key distribution (mGKD) protocol and define two main associated models: a weaker Sec-mGKD-pki model captures basic group key authentication, secrecy, and perfect forward secrecy (across sessions), assuming a trusted PKI; and a stronger Sec-mGKD-gs model that additionally captures group member authorization[1] without any assumptions about the server. Notably, the Sec-mGKD-gs model captures a strong group member authorization property, which ensures that only authorized parties that can join their intended groups, without relying on any trusted ACM from the server. They first prove the security

[1] This property was originally called "group member authentication" in [24]. We rename this security guarantee to authorization.

of Zoom in the weaker model, and then propose a generic transformation for mGKD, and applies this transformation to Zoom to obtain a new ZoomPAKE that is provably secure in the stronger model. However, the stronger security model in [24] heavily relies on an assumption that every group has a long-live leader that never leaves. This assumption enforces group roster authentication as a natural property in an impractical way and weakens the malicious servers' capabilities from two aspects: On one hand, it prevents a leader from unexpectedly leaving the group, e.g., due to network loss. On the other hand, it prevents attackers from freely injecting malicious leave notification on behalf the leader and performing group-split attacks, thereby restricting the resulting security guarantees to a partially malicious server setting. Moreover, the security model in [24] focuses only on group key authenticity, leaving group roster authenticity unexplored.

In particular, both works [25,24] consider compromise of the long-term states of honest parties, while assuming their group-specific session states incorruptible.

2.2 Asynchronous Group Communication

Asynchronous group communication (aka. secure group messaging, SGM) allows a set of users to communicate securely over an untrusted network in an asynchronous setting. For example, Alice can initialize a group and continuously send messages within the group even when some members are offline.

Intuitively, SGM can be trivially constructed using pairwise channels among group members; however, this approach incurs a total computational cost of $O(n^2)$, where n denotes the number of group members. To improve efficiency, Cohn-Gordon, Cremers, Garratt, Millican, and Milner propose the *Asynchronous Ratchet Tree* (ART) protocol [21], which employs recursive Diffie-Hellman computations to derive a group key shared by all members for encryption and decryption, reducing the total computational cost to $O(n \log n)$. ART was adopted as the initial draft of the IETF *Message Layer Security* (MLS) project [12]. Since Draft 2, MLS has replaced its core component ART with TreeKEM, a protocol that retains a similar tree structure but relies on a generic Key Encapsulation Mechanism (KEM) and therefore enables post-quantum security. Alwen, Coretti, Dodis, and Tselekounis [7] formalize TreeKEM as a *Continuous Group Key Agreement* (CGKA) protocol. Subsequently, numerous analyses of MLS and CGKA, as well as improvements targeting stronger security and higher efficiency, have been proposed [8,10,20,23,9,34,35]. After a total of twenty drafts, MLS was standardized as RFC 9420 in July 2023, and its security was formally analyzed by Cremers, Günsay, Wesselkamp, and Zhao in [22].

In comparison between SGM and real-time group communication, there are three core differences: (1) While SGM focuses on asynchronous communication, i.e., messages can be encrypted for offline recipients, real-time group communication is synchronous. This generally leads to lower protocol complexity in real-time group communication compared to SGM; (2) While SGM typically assigns all group members equal roles and administrative power, real-time group communication often includes a centralized "leader" who schedules the group,

assigns permissions to authorized members, and therefore holds stronger administrative authority than other participants; and (3) While SGM often serves long-lived groups and assumes users with long-term identities, whose authentication is implicitly guaranteed by the trustworthiness of the underlying PKI or service providers, real-time group communication emphasizes rapid joining and often allows unauthenticated participation (e.g., guest log-in), making admission control and authorization more critical.

2.3 Multi-Recipient Cryptography

Multi-recipient cryptography, which enables one party to simultaneously send messages to a set of recipients, can be viewed as a natural extension of conventional single-recipient cryptography. Intuitively, most classical cryptographic primitives can be transformed into the multi-recipient setting by having the sender repeatedly execute the single-recipient primitive for each recipient. However, this straightforward approach incurs $O(n)$ bandwidth and computational costs, where n denotes the number of recipients.

From a methodological perspective, formalizing cryptographic primitives directly in the multi-recipient setting as distinct schemes enables a precise characterization of their security notions and facilitates the design of more generic and efficient constructions. For example, Stéphanie Alt [5] extends the conventional authenticated encryption (AE) to a so-called "1→n" AE, defines its confidentiality and authenticity notations, and proposes generic constructions by using concealment schemes. Kaoru Kurosawa [31] extends the conventional public key encryption (PKE) to multi-recipient PKE, formalizes its formal security, and propose efficient constructions. This line of work further motivates Bellare, Boldyreva, and Staddon [14] to analyze the reproducibility properties of multi-recipient PKE. Nigel Paul Smart [33] extends the conventional key encapsulation mechanism (KEM) to multi-recipient KEM, formalizes its formal security, and propose both a generic construction and an efficient ElGamal-based construction.

Building on the authenticated KEM (AKEM) formalized by Alwen, Blanchet, Hauck, Kiltz, Lipp, and Riepel in [6], we extend AKEM to the multi-recipient setting and introduce a new primitive, which we call multi-recipient authenticated key encapsulation mechanism (maKEM).

Notation We assume that all algorithms defined in this paper are parameterized implicitly by the security parameter λ. We write $[n]$ to denote the set of integers $\{1, \ldots, n\}$. We write $\perp$ to denote a special error symbol that is not included in any set or list (unless specified), $*$ to denote a variable that is irrelevant, $\mathbf{0}$ to denote an empty string, $\emptyset$ to denote an empty set, and $()$ to denote an empty list. The error symbol $\perp$ must *not* be included in the input of any algorithm or oracle. Let $x \xleftarrow{\$} \mathtt{Z}$ and $x \xleftarrow{\$} \mathcal{D}$ respectively denote sampling a variable x uniformly at random from a set $\mathtt{Z}$ and following a distribution $\mathcal{D}$. Let $x \xleftarrow{\$} X(y)$ denote the execution of a probabilistic algorithm X with an input y followed by assigning the output to a variable x. We write $x \leftarrow X(y)$ if the algorithm X is deterministic. Let $\mathtt{L} \xleftarrow{+} x$ denote adding x into an ordered list or an unordered

set $\mathtt{L}$ ($\mathtt{L} \leftarrow \mathtt{L} \cup \{x\}$) and let $\mathtt{L} \xleftarrow{-} x$ denote removing x from the list or set $\mathtt{L}$ ($\mathtt{L} \leftarrow \mathtt{L} \setminus \{x\}$). Let $\mathtt{L} \xleftarrow{-/+} (x, y)$ denote replacing x in the list or set $\mathtt{L}$ with y at the same location if such x exists. We use $\mathrm{Dic}[*] \leftarrow x$ denote initializing all elements of a dictionary Dic with x. We use the keyword **req** to indicate that the subsequent condition must be satisfied and the keyword **parse** to indicate that the subsequent expression must hold. If any such requirement fails, the current algorithm, as well as all recursively invoked algorithms, abort and roll back all execution effects, returning an error. For simplicity, we assume all cryptographic primitives in this work share the same symmetric key space $\mathcal{K}$.

3 Multi-Recipient Authenticated Key Encapsulation Mechanism

In Section 3.1, we propose a formal definition for the *multi-recipient authenticated key encapsulation mechanism* (maKEM) scheme, extending AKEM [6] from two-party setting to multi-party setting. In Section 3.2, we define two core security properties: *indistinguishability under chosen-ciphertext attacks* (IND-CCA) and *ciphertext integrity under chosen-ciphertext attacks* (CTI-CCA), which naturally extend the Outsider-CCA and Outsider-Auth notions for AKEM in [6], respectively, with the only addition of allowing secret-key corruption. In Section 3.3, we propose two instantiations that can provably satisfy IND-CCA and CTI-CCA security. While the first instantiation is a concrete design based on elliptic curve Diffie-Hellman (ECDH) key exchange and authenticated encryption (AE), following the intuition of the Box algorithm in Zoom [17], the second instantiation is a generic design based on digital signature (DS) and public key encryption (PKE), following the intuition of authenticated PKE in [6].

3.1 Syntax

Definition 1 (maKEM). *A* multi-recipient authenticated key encapsulation mechanism *(maKEM) scheme consists of the following algorithms:*

Key Generation $(\mathsf{aek}, \mathsf{adk}) \xleftarrow{\$} \mathsf{maKEM.KGen}$ *outputs a pair of public authenticated encapsulation key* aek *and secret authenticated decapsulation key* adk.

Encapsulation $(k, \{\mathsf{ct}_i\}_{i\in[n]}) \xleftarrow{\$} \mathsf{maKEM.Encaps}(\mathsf{adk}, \{\mathsf{aek}_i\}_{i\in[n]})$ *inputs a secret authenticated decapsulation key and a sequence of* n *public authenticated encapsulation keys* $\{\mathsf{aek}_i\}_{i\in[n]}$ *for some* n*, and outputs a key* k *and a sequence of* n *ciphertexts* $\{\mathsf{ct}_i\}_{i\in[n]}$.

Decapsulation $k/\perp \leftarrow \mathsf{maKEM.Decaps}(\mathsf{adk}, \mathsf{aek}, \mathsf{ct})$ *inputs a secret authenticated decapsulation key* adk*, a public authenticated encapsulation key* aek*, and a ciphertext* ct*, and outputs a key* $k \in \mathcal{K}$ *or an error symbol* $\perp$.

Definition 2 (Correctness of maKEM). *Let* maKEM *denote a maKEM scheme. We say* maKEM *is* δ_{maKEM}*-correct, if it holds for any* $n+1$ *key pairs* $(\mathsf{aek}_i, \mathsf{adk}_i) \xleftarrow{\$} \mathsf{maKEM.KGen}$ *for* $i \in [n]$ *and* $(\mathsf{aek}^\star, \mathsf{adk}^\star) \xleftarrow{\$} \mathsf{maKEM.KGen}$ *that*

$$\Pr\left[\exists i \in [n], k_i \neq k \,\middle|\, \begin{array}{l}(k, \{\mathsf{ct}_i\}_{i\in[n]}) \xleftarrow{\$} \mathsf{maKEM.Encaps}(\mathsf{adk}^\star, \{\mathsf{aek}_i\}_{i\in[n]}) \\ k_i \leftarrow \mathsf{maKEM.Decaps}(\mathsf{adk}_i, \mathsf{aek}^\star, \mathsf{ct}_i), \forall i \in [n]\end{array}\right] \leq \delta_{\mathsf{maKEM}}$$

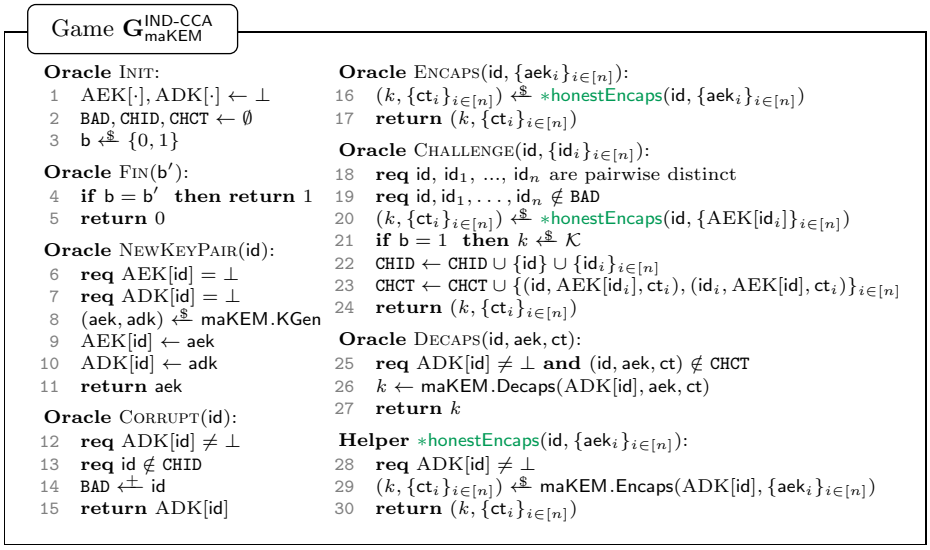


Fig. 1. The IND-CCA game for a maKEM scheme.

3.2 Security Notions

The IND-CCA security provides the confidentiality for maKEM. More concretely, it ensures that no attackers can distinguish the real encapsulated keys from random keys, even if they have the capabilities of eavesdropping honest encapsulations via the Encaps oracle, injecting malicious ciphertexts via the Decaps oracle, and corrupting the secret keys of some irrelevant parties via the Corrupt oracle.

Definition 3 (IND-CCA Security of maKEM). *Let* maKEM *denote a maKEM scheme. We say* maKEM *is* $\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{maKEM}}$*-IND-CCA secure if the advatange of any attacker* $\mathcal{A}$ *in the game* $\mathbf{G}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}$ *in Figure 1 is bounded by*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) := \left| \Pr[\mathbf{G}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) = \mathsf{true}] - \frac{1}{2} \right| \leq \epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{maKEM}}$$

The CTI-CCA security provides the integrity for maKEM. More concretely, it ensures that no attackers can forge any ciphertext between two honest parties, even if they have the capabilities of eavesdropping honest encapsulations via the Encaps oracle, injecting malicious ciphertexts via the Decaps oracle, and corrupting the secret keys of some irrelevant parties via the Corrupt oracle.

Definition 4 (CTI-CCA Security of maKEM). *Let* maKEM *denote a maKEM scheme. We say* maKEM *is* $\epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{maKEM}}$*-CTI-CCA secure if the advatange of any attacker* $\mathcal{A}$ *in the game* $\mathbf{G}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{maKEM}}$ *in Figure 2 is bounded by*

$$\mathsf{Adv}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) := \Pr[\mathbf{G}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) = \mathsf{true}] \leq \epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{maKEM}}$$

Game $\mathbf{G}^{\text{CTI-CCA}}_{\mathsf{maKEM}}$

Oracle INIT:
31 $\text{AEK}[\cdot], \text{ADK}[\cdot] \leftarrow \bot$
32 $\texttt{BAD}, \texttt{CT} \leftarrow \emptyset$

Oracle NEWKEYPAIR(id):
33 **req** $\text{AEK}[\mathsf{id}] = \bot$
34 **req** $\text{ADK}[\mathsf{id}] = \bot$
35 $(\mathsf{aek}, \mathsf{adk}) \xleftarrow{\$} \mathsf{maKEM.KGen}$
36 $\text{AEK}[\mathsf{id}] \leftarrow \mathsf{aek}$
37 $\text{ADK}[\mathsf{id}] \leftarrow \mathsf{adk}$
38 **return** aek

Oracle CORRUPT(id):
39 **req** $\text{ADK}[\mathsf{id}] \neq \bot$
40 $\texttt{BAD} \xleftarrow{\cup} \mathsf{id}$
41 **return** $\text{ADK}[\mathsf{id}]$

Oracle FIN(id, id′, ct):
42 **req** $\mathsf{id} \neq \mathsf{id}'$ **and** $\{\mathsf{id}, \mathsf{id}'\} \cap \texttt{BAD} = \emptyset$
43 **req** $\{(\mathsf{id}, \text{AEK}[\mathsf{id}'], \mathsf{ct}), (\mathsf{id}', \text{AEK}[\mathsf{id}], \mathsf{ct})\} \cap \texttt{CT} = \emptyset$
44 $\mathsf{adk} \leftarrow \text{ADK}[\mathsf{id}]$; $\mathsf{aek} \leftarrow \text{AEK}[\mathsf{id}']$
45 $k \leftarrow \mathsf{maKEM.Decaps}(\mathsf{adk}, \mathsf{aek}, \mathsf{ct})$
46 **if** $k \neq \bot$ **then return** true
47 **return** false

Oracle ENCAPS(id, $\{\mathsf{aek}_i\}_{i\in[n]}$):
48 $(k, \{\mathsf{ct}_i\}_{i\in[n]}) \xleftarrow{\$} *\mathsf{honestEncaps}(\mathsf{id}, \{\mathsf{aek}_i\}_{i\in[n]})$
49 $\texttt{CT} \leftarrow \texttt{CT} \cup \{(\mathsf{id}, \mathsf{aek}_i, \mathsf{ct}_i)\}_{i\in[n]}$
50 **return** $(k, \{\mathsf{ct}_i\}_{i\in[n]})$

Oracle DECAPS(id, aek, ct):
51 **req** $\text{ADK}[\mathsf{id}] \neq \bot$
52 $k \leftarrow \mathsf{maKEM.Decaps}(\text{ADK}[\mathsf{id}], \mathsf{aek}, \mathsf{ct})$
53 **return** k

Fig. 2. The CTI-CCA game for a maKEM scheme. The $*\mathsf{honestEncaps}$ helper function is defined in Figure 1.

Our maKEM Construction based on ECDH and AE

Algorithm KGen:
54 $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{ECDH.KGen}$; $\mathsf{aek} \leftarrow \mathsf{pk}$; $\mathsf{adk} \leftarrow \mathsf{sk}$
55 **return** (aek, adk)

Algorithm Decaps(adk, aek, ct):
56 $\mathsf{sdk} \leftarrow \mathsf{ECDH.Derive}(\mathsf{adk}, \mathsf{aek})$; $k^{\mathsf{AE}} \leftarrow \mathsf{KDF}(\mathsf{sdk})$
57 $k \leftarrow \mathsf{AE.Dec}(k^{\mathsf{AE}}, c)$
58 **return** k

Algorithm Encaps(adk, $\{\mathsf{aek}_i\}_{i\in[n]}$):
59 $k \xleftarrow{\$} \mathcal{K}$
60 **foreach** $i \in [n]$ **do**
61 $\quad \mathsf{sdk}_i \leftarrow \mathsf{ECDH.Derive}(\mathsf{adk}, \mathsf{aek}_i)$
62 $\quad k_i^{\mathsf{AE}} \leftarrow \mathsf{KDF}(\mathsf{sdk}_i)$
63 $\quad c_i \xleftarrow{\$} \mathsf{AE.Enc}(k_i^{\mathsf{AE}}, k)$
64 **return** $(k, \{c_i\}_{i\in[n]})$

Fig. 3. Our maKEM construction based on elleptic curve Diffie-Hellman key exchange and authenticated encryption.

3.3 Instantiations

We depict our first concrete construction based on ECDH and AE in Figure 3. The KGen algorithm simply samples and outputs a new ECDH key pair. The Encaps algorithm inputs a secret key adk and n public keys aek_i for $i \in [n]$ and samples a key k uniformly at random. By applying a KDF to the ECDH key exchange between every pair $(\mathsf{adk}, \mathsf{aek}_i)$ for $i \in [n]$, the Encaps algorithm obtains n symmetric keys k^{AE} and uses them to encrypt the sampled key k. The Decaps algorithm applies a KDF to the ECDH key exchange between the input key pair $(\mathsf{adk}, \mathsf{aek})$ for a symmetric key k^{AE}, which is further used to recover the encrypted key k from the input ciphertext.

Theorem 1. *Let* maKEM *denote the maKEM construction in Figure 3. If the underlying* AE *is* δ_{AE}*-correct, then* maKEM *is* δ_{maKEM}*-correct, where*

$$\delta_{\mathsf{maKEM}} \leq n\delta_{\mathsf{AE}}$$

Here, n *denotes the number of* aeks *included in the inputs of* maKEM.Encaps.

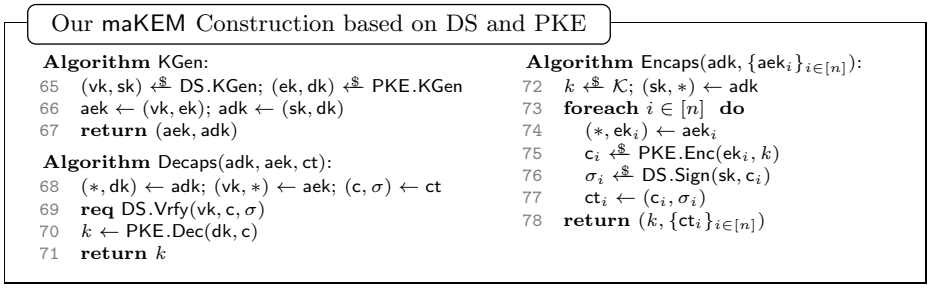


Fig. 4. Our maKEM construction based on digital signature and public key encryption.

Theorem 2. *Let* maKEM *denote the maKEM construction in Figure 3. If the* mm-prf-ODH *problem over* ECDH *and* KDF *has* $\epsilon^{\text{mm-prf-ODH}}_{\text{ECDH,KDF}}$ *hardness, and* AE *is* $\epsilon^{\text{ind-cca}}_{\text{AE}}$-IND-CCA *secure, then* maKEM *is* $\epsilon^{\text{ind-cca}}_{\text{maKEM}}$-IND-CCA *secure, where*

$$\epsilon^{\text{ind-cca}}_{\text{maKEM}} \leq q^{5}_{\text{NKP}} \epsilon^{\text{mm-prf-ODH}}_{\text{ECDH,KDF}} + q^{2}_{\text{NKP}} \epsilon^{\text{ind-cca}}_{\text{AE}}$$

Here, q_{NKP} *denotes the number of queries to the* NewKeyPair *oracle.*

Theorem 3. *Let* maKEM *denote the maKEM construction in Figure 3. If the* mm-prf-ODH *problem over* ECDH *and* KDF *has* $\epsilon^{\text{mm-prf-ODH}}_{\text{ECDH,KDF}}$ *hardness, and* AE *is* $\epsilon^{\text{cti-cca}}_{\text{AE}}$-CTI-CCA *secure, then* maKEM *is* $\epsilon^{\text{cti-cca}}_{\text{maKEM}}$-CTI-CCA *secure, where*

$$\epsilon^{\text{cti-cca}}_{\text{maKEM}} \leq q^{2}_{\text{NKP}} \epsilon^{\text{mm-prf-ODH}}_{\text{ECDH,KDF}} + q^{2}_{\text{NKP}} \epsilon^{\text{cti-cca}}_{\text{AE}}$$

Here, q_{NKP} *denotes the number of queries to the* NewKeyPair *oracle.*

We depict our second generic construction based on DS and PKE in Figure 4. The KGen algorithm samples a DS public-secret key pair and a PKE public-secret key pair, followed by outputting the tuple of both public keys as aek and the tuple of both secret keys as adk. The Encaps algorithm samples a key randomly, extracts a DS secret key from the input adk, and extracts PKE public keys from every input aek_i for $i \in [n]$. Then, the Encaps algorithm encrypts this random key n times by using each PKE public keys, signs every encryption by using DS, and outputs the random key together with all PKE ciphertexts and DS signatures. The Decaps algorithm verifies and decrypts the input ciphertext to recover the encapsulated keys in a similar reverse way.

Theorem 4. *Let* maKEM *denote the maKEM construction in Figure 4. If the underlying* DS *is* δ_{DS}*-correct and* PKE *is* δ_{PKE}*-correct, then* maKEM *is* δ_{maKEM}*-correct, where*

$$\delta_{\text{maKEM}} \leq n(\delta_{\text{DS}} + \delta_{\text{PKE}})$$

Here, n *denotes the number of* aeks *included in the inputs of* maKEM.Encaps.

Theorem 5. *Let* maKEM *denote the maKEM construction in Figure 4. If the underlying* PKE *is* $\epsilon^{\text{ind-cca}}_{\mathsf{PKE}}$-IND-CCA *secure, then* maKEM *is* $\epsilon^{\text{ind-cca}}_{\mathsf{maKEM}}$-IND-CCA *secure, where*

$$\epsilon^{\text{ind-cca}}_{\mathsf{maKEM}} \leq q^2_{\text{NKP}} \epsilon^{\text{ind-cca}}_{\mathsf{PKE}}$$

Here, q_{NKP} *denotes the number of queries to the* NewKeyPair *oracle.*

Theorem 6. *Let* maKEM *denote the maKEM construction in Figure 4. If the underlying* DS *is* $\epsilon^{\text{suf-cma}}_{\mathsf{DS}}$-SUF-CMA *secure, then* maKEM *is* $\epsilon^{\text{cti-cca}}_{\mathsf{maKEM}}$-CTI-CCA *secure, where*

$$\epsilon^{\text{cti-cca}}_{\mathsf{maKEM}} \leq q_{\text{NKP}} \epsilon^{\text{suf-cma}}_{\mathsf{DS}}$$

Here, q_{NKP} *denotes the number of queries to the* NewKeyPair *oracle.*

Instantiations. We provide concrete instantiations for our maKEM construction in Figure 4 in both classical and post-quantum settings. The underlying DS can be instantiated with Ed25519 IETF version [30] for classical security, whose SUF-CMA security is proven in [18], and the NIST suggested CRYSTALS-Dilithium for post-quantum security, whose SUF-CMA security is proven in [11]. The underlying PKE can be instantiated by applying Fujisaki-Okamoto transformation [26,28] to obtain IND-CCA security in both classical and post-quantum settings.

4 Continuous Group Key Distribution

In this section, we propose our novel *continuous group key distribution* (CGKD) protocol for real-time group communication and provide its correctness definition. A CGKD protocol allows a party, in the role of leader, to distribute keys to other parties, in the role of participants, within a group. Every party must be uniquely identified by an identifier id. Similarly, every group must be uniquely identified by an identifier gid. Every group is associated with a group secret, which is securely sent to selected parties over some out-of-band channels, e.g., email or SMS. The group secret is expected to be sampled from a distribution with sufficiently high entropy, though in practice it might also be picked arbitrarily, e.g., "123456". We call parties holding the correct secret of a group *authorized*.

Within every group, only one member can take on the role of leader at any given time, while all other members act as participants. Every group member maintains a *roster*, which includes the *record* of all group members. Briefly speaking, records reflect the public views of group members' states and can be periodically *refreshed*. The leader has stronger capabilities of access control and group management than participants, such as accepting joining requests, updating group rosters, and rotating group keys. The leader can continuously announce these modifications on groups by an "*evolve*" operation. Every group member may leave the group at any time (e.g., due to a loss of network connection). In particular, when the current leader leaves the group, the remaining group members selects the next group leader according to their locally stored roster. The role of each party in different groups might be distinct: a party can be leader in one group and participant in others.

Epoch and Stage. We use two counters to identify every group evolution: *epoch* and *stage*. The epoch counter tracks the number of state "healing" events within every group. It is initialized with 1 when the group is created, and is incremented by 1 after each designated time window during which group members are expected to refresh their states (e.g., every 15 minutes). In practice, the exact duration of this time window may be determined by the implementation, chosen by the leader, or adjusted dynamically during group communication. For generality, we do not model that exact time window explicitly; instead, we use a flag $\mathsf{nxtEp} = \mathsf{true}$ to capture the event of an epoch increment, upon which every group member should refresh their local states and public records. If the group members are never required to refresh their states, then no epoch increment occurs.

The stage counter tracks the number of times the "evolve" operation is invoked within an epoch. Whenever a group enters a new epoch, the stage counter is reset to 0. It is incremented by 1 each time the leader evolves the group, e.g., by updating the group roster and/or rotating group keys.

Record and Roster. A record rec is a data structure that reflects the public view of a group member's state within an epoch. Each record rec contains its identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep})$, identifying its epoch ep in the session sid of a party id, and associated cryptographic metadata, including identity public key ipk and session public key spk. We write $\mathsf{rec.var}$ to refer to any variable var contained in a record rec, and use "record identifier" $\mathsf{rec.getIds}$ to refer to the tuple $(\mathsf{rec.id}, \mathsf{rec.sid}, \mathsf{rec.ep})$.

A roster is an ordered list that stores the records of group members. We require that the ordering of records within a roster must indicate the current leader and the succession priority among members. This allows all members sharing the same roster to immediately agree on the next leader when the current leader leaves the group, without requiring explicit out-of-band leader selection. The ordering of records within a roster may vary across implementations, for example, according to implementation-specific roles (e.g., leader, creator, managers, assistants, and members), join time, alphabetical order, or a leader-defined sequence.

Moreover, we use a helping function $*\mathsf{leader}(\mathsf{ros})$ to denote identifying the record of the (unique) group leader from any roster ros.

Remark 1. Compared with existing works [25] and [24], where the underlying rosters (i.e., the leader-participant list LPL in [25] and the set of group parties GP in [24]) contain only group members' identifiers, the rosters in our work include both identifiers and cryptographic metadata. Moreover, our roster is ordered according to the current leader and the succession priority among members. This allows group members to agree on (i) who they are communicating with, (ii) which cryptographic keys the communication partners are using, and (iii) who the next leader will be.

We stress that this design choice is particularly important when accounting for the unexpected departure of group leaders, especially in the presence of fully malicious servers. As we show in Section 6, our protocol handles leader transitions smoothly in the end-to-end setting, without assuming the existence of a trusted server as in [25] or preventing leaders from leaving as in [24].

State. Every party id is associated with a long-term state $\mathsf{lst}_{\mathsf{id}}$. To model concurrent or sequential real-time group communication, we use $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}$ to denote id's session state indexed by sid. We assume that the long-term state $\mathsf{lst}_{\mathsf{id}}$ is shared by all session states $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}$ of the same party.

Definition 5 (State). *In a CGKD protocol, each party* id *maintains the following state variables:*

- $\mathsf{lst.id}$*: the associated party identifier* id.
- $\mathsf{lst.ipk}$*: the identity public key.*
- $\mathsf{lst.isk}$*: the identity secret key.*
- $\mathsf{sst.id}$*: the associated party identifier* id*; identical to* $\mathsf{lst.id}$.
- $\mathsf{sst.sid}$*: the associated session identifier* sid.
- $\mathsf{sst.spk}$*: the session public key.*
- $\mathsf{sst.ssk}$*: the session secret key.*
- $\mathsf{sst.gid}$*: the associated group identifier* gid.
- $\mathsf{sst.role} \in \{\mathsf{leader}, \mathsf{parti}, \mathsf{joiner}\}$*: the role of the party* id*, either the group leader, or a group participant that has already been announced as a group participant, or a party that is joining a group.*
- $\mathsf{sst.ep}$*: the current epoch; initialized with* 0.
- $\mathsf{sst.stg}$*: the current stage; initialized with* 0.
- $\mathsf{sst.gk}$*: the current group key; initialized with* $\perp$.
- $\mathsf{sst.ros}$*: the current group roster (see the explanation below); initialized with* $()$.
- $\mathsf{sst.rec}$*: the record of the session state for the current epoch; initialized with* $\perp$.

Definition 6. *A CGKD protocol consists of the following algorithms.*

Sign Up: $(\mathsf{lst}, \mathsf{m}^{\mathsf{siup}}) \xleftarrow{\$} \mathsf{SignUp}(\mathsf{id})$ *allows a party* id *to initialize a long-term state* lst *and to output a message* $\mathsf{m}^{\mathsf{siup}}$.

Group Schedule: $\mathsf{gs} \xleftarrow{\$} \mathsf{Schedule}(\mathsf{gid}, \mathcal{D})$ *schedules a group secret* gs *for a group* gid *using a distribution* $\mathcal{D}$*. The group secret is securely sent to authorized parties, e.g., over some out-of-band channels.*

Register: $\mathsf{Register} = (\mathsf{Register\text{-}L}, \mathsf{Register\text{-}P})$ *allows a party to register for joining a group. It consists of two sub-algorithms depending on the role of the caller:*

- $(\mathsf{sst}, \mathsf{m}^{\mathsf{reg}}) \xleftarrow{\$} \mathsf{Register\text{-}L}(\mathsf{lst}, \mathsf{gid})$ *(resp.* $(\mathsf{sst}, \mathsf{m}^{\mathsf{reg}}) \xleftarrow{\$} \mathsf{Register\text{-}P}(\mathsf{lst}, \mathsf{gid})$*) allows a party with long-term state* lst *to register for a group* gid *as leader (resp. joiner). The output is a session state* sst *and a message* $\mathsf{m}^{\mathsf{reg}}$.

Participant Join: $\mathsf{Join} = (\mathsf{Join\text{-}L}, \mathsf{Join\text{-}P})$ *allows a joiner to interact with a leader for joining a group:*

- $(\mathsf{sst}', \mathsf{m}^{\mathsf{join}}, \mathsf{rec}) \xleftarrow{\$} \mathsf{Join\text{-}L}(\mathsf{sst}, \mathsf{gs}, \mathsf{m})$ *(resp.* $(\mathsf{sst}', \mathsf{m}^{\mathsf{join}}, \mathsf{rec}) \xleftarrow{\$} \mathsf{Join\text{-}P}(\mathsf{sst}, \mathsf{gs}, \mathsf{m})$*) allows a leader (resp. joiner) with a session state* sst *to input a group secret* gs *and an incoming message* m*, and to output a new session state* sst'*, a message* $\mathsf{m}^{\mathsf{join}}$*, and the partner's record* rec*. The record* $\mathsf{rec} = \mathbf{0}$ *if the join phase has not completed, and* $\mathsf{rec} \neq \mathbf{0}$ *otherwise.*

Refresh: $\mathsf{Refresh} = (\mathsf{Refresh\text{-}L}, \mathsf{Refresh\text{-}P})$ *allows group participants to refresh their states and to synchronize updated records with the leader.*

- $(\mathsf{sst}', \mathsf{rec}) \xleftarrow{\$} \mathsf{Refresh\text{-}L}(\mathsf{sst}, \mathsf{m})$ *allows the leader to recover a record* rec *from an incoming message* m *and to update its state.*
- $(\mathsf{sst}', \mathsf{m}^{\mathsf{refr}}, \mathsf{rec}) \xleftarrow{\$} \mathsf{Refresh\text{-}P}(\mathsf{sst})$ *allows a participant to refresh its own session state* sst*, and to output a message* $\mathsf{m}^{\mathsf{refr}}$ *and a record* rec*.*

Stage Evolution: $\mathsf{Evolve} = (\mathsf{Evolve\text{-}L}, \mathsf{Evolve\text{-}P})$ *allows a group leader to advance the group to the next stage (and optionally to the next epoch), and to synchronize resulting updates with the participants.*

- $(\mathsf{sst}', \mathsf{m}^{\mathsf{evol}}) \xleftarrow{\$} \mathsf{Evolve\text{-}L}(\mathsf{sst}, \mathsf{nxtEp})$ *allows a leader to evolve its session state* sst *and to output a message* $\mathsf{m}^{\mathsf{evol}}$*. If the flag* $\mathsf{nxtEp} = \mathsf{true}$*, then the leader additionally advances the epoch and refreshes its own state.*
- $(\mathsf{sst}', \mathsf{nxtEp}) \xleftarrow{\$} \mathsf{Evolve\text{-}P}(\mathsf{sst}, \mathsf{m})$ *allows a party to apply an evolve message* m *to its session state* sst*. The output flag* $\mathsf{nxtEp} = \mathsf{true}$ *if the epoch is advanced, and* $\mathsf{nxtEp} = \mathsf{false}$ *otherwise.*

Member Leave: $\mathsf{Leave} = (\mathsf{Leave\text{-}L}, \mathsf{Leave\text{-}P})$ *allows a member to process the notification that a party has left the group:*

- $\mathsf{sst}' \xleftarrow{\$} \mathsf{Leave\text{-}L}(\mathsf{sst}, \mathsf{id}, \mathsf{sid})$ *(resp.* $\mathsf{sst}' \xleftarrow{\$} \mathsf{Leave\text{-}P}(\mathsf{sst}, \mathsf{id}, \mathsf{sid})$*) allows a leader (resp. joiner or participant) with session state* sst *to react to the leaving of a party* id*'s session* sid*. The output is a new session state* sst*. If* $(\mathsf{id}, \mathsf{sid})$ *indicates the session state* sst *itself, then the session state* sst *is erased.*

We present an example of an honest CGKD protocol execution in Figure 5. All incoming and outgoing messages of a CGKD protocol are publicly accessible, e.g., via a "bulletin board" executed by the untrusted service provider. In contrast, the group secret gs of every group gid is shared only among authorized parties and therefore invisible to the service provider.

The $\mathsf{Register}$ phase allows an authorized party to register for the group. The first party who initializes the group immediately takes the initial leader role of the group. In this paper, we assume that every group is activated by authorized members only once. This captures real-world scenarios that a group is activated by the unique party who schedules this group, and usually, only once. The Join phase may require multiple communication rounds for the group leader to verify joining requests from authorized parties. In contrast, the Leave phase is non-interactive for leaders, joiners, and participants: they simply process the notifications that some members have left the group. This captures real-world scenarios in which parties may leave the group unexpectedly (e.g., due to network disconnection) without producing any message or a malicious server may manipulate network traffic. Every participant can update their session states during the $\mathsf{Refresh}$ phase. All changes to groups resulting from the Join, Leave, and $\mathsf{Refresh}$ phases are buffered at the leader and are synchronized with group participants during the Evolve phase, in which the leader may also refresh its own states, rotate group keys, and remove participants who fail to refresh within the current epoch.

We provide the correctness definition for our CGKD protocol in Section B.1. Informally, our correctness definition captures the following four properties of an honest CGKD protocol execution:

- **Record Agreement During Join Phase:** When a participant completes the join-phase communication with the group leader, they must agree on the group

identifier as well as their respective records and roles. This property ensures that the group leader and the newly joined participant correctly exchange the information necessary for subsequent group key distribution.
- **Correct Stage Evolution:** When a leader advances the group to the next stage (optionally to the next epoch), it must correctly update the roster, and all group participants must agree with the leader on the updated epoch, stage, group key, and roster. This property ensures that every group member agrees on a consistent view of the system state during stage evolution, including any updates resulting from the refresh phase.
- **Correct Leader Leave:** When a leader leaves the group, the leader's session state and the session states of all newly joined participants that have not yet been announced to the group must be erased. In addition, all remaining participants must agree on the updated roster. This property ensures that the leaving leader's session and any unannounced joining sessions cannot participant in future group communication, and that the remaining participants agree on the next leader as determined by the roster order.
- **Correct Participant Leave:** When a participant leaves the group, its session state must be erased and its record must be removed from the leader's roster. This property ensures that the leaving participant's session cannot participant in future group communication.

In this work, we only focus on a class of CGKD protocols that satisfy the following basic consistency properties.

1. A session's role must align with its local roster. That is, $\mathsf{sst.role} = \mathsf{leader}$ if and only if $\mathsf{sst.rec} = *\mathsf{leader}(\mathsf{sst.ros})$; $\mathsf{sst.role} = \mathsf{parti}$ if and only if $\mathsf{sst.ros} \neq ()$; otherwise, $\mathsf{sst.role} = \mathsf{joiner}$.
2. The record of every session must correctly reflect the public view of its state, i.e., for $\mathsf{rec} \leftarrow \mathsf{sst.rec}$, it must hold that $(\mathsf{rec.id}, \mathsf{rec.sid}, \mathsf{rec.ep}, \mathsf{rec.ipk}, \mathsf{rec.spk}) = (\mathsf{sst.id}, \mathsf{sst.sid}, \mathsf{sst.ep}, \mathsf{lst}_{\mathsf{sst.id}}.\mathsf{ipk}, \mathsf{sst.spk})$.
3. A party accepts a record only if the received record rec corresponds to the next epoch of the session state sst, i.e., $\mathsf{rec.ep} = \mathsf{sst.ep} + 1$.
4. In the Evolve phase, a leader (resp. participant) increments the local epoch counter by 1 if and only if the input (resp. output) flag $\mathsf{nxtEp} = \mathsf{true}$. If $\mathsf{nxtEp} = \mathsf{false}$, then they increment the local stage counter by 1.

5 CGKD Security Model

In Section 5.1, we introduce a set of strong security guarantees that a CGKD protocol is expected to achieve. In Section 5.2, we propose a security model for CGKD protocols that captures these guarantees.

5.1 Strong Security Guarantees

A CGKD protocol is expected to achieve the following security guarantees, which are originally introduced in [25,24] without considering session state compromise within a target group.

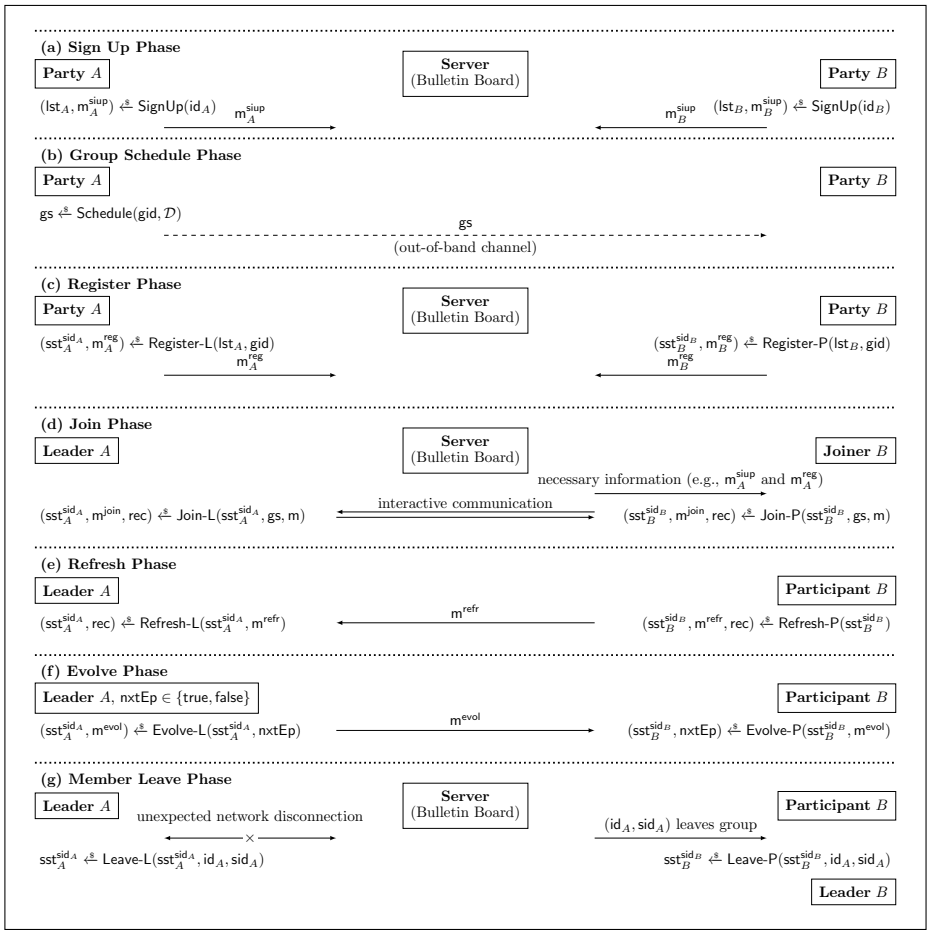


Fig. 5. An example of CGKD protocol execution

- **Group Key Authentication**: If a group member accepts a group key at some stage, then its leader must have produced the same group key at the same stage, if the attacker does not perform any session state compromise.
- **Group Roster Authentication**: If a group member accepts a roster at some stage, then its leader must have produced the same roster at the same stage, if the attacker does not perform any session state compromise.
- **Group Key Secrecy**: If a group member accepts a group key at some stage, then an attacker who does not perform any session state compromise cannot derive this key, even if it learns other group keys.
- **Perfect Forward Secrecy (across sessions)**: An attacker cannot learn any group key of any group that a party was previously in, even if that party's long-term state and session states of other groups are compromised.

– **Group Member Authorization**: When a participant that successfully joins a group, if the leader is authorized for the group, then so is the joined participant, and vice versa, unless the group secret is leaked.

We also consider the following novel security guarantees taking session state corruption in any group into account:

– **Strong Group Key Authentication**: If a group member accepts a group key at some stage, then its leader must have produced the same group key at the same stage, unless the leader's or that group member's session states are compromised.
– **Strong Group Roster Authentication**: If a group member accepts a roster at some stage, then its leader must have produced the same roster at the same stage, unless the leader's or that group member's session states are compromised.
– **Perfect Forward Secrecy (within sessions)**: An attacker cannot learn group keys of any group at any past stage, even if some group member's long-term state and session states of the same group at later stages are compromised.
– **Post-Compromise Security**: If a group member's local state is fully compromised (i.e., both the long-term and the session state), then its session state can be recovered in the presence of a passive attacker.
– **Post-Compromise Group Member Authorization**: Even after fully compromising a party's local state, an attacker still cannot cause the group leader to accept any join request from an unauthorized party.

5.2 Security Model

We depict our security game $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ for a CGKD protocol with a minimum entropy requirement α_{min} on the group secret in Figures 6 and 7.

Trust Model. We assume that randomness sampled by all parties is independent, uniform, and unpredictable. We assume that all authorized parties honestly follow the protocol specification, in particular by using the correct group secret when joining groups. We assume that every group is initialized by authorized parties only once, which captures the real-world scenario where every group is usually initialized by the unique party that schedules this group, and typically, only once. We assume that every group is terminated when all its group members leave the group, which captures the real-world scenario where a group session is usually scheduled with an expected duration and naturally ends after all participants depart. We assume that group secrets are honestly sampled from some distributions chosen by attackers and sent to authorized parties over some out-of-band channels, i.e., unknown to attackers if distribution has sufficiently high entropy and these out-of-band channels are secure.

We emphasize that the trust assumptions in our model are intended to reflect realistic deployment scenarios and do not impose any stronger assumptions that would weaken the capabilities of malicious servers.

Threat Model. We assume that the attacker has full control over the network and can eavesdrop on, drop, and inject messages during all phases. We allow the attacker to choose the distribution of the group secret for each group, enabling us to analyze how the choice of distribution affects the resulting security guarantees. We allow the attacker to corrupt the following secrets: (1) the long-term state of any party, capturing the real-world scenario where hardware devices maybe be stolen, (2) the group secret of any group, capturing the real-world scenario where out-of-band channels may be vulnerable, (3) arbitrarily many group keys, which enables us to analyze its impact on the security of previous and future group keys, and (4) the session state of any party after it has successfully joined a group, capturing the real-world scenario where session states may be leaked due to malware or improper state erasure. However, we exclude attackers from compromising a party's session state during the joining phase, as this phase is typically ephemeral in real-world deployments.

Oracles. Our security game $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ between a challenger and an attacker $\mathcal{A}$ starts with a query from the attacker $\mathcal{A}$ to the INIT oracle, where the challenger initializes seven global dictionaries and five global sets: (1) GS stores the values of each group secret, (2) RG stores the groups that each session registers for, (3) Rec stores the records of each session at each epoch, (4) GK stores the group keys of each session at each epoch and each stage, (5) Ros stores the rosters of each session at each epoch and each stage, (6) AP stores the identifiers of each authorized party of each group, (7) CanInj stores the identifiers of records that the attacker can inject in each group, (8) `actGrp` stores the activated groups, (9) `CPar` stores the identifiers of parties that are corrupted, (10) `CGrp` stores the identifiers of groups whose secrets are leaked, (11) `CEp` stores the identifiers of records, whose corresponding session states are compromised, and (12) `CStg` stores the locations of group keys that are revealed. Recall that group keys are rotated only within each stage and can therefore be identified by the combination of party identifiers, session identifiers, epoch indices, and stage indices. Moreover, the challenger also initializes a global variable tested with $\perp$, samples a challenge bit b, and sets a flag win to false.

Our security game $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ ends with a query from $\mathcal{A}$ to the FIN oracle that takes as input a bit b'. The flag win is set to true if the evaluation of helper function $*\mathsf{winChl}$ on b' is true. Finally, this oracle outputs the flag win.

The attacker $\mathcal{A}$ also has access to the following oracles.

NewParty(id)**:** inputs the identifier id of a party, whose long-term state has not been initialized. The challenger runs the SignUp algorithm on the input id honestly, followed by outputting the generated sign-up message $\mathsf{m}^{\mathsf{siup}}$.

NewGroup($\mathsf{gid}, \mathcal{D}$)**:** inputs the identifier of a group, whose secret has not been initialized, and a distribution $\mathcal{D}$. The challenger runs the $\mathsf{Schedule}$ algorithm on gid and $\mathcal{D}$ honestly, followed by storing the generated group secret gs into the dictionary GS. If the entropy of the distribution $\mathcal{D}$ is smaller than the required minimum entropy α_{min}, the challenger adds the identifier gid into the set `CGrp` and outputs the generated gs. Otherwise, the challenger simply outputs an empty string $\mathbf{0}$.

Game $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$

Oracle INIT:
79 $\mathrm{GS}[\cdot], \mathrm{RG}[\cdot], \mathrm{Rec}[\cdot], \mathrm{GK}[\cdot] \leftarrow \bot$
80 $\mathrm{Ros}[\cdot] \leftarrow (); \mathrm{AP}[\cdot], \mathrm{CanInj}[\cdot] \leftarrow \emptyset$
81 $\mathtt{actGrp}, \mathtt{CPar}, \mathtt{CGrp}, \mathtt{CEp}, \mathtt{CStg} \leftarrow \emptyset$
82 $\mathsf{tested} \leftarrow \bot; \mathsf{b} \xleftarrow{\$} \{0,1\}; \mathsf{win} \leftarrow \mathsf{false}$

Oracle FIN(b'):
83 **if** $*\mathsf{winChl}(\mathsf{b}')$ **then** $\mathsf{win} \leftarrow \mathsf{true}$
84 **return** win

Test Challeng Bit
Oracle TEST($\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}$):
85 **req** $\mathsf{tested} = \bot$ **and** $\mathrm{GK}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}] \neq \bot$
86 $\mathsf{tested} \leftarrow (\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}); k \leftarrow \mathrm{GK}[\mathsf{tested}]$
87 **if** $\mathsf{b} = 1$ **then**
88 $k \xleftarrow{\$} \mathcal{K}$
89 **return** k

Party and Group Creation
Oracle NEWPARTY(id):
90 **req** $\mathsf{lst}_{\mathsf{id}} = \bot$
91 $(\mathsf{lst}_{\mathsf{id}}, \mathsf{m}^{\mathsf{siup}}) \xleftarrow{\$} \mathsf{SignUp}(\mathsf{id})$
92 **return** $\mathsf{m}^{\mathsf{siup}}$

Oracle NEWGROUP($\mathsf{gid}, \mathcal{D}$):
93 **req** $\mathrm{GS}[\mathsf{gid}] = \bot$
94 $\mathsf{gs} \xleftarrow{\$} \mathsf{Schedule}(\mathsf{gid}, \mathcal{D})$
95 $\mathrm{GS}[\mathsf{gid}] \leftarrow \mathsf{gs}$
96 **if** $\alpha_{\mathcal{D}} < \alpha_{\mathsf{min}}$ **then**
97 $\mathtt{CGrp} \xleftarrow{\perp} \mathsf{gid}$; **return** gs
98 **return 0**

Oracle AUTHORIZE($\mathsf{gid}, \mathsf{id}$):
99 **req** $\mathrm{GS}[\mathsf{gid}] \neq \bot$
100 **req** $\mathsf{lst}_{\mathsf{id}} \neq \bot$
101 $\mathrm{AP}[\mathsf{gid}] \xleftarrow{\perp} \mathsf{id}$

Secret Exposure
Oracle CORRUPT(id):
102 $\mathtt{CPar} \xleftarrow{\perp} \mathsf{id}$
103 **return** $\mathsf{lst}_{\mathsf{id}}$

Oracle REVEAL(gid):
104 $\mathtt{CGrp} \xleftarrow{\perp} \mathsf{gid}$
105 **return** $\mathrm{GS}[\mathsf{gid}]$

Oracle COMPROMISE($\mathsf{id}, \mathsf{sid}$):
106 $\mathsf{sst} \leftarrow \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}$; $\mathsf{ep} \leftarrow \mathsf{sst.ep}$
107 **req** $\mathsf{sst.role} = \{\mathsf{leader}, \mathsf{parti}\}$
108 $\mathtt{CEp} \xleftarrow{\perp} (\mathsf{id}, \mathsf{sid}, \mathsf{ep})$
109 **if** $\mathrm{Rec}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}+1] \neq \bot$ **then**
110 $\mathtt{CEp} \xleftarrow{\perp} \mathsf{id}, (\mathsf{sid}, \mathsf{ep}+1)$
111 $\mathtt{CStg} \xleftarrow{\perp} (\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{sst.stg})$
112 **return** sst

Oracle LEAK($\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}$):
113 **req** $\mathrm{GK}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}] \neq \bot$
114 $\mathtt{CStg} \xleftarrow{\perp} (\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg})$
115 **return** $\mathrm{GK}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}]$

Register Phase
Oracle INITGROUP($\mathsf{gid}, \mathsf{id}$):
116 **req** $\mathsf{id} \in \mathrm{AP}[\mathsf{gid}]$ **and** $\mathsf{gid} \notin \mathtt{actGrp}$
117 $(\mathsf{sst}, \mathsf{m}^{\mathsf{reg}}) \xleftarrow{\$} \mathsf{Register\text{-}L}(\mathsf{lst}_{\mathsf{id}}, \mathsf{gid})$
118 $\mathtt{actGrp} \xleftarrow{\perp} \mathsf{gid}$; $\mathsf{sid} \leftarrow \mathsf{sst.sid}$
119 **if** $\mathsf{gid} \in \mathtt{CGrp}$ **then**
120 $\mathrm{CanInj}[\mathsf{gid}] \xleftarrow{\perp} \mathsf{sst.rec.getIds}$
121 $*\mathsf{stg\text{-}mgmt}(\mathsf{id}, \mathsf{sid})$; $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] \leftarrow \mathsf{gid}$
122 $\mathrm{Rec}[\mathsf{id}, \mathsf{sid}, \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep}] \leftarrow \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{rec}$
123 **return** $(\mathsf{sid}, \mathsf{m}^{\mathsf{reg}})$

Oracle REGPARTICIPANT($\mathsf{gid}, \mathsf{id}$):
124 **req** $\mathsf{id} \in \mathrm{AP}[\mathsf{gid}]$
125 $(\mathsf{sst}, \mathsf{m}^{\mathsf{reg}}) \xleftarrow{\$} \mathsf{Register\text{-}P}(\mathsf{lst}_{\mathsf{id}}, \mathsf{gid})$
126 $\mathsf{sid} \leftarrow \mathsf{sst.sid}$; $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] \leftarrow \mathsf{gid}$
127 **return** $(\mathsf{sid}, \mathsf{m}^{\mathsf{reg}})$

Join Phase
Oracle JOINLEADER($\mathsf{id}, \mathsf{sid}, \mathsf{m}$):
128 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{role} = \mathsf{leader}$
129 $\mathsf{gid} \leftarrow \mathrm{RG}[\mathsf{id}, \mathsf{sid}]$; $\mathsf{gs} \leftarrow \mathrm{GS}[\mathsf{gid}]$
130 $(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m}^{\mathsf{join}}, \mathsf{rec}) \leftarrow \mathsf{Join\text{-}L}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{gs}, \mathsf{m})$
131 **if** $\mathsf{rec} \neq \mathbf{0}$ **then**
132 **if** $\mathsf{gid} \notin \mathtt{CGrp}$ **then**
133 $\mathsf{win} \leftarrow *\mathsf{winAthrz}(\mathsf{gid}, \mathsf{rec})$
134 **else** $\mathrm{CanInj}[\mathsf{gid}] \xleftarrow{\perp} \mathsf{rec.getIds}$
135 **return** $\mathsf{m}^{\mathsf{join}}$

Oracle JOINPARTICIPANT($\mathsf{id}, \mathsf{sid}, \mathsf{m}$):
136 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{role} = \mathsf{joiner}$
137 $\mathsf{gid} \leftarrow \mathrm{RG}[\mathsf{id}, \mathsf{sid}]$; $\mathsf{gs} \leftarrow \mathrm{GS}[\mathsf{gid}]$
138 $(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m}^{\mathsf{join}}, \mathsf{rec}) \leftarrow \mathsf{Join\text{-}P}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{gs}, \mathsf{m})$
139 **if** $\mathsf{rec} \neq \mathbf{0}$ **then**
140 $\mathrm{Rec}[\mathsf{id}, \mathsf{sid}, \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep}] \leftarrow \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{rec}$
141 **if** $\mathsf{gid} \notin \mathtt{CGrp}$ **then**
142 $\mathsf{win} \leftarrow *\mathsf{winAthrz}(\mathsf{gid}, \mathsf{rec})$
143 **else** $\mathrm{CanInj}[\mathsf{gid}] \xleftarrow{\perp} \mathsf{rec.getIds}$
144 **return** $\mathsf{m}^{\mathsf{join}}$

Refresh Phase
Oracle REFRESHLEADER($\mathsf{id}, \mathsf{sid}, \mathsf{m}$):
145 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{role} = \mathsf{leader}$
146 $(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{rec}) \xleftarrow{\$} \mathsf{Refresh\text{-}L}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m})$
147 $*\mathsf{refr\text{-}mgmt}(\mathsf{rec})$

Oracle REFRESHPARTICIPANT($\mathsf{id}, \mathsf{sid}$):
148 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{role} = \mathsf{parti}$
149 $(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m}^{\mathsf{refr}}, \mathsf{rec}) \xleftarrow{\$} \mathsf{Refresh\text{-}P}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}})$
150 $\mathrm{Rec}[\mathsf{id}, \mathsf{sid}, \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep}+1] \leftarrow \mathsf{rec}$
151 **return** $\mathsf{m}^{\mathsf{refr}}$

Stage Evolution Phase
Oracle EVOLVELEADER($\mathsf{id}, \mathsf{sid}, \mathsf{nxtEp}$):
152 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{role} = \mathsf{leader}$
153 $(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m}^{\mathsf{evol}}) \xleftarrow{\$} \mathsf{Evolve\text{-}L}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{nxtEp})$
154 $*\mathsf{stg\text{-}mgmt}(\mathsf{id}, \mathsf{sid})$
155 **if** $\mathsf{nxtEp} = \mathsf{true}$ **then**
156 $\mathrm{Rec}[\mathsf{id}, \mathsf{sid}, \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep}] \leftarrow \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{rec}$
157 **return** $\mathsf{m}^{\mathsf{evol}}$

Oracle EVOLVEPARTICIPANT($\mathsf{id}, \mathsf{sid}, \mathsf{m}$):
158 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{role} = \mathsf{parti}$
159 $(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{nxtEp}) \xleftarrow{\$} \mathsf{Evolve\text{-}P}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m})$
160 **if** $\mathsf{nxtEp} = \mathsf{true}$ **then**
161 $*\mathsf{refr\text{-}mgmt}(\mathsf{id}, \mathsf{sid}, *\mathsf{leader}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros}))$
162 $*\mathsf{stg\text{-}mgmt}(\mathsf{id}, \mathsf{sid})$
163 $\mathsf{win} \leftarrow *\mathsf{winGKRosAthn}(\mathsf{id}, \mathsf{sid})$

Leave Phase
Oracle LEAVELEADER($\mathsf{id}, \mathsf{sid}, \mathsf{id}', \mathsf{sid}'$):
164 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}} = \mathsf{leader}$
165 $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}} \xleftarrow{\$} \mathsf{Leave\text{-}L}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{id}', \mathsf{sid}')$

Oracle LEAVEJOINERPARTI($\mathsf{id}, \mathsf{sid}, \mathsf{id}', \mathsf{sid}'$):
166 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}} \in \{\mathsf{joiner}, \mathsf{parti}\}$
167 $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}} \xleftarrow{\$} \mathsf{Leave\text{-}P}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{id}', \mathsf{sid}')$

Fig. 6. The security game for a CGKD protocol.

Game $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ - Helper Functions

Helper $*$stg-mgmt(id, sid):
168 **req** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{gk} \neq \bot$ **and** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros} \neq \bot$
169 $\mathrm{GK}[\mathsf{id},\mathsf{sid},\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep},\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{stg}] \leftarrow \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{gk}$
170 $\mathrm{Ros}[\mathsf{id},\mathsf{sid},\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep},\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{stg}] \leftarrow \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros}$
171 $\mathsf{rec} \leftarrow *\mathsf{leader}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros})$
172 **if** $\mathsf{rec.getIds} \in \mathtt{CEp} \cup \mathrm{CanInj}[\mathsf{gid}]$ **then**
173 **foreach** $\mathsf{rec}' \in \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros}$ **do**
174 $\mathrm{CanInj}[\mathsf{gid}] \xleftarrow{\perp} \mathsf{rec.getIds}$

Helper $*$winAthrz(gid, rec):
175 $(\mathsf{id}', \mathsf{sid}', *) \leftarrow \mathsf{rec.getIds}$
176 **if** $\mathrm{RG}[\mathsf{id}', \mathsf{sid}'] \neq \mathsf{gid}$ **then return** true
177 **if** $\mathsf{id}' \notin \mathrm{AP}[\mathsf{gid}]$ **then return** true
178 **return** false

Helper $*$winGKRosAthn(id, sid):
179 $\mathsf{gid} \leftarrow \mathrm{RG}[\mathsf{id},\mathsf{sid}]$
180 $\mathsf{rec} \leftarrow *\mathsf{leader}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros})$
181 **if** $\mathsf{rec.getIds} \in \mathtt{CEp} \cup \mathrm{CanInj}[\mathsf{gid}]$ **then**
182 **return** false
183 **if** $(\mathsf{id},\mathsf{sid},\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep}) \in \mathtt{CEp}$ **then**
184 **return** false
185 $\mathsf{stg} \leftarrow \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{stg}$
186 $(\mathsf{lid},\mathsf{lsid},\mathsf{lep}) \leftarrow \mathsf{rec.getIds}$
187 **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{gk} \neq \mathrm{GK}[\mathsf{lid},\mathsf{lsid},\mathsf{lep},\mathsf{stg}]$ **then**
188 **return** true
189 **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros} \neq \mathrm{Ros}[\mathsf{lid},\mathsf{lsid},\mathsf{lep},\mathsf{stg}]$ **then**
190 **return** true
191 **return** false

Helper $*$refr-mgmt(id, sid, rec):
192 $\mathsf{gid} \leftarrow \mathrm{RG}[\mathsf{id},\mathsf{sid}]$; $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}') \leftarrow \mathsf{rec.getIds}$
193 **if** $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}' - 1) \in \mathrm{CanInj}[\mathsf{gid}]$ **then**
194 $\mathrm{CanInj}[\mathsf{gid}] \xleftarrow{\perp} (\mathsf{id}', \mathsf{sid}', \mathsf{ep}')$
195 **if** $\mathsf{id}' \in \mathtt{CPar}$ **and** $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}'-1) \in \mathtt{CEp}$ **then**
196 **if** $\mathsf{rec} \neq \mathrm{Rec}[\mathsf{id}', \mathsf{sid}', \mathsf{ep}']$ **then**
197 $\mathrm{CanInj}[\mathsf{gid}] \xleftarrow{\perp} (\mathsf{id}', \mathsf{sid}', \mathsf{ep}')$

Helper $*$winChl(b'):
198 **if** $\mathsf{b} = \mathsf{b}'$ **then**
199 **if** $\mathsf{tested} = \bot$ **or** $*\mathsf{fresh}(\mathsf{tested})$ **then**
200 **return** true
201 **return** false

Helper $*$fresh(id, sid, ep, stg):
202 $\mathsf{gid} \leftarrow \mathrm{RG}[\mathsf{id},\mathsf{sid}]$
203 $\mathsf{rec} \leftarrow *\mathsf{leader}(\mathrm{Ros}[\mathsf{id},\mathsf{sid},\mathsf{ep},\mathsf{stg}])$
204 **if** $\mathsf{rec.getIds} \in \mathtt{CEp} \cup \mathrm{CanInj}[\mathsf{gid}]$ **then**
205 **return** false
206 **if** $(\mathsf{id},\mathsf{sid},\mathsf{ep}) \in \mathtt{CEp}$ **then**
207 **return** false
208 $(\mathsf{lid},\mathsf{lsid},\mathsf{lep}) \leftarrow \mathsf{rec.getIds}$
209 $\mathsf{ros} \leftarrow \mathrm{Ros}[\mathsf{lid},\mathsf{lsid},\mathsf{lep},\mathsf{stg}]$
210 **foreach** $\mathsf{rec}' \in \mathsf{ros}$ **do**
211 $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}') \leftarrow \mathsf{rec}'.\mathsf{getIds}$
212 **if** $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}', \mathsf{stg}) \in \mathtt{CStg}$ **then**
213 **return** false
214 **if** $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}') \in \mathtt{CEp} \cup \mathrm{CanInj}[\mathsf{gid}]$ **then**
215 **return** false
216 **return** true

Fig. 7. Helper functions in our security game for a CGKD protocol.

Authorize$(\mathsf{gid},\mathsf{id})$**:** inputs the identifier gid of a group, whose secret has been initialized, and the identifier id of a party, whose long-term state has been initialized. The challenger adds id into the set AP[gid].

InitGroup$(\mathsf{gid},\mathsf{id})$**:** inputs the identifier gid of a group that has not yet been activated and the identifier id of a party authorized to join the group gid. The challenger runs Register-L on the long-term state $\mathsf{lst}_{\mathsf{id}}$ and the group identifier gid to create a session state sst and a registration message $\mathsf{m}^{\mathsf{reg}}$. Then, the challenger marks gid as activated by adding it to the set `actGrp`. If the secret of the group gid is leaked, i.e., $\mathsf{gid} \in \mathtt{CGrp}$, then the challenger marks the record underlying this session as *injectable*, i.e., adds the identifier of this record to the set CanInj[gid]. This models the real-world scenario, where an attacker can impersonate any party to join a group whose secret has been leaked. Afterwards, the group identifier, record, group key, and roster underlying sst are respectively stored in the global dictionaries. The output is the session identifier sst.sid and the registration message $\mathsf{m}^{\mathsf{reg}}$.

RegParticipant$(\mathsf{gid},\mathsf{id})$**:** inputs the identifiers gid of a group and id of a party authorized to join group gid. The challenger runs Register-P on the long-term state $\mathsf{lst}_{\mathsf{id}}$ and the group identifier gid to create a session state sst and a registration message $\mathsf{m}^{\mathsf{reg}}$. Then, the challenger stores the group identifier gid

as the registered group of the session sst. The output is the session identifier $\mathsf{sst.sid}$ and the registration message $\mathsf{m}^{\mathsf{reg}}$.

JoinLeader$(\mathsf{id}, \mathsf{sid}, \mathsf{m})$**:** inputs the identifiers id of a party and sid of its session, whose role is leader, as well as an incoming message m. The challenger runs $\mathsf{Join\text{-}L}$ using the correct secret of the group gid previously registered by this session, and obtains a message $\mathsf{m}^{\mathsf{join}}$ and the record rec of a joiner session. If the joining phase is completed, indicated by $\mathsf{rec} \neq \mathbf{0}$, the challenger evaluates the flag win by the helper function $*\mathsf{winAthrz}$ if the group gid is not compromised; otherwise, the challenger marks the joiner's record as injectable. This models the real-world scenario, where an attacker can impersonate any party to join a group whose secret has been leaked, while authorization is otherwise guaranteed. Finally, the challenger outputs the message $\mathsf{m}^{\mathsf{join}}$.

JoinParticipant$(\mathsf{id}, \mathsf{sid}, \mathsf{m})$**:** inputs the identifiers id of a party and sid of its session, whose role is joiner, as well as an incoming message m. The challenger runs $\mathsf{Join\text{-}P}$ using the correct secret of the group gid previously registered by this session, and obtains a message $\mathsf{m}^{\mathsf{join}}$ and the record rec of a leader session. If the joining phase is completed, indicated by $\mathsf{rec} \neq \mathbf{0}$, the challenger stores the record of this session. Then, the challenger either evaluates the flag win or marks the leader's record as injectable, analogously to the execution of the JoinLeader oracle. Finally, the challenger outputs the message $\mathsf{m}^{\mathsf{join}}$.

RefreshLeader$(\mathsf{id}, \mathsf{sid}, \mathsf{m})$**:** inputs the identifiers id of a party and sid of its session, whose role is leader, as well as an incoming message m. The challenger runs $\mathsf{Refresh\text{-}L}$ and obtains the next record rec of a participant session. The challenger performs refresh management by marking this record as injectable, if either (1) its current record underlying the that participant's session state is already injectable, or (2) both the long-term state and the session state of that participant are compromised and the received record rec is injected.

RefreshParticipant$(\mathsf{id}, \mathsf{sid})$**:** inputs the identifiers id of a party and sid of its session, whose role is parti. The challenger runs $\mathsf{Refresh\text{-}P}$ and obtains the next record rec of a participant session as well as a refresh message $\mathsf{m}^{\mathsf{refr}}$. The challenger stores the record rec and outputs the refresh message $\mathsf{m}^{\mathsf{refr}}$.

EvolveLeader$(\mathsf{id}, \mathsf{sid}, \mathsf{nxtEp})$**:** inputs the identifiers id of a party and sid of its session, whose role is leader, as well as a flag nxtEp indicating whether the session should advance to the next epoch. The challenger runs $\mathsf{Evolve\text{-}L}$ and obtains an evolve message $\mathsf{m}^{\mathsf{evol}}$. Then, the challenger performs stage management, i.e., storing the group key and roster underlying the session $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}$ in their respective global dictionaries, and marking all records included in the roster as injectable if the leader specified by the local roster is either compromised or already injectable. If the flag nxtEp is true, the challenger additionally stores the record underlying $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}$. Finally, the challenger outputs the evolve message $\mathsf{m}^{\mathsf{evol}}$.

EvolveParticipant$(\mathsf{id}, \mathsf{sid}, \mathsf{m})$**:** inputs the identifiers id of a party and sid of its session, whose role is parti, as well as an incoming message m. The challenger runs $\mathsf{Evolve\text{-}P}$ and obtains a flag nxtEp indicating whether the participant session advances to the next epoch. If the flag nxtEp is true, the challenger performs the refresh management for the leader's record. Finally, the challenger

performs stage management and evaluates the flag win by the helper function $*$winGKRosAthn.

LeaveLeader$(\mathsf{id}, \mathsf{sid}, \mathsf{id}', \mathsf{sid}')$**:** inputs the identifiers id of a party and sid of its session, whose role is leader, as well as id' of a party and sid' of its session that is leaving the group. The challenger simply runs Leave-L.

LeaveJoinerParti$(\mathsf{id}, \mathsf{sid}, \mathsf{id}', \mathsf{sid}')$**:** inputs the identifier id of a party and sid of its session, whose role is parti or joiner, as well as id' of a party and the identifier sid' of its session that is leaving the group. The challenger simply runs Leave-P.

Corrupt(id)**:** inputs the identifier id of a party to be corrupted. The challenger simply adds id to the set CPar and outputs the long-term state of that party.

Reveal(gid)**:** inputs the identifier gid of a group, whose secret to be revealed. The challenger adds gid to the set CGrp and outputs the secret of that group.

Compromise$(\mathsf{id}, \mathsf{sid})$**:** inputs the identifiers id of a party and sid of its session, whose role is either leader or parti. The challenger marks the current record, the next record (if available), and the current stage of the session state $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}$ as compromised by adding their respective identifiers to the global dictionaries. Finally, the challenger outputs the session state $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}$.

Leak$(\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg})$ inputs the identifier id of a party, the identifier sid of its session, an epoch ep, and a stage stg, such that the corresponding group key has been generated. The challenger adds the input tuple to the set CStg and outputs that group key.

Test$(\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg})$**:** inputs the identifier id of a party, the identifier sid of its session, an epoch ep, and a stage stg, such that the corresponding group key has been generated. This oracle can be queried at most once, indicated by the global variable $\mathsf{tested} = \bot$. The challenger sets tested to the input tuple and outputs the real corresponding group key if the challenge bit $\mathsf{b} = 0$ or a random key if $\mathsf{b} = 1$.

Our security game $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ captures all security properties in Section 5.1.

1. **(Strong) Group Key Authentication** and **(Strong) Group Roster Authentication** are captured by the helper function $*$winGKRosAthn. This helper function outputs true if and only if the group key or the roster of a member is unequal to the one of the leader, unless (1) the leader's session state is compromised or its record is injectable (indicating that the leader is not honest), or (2) that group member's session state is compromised.
2. **Group Key Secrecy**, **Perfect Forward Secrecy (across sessions)**, and **Perfect Forward Secrecy (within sessions)** are captured by the helper function $*$winChl. This helper function outputs true if and only if the bit b' output by the attacker equals the challenge bit b, and either the global variable $\mathsf{tested} = \bot$ or the helper function $*$fresh evaluates to true on tested. The freshness condition on $\mathsf{tested} = (\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg})$ fails only if there exists any member in the group of party id's session sid at epoch ep for which one of the following conditions holds: (1) the group key at stage stg is leaked, (2) the session state is compromised, or (3) the record is injected (indicating that this member is not honest).

3. **(Post-Compromise) Group Member Authorization** are captured by the helper function $*\mathsf{winAthrz}$, which is invoked only if the caller's group secret has not been revealed. This helper function outputs true if and only if the partner session, as indicated by the accepted record, is both registered and authorized for the same group as the caller's session.
4. **Post-Compromise Security** is captured by two facts during the refresh phase. On one hand, after each participant's session refreshment, the identifier of every newly generated record is not added to the set $\mathtt{CEp}$, reflecting that this refreshed session state is not considered compromised. On the other hand, after each record recovery by the leader, the identifiers of the recovered record rec is not added to the dictionary CanInj if the record was honestly generated by the party $\mathsf{rec.id}$ in the session $\mathsf{rec.sid}$ at the epoch $\mathsf{rec.ep}$, reflecting that the record is not considered injectable against a passive attacker.

Definition 7 (Security of CGKD). *Let* CGKD *denote a continuous group key distribution protocol and let* α_{min} *denote its required minimum entropy. We say* CGKD *is* $\epsilon^{\mathsf{sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$*-secure, if the below defined advantage* $\mathsf{Adv}^{\mathsf{sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ *of any attacker* $\mathcal{A}$ *in the game* $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ *in Figure 6 is bounded by*

$$\mathsf{Adv}^{\mathsf{sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}(\mathcal{A}) := \Pr[\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}(\mathcal{A}) = \mathsf{true}] - \frac{1}{2} \leq \epsilon^{\mathsf{sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$$

6 Our Generic Construction

In this section, we propose our CGKD construction, depicted in Figures 8 and 9. We prove that our construction can satisfy both correctness and security in Appendix K and Appendix J, respectively.

Additional State Variables. Our CGKD construction includes the following additional state variables:

- $\mathsf{sst.nxtSpk}$: the session public key used for the next epoch; initialized with $\perp$.
- $\mathsf{sst.nxtSsk}$: the session secret key used for the next epoch; initialized with $\perp$.
- $\mathsf{sst.nxtRec}$: the record used for the next epoch; initialized with $\perp$.
- $\mathsf{sst.nxtRos}$: the roster used for the next epoch; specific to leaders and initialized with $()$.
- $\mathsf{sst.Temp}[\cdot,\cdot]$: a dictionary that stores the temporal data during the participant-join phase; specific to leaders and initialized with $\perp$ for every entry.
- $\mathsf{sst.temp}$: a value that stores the temporal data during the participant-join phase; specific to joiners and initialized with $\perp$.

Our CGKD construction follows the syntax in Definition 6 and makes use of five primitives: a two-round password-authenticated key exchange (PAKE) following the syntax in [32], a multi-recipient authenticated key encapsulation mechanism (maKEM) in Section 3, a digital signature (DS), a message authentication code (MAC), and a key derivation function (KDF). We present a high-level intuition behind our construction below and provide a detailed description in Appendix C.

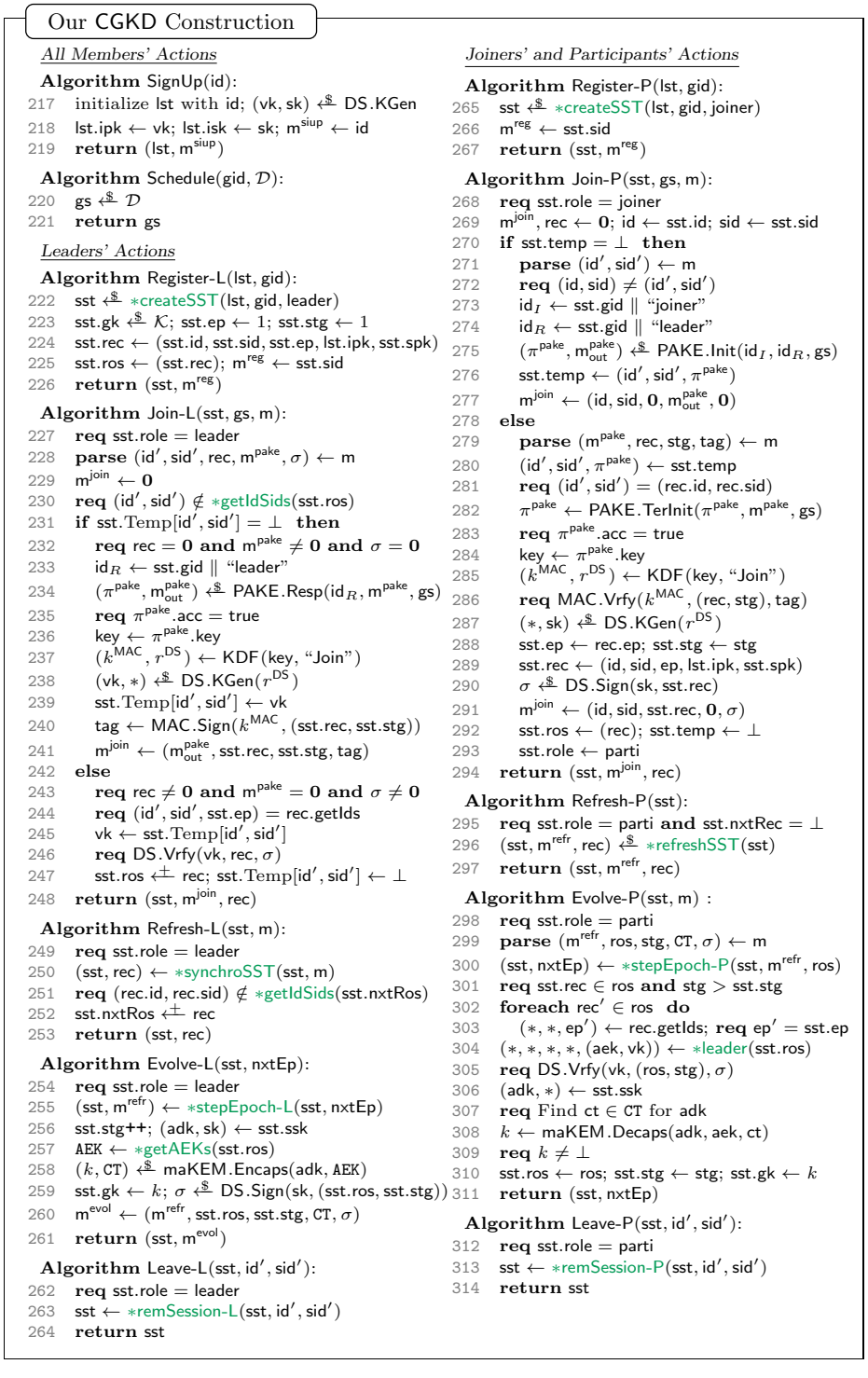


Fig. 8. Our CGKD construction.



Fig. 9. The helper functions in our CGKD protocol.

SignUp(id)**:** inputs the identifier id of a party. This party initializes its long-term state lst, generates a DS public-secret key pair (vk, sk) as its identity key pair, and outputs an outgoing sign-up message $\mathsf{m}^{\mathsf{siup}}$ that equals id.

Schedule(gid, $\mathcal{D}$)**:** inputs a group identifier gid and a group secret distribution $\mathcal{D}$. The group secret gs is sampled following the distribution $\mathcal{D}$ and output.

Register-L(lst, gid) **and** Register-P(lst, gid)**:** both algorithms input a long-term state lst and a group identifier gid. First, the caller creates a new session state in the following four steps: (1) sample a session identifier sid uniformly at random, (2) set the role of the session state to either leader or joiner, (3) generate a maKEM key pair (aek, adk) and a DS key pair (vk, sk), (4) set (aek, vk) as the session public key and their secret keys as session secret key. If the caller is the leader, then it additionally initializes more session state variables in the following four steps: (1) sample a new group key uniformly at random, (2) set both the epoch and stage to 1, (3) generate a new record using the local state, and (4) set the local roster to a list that only contains the new record. Finally, the caller outputs the created session state and a registration message that equals its session identifier.

Join-L(sst, gs, m) **and** Join-P(sst, gs, m)**:** The join phase consists of four steps.

1. The joiner parses a party-session identifier pair of the potential leader from the incoming message m, followed by running the PAKE protocol using the correct group secret. Afterwards, the joiner stores the parsed identifiers and the PAKE states in the temporal variable sst.temp. The output message includes its own party-session identifiers and the outgoing PAKE message.
2. Upon receiving $\mathsf{m}^{\mathsf{join}}$, the leader runs the PAKE protocol using the PAKE message included in $\mathsf{m}^{\mathsf{join}}$ and the correct group secret. Recall that the underlying PAKE protocol is two-round. The leader terminates the PAKE protocol and derives a MAC key and a DS public key from this PAKE key from the resulting PAKE key. The MAC key is used to authenticate its record and stage in this step, while the DS public key is temporally stored in the state and will be used to verify the messages from the joiner in the next step. This prevents an attacker from impersonating the joiner towards the leader, even if it can compromise the leader's state, thereby enabling post-compromise authorization. The output message $\mathsf{m}^{\mathsf{join}}$ includes the leader's PAKE outgoing message, record, stage, and MAC tag.
3. Upon receiving the $\mathsf{m}^{\mathsf{join}}$, the joiner runs the PAKE protocol followed by deriving a MAC key and a DS secret key from the key PAKE. Then, the joiner verifies the leader's record and stage by using MAC, generates its own record, and signs this record by using DS. Afterwards, the joiner changes its role to parti and outputs the leader's record. The output message includes the participant's party-session identifiers, record, and DS signature.
4. Upon receiving the second message $\mathsf{m}^{\mathsf{join}}$, the leader verifies the record included in the input message by using the stored DS public key. If the verification passes, the leader adds it into the local roster, erases the data in the Temp for this joining phase, and outputs an empty joining message as well as the verified record.

Refresh-L(sst, m) **and** Refresh-P(sst)**:** The refresh phase starts with a participant running Refresh-P. The participant generates the a session public-secret key pair and a new record for the next epoch. The new record is signed twice, respectively by using DS secret keys included in the long-term and the current session secret keys. The output message $\mathsf{m}^{\mathsf{refr}}$ consisting of the next record and both DS signatures.

Upon receiving the message $\mathsf{m}^{\mathsf{refr}}$, the leader checks whether the included record (1) is a successor of a record in the leader's current roster, (2) indicates the next epoch, and (3) can pass the verification on both DS signatures in $\mathsf{m}^{\mathsf{refr}}$ respectively under the identity and session public keys contained in its predecessor record in the leader's current roster. If any above check fails, the leader aborts the execution and rolls back its session state. Otherwise, the leader adds the incoming record rec into its local next roster $\mathsf{sst.nxtRos}$.

$\mathsf{Evolve\text{-}L}(\mathsf{sst}, \mathsf{nxtEp})$ **and** $\mathsf{Evolve\text{-}P}(\mathsf{sst}, \mathsf{m})$**:** The evolve phase starts with a leader running $\mathsf{Evolve\text{-}L}$. If the flag $\mathsf{nxtEp} = \mathsf{true}$, the leader refreshes its own session states for a refresh message $\mathsf{m}^{\mathsf{refr}}$, similar to the $\mathsf{Refresh\text{-}P}$ execution. Then, the leader immediately overwrites its current session public key, session secret key, record, and roster by the next ones, which are subsequently erased. Afterwards, the leader reinitializes the next roster with a list that only includes its own record, increments the local epoch by 1 and sets the stage to 0. Otherwise, the leader sets the outgoing message $\mathsf{m}^{\mathsf{refr}}$ to an empty string. In both cases, the leader increments its stage by 1.

Then, the leader extracts the maKEM and DS secret keys in its session secret key, uses DS to sign the current roster and stage, and uses maKEM to encapsulates a new group key. Here, the maKEM public keys are the ones contained in all records included in its local roster. The output message $\mathsf{m}^{\mathsf{evol}}$ consists of the leader's refresh message $\mathsf{m}^{\mathsf{refr}}$, its current roster, its current stage, the set of maKEM ciphertexts $\mathtt{CT}$, and the DS signature.

Upon receiving the message $\mathsf{m}^{\mathsf{evol}}$, if this message includes a non-empty fresh message, the participant recovers the leader's next record rec and checks whether it is a successor of the current leader record, followed by updating session states similar to the leader's execution in the $\mathsf{Evolve\text{-}L}$. The current roster is set to a list that only includes the recovered leader record. Then, the participant performs a sequence of checks on the validity of the roster, the stage, and the maKEM ciphertexts in the input[2], followed by recovering a group key. When all checks pass, the participant sets its roster and stage to the input values, and outputs its updated session state and the flag nxtEp.

$\mathsf{Leave\text{-}L}(\mathsf{sst}, \mathsf{id}', \mathsf{sid}')$ **and** $\mathsf{Leave\text{-}P}(\mathsf{sst}, \mathsf{id}', \mathsf{sid}')$**:** both algorithms input a session state sst, an identifier id' of a leaving party, and a session identifier sid' of its session. If the input identifiers refer to the caller's own session, then the caller simply erases its session state. Otherwise, the caller removes all records corresponding to the input identifiers from the local roster.

If the caller is a leader, it also removes all corresponding records from its next-epoch roster $\mathsf{sst.nxtRos}$ if the leaving party is a participant, and erases the corresponding temporary data in the dictionary sst.Temp if the leaving party is a joiner. Otherwise, the caller is a participant or joiner. In this case, it erases

[2] Here, we simply assume that there exists an efficient approach for participants to find the desired maKEM ciphertext ct from the input set $\mathtt{CT}$ at Line 307. In practice, this can be easily realized by (1) having the leader attach each participant's maKEM public key to the corresponding maKEM ciphertext, or (2) having participants attempt to decapsulate the maKEM ciphertexts one by one.

its session state if the current roster becomes empty, and switches to the leader role if it becomes the leader in the current roster.
Finally, the caller outputs its updated session state.

Theorem 7. *Let* CGKD *denote the CGKD construction in Figure 8. If the underlying* DS *is* $\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}$-EUF-CMA *secure,* maKEM *is* $\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{maKEM}}$-IND-CCA *and* $\epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{maKEM}}$-CTI-CCA *secure,* MAC *is* $\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}}$-EUF-CMA *secure,* KDF *is* $\epsilon^{\mathsf{prf}}_{\mathsf{KDF}}$-PRF *secure, and* PAKE *is* $\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha_{\mathsf{min}}}$-wBPR[3] *secure, then* CGKD *is* $\epsilon^{\mathsf{sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$-Sec *secure in Definition 7, where*

$$\begin{aligned}\epsilon^{\mathsf{sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}} \leq & (q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 \cdot 2^{-\lambda} + q_{\mathrm{NG}}\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha_{\mathsf{min}}} + q_{\mathrm{NG}}(q_{\mathrm{JL}} + q_{\mathrm{JP}})\epsilon^{\mathsf{prf}}_{\mathsf{KDF}} \\ & + q_{\mathrm{NG}}q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}} + q_{\mathrm{NG}}(q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}} \\ & + q_{\mathrm{NG}}\epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{maKEM}} + q_{\mathrm{NG}}q_{\mathrm{EL}}\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{maKEM}}\end{aligned}$$

Here, q_{IG}*,* q_{RegP}*,* q_{NG}*,* q_{JL}*,* q_{NP}*,* q_{RefP}*, and* q_{EL} *respectively denote the number of queries to the* InitGroup*,* RegParticipant*,* NewGroup*,* JoinLeader*,* NewParty*,* RefreshParticipant*, and* EvolveLeader *oracles.*

Proof (Sketch). Note that session identifiers are sampled only in the Register-L and Register-P algorithms from the set $\{0,1\}^\lambda$, and that these two algorithms are executed only in the InitGroup and RegParticipant oracles, respectively. All session identifiers in the game are distinct except with probability at most $(q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 \cdot 2^{-\lambda}$. The challenger first guesses the group identifier gid that the attacker targets by guessing the index of the queries to the NewGroup oracle. Then, we separately consider the cases that the attacker triggers win $\leftarrow$ true via $*$winAthrz, $*$winGKRosAthn, and $*$winChl helper functions.

Case 1: Note that the attacker can trigger win $\leftarrow$ $*$winAthrz only if the target group secret is not revealed. This means that the entropy of that group secret distribution must be at least α_{min}. During the Join phase within the target group, every PAKE key is random and can be known by (1) either a leader, (2) or joiner, (3) or both of them if they share the same group secret and the same PAKE transcript, except for the probability at most $\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha_{\mathsf{min}}}$. Due to the PRF security of KDF, the MAC key and DS key pair derived from every PAKE key must also be random, except for the probability at most $(q_{\mathrm{JL}} + q_{\mathrm{JP}})\epsilon^{\mathsf{prf}}_{\mathsf{KDF}}$. Finally, an authorized party (either a leader or a joiner) can complete the joining phase with an unauthorized party within the target group, only if the attacker can forge a corresponding MAC tag or DS signature, which happens with probability at most $q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}} + q_{\mathrm{JL}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}$.

Case 2: Note that the attacker can trigger win $\leftarrow$ $*$winGKRosAthn when a participant id's session sid is evolving within the target group, only if its leader's record is not injectable. Note also that this can happen only if all past group leaders are recursively non-injectable. This means, the secret of the target group

[3] Our weak BPR (wBPR) model, see Appendix A.7, is a weaker variant of the BPR model in [32]: it (1) considers only one initiator-responder pair, and (2) does not allow key-reveal or password-corruption queries. .

must not be revealed. Following Case 1, every leader within the target group exchanges honest records with authorized joiners, and vice versa. For leaders and participants, note that their records will be signed twice in the EvolveLeader or RefreshParticipant oracle, respectively by using its long-term key and session public key. Even if either DS secret key is leaked, the probability that the attacker can forge a new record of any leader or participant within the target group is bounded by $(q_{\text{NP}} + q_{\text{RefP}} + q_{\text{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$. Particularly, this means that the participant that the attacker is aiming at and its leader must agree on each other's honest record. Note that every roster is signed by DS using the leader's session secret key, which is included in its record. The probability that the attacker can forge a roster is therefore bounded by $(q_{\text{RefP}} + q_{\text{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$. By the CTI-CCA security of maKEM, the probability that the attacker can forge a group key is bounded by $\epsilon_{\mathsf{maKEM}}^{\mathsf{cti\text{-}cca}}$.

Case 3: Note that $\mathsf{win} \leftarrow *\mathsf{winChl}$ holds only if the attacker can distinguish a real group key at an epoch ep and stage stg of a party id's session sid from random, without violating the refreshness condition. If $\mathsf{tested} = \perp$, then the attacker obtains no information about the challenge bit and therefore can guess it correctly with probability at most $\frac{1}{2}$. Otherwise, note that the freshness condition covers $*\mathsf{winGKRosAthn}$ evaluation. Following Case 2, the tested leader obtains the honest records of every participants at the tested epoch and stage. The challenger guesses the query to the EvolveLeader oracle that produces the challenged group key with probability at least $\frac{1}{q_{\text{EL}}}$. Due to the IND-CCA security of the underlying maKEM, the probability that the attacker can distinguish the encapsulated group key from random with probability at most $\epsilon_{\mathsf{maKEM}}^{\mathsf{ind\text{-}cca}} + \frac{1}{2}$.

The proof is concluded by the statements above.

Instantiations. We provide concrete instantiations for our CGKD construction in both classical and post-quantum settings. The underlying two-round PAKE can be instantiated with the generic construction in [32] or concrete constructions such as CPace [27,3] and SPAKE2 [4,1,2] for classical security, and with the generic construction CAKE [13] for post-quantum security. The underlying DS can be instantiated with Ed25519 original version [16] for classical security, whose EUF-CMA security is proven in [18], and the NIST suggested CRYSTALS-Dilithium for post-quantum security, whose EUF-CMA security is proven in [11]. The underlying KDF can be instantiated with HMAC-SHA256 or HKDF for classical security, and by doubling the key length for post-quantum security. The underlying MAC can be instantiated with HMAC-SHA256 or Poly1305-AES for classical security, and by doubling the key length for post-quantum security. The underlying maKEM can be instantiated with our constructions in Section 3.

7 Summary

In this paper, we propose a new CGKD protocol for real-time group communication that allows parties to refresh their states, along with an associated correctness model that captures the intended functionality under parties' honest execution, assuming a trusted server.

While a malicious server may break correctness, the security of the group communication should remain intact. To model the impact of malicious servers on security, we define a set of core security guarantees and an associated game-based security model that captures all of these guarantees. Our model relies only on realistic assumptions common in practice and does not require any trust in the servers. Moreover, our model allows attackers to perform session-state compromise on some group-specific secrets, enabling us to analyze the impact of their potential leakage in unexpected scenarios. For example, if a party's group-specific state is not properly erased and becomes vulnerable, our model captures that secrets in the group communication prior to the party's last state refresh remain secure. In the literature, a number of UC-style security models are defined for related domain CGKA (e.g., [9,10,22]. We leave the UC-style security models and analyses for CGKD as an open question.

We also provide a generic construction that provably satisfies both the correctness and security models, and suggest concrete instantiations for both classical and post-quantum security. Notably, this makes our construction the first provably post-quantum secure end-to-end real-time group communication protocol.

In addition to these main contributions, we introduce a novel maKEM primitive, which serves as a building block in our generic CGKD construction and may be of independent research interest. We define two associated security notions for confidentiality and authenticity, provide both a concrete and a generic constructions, and prove their security in both models. This work aims only to initiate the study of maKEM and to demonstrate its feasibility, targeting neither efficiency optimization nor a comprehensive exploration of its security requirements. We leave a systematic study of the security landscape of maKEM, as well as efficient constructions, as an open problem.

Acknowledgements. The first author Mang Zhao's work was supported by National Natural Science Foundation of China (NSFC) under Grant U25A20435. The corresponding author Qian Wang's work was supported by the NSFC under Grant U2441240 ("Ye Qisun" Science Foundation).

References

1. Abdalla, M., Barbosa, M.: Perfect Forward Security of SPAKE2. Cryptology ePrint Archive, Paper 2019/1194 (2019), https://eprint.iacr.org/2019/1194
2. Abdalla, M., Barbosa, M., Bradley, T., Jarecki, S., Katz, J., Xu, J.: Universally Composable Relaxed Password Authenticated Key Exchange. Cryptology ePrint Archive, Paper 2020/320 (2020), https://eprint.iacr.org/2020/320
3. Abdalla, M., Haase, B., Hesse, J.: Security analysis of CPace. In: Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part IV. pp. 711–741. Springer (2021)
4. Abdalla, M., Pointcheval, D.: Simple password-based encrypted key exchange protocols. In: Topics in Cryptology–CT-RSA 2005: The Cryptographers' Track at

the RSA Conference 2005, San Francisco, CA, USA, February, 2005. Proceedings. pp. 191–208. Springer (2005)
5. Alt, S.: Authenticated hybrid encryption for multiple recipients. Cryptology ePrint Archive, Paper 2006/029 (2006), https://eprint.iacr.org/2006/029
6. Alwen, J., Blanchet, B., Hauck, E., Kiltz, E., Lipp, B., Riepel, D.: Analysing the hpke standard. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 87–116. Springer (2021)
7. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, Santa Barbara, CA, USA, August, 2020, Proceedings, Part I. pp. 248–277. Springer (2020)
8. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of mls. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 1463–1483 (2021)
9. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Theory of Cryptography Conference. pp. 261–290. Springer (2020)
10. Alwen, J., Jost, D., Mularczyk, M.: On the insider security of mls. In: Annual International Cryptology Conference. pp. 34–68. Springer (2022)
11. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation (Version 3.1), https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf
12. Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., Cohn-Gordon, K.: The Messaging Layer Security (MLS) Protocol, https://datatracker.ietf.org/doc/rfc9420/
13. Beguinet, H., Chevalier, C., Pointcheval, D., Ricosset, T., Rossi, M.: Get a cake: Ge neric t ransformations from k ey encaspulation mechanisms to password a uthenticated k ey e xchanges. In: International Conference on Applied Cryptography and Network Security. pp. 516–538. Springer (2023)
14. Bellare, M., Boldyreva, A., Staddon, J.: Multi-recipient encryption schemes: Security notions and randomness re-use. In: PKC. vol. 2003, pp. 85–99 (2003)
15. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated Key Exchange Secure against Dictionary Attacks. In: Preneel, B. (ed.) Advances in Cryptology — EUROCRYPT 2000. pp. 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
16. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of cryptographic engineering **2**(2), 77–89 (2012)
17. Blum, J., Booth, S., Chen, B., Gal, O., Krohn, M., Len, J., Lyons, K., Marcedone, A., Maxim, M., Mou, M.E., Namavari, A., O'Connor, J., Rien, S., Steele, M., Green, M., Kissner, L., Stamos, A.: Zoom cryptography whitepaper (version 4.7), https://github.com/zoom/zoom-e2e-whitepaper/tree/master Accessed on Feb. 2026
18. Brendel, J., Cremers, C., Jackson, D., Zhao, M.: The provable security of Ed25519: theory and practice. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1659–1676. IEEE (2021)
19. Brendel, J., Fischlin, M., Günther, F., Janson, C.: PRF-ODH: Relations, instantiations, and impossibility results. In: Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part III 37. pp. 651–681. Springer (2017)

20. Brzuska, C., Cornelissen, E., Kohbrok, K.: Security analysis of the mls key derivation. In: 2022 IEEE Symposium on security and privacy (SP). pp. 2535–2553. IEEE (2022)
21. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On Ends-to-Ends Encryption: Asynchronous group messaging with strong security guarantees. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (2018)
22. Cremers, C., Günsay, E., Wesselkamp, V., Zhao, M.: Etk: External-operations treekem and the security of mls in rfc 9420. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 3–33. Springer (2026)
23. Cremers, C., Hale, B., Kohbrok, K.: The complexities of healing in secure group messaging: Why {Cross-Group} effects matter. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 1847–1864 (2021)
24. Cremers, C., Ronen, E., Zhao, M.: Multi-stage group key distribution and pakes: Securing zoom groups against malicious servers without new security elements. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 2686–2704. IEEE (2024)
25. Dodis, Y., Jost, D., Kesavan, B., Marcedone, A.: End-to-end encrypted zoom meetings: Proving security and strengthening liveness. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 157–189. Springer (2023)
26. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: Annual international cryptology conference. pp. 537–554. Springer (1999)
27. Haase, B., Labrique, B.: AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(2), 1–48 (Feb 2019). https://doi.org/10.13154/tches.v2019.i2.1-48, https://tches.iacr.org/index.php/TCHES/article/view/7384
28. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the fujisaki-okamoto transformation. In: Theory of Cryptography Conference. pp. 341–371. Springer (2017)
29. Isobe, T., Ito, R.: Security analysis of end-to-end encryption for Zoom meetings. IEEE Access **9**, 90677–90689 (2021)
30. Josefsson, S., Liusvaara, I.: Rfc 8032: Edwards-curve digital signature algorithm (eddsa) (2017)
31. Kurosawa, K.: Multi-recipient public-key encryption with shortened ciphertext. In: Naccache, D., Paillier, P. (eds.) Public Key Cryptography. pp. 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
32. Pan, J., Zeng, R.: A generic construction of tightly secure password-based authenticated key exchange. In: Guo, J., Steinfeld, R. (eds.) Advances in Cryptology – ASIACRYPT 2023. pp. 143–175. Springer Nature Singapore, Singapore (2023)
33. Smart, N.P.: Efficient key encapsulation to multiple parties. In: International Conference on Security in Communication Networks. pp. 208–219. Springer (2004)
34. Wallez, T., Protzenko, J., Beurdouche, B., Bhargavan, K.: {TreeSync}: authenticated group management for messaging layer security. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 1217–1233 (2023)
35. Wallez, T., Protzenko, J., Bhargavan, K.: Treekem: A modular machine-checked symbolic security analysis of group key agreement in messaging layer security. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 4375–4390. IEEE (2025)

Appendix: Supplementary Material

In the following sections, we provide supplementary material that includes preliminaries, detailed descriptions, and all proofs. In Appendix A, we recall all preliminaries. In Appendix B, we define correctness model for a CGKD protocol and prove that our construction provably satisfies it. In Appendix C, we provide a detailed description for our CGKD construction in Section 6. In Appendix D, we provide our proof for Theorem 1. In Appendix E, we provide our proof for Theorem 2. In Appendix F, we provide our proof for Theorem 3. In Appendix G, we provide our proof for Theorem 4. In Appendix H, we provide our proof for Theorem 5. In Appendix I, we provide our proof for Theorem 6. In Appendix K, we provide our proof for Theorem 8. In Appendix J, we provide our proof for Theorem 7 and its related Lemma 1.

A Preliminaries

A.1 Message Authentication Code

Definition 8. *Let $\mathcal{K}$ and $\mathcal{M}$ respectively denote the space of keys and messages. A* message authentication code *(MAC) scheme* $\mathsf{MAC} = (\mathsf{MAC.Sign}, \mathsf{MAC.Vrfy})$ *is a tuple of algorithms, where*

Signing $\mathsf{tag} \leftarrow \mathsf{MAC.Sign}(k, m)$ *inputs a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$ and (deterministically) outputs a tag* tag*.*

Verification $\mathsf{true/false} \leftarrow \mathsf{MAC.Vrfy}(k, m, \mathsf{tag})$ *inputs a key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$, and a tag* tag*, and deterministically outputs* true *if the verification passes and* false *otherwise.*

Definition 9 (Correctness of MAC). *Let* $\mathsf{MAC} = (\mathsf{MAC.Sign}, \mathsf{MAC.Vrfy})$ *be a message authentication code scheme. We say* MAC *is δ_{MAC}-correct, if for every key $k \xleftarrow{\$} \mathcal{K}$ and every message $m \in \mathcal{M}$, the probability below is bounded by δ_{MAC}:*

$$\Pr\left[\mathsf{false} = \mathsf{MAC.Vrfy}(k, m, \mathsf{tag}) \,|\, \mathsf{tag} \leftarrow \mathsf{MAC.Sign}(k, m)\right] \leq \delta_{\mathsf{MAC}}$$

Definition 10. *We say a* MAC *is $\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}}$-*EUF-CMA *secure, if the below defined advantage of any attacker $\mathcal{A}$ against $\mathbf{G}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}$ game in Figure 10 is bounded by,*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}} := \Pr[\mathbf{G}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A}) = 1] \leq \epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}}$$

A.2 (Randomized) Authenticated Encryption

Definition 11. *Let $\mathcal{K}$, $\mathcal{M}$, and $\mathcal{CT}$ respectively denote the space of keys, messages, and ciphertexts. A* (randomized) authenticated encryption *(AE) scheme* $\mathsf{AE} = (\mathsf{AE.Enc}, \mathsf{AE.Dec})$ *is a tuple of algorithms, where*

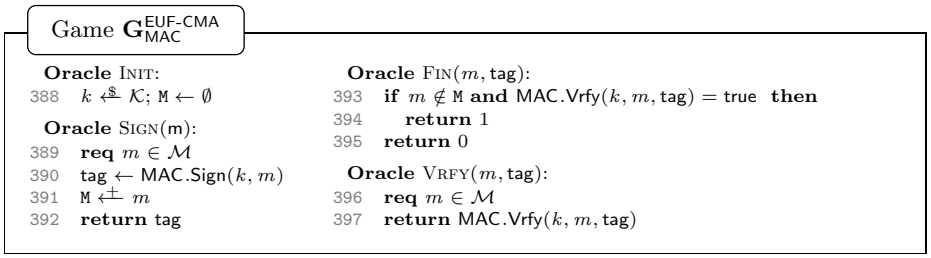


Fig. 10. The EUF-CMA security game for a MAC scheme.

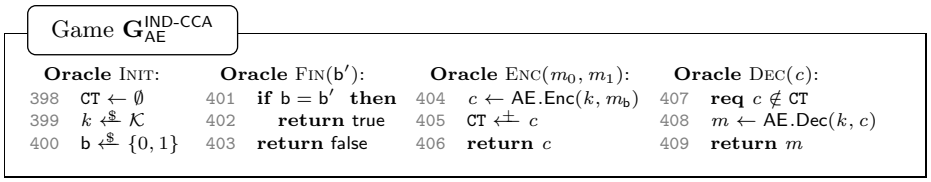


Fig. 11. The IND-CCA security for a (randomized) AE scheme.

Encryption $c \leftarrow \mathsf{AE.Enc}(k, m)$ *inputs a key* $k \in \mathcal{K}$ *and a message* $m \in \mathcal{M}$ *and outputs a ciphertext* $c \in \mathcal{CT}$.

Decryption $m \leftarrow \mathsf{AE.Dec}(k, c)$ *inputs a key* $k \in \mathcal{K}$ *and a ciphertext* $c \in \mathcal{CT}$ *and deterministically outputs a message* $m \in \mathcal{M} \cup \{\perp\}$.

Definition 12 (Correctness of (Randomized) AE). *Let* $\mathsf{AE} = (\mathsf{AE.Enc}, \mathsf{AE.Dec})$ *be a (randomized) AE scheme. We say* AE *is* δ_{AE}*-correct, if for every key* $k \xleftarrow{\$} \mathcal{K}$ *and every message* $m \in \mathcal{M}$*, the probability below is bounded by* δ_{AE}*:*

$$\Pr\left[m \neq \mathsf{AE.Dec}(k, c) \mid c \leftarrow \mathsf{AE.Enc}(k, m)\right] \leq \delta_{\mathsf{AE}}$$

Definition 13 (IND-CCA Security of (Randomized) AE). *We say a (randomized)* AE *is* $\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{AE}}$*-IND-CCA secure, if the below defined advantage of any attacker* $\mathcal{A}$ *against* $\mathbf{G}^{\mathsf{IND\text{-}CCA}}_{\mathsf{AE}}$ *game in Figure 11 is bounded by,*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{AE}} := \left| \Pr[\mathbf{G}^{\mathsf{IND\text{-}CCA}}_{\mathsf{AE}}(\mathcal{A}) = \mathsf{true}] - \frac{1}{2} \right| \leq \epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{AE}}$$

Definition 14 (CTI-CCA Security of (Randomized) AE). *We say a (randomized)* AE *is* $\epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{AE}}$*-CTI-CCA secure, if the below defined advantage of any attacker* $\mathcal{A}$ *against* $\mathbf{G}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{AE}}$ *game in Figure 12 is bounded by,*

$$\mathsf{Adv}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{AE}} := \Pr[\mathbf{G}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{AE}}(\mathcal{A}) = 1] \leq \epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{AE}}$$

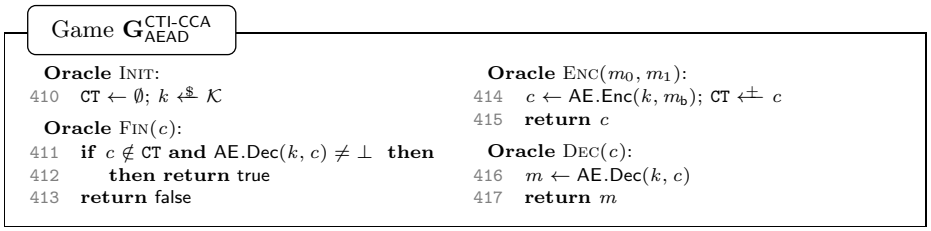


Fig. 12. The CTI-CPA security for a (randomized) AE scheme.

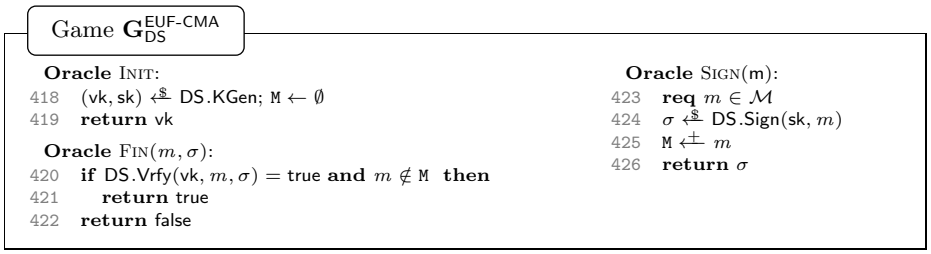


Fig. 13. The EUF-CMA security game for a DS scheme.

A.3 Digital Signature

Definition 15. *Let $\mathcal{M}$ denote the space of messages. A* digital signature *(DS) scheme* $\mathsf{DS} = (\mathsf{DS.KGen}, \mathsf{DS.Sign}, \mathsf{DS.Vrfy})$ *is a tuple of algorithms, where*

Key Generation $(\mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{DS.KGen}$ *outputs a public verification key* vk *and a private signing key* sk*.*

Signing $\sigma \xleftarrow{\$} \mathsf{DS.Sign}(\mathsf{sk}, m)$ *inputs a private signing key* sk*, a message $m \in \mathcal{M}$, and outputs a signature σ.*

Verification $\mathsf{true/false} \leftarrow \mathsf{DS.Vrfy}(\mathsf{vk}, m, \sigma)$ *inputs a public verification key* vk*, a message $m \in \mathcal{M}$, and a signature σ, and deterministically outputs* true *if the verification passes and* false *otherwise.*

Definition 16 (Correctness of DS). *Let* $\mathsf{DS} = (\mathsf{DS.KGen}, \mathsf{DS.Sign}, \mathsf{DS.Vrfy})$ *be a DS scheme. We say* DS *is δ_{DS}-correct, if for every key pair* $(\mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{DS.KGen}$ *and every message $m \in \mathcal{M}$, the probability below is bounded by δ_{DS}:*

$$\Pr\left[\mathsf{false} = \mathsf{DS.Vrfy}(\mathsf{vk}, m, \sigma) \,|\, \sigma \xleftarrow{\$} \mathsf{DS.Sign}(\mathsf{sk}, m)\right] \leq \delta_{\mathsf{DS}}$$

Definition 17. *We say a* DS *is $\epsilon^{\text{euf-cma}}_{\mathsf{DS}}$-*EUF-CMA *secure, if the below defined advantage of any attacker $\mathcal{A}$ against $\mathbf{G}^{\text{EUF-CMA}}_{\mathsf{DS}}$ game in Figure 13 is bounded by,*

$$\mathsf{Adv}^{\text{EUF-CMA}}_{\mathsf{DS}} := \Pr[\mathbf{G}^{\text{EUF-CMA}}_{\mathsf{DS}}(\mathcal{A}) = 1] \leq \epsilon^{\text{euf-cma}}_{\mathsf{DS}}$$

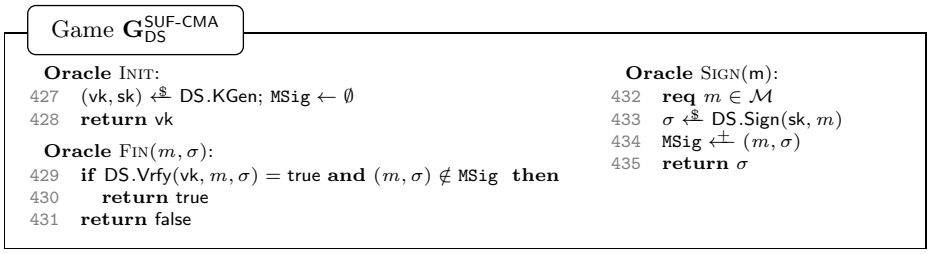


Fig. 14. The SUF-CMA security game for a DS scheme.

Definition 18. *We say a* DS *is* $\epsilon^{\mathsf{suf\text{-}cma}}_{\mathsf{DS}}$-SUF-CMA *secure, if the below defined advantage of any attacker* $\mathcal{A}$ *against* $\mathbf{G}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}$ *game in Figure 14 is bounded by,*

$$\mathsf{Adv}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}} := \Pr[\mathbf{G}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{A}) = 1] \leq \epsilon^{\mathsf{suf\text{-}cma}}_{\mathsf{DS}}$$

A.4 Public Key Encryption

Definition 19. *Let* $\mathcal{M}$ *denote the space of messages. A* public key encryption *(PKE) scheme* $\mathsf{PKE} = (\mathsf{PKE.KGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *is a tuple of algorithms, where*

Key Generation $(\mathsf{ek}, \mathsf{dk}) \xleftarrow{\$} \mathsf{PKE.KGen}$ *outputs a public encryption key* ek *and a private decryption key* dk*.*

Encryption $\mathsf{c} \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{ek}, m)$ *inputs a public encryption key* ek *and a message* m*, and a ciphertext* c*.*

Decryption $m \leftarrow \mathsf{PKE.Dec}(\mathsf{dk}, \mathsf{c})$ *inputs a private decryption key* dk *and a ciphertext* c*, and deterministically outputs a message* $m \in \mathcal{M}$*.*

Definition 20 (Correctness of PKE). *Let* $\mathsf{PKE} = (\mathsf{PKE.KGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *be a PKE scheme. We say* PKE *is* δ_{PKE}*-correct, if for every key pair* $(\mathsf{ek}, \mathsf{dk}) \xleftarrow{\$} \mathsf{PKE.KGen}$ *and every* $m \in \mathcal{M}$*, the probability below is bounded by* δ_{PKE}*:*

$$\Pr\left[m \neq \mathsf{PKE.Dec}(\mathsf{dk}, \mathsf{c}) \,|\, \mathsf{c} \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{ek}, m)\right] \leq \delta_{\mathsf{PKE}}$$

Definition 21. *We say a* PKE *is* $\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{PKE}}$-IND-CCA *secure, if the below defined advantage of any attacker* $\mathcal{A}$ *against* $\mathbf{G}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}}$ *game in Figure 15 is bounded by,*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}} := \left|\Pr[\mathbf{G}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}}(\mathcal{A}) = 1] - \frac{1}{2}\right| \leq \epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{PKE}}$$

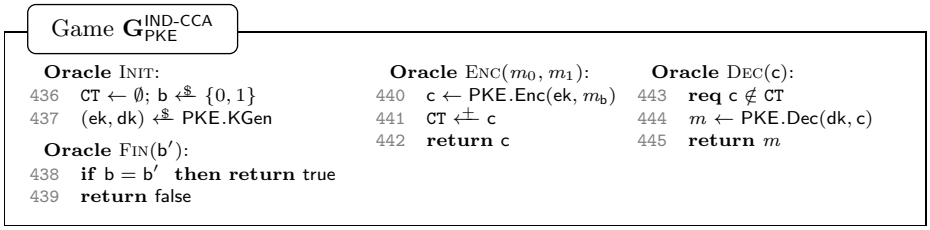


Fig. 15. The IND-CCA security game for a PKE scheme.

A.5 Hash Functions and Pseudorandom Functions

Definition 22. *Let* $\mathsf{KDF} : \mathcal{K} \times \mathcal{M} \to \mathcal{O}$ *be a function that maps a key* $\mathsf{k} \in \mathcal{K}$ *and a string* $\mathsf{m} \in \mathcal{M}$ *to an output* $y \in \mathcal{O}$*. Let* $\mathsf{RF} : \mathcal{K} \to \mathcal{O}$ *be any truly random function. We say* KDF *is* $\epsilon^{\mathsf{prf}}_{\mathsf{KDF}}$*-*PRF*-secure if the below defined advantage of any attacker* $\mathcal{A}$ *is bounded by*

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF}}(\mathcal{A}) := \left| \Pr[\mathcal{A}^{\mathsf{KDF}(\mathsf{k},\cdot)} = 1 \middle| \mathsf{k} \xleftarrow{\$} \mathcal{K}] - \Pr[\mathcal{A}^{\mathsf{RF}(\cdot)} = 1] \right| \leq \epsilon^{\mathsf{prf}}_{\mathsf{KDF}}$$

A.6 lr-prf-ODH Assumption

We recall the generic lr-prf-ODH definition in [19].

Definition 23. *Let* ECDH *denote a cyclic group* G *with order* q *over an elliptic curve* EC *with a generator* g*. Let* $\mathsf{H} : \mathbb{G} \times \{0,1\}^\star \to \{0,1\}^{l_\mathsf{H}}$ *denote a function. We define a generic security notion* lr-prf-ODH *which is parameterized by* $\mathsf{l}, \mathsf{r} \in \{\mathsf{n}, \mathsf{s}, \mathsf{m}\}$ *indicating how often the attacker is allowed to query a certain "left" resp. "right" oracle (*ODH_u *resp.* ODH_v*) where* n *indicates that no query is allowed,* s *that a single query is allowed, and* m *that multiple (polynomially many) queries are allowed to the respective side. Consider the following security game* $\mathbf{G}^{\mathsf{lr\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{H}}$ *between a challenger* $\mathcal{C}$ *and a* PPT *attacker* $\mathcal{A}$*.*

1. *The challenger* $\mathcal{C}$ *samples* $u \xleftarrow{\$} \mathbb{Z}_q$ *and provides* $\mathbb{G}$*,* g*, and* g^u *to the attacker* $\mathcal{A}$*.*
2. *If* $\mathsf{l} = \mathsf{m}$*,* $\mathcal{A}$ *can issue arbitrarily many queries to the following oracle* ODH_u*.*
   ODH_u **oracle.** *On a query of the form* (S, x)*,* $\mathcal{C}$ *first checks if* $S \notin \mathbb{G}$ *and returns* $\perp$ *if this is the case. Otherwise, it computes* $y \leftarrow \mathsf{H}(S^u, x)$ *and returns* y*.*
3. *Eventually,* $\mathcal{A}$ *issues a challenge query* $x^\star$*. On this query,* $\mathcal{C}$ *samples* $v \xleftarrow{\$} \mathbb{Z}_q$ *and a bit* $\mathsf{b} \xleftarrow{\$} \{0,1\}$ *uniformly at random. It then computes* $y^\star_0 = \mathsf{H}(g^{uv}, x^\star)$ *and samples* $y^\star_q \xleftarrow{\$} \{0,1\}^{l_\mathsf{H}}$ *uniformly at random. The challenger returns* $(g^v, y^\star_\mathsf{b})$ *to* $\mathcal{A}$*.*
4. *Next,* $\mathcal{A}$ *may issue (arbitrarily interleaved) queries to the following oracles* ODH_u *and* ODH_v *(depending on* l *and* r*).*
   ODH_u **oracle.** *The attacker* $\mathcal{A}$ *may ask no (*$\mathsf{l} = \mathsf{n}$*), a single (*$\mathsf{l} = \mathsf{s}$*), or arbitrarily many (*$\mathsf{l} = \mathsf{m}$*) queries to this oracle. On a query of the form* (S, x)*, the challenger first checks if* $S \notin G$ *or* $(S, x) = (g^v, x^\star)$ *and returns* $\perp$ *if this is the case. Otherwise, it computes* $y \leftarrow \mathsf{H}(S^u, x)$ *and returns* y*.*

ODH_v **oracle.** *The attacker $\mathcal{A}$ may ask no ($\mathsf{r} = \mathsf{n}$), a single ($\mathsf{r} = \mathsf{s}$), or arbitrarily many ($\mathsf{r} = \mathsf{m}$) queries to this oracle. On a query of the form (T, x), the challenger first checks if $T \notin G$ or $(T, x) = (g^u, x^\star)$ and returns $\perp$ if this is the case. Otherwise, it computes $y \leftarrow \mathsf{H}(T^v, x)$ and returns y.*

5. *At some point, $\mathcal{A}$ stops and outputs a guess $\mathsf{b}' \in \{0, 1\}$.*

We say that the attacker wins the lr-prf-ODH *game if $\mathsf{b}' = \mathsf{b}$. We say the* lr-prf-ODH *problem is ϵ-hard over* ECDH *and* H*, if the advantage of any* PPT *attacker $\mathcal{A}$ that wins above $\mathbf{G}^{\mathsf{lr\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{H}}$ experiment is bounded by ϵ.*

In this paper, we only need the mn-prf-ODH assumption.

A.7 Password-Authenticated Key Exchange

The password-authenticated key exchange (PAKE) protocols allow two parties to establish a high-entropy key over an insecure channel using a shared low-entropy password. We first recall the definition of *two-round PAKE* in Definition 24 following [32,15].

Then, we simplify the BPR security model from [32], which is a "multi-test" variant of the original definition in [15], and introduce a weaker model, denoted wBPR, in Definition 29. Compared with the original "single-test" definition [15] and its "multi-test" variant BPR model in [32], our wBPR model (1) considers only one initiator-responder pair, (2) does not allow attackers to reveal the keys output by every session, and (3) does not allow attackers to corrupt passwords. Note that our wBPR model is also weaker than and therefore implied by the security model defined in [3,1,2]. Thus, some modern and widely used two-round PAKE schemes, including CPace [27] or SPAKE2 [4] that are respectively proven secure in [3] and [1,2], are also secure in our wBPR model.

To model parallel and sequential sessions of every party, we use π_P^{idx} to denote idx-th session state of a party P. When it causes no ambiguity, we also use π_I^{idx} (resp. π_R^{idx}) to denote idx-th session state of an initiator I (resp. a responder R).

Definition 24 (Two-round PAKE). *Let I be an initiator, R be a responder, $\mathcal{D}$ be an associated distribution with some entropy α, and pw be the password that is randomly sampled from $\mathcal{D}$ and is shared between I and R. A two-round PAKE protocol* $\mathsf{PAKE} = (\mathsf{PAKE.Init}, \mathsf{PAKE.Resp}, \mathsf{PAKE.TerInit})$ *consists of the following three algorithms:*

- ***Initialize*** $(\pi_I^{\mathsf{idx}}, \mathsf{m}^{\mathsf{init}}) \xleftarrow{\$} \mathsf{PAKE.Init}(I, R, pw)$*: allows an initiator I to input a responder R and a password pw, and to output its* idx*-th session state π_I^{idx} and a message $\mathsf{m}^{\mathsf{init}}$.*
- ***Response*** $(\pi_R^{\mathsf{idx}}, \mathsf{m}^{\mathsf{resp}}) \xleftarrow{\$} \mathsf{PAKE.Resp}(R, \mathsf{m}, pw)$*: allows a responder R to input an incoming message* m *and a password pw, and to output its* idx*-th session state π_R^{idx} and a message $\mathsf{m}^{\mathsf{resp}}$.*
- ***Terminate Initialization*** $\pi_I^{\mathsf{idx}} \leftarrow \mathsf{PAKE.TerInit}(\pi_I^{\mathsf{idx}}, \mathsf{m}, pw)$*: inputs an initiator I's* idx*-th session state π_I^{idx}, an incoming message* m*, and a password pw, and to output an updated session state π_I^{idx}.*

Definition 25 (Session States in PAKE). *Every session state π in a* PAKE *protocol has the following state variables:*

- $\pi.\mathsf{id}$*: the owner of the session state; specified at session state creation.*
- $\pi.\mathsf{idx}$*: the index of the session state; specified at session state creation.*
- $\pi.\mathsf{acc} \in \{\mathsf{true}, \mathsf{false}\}$*: indicating whether the session is accepted; initialized with* false.
- $\pi.\mathsf{tr}$*: the trace of the instance, i.e., the names of initiator and responder involved in the instance and the messages sent and received by $\pi.\mathsf{id}$ in the instance.*
- $\pi.\mathsf{key}$*: the session key of the session state; initialized with $\perp$.*

Definition 26 (Correctness of PAKE). *Let* $\mathsf{PAKE} = (\mathsf{PAKE.Init}, \mathsf{PAKE.Resp}, \mathsf{PAKE.TerInit})$ *be a two-round PAKE protocol associated with any distribution $\mathcal{D}$, I be any initiator, R be any responder, and pw be any password that is shared between I and R. We say* PAKE *is δ_{PAKE}-correct, if the following probability is bounded by δ_{PAKE}:*

$$\Pr\left[\begin{array}{c} \pi_I^{\mathsf{idx}}.\mathsf{acc} = \pi_R^{\mathsf{idx}'}.\mathsf{acc} = \mathsf{true} \\ \pi_I^{\mathsf{idx}}.\mathsf{key} = \pi_R^{\mathsf{idx}'}.\mathsf{key} \neq \perp \end{array} \middle| \begin{array}{c} (\pi_I^{\mathsf{idx}}, \mathsf{m}^{\mathsf{init}}) \xleftarrow{\$} \mathsf{PAKE.Init}(I, R, pw) \\ (\pi_R^{\mathsf{idx}'}, \mathsf{m}^{\mathsf{resp}}) \xleftarrow{\$} \mathsf{PAKE.Resp}(R, \mathsf{m}^{\mathsf{init}}, pw) \\ \pi_I^{\mathsf{idx}} \leftarrow \mathsf{PAKE.TerInit}(\pi_I^{\mathsf{idx}}, \mathsf{m}^{\mathsf{resp}}, pw) \end{array}\right] \leq \delta_{\mathsf{PAKE}}$$

Definition 27 (Partnered Sessions in PAKE [32]). *An initiator's session state π_I^{idx} and a responder's session state $\pi_R^{\mathsf{idx}'}$ are partnered if and only if*

$$\pi_I^{\mathsf{idx}}.\mathsf{acc} = \pi_R^{\mathsf{idx}'}.\mathsf{acc} = \mathsf{true} \ \textbf{\textit{and}} \ \pi_I^{\mathsf{idx}}.\mathsf{tr} = \pi_R^{\mathsf{idx}'}.\mathsf{tr} \neq \perp$$

Two initiators' session states are never partnered, neither are two responders' session states. We define a partnering helper function $*\mathsf{isPartnered}(\pi_I^{\mathsf{idx}}, \pi_R^{\mathsf{idx}'})$ which outputs true if and only if π_I^{idx} and $\pi_R^{\mathsf{idx}'}$ are partnered.

Definition 28 (Freshness of PAKE [32]). *A session state π_P^{idx} is fresh, where P is either an initiator or a responder, if and only if:*

- *It has more than one partnered session states, or*
- *The following conditions hold:*
  - *π_P^{idx} is accepted and was not queried to* Test *oracle before.*
  - *At least one of the following conditions holds:*
    1. *A unique fresh partnered session state exists.*
    2. *No partnered session state exists.*

We define a freshness helper function $*\mathsf{isFresh}(P, \mathsf{idx})$ which outputs true if and only if π_P^{idx} is fresh.

Definition 29 (wBPR Security). *Let* PAKE *be a two-round PAKE protocol that is associated with a dictionary $\mathcal{D}$ with entropy α. We say* PAKE *is $\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha}$-*wBPR *secure if the below defined advantage of any attacker $\mathcal{A}$ against $\mathbf{G}^{\mathsf{wBPR}}_{\mathsf{PAKE}}$ game in Figure 16 is bounded by*

$$\mathsf{Adv}^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha}(\mathcal{A}) := \left| \Pr[\mathbf{G}^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha}(\mathcal{A}) = 1] - \frac{1}{2} \right| \leq \epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha}$$

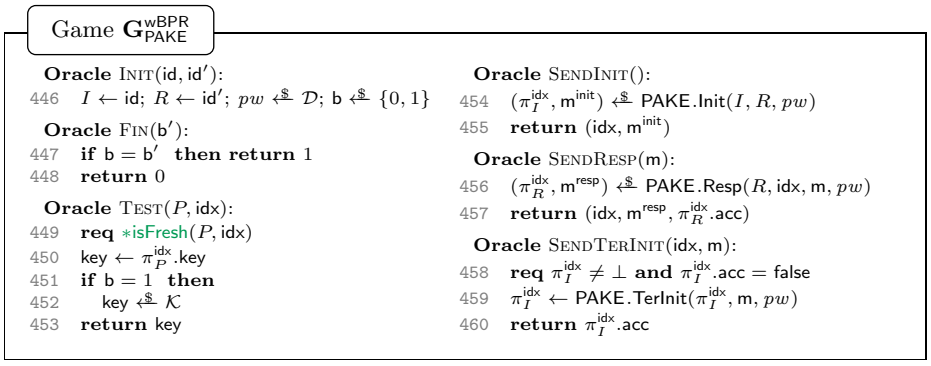


Fig. 16. The weak BPR wBPR game for a two-round PAKE scheme.

B Correctness Analysis for CGKD

B.1 Detailed Correctness Model for CGKD

We now present the correctness game for a CGKD protocol in Figures 17 and 18.

The correctness game between a challenger and an attacker $\mathcal{A}$ starts with a query from $\mathcal{A}$ to the Init oracle, where the challenger initializes a flag win with false, as well as six global dictionaries and one global set: (1) GS stores the values of each group secret, (2) MSiup stores the sign-up messages of each party, (3) MReg stores the register messages of each session, (4) AP stores the authorized parties of each group, (5) Ros stores the latest rosters of each group, (6) ToAdd stores the participants of each group that have completed joining but have not been announced, and (7) `actGrp` stores the activated groups. The game ends with a query to the Fin oracle that simply returns the flag win.

During the game, the attacker $\mathcal{A}$ has access to the following oracles.

NewParty(id)**:** inputs the identifier id of a party, whose long-term state has not been initialized. The challenger runs the SignUp algorithm to create a long-term state and a sign-up message for id, stores the sign-up message in the dictionary MSiup, and outputs both the long-term state and the sign-up message.

NewGroup(gid, $\mathcal{D}$)**:** inputs the identifier gid of a group, whose secret has not been initialized, and a distribution $\mathcal{D}$. The challenger samples a group secret gs following the distribution $\mathcal{D}$, stores it into the dictionary GS, and outputs gs.

Authorize(gid, id)**:** inputs the identifier gid of a group, whose secret has been set, and the identifier id of a party, whose sign-up message has been generated. The challenger simply adds the party identifier into the set AP[gid].

InitGroup(gid, id)**:** inputs the identifier gid of a group, which has not yet been activated, and the identifier id of a party authorized for gid. The challenger generates a new session state and a registration message by running the Register-L algorithm. Then, the challenger stores the registration message into the dictionary MReg, adds the session's record to Ros[gid], and adds the group identifier

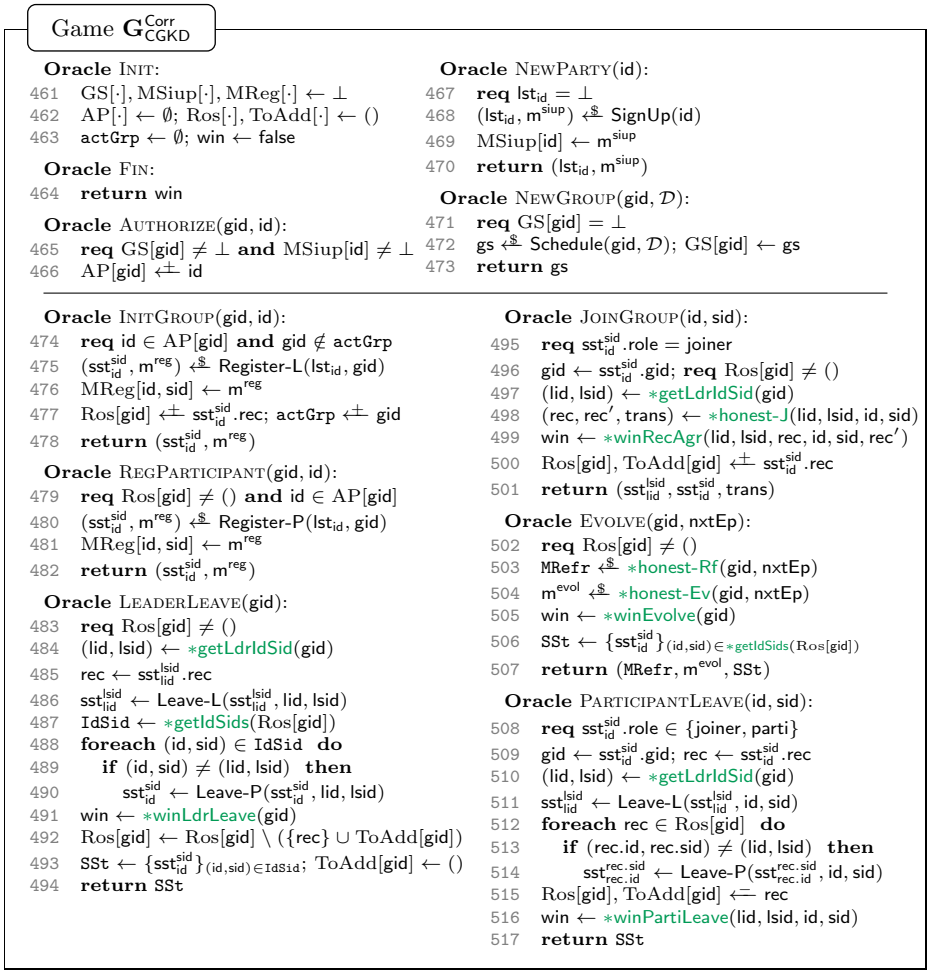


Fig. 17. Correctness game for a CGKD protocol.

gid to actGrp. Finally, the challenger outputs both the generated session state and registration message.

RegParticipant(gid, id)**:** inputs the identifier gid of a group, whose roster is non-empty, and the identifier id of a party authorized for gid. The challenger generates a new session state and a registration message by running the Register-P algorithm. Then, the challenger stores the registration message in the dictionary MReg. Finally, the challenger outputs both the generated session state and registration message.

JoinGroup(id, sid)**:** inputs the identifiers id of a party and sid of its session, whose role is joiner. Let gid denote the group identifier of this session. The global roster corresponding to gid must be non-empty. The challenger runs an honest

Game $\mathbf{G}^{\mathsf{Corr}}_{\mathsf{CGKD}}$ - Helper Functions

Helper $*$getLdrIdSid(gid):
518 **req** $\mathrm{Ros}[\mathsf{gid}] \neq ()$
519 $\mathsf{rec} \leftarrow *\mathsf{leader}(\mathrm{Ros}[\mathsf{gid}])$
520 **return** $(\mathsf{rec.id}, \mathsf{rec.sid})$

Helper $*$getIdSids(Target):
521 $\mathtt{IdSid} \leftarrow \emptyset$
522 **foreach** $\mathsf{rec} \in \mathtt{Target}$ **do**
523 $\quad \mathtt{IdSid} \xleftarrow{\cup} (\mathsf{rec.id}, \mathsf{rec.sid})$
524 **return** IdSid

Helper $*$honest-J(lid, lsid, id, sid):
525 $\mathsf{m}^{\mathsf{join}} \leftarrow (\mathrm{MSiup}[\mathsf{lid}], \mathrm{MReg}[\mathsf{lid}, \mathsf{lsid}])$
526 $\mathsf{trans} \leftarrow ()$; $\mathsf{gs} \leftarrow \mathrm{GS}[\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{gid}]$
527 $\mathsf{rec}_{\mathsf{leader}}, \mathsf{rec}_{\mathsf{joiner}} \leftarrow \mathbf{0}$
528 **while** $\mathsf{m}^{\mathsf{join}} \neq \mathbf{0}$ **do**
529 $\quad (\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m}^{\mathsf{join}}, \mathsf{rec}) \xleftarrow{\$} \text{Join-P}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{gs}, \mathsf{m}^{\mathsf{join}})$
530 $\quad \mathsf{trans} \xleftarrow{+} \mathsf{m}^{\mathsf{join}}$; $\mathsf{rec}_{\mathsf{leader}} \leftarrow \mathsf{rec}$
531 $\quad$ **if** $\mathsf{m}^{\mathsf{join}} = \mathbf{0}$ **then break**
532 $\quad (\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}, \mathsf{m}^{\mathsf{join}}, \mathsf{rec}) \xleftarrow{\$} \text{Join-L}(\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}, \mathsf{gs}, \mathsf{m}^{\mathsf{join}})$
533 $\quad \mathsf{trans} \xleftarrow{+} \mathsf{m}^{\mathsf{join}}$; $\mathsf{rec}_{\mathsf{joiner}} \leftarrow \mathsf{rec}$
534 **return** $(\mathsf{rec}_{\mathsf{leader}}, \mathsf{rec}_{\mathsf{joiner}}, \mathsf{trans})$

Helper $*$honest-Rf(gid, nxtEp):
535 **if** $\mathsf{nxtEp} = \mathsf{false}$ **then return** $\emptyset$
536 $(\mathsf{lid}, \mathsf{lsid}) \leftarrow *\mathsf{getLdrIdSid}(\mathsf{gid})$; $\mathtt{MRefr} \leftarrow \emptyset$
537 **foreach** $\mathsf{rec} \in \mathrm{Ros}[\mathsf{gid}]$ **do**
538 $\quad \mathsf{id} \leftarrow \mathsf{rec.id}$; $\mathsf{sid} \leftarrow \mathsf{rec.sid}$
539 $\quad$ **if** $(\mathsf{id}, \mathsf{sid}) \neq (\mathsf{lid}, \mathsf{lsid})$ **then**
540 $\quad\quad (\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m}^{\mathsf{refr}}, *) \xleftarrow{\$} \text{Refresh-P}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}})$
541 $\quad\quad (\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}, *) \xleftarrow{\$} \text{Refresh-L}(\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}, \mathsf{m}^{\mathsf{refr}})$
542 $\quad\quad \mathtt{MRefr} \xleftarrow{+} \mathsf{m}^{\mathsf{refr}}$
543 $\quad\quad \mathrm{Ros}[\mathsf{gid}], \mathrm{ToAdd}[\mathsf{gid}] \xleftarrow{=/+} (\mathsf{rec}, \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{rec})$
544 **return** MRefr

Helper $*$honest-Ev(gid, nxtEp):
545 $(\mathsf{lid}, \mathsf{lsid}) \leftarrow *\mathsf{getLdrIdSid}(\mathsf{gid})$
546 $\mathsf{rec} \leftarrow \mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{rec}$; $\mathrm{ToAdd}[\mathsf{gid}] \leftarrow ()$
547 $(\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}, \mathsf{m}^{\mathsf{evol}}) \xleftarrow{\$} \text{Evolve-L}(\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}, \mathsf{nxtEp})$
548 **if** $\mathsf{nxtEp} = \mathsf{true}$ **then**
549 $\quad \mathrm{Ros}[\mathsf{gid}] \xleftarrow{=/+} (\mathsf{rec}, \mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{rec})$
550 $\mathtt{IdSid} \leftarrow *\mathsf{getIdSids}(\mathrm{Ros}[\mathsf{gid}])$
551 **foreach** $(\mathsf{id}, \mathsf{sid}) \in \mathtt{IdSid} \setminus \{(\mathsf{lid}, \mathsf{lsid})\}$ **do**
552 $\quad (\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, *) \xleftarrow{\$} \text{Evolve-P}(\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}, \mathsf{m}^{\mathsf{evol}})$
553 **return** $\mathsf{m}^{\mathsf{evol}}$

Helper $*$winRecAgr(lid, lsid, rec, id, sid, rec′):
554 **if** $\mathsf{rec} = \perp$ **or** $\mathsf{rec}' = \perp$ **then**
555 $\quad$ **return** true
556 **if** $\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{rec} \neq \mathsf{rec}$ **then return** true
557 **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{rec} \neq \mathsf{rec}'$ **then return** true
558 **if** $\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{gid} \neq \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{gid}$ **then**
559 $\quad$ **return** true
560 **if** $\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{role} \neq \mathsf{leader}$ **then return** true
561 **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{role} \neq \mathsf{parti}$ **then return** true
562 **return** false

Helper $*$winEvolve(gid):
563 $(\mathsf{lid}, \mathsf{lsid}) \leftarrow *\mathsf{getLdrIdSid}(\mathsf{gid})$
564 $\mathtt{IdSid} \leftarrow *\mathsf{getIdSids}(\mathrm{Ros}[\mathsf{gid}])$
565 **if** $\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{ros} \neq \mathrm{Ros}[\mathsf{gid}]$ **then**
566 $\quad$ **return** true
567 **foreach** $(\mathsf{id}, \mathsf{sid}) \in \mathtt{IdSid}$ **do**
568 $\quad$ **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep} \neq \mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{ep}$ **then**
569 $\quad\quad$ **return** true
570 $\quad$ **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{stg} \neq \mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{stg}$ **then**
571 $\quad$ **return** true
572 $\quad$ **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{gk} \neq \mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{gk}$ **then**
573 $\quad\quad$ **return** true
574 $\quad$ **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros} \neq \mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{ros}$ **then**
575 $\quad\quad$ **return** true
576 **return** false

Helper $*$winLdrLeave(gid):
577 $\mathsf{ros} \leftarrow \mathrm{Ros}[\mathsf{gid}]$
578 $(\mathsf{lid}, \mathsf{lsid}) \leftarrow *\mathsf{getLdrIdSid}(\mathsf{ros})$
579 **if** $\mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}} \neq \perp$ **then return** true
580 **foreach** $\mathsf{rec} \in \mathrm{ToAdd}[\mathsf{gid}]$ **do**
581 $\quad \mathsf{id} \leftarrow \mathsf{rec.id}$; $\mathsf{sid} \leftarrow \mathsf{rec.sid}$
582 $\quad$ **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}} \neq \perp$ **then return** true
583 $\mathsf{ros} \xleftarrow{-} *\mathsf{leader}(\mathsf{ros})$
584 **foreach** $(\mathsf{id}, \mathsf{sid}) \in *\mathsf{getIdSids}(\mathsf{ros})$ **do**
585 $\quad$ **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ros} \neq \mathsf{ros}$ **then return** true
586 **return** false

Helper $*$winPartiLeave(lid, lsid, id, sid):
587 **if** $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}} \neq \perp$ **then return** true
588 $\mathsf{ros} \leftarrow \mathsf{sst}^{\mathsf{lsid}}_{\mathsf{lid}}.\mathsf{ros}$
589 **if** $(\mathsf{id}, \mathsf{sid}) \in *\mathsf{getIdSids}(\mathsf{ros})$ **then**
590 $\quad$ **return** true
591 **return** false

Fig. 18. Helper functions in the correctness game for a CGKD protocol.

interaction between the joiner's session and the leader's session, after which the win flag is evaluated by the helper function $*$winRecAgr. Then, the challenger adds the session's record into the set Ros[gid] and ToAdd[gid]. Finally, the challenger outputs the session states of both the leader and the joiner, along with the transcript of the joining interaction.

Evolve(gid, nxtEp)**:** inputs the identifier gid of a non-empty group, and a flag nxtEp indicating whether the epoch should be advanced. If nxtEp = true, the challenger runs an honest refresh phase interaction between the leader and

all participants of the group gid and obtains a set of refresh messages MRefr, followed by replacing old records of every participants in Ros[gid] and ToAdd[gid] with their new records. Otherwise, the challenger sets MRefr to an empty set. Second, the challenger runs an honest evolve phase interaction between the leader and all participant of the group gid and obtains an evolve message $\mathsf{m}^{\mathsf{evol}}$. If nxtEp = true, the challenger additionally replaces the record of the leader in Ros[gid] with its new one. Afterwards, the challenger resets the ToAdd[gid] to an empty list, which captures that the leader announces all newly joined participants to all group members.
Finally, the challenger evaluates the flag win by the helper function $*$winEvolve, and outputs the set of refresh message, the evolve message, and the session states of all parties in the group gid.

LeaderLeave(gid)**:** inputs the identifier gid of a non-empty group, whose leader is leaving. Let lid and lsid respectively denote the identifier of the leader and its session. The challenger runs Leave-L and Leave-P algorithms for all group members to remove the leader lid's session lsid. Then, the challenger evaluates the flag win by the helper function $*$winLdrLeave. Afterwards, the challenger removes both the leader's record and all records in ToAdd[gid] from the roster Ros[gid] and resets ToAdd[gid] to an empty list, which captures that all participants who have completed the joining phase with the leader but have not yet been announced to the group cannot be acknowledged by other participants. Finally, the challenger outputs the session states of parties in the group gid.

ParticipantLeave(id, sid)**:** inputs the identifiers (id, sid) of a party and its session that is to be removed from the group. Let gid denote the group identifier of this session. The challenger respectively runs Leave-P and Leave-L algorithms for all participant's session in this group and the leader's session state to remove the party id's session sid. Afterwards, the challenger removes the record of the session specified by (id, sid) from both Ros[gid] and ToAdd[gid]. Finally, the challenger evaluates the flag win by the helper function $*$winPartiLeave and outputs the party id's session state sid and its leader's session state.

Correctness Properties. Our correctness game captures the following properties:

- **Record Agreement During Join Phase:** When a participant completes the join-phase communication with the group leader, they must agree on the group identifier as well as their respective records and roles. This property ensures that the group leader and the newly joined participant correctly exchange the information necessary for subsequent group key distribution. This property is captured by the helper function $*$winRecAgr.
- **Correct Stage Evolution:** When a leader advances the group to the next stage (optionally to the next epoch), it must correctly update the roster, and all group participants must agree with the leader on the updated epoch, stage, group key, and roster. This property ensures that every group member agrees on a consistent view of the system state during stage evolution, including any updates resulting from the refresh phase. This property is captured by the helper function $*$winRecAgr.

– **Correct Leader Leave:** When a leader leaves the group, the leader's session state and the session states of all newly joined participants that have not yet been announced to the group must be erased. In addition, all remaining participants must agree on the updated roster. This property ensures that the leaving leader's session and any unannounced joining sessions cannot participant in future group communication, and that the remaining participants agree on the next leader as determined by the roster order. This property is captured by the helper function $*\mathsf{winLdrLeave}$.
– **Correct Participant Leave:** When a participant leaves the group, its session state must be erased and its record must be removed from the leader's roster. This property ensures that the leaving participant's session cannot participant in future group communication. This property is captured by the helper function $*\mathsf{winPartiLeave}$.

Definition 30 (Correctness of CGKD). *Let* CGKD *denote a continuous group key distribution protocol. We say* CGKD *is* δ_{CGKD}*-correct if the below defined advatange of any attacker* $\mathcal{A}$ *in the game* $\mathbf{G}^{\mathsf{Corr}}_{\mathsf{CGKD}}$ *in Figure 17 is bounded by*

$$\mathsf{Adv}^{\mathsf{Corr}}_{\mathsf{CGKD}}(\mathcal{A}) := \Pr[\mathbf{G}^{\mathsf{Corr}}_{\mathsf{CGKD}}(\mathcal{A}) = \mathsf{true}] \leq \delta_{\mathsf{CGKD}}$$

B.2 Correctness of Our CGKD Construction

Theorem 8. *Let* CGKD *denote the CGKD construction in Figure 8. If the underlying* DS *is* δ_{DS}*-correct,* maKEM *is* δ_{maKEM}*-correct,* MAC *is* δ_{MAC}*-correct, and* PAKE *is* δ_{PAKE}*-correct, then* CGKD *is* δ_{CGKD}*-correct in Definition 30, where*

$$\begin{aligned}\delta_{\mathsf{CGKD}} \leq & (q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 \cdot 2^{-\lambda} + q_{\mathrm{JG}}\delta_{\mathsf{PAKE}} + q_{\mathrm{JG}}\delta_{\mathsf{MAC}} + q_{\mathrm{E}}\delta_{\mathsf{maKEM}} \\ & + (q_{\mathrm{E}}q_{\mathrm{JG}} + q_{\mathrm{E}} + q_{\mathrm{JG}})\delta_{\mathsf{DS}}\end{aligned}$$

Here, q_{IG}*,* q_{RegP}*,* q_{JG}*, and* q_{E} *respectively denotes the number of queries to* InitGroup*,* RegParticipant*,* JoinGroup*, and* Evolve *oracles.*

C Detailed Description for Our CGKD Construction

Following the syntax in Definition 6, our CGKD construction makes use of a two-round PAKE (PAKE) following the syntax in [32], a maKEM (maKEM) in Section 3, a digital signature (DS), a message authentication code (MAC), and a key derivation function (KDF), and consists of following six algorithms below.

$\mathsf{SignUp}(\mathsf{id})$**:** inputs the identifier id of a party and initializes its long-term state lst. Then, this party generates a DS public-secret key pair $(\mathsf{vk}, \mathsf{sk})$, uses vk as the identity public key and sk as the identity secret key, and sets the outgoing sign-up message $\mathsf{m}^{\mathsf{siup}}$ to id.
$\mathsf{Schedule}(\mathsf{gid}, \mathcal{D})$**:** inputs a group identifier gid and a group secret distribution $\mathcal{D}$. The group secret gs is sampled following the distribution $\mathcal{D}$ and finally output.

Register-L($\mathsf{lst}, \mathsf{gid}$): inputs a long-term state lst and a group identifier gid. First, the leader creates a new session state in the following four steps: (1) sample a session identifier sid uniformly at random, (2) setting the role of the session state to leader, (3) generate a maKEM key pair $(\mathsf{aek}, \mathsf{adk})$ and a DS key pair $(\mathsf{vk}, \mathsf{sk})$, (4) set $(\mathsf{aek}, \mathsf{vk})$ as the session public key and their secret keys as session secret key. Then, the leader initializes more session state variables in the following four steps: (1) sample a new group key uniformly at random, (2) set both the epoch and stage to 1, (3) generate a new record using the identifiers and public keys in long-term and session states, and (4) set the local roster to a list that only contains the new record. The outputs consists of the created session state and a registration message that equals to the sampled session identifier.

Register-P($\mathsf{lst}, \mathsf{gid}$): inputs a long-term state lst and a group identifier gid. The participant creates a new session state, similar to the four steps in Register-L algorithm, except for setting the role of the session state to joiner. The outputs consists of the created session state and a registration message that equals to the sampled session identifier.

Join-L($\mathsf{sst}, \mathsf{gs}, \mathsf{m}$): The leader's joining algorithm inputs a session state sst, whose role is leader, a group secret gs, and an incoming message m. The leader parses the incoming message into a five-element tuple: a party identifier id', a session identifier sid', a record rec, a PAKE message $\mathsf{m}^{\mathsf{PAKE}}$, and a DS signature σ. The outgoing message $\mathsf{m}^{\mathsf{join}}$ is initialized with an empty string. The parsed party and session identifiers $(\mathsf{id}', \mathsf{sid}')$ must not conflict with any identifier pairs specified by any record included in the local roster. Then, the leader performs different actions according to whether $\mathsf{sst}.\mathsf{Temp}$ has been initialized on the identifiers $(\mathsf{id}', \mathsf{sid}')$.

(a) If $\mathsf{sst}.\mathsf{Temp}[\mathsf{id}', \mathsf{sid}']$ has not been initialized, then the parsed record rec and the DS signature σ must be empty strings, while the PAKE message $\mathsf{m}^{\mathsf{PAKE}}$ must not. Next, the leader runs $\mathsf{PAKE}.\mathsf{Resp}$ algorithm on a responder identifier, which is the concatenation of its group identifier and role, the incoming PAKE message $\mathsf{m}^{\mathsf{PAKE}}$, and the group secret gs, and obtains a PAKE state π^{pake} and a PAKE message $\mathsf{m}^{\mathsf{pake}}_{\mathsf{out}}$. The PAKE state π^{pake} must successfully terminate and derive a symmetric PAKE key. Then, the leader uses KDF and this PAKE key to derive a MAC key and a DS randomness, which is further used to derive a DS public key by using DS key generation algorithm. Afterwards, the leader stores the DS public key at $\mathsf{sst}.\mathsf{Temp}[\mathsf{id}', \mathsf{sid}']$ and signs its record and current stage by using MAC for a tag. The outgoing message $\mathsf{m}^{\mathsf{join}}$ consists of the PAKE message $\mathsf{m}^{\mathsf{pake}}_{\mathsf{out}}$, the leader's session record and stage, and the computed MAC tag.

(b) If $\mathsf{sst}.\mathsf{Temp}[\mathsf{id}', \mathsf{sid}']$ has been initialized by a DS public key, then parsed PAKE message $\mathsf{m}^{\mathsf{PAKE}}$ must be an empty string, while the record rec and the DS signature σ must not. The leader checks (1) whether the parsed record specifies the party id', the session sid', and the leader's current epoch $\mathsf{sst}.\mathsf{ep}$, and (2) whether the parsed record and DS signature can pass the verification by using the stored DS public key. Finally, the leader adds the parsed record into its local roster and erases $\mathsf{sst}.\mathsf{Temp}[\mathsf{id}', \mathsf{sid}']$.

If any above check fails, the leader aborts the execution and rolls back its session state. The output consists of the leader's updated session state, the message $\mathsf{m}^{\mathsf{join}}$, and the parsed record rec.

$\mathsf{Join\text{-}P}(\mathsf{sst}, \mathsf{gs}, \mathsf{m})$**:** inputs a session state sst, whose role is joiner, a group secret gs, and an incoming message m. The joiner initializes the outgoing message $\mathsf{m}^{\mathsf{join}}$ and record rec to empty strings, and performs different actions according to whether $\mathsf{sst.temp}$ has been initialized.

(a) If $\mathsf{sst.temp}$ has not been initialized, then the joiner parses the incoming message to a party and session identifier pair $(\mathsf{id}', \mathsf{sid}')$ that is district from its own pair. Next, the joiner runs $\mathsf{PAKE.Init}$ algorithm on an initiator identifier, which is the concatenation of its group identifier and its role, and a responder identifier, which is the concatenation of its group identifier and the leader role, and the group secret gs, and obtains a PAKE state π^{pake} and a PAKE message $\mathsf{m}^{\mathsf{pake}}_{\mathsf{out}}$. Finally, the joiner stores the parsed identifier pair and the derived PAKE state to $\mathsf{sst.temp}$ and updates the outgoing message $\mathsf{m}^{\mathsf{join}}$ to its party identifier, session identifier, an empty string, the PAKE message, and an empty string.

(b) If $\mathsf{sst.temp}$ has been initialized by a tuple of party identifier id', session identifier sid', and a PAKE state π^{pake}, the joiner parses the incoming message to a PAKE message $\mathsf{m}^{\mathsf{PAKE}}$, a record rec, a stage stg, and a MAC tag tag. Next, the joiner checks whether the stored party and session identifiers $(\mathsf{id}', \mathsf{sid}')$ equals to the one specified by the record rec. Then, the joiner runs $\mathsf{PAKE.TerInit}$ algorithm on the PAKE state π^{pake}, PAKE message $\mathsf{m}^{\mathsf{PAKE}}$, and group secret gs. The PAKE state π^{pake} must successfully terminate and derive a symmetric PAKE key. Afterwards, the joiner uses KDF and the PAKE key to derive a MAC key and a DS randomness, which is further used to derive a DS secret key by using DS key generation algorithm. The joiner checks whether the incoming record rec, stage stg, and tag tag can pass the MAC verification by using the derived MAC key. The joiner (1) locally stores both the epoch included in the incoming record and the incoming stage, (2) derives its own record, and (3) signs its own record by using the derived DS secret key for a DS signature. The outgoing message $\mathsf{m}^{\mathsf{join}}$ is updated to the tuple consisting of the joiner's party identifier, session identifier, local record, an empty string, and the computed DS signature. Finally, the joiner sets its roster as a list that only includes the incoming record, erases its $\mathsf{sst.temp}$, and sets its role to parti.

The output consists of the participant's updated session state, the message $\mathsf{m}^{\mathsf{join}}$, and the parsed record rec.

$\mathsf{Refresh\text{-}L}(\mathsf{sst}, \mathsf{m})$**:** inputs a session state sst, whose role is leader, and an incoming message m. The leader checks (1) whether the incoming message m can be parsed into a record rec and two DS signatures σ and σ', (2) whether a party identifier id, a session identifier sid, an epoch ep, and an identity public key ipk can be parsed from the record rec, (3) whether the parsed ep equals the leader session epoch plus one, (4) whether a record included in the leader's roster specifies the parsed party id and its session sid with the epoch $\mathsf{ep} - 1$, the same identity public key ipk, and some session public key spk. (5) whether

the incoming record can be verified on the identity public key ipk and the DS signature σ , and (6) whether the incoming record can be verified on the DS public key included in the spk and the DS signature σ'. If any above check fails, the leader aborts the execution and rolls back its session state. Otherwise, the leader adds the incoming record rec into its local next roster sst.nxtRos.

The outputs consists of the leader's updated session state and the incoming record rec.

Refresh-P(sst)**:** The participant's refresh algorithm inputs a session state sst, whose role is parti. The participant generates a new maKEM key pair and a DS key pair and respectively stores their public keys into the next session public key sst.nxtSpk and their secret keys into the next session secret key sst.nxtSsk. Next, the participant uses the next public-secret session key pairs to generate the next record, which is then signed twice, respectively by using the long-term secret key and the DS secret key included in the current session secret key. The outgoing message $\mathsf{m}^{\mathsf{refr}}$ consists of the next record and both DS signatures.

The outputs consists of the updated session state, the message $\mathsf{m}^{\mathsf{refr}}$, and the next record.

Evolve-L(sst, nxtEp)**:** inputs a session state sst, whose role is leader, and a flag nxtEp.

If the flag nxtEp = true, the leader refreshes its own sessions states, similar to the execution of participant's state refresh Refresh-P, and obtains a refresh message $\mathsf{m}^{\mathsf{refr}}$. Then, the leader immediately moves its next session public key, session secret key, record, and roster to (and therefore overwrites) the current ones. The next session public key, session secret key, and record are erased, and the next local roster is initialized to a list that only includes the leader's current record. The local epoch is incremented by 1 and the stage is reset to 0. Otherwise, the leader sets the message $\mathsf{m}^{\mathsf{refr}}$ to an empty string.

Afterwards, the leader increments its stage by 1, followed by extracting a maKEM secret key from its session secret key and a set of maKEM public keys from all records included in its local roster. Next, the leader runs maKEM.Encaps on the extracted maKEM secret key and the set of public keys to obtain a key k and a set of ciphertext CT. Then, the leader signs the current roster and stage by using the DS secret key included in the session secret key for a DS signature. Finally, the leader sets the message $\mathsf{m}^{\mathsf{evol}}$ to a tuple consists of the refresh message $\mathsf{m}^{\mathsf{refr}}$, its current roster, its current stage, the set of maKEM ciphertexts CT, and the DS signature, and sets the group key to the encapsulated key k .

The outputs consists of the leader's updated session state and the message $\mathsf{m}^{\mathsf{evol}}$.

Evolve-P(sst, m)**:** inputs a session state sst, whose role is parti, and an incoming message m. The participant parses the incoming message m to a tuple that consists of a refresh message $\mathsf{m}^{\mathsf{refr}}$, a roster, a stage, a set of maKEM ciphertexts CT, and a DS signature.

If the message $\mathsf{m}^{\mathsf{refr}}$ is not an empty string, similar to the leader's execution in the Refresh-L algorithm, the participant recovers a record rec. The recovered rec must be the leader record in the parsed roster and the participant's next record

must have been set. Then, similar to the leader's execution in the Evolve-L when $\mathsf{nxtEp} = \mathsf{true}$, the participant overwrites its session state variables for the current epoch, erases the session state variables for the next epoch, increments the epoch counter, and resets the stage counter, with the only except that the current roster is set to a list that only includes the recovered record. Finally, the participant sets the output flag nxtEp to true. Otherwise, the participant simply sets the output flag nxtEp to false.

Afterwards, the participant checks (1) whether its record is included in the input roster, (2) whether the input stage is greater than its current stage, (3) whether all records in the input roster share the same epoch as the participant, (4) whether the input roster and stage pass the DS verification on the leader's identity public key and the input signature, and (5) whether the input set of maKEM ciphertexts includes one ciphertext that can be decapsulated to a non-$\perp$ key by using the maKEM secret key included in its session secret key and the maKEM public key included in the leader's record. If any above check fails, the participant aborts the execution and rolls back its session state. Otherwise, the participant sets its roster to the input roster, its stage to the input stage, and its group key to the decapsulated key.

The outputs consists of the participant's updated session state and the flag nxtEp.

Leave-L$(\mathsf{sst}, \mathsf{id}', \mathsf{sid}')$: inputs a session state sst with role leader, a party identifier id', and a session identifier sid'. If the input identifier pair indicates the leader session itself, the leader simply erases its session state. Otherwise, the leader removes all records, which indicate party id''s session sid', from the local rosters for the current and the next epoch. Besides, the element in the local dictionary $\mathsf{sst}.\mathsf{Temp}$ on the index $(\mathsf{id}', \mathsf{sid}')$ is erased.

The output is the leader's updated session state.

Leave-P$(\mathsf{sst}, \mathsf{id}', \mathsf{sid}')$: inputs a session state sst, whose role is leader, a party identifier id', and a session identifier sid'. If the input identifier pair indicates the participant session itself, the participant simply erases its session state. If the input identifier pair indicates the leader's record in the local roster, the participant removes that record. Now, if the local roster is empty, the participant also erases its session state. If the participant's record is the leader record of the current local roster, the participant changes its role to leader.

The output is the participant's updated session state.

D Proof of Theorem 1

Proof. Let n denote the number of aeks included in the inputs of maKEM.Encaps algorithm. The encapsulation algorithm maKEM.Encaps computes ECDH key exchange between $(\mathsf{aek}^\star, \mathsf{adk}^\star) = (\mathsf{pk}^\star, \mathsf{sk}^\star)$ and every pair of ECDH key pairs $(\mathsf{aek}_i, \mathsf{adk}_i) = (\mathsf{pk}_i, \mathsf{sk}_i)$, where $i \in [n]$.

For every $i \in [n]$, we always have that $\mathsf{ECDH.Derive}(\mathsf{pk}^\star, \mathsf{adk}_i) = \mathsf{sdk}_i = \mathsf{ECDH.Derive}(\mathsf{aek}_i, \mathsf{sk}^\star)$, since the ECDH key exchange have perfect correctness. Furthermore, for every $i \in [n]$, we have that $\mathsf{AE.Dec}(\mathsf{sdk}_i, \mathsf{AE.Enc}(\mathsf{sdk}_i, k)) = k$ except with the probability at most δ_{AE}.

Thus, for every $i \in [n]$, the decapsulation is correct except with the probability at most δ_{AE}. Since the correctness of maKEM is measured for all n decapsulations, we have that

$$\delta_{\mathsf{maKEM}} \leq n\delta_{\mathsf{AE}}$$

E Proof of Theorem 2

Proof. We give the proof by a sequence of games. Let $\mathsf{Adv}_i(\mathcal{A}) = |\Pr[\mathbf{G}_i(\mathcal{A}) = \mathsf{true}] - \frac{1}{2}|$ denote the advantage of the attacker $\mathcal{A}$ in winning game $\mathbf{G}_i$.

Game $\mathbf{G}_0$. This game is identical to the $\mathsf{IND\text{-}CCA}$ security game of maKEM scheme in Figure 1. Thus, it holds that

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) = \mathsf{Adv}_0(\mathcal{A})$$

Game $\mathbf{G}_1$. This game is identical to $\mathbf{G}_0$ except that the challenger guesses the number of parties $n_{\mathsf{unCorrupt}}$ that will not be corrupted at the end of the game, and let $\mathcal{A}$ immediately loses if the guess is wrong.

Note that the attacker $\mathcal{A}$ can query NewKeyPair in total q_{NKP} times and therefore there are in total at most q_{NKP} parties that are not be corrupted. Thus, the challenger can guess correctly with probability at least $\frac{1}{q_{\mathrm{NKP}}}$ and we have that

$$\mathsf{Adv}_0(\mathcal{A}) \leq q_{\mathrm{NKP}}\mathsf{Adv}_1(\mathcal{A})$$

In particular, this game ensures that there are at most $n_{\mathsf{pair}} \leq n^2_{\mathsf{unCorrupt}} \leq q^2_{\mathrm{NKP}}$ pairs of parties, where neither of them is corrupted by the attacker $\mathcal{A}$. The number n_{pair} is used in the following games.

Game $\mathbf{G}_2$. This game is identical to $\mathbf{G}_1$ except that the challenger does the following modification:

- The challenger guesses one pair of indices of queries to the NewKeyPair oracle $(\mathsf{idx}_{(1,0)}, \mathsf{idx}_{(1,1)})$ and samples a random key $\widetilde{k^{\mathsf{AE}}}_1$.
- Whenever the attacker sends $\mathsf{idx}_{(1,j)}$-th query to the NewKeyPair oracle for $j \in \{0,1\}$, the challenger records the generated $(\mathsf{pk}_{(1,j)}, \mathsf{sk}_{(1,j)})$.
- Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}_{(1,0)}, \mathsf{sk}_{(1,1)}))$ and $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}_{(1,1)}, \mathsf{sk}_{(1,0)}))$, the challenger replaces them with $\widetilde{k^{\mathsf{AE}}}_1$.

We prove that $\mathbf{G}_1$ and $\mathbf{G}_2$ are indistinguishable by reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_1$ and $\mathbf{G}_2$, then we can construct an attacker $\mathcal{B}_1$ that solves the $\mathsf{mm\text{-}prf\text{-}ODH}$ problem over ECDH and KDF. The attacker $\mathcal{B}_1$ obtains $(g, U, V, y^\star)$ from its challenger, where g is the group generator and that there exists some unknown u and v such that $U = g^u$ and $V = g^v$. Then, $\mathcal{B}_1$ simulates $\mathbf{G}_1$ honestly to $\mathcal{A}$ except for the following modifications:

- $\mathcal{B}_1$ guesses one pair of indices of queries to the NewKeyPair oracle $(\mathsf{idx}_{(1,0)}, \mathsf{idx}_{(1,1)})$.

– Whenever the attacker sends $\mathsf{idx}_{(1,0)}$-th query to the NewKeyPair oracle, $\mathcal{B}_1$ returns U. Whenever the attacker sends $\mathsf{idx}_{(1,1)}$-th query to the NewKeyPair oracle, $\mathcal{B}_1$ returns V.
– Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(U, v))$ and to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(V, u))$, $\mathcal{B}_1$ replaces them with $y^\star$.
– Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{aek}, u))$, the $\mathcal{B}_1$ sends aek to its ODH_u oracle and uses the received reply to replace this KDF computation.
– Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{aek}, v))$, the $\mathcal{B}_1$ sends aek to its ODH_v oracle and uses the received reply to replace this KDF computation.

Assume that the parties that are input to the $\mathsf{idx}_{(1,0)}$-th and $\mathsf{idx}_{(1,1)}$)-th queries to the NewKeyPair oracle both are not corrupted, which happens with probability at least $\frac{1}{n_{\mathsf{pair}}}$

Note that $\mathcal{B}_1$ perfectly simulates $\mathbf{G}_1$ to $\mathcal{A}$ honestly if the challenge bit is 0 and $\mathbf{G}_2$ to $\mathcal{A}$ if the challenge bit is 1. If $\mathcal{A}$ can distinguish $\mathbf{G}_1$ and $\mathbf{G}_2$, then $\mathcal{B}_1$ can always solves the $\mathsf{mm\text{-}prf\text{-}ODH}$ problem over ECDH and KDF. Thus, we have that

$$\mathsf{Adv}_1(\mathcal{A}) \leq \mathsf{Adv}_2(\mathcal{A}) + n_{\mathsf{pair}} \epsilon^{\mathsf{mm\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{KDF}}$$

Game $\mathbf{G}_3$. This game is identical to $\mathbf{G}_2$ except that the challenger does the following modification:

– The challenger guesses two pairs of indices of queries to the NewKeyPair oracle $(\mathsf{idx}_{(i,0)}, \mathsf{idx}_{(i,1)})_{i\in\{1,2\}}$ and samples two random key $\{\widetilde{k^{\mathsf{AE}}}_i\}_{i\in\{1,2\}}$.
– Whenever the attacker sends $\mathsf{idx}_{(i,j)}$-th query to the NewKeyPair oracle for $i \in \{1, 2\}$ and $j \in \{0, 1\}$, the challenger records the generated $(\mathsf{pk}_{(i,j)}, \mathsf{sk}_{(i,j)})$.
– Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}_{(i,0)}, \mathsf{sk}_{(i,1)}))$ and $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}_{(i,1)}, \mathsf{sk}_{(i,0)}))$ for every $i \in \{1, 2\}$, the challenger replaces them with $\widetilde{k^{\mathsf{AE}}}_i$.

If the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_2$ and $\mathbf{G}_3$, then we can construct an attacker $\mathcal{B}_2$ that solves the $\mathsf{mm\text{-}prf\text{-}ODH}$ problem over ECDH and KDF, similar to the proof in $\mathbf{G}_2$. The attacker $\mathcal{B}_2$ obtains $(g, U, V, y^\star)$ from its challenger, where g is the group generator and that there exists some unknown u and v such that $U = g^u$ and $V = g^v$. Then, $\mathcal{B}_2$ simulates $\mathbf{G}_2$ honestly to $\mathcal{A}$ except for the modifications for $i = 2$ similar to what $\mathcal{B}_1$ did.

Similarly, we can easily have that

$$\mathsf{Adv}_2(\mathcal{A}) \leq \mathsf{Adv}_3(\mathcal{A}) + n_{\mathsf{pair}} \epsilon^{\mathsf{mm\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{KDF}}$$

Game $\mathbf{G}_4$. This game is identical to $\mathbf{G}_3$ except that the challenger does the following modification:

– The challenger guesses n_{pair} pairs of indices of queries to the NewKeyPair oracle $(\mathsf{idx}_{(i,0)}, \mathsf{idx}_{(i,1)})_{i\in[n_{\mathsf{pair}}]}$ and samples two random key $\{\widetilde{k^{\mathsf{AE}}}_i\}_{i\in[n_{\mathsf{pair}}]}$.
– Whenever the attacker sends $\mathsf{idx}_{(i,j)}$-th query to the NewKeyPair oracle for $i \in [n_{\mathsf{pair}}]$ and $j \in \{0,1\}$, the challenger records the generated $(\mathsf{pk}_{(i,j)}, \mathsf{sk}_{(i,j)})$.
– Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}_{(i,0)}, \mathsf{sk}_{(i,1)}))$ and $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}_{(i,1)}, \mathsf{sk}_{(i,0)}))$ for every $i \in [n_{\mathsf{pair}}]$, the challenger replaces them with $\widetilde{k^{\mathsf{AE}}}_i$.

Similar to the argument above, we can easily construct a sequence of attackers that can solve the mm-prf-ODH problem over ECDH and KDF. Thus, we have that

$$\mathsf{Adv}_3(\mathcal{A}) \leq \mathsf{Adv}_4(\mathcal{A}) + (n_{\mathsf{pair}} - 2)n_{\mathsf{pair}}\epsilon^{\mathsf{mm\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{KDF}}$$

Game $\mathbf{G}_5$. This game is identical to $\mathbf{G}_4$ except that the challenger does the following modification:

– For every query to Challenge oracle, the challenger simply replaces all k that will be output by random keys.

Note also that all honest k^{AE} keys have been replaced with random keys $\widetilde{k^{\mathsf{AE}}}_i$ for some i in $\mathbf{G}_4$. We can prove that $\mathbf{G}_4$ and $\mathbf{G}_5$ are indistinguishable from the attacker $\mathcal{A}$'s view by n_{pair} hybrid games. In the $i^\star$-th hybrid game for every $1 \leq i^\star \leq n_{\mathsf{pair}}$, the challenger replaces all k encrypted by using the first i keys $\{\widetilde{k^{\mathsf{AE}}}_i\}_{1\leq i\leq i^\star}$ with random keys.

If the attacker $\mathcal{A}$ can distinguish every two adjacent hybrid games, then we can easily construct an attacker $\mathcal{B}_3$ that breaks the IND-CCA security of AE by a simply reduction. Note that these n_{pair} hybrid games are independent. Thus, we have that

$$\mathsf{Adv}_4(\mathcal{A}) \leq \mathsf{Adv}_5(\mathcal{A}) + n_{\mathsf{pair}}\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{AE}}$$

Final Analysis. In $\mathbf{G}_5$, the attacker $\mathcal{A}$ obtains no information about the challenge bit, and therefore can win only by randomly guessing the challenge bit. This leads the probability of $\mathcal{A}$'s winning to be $\frac{1}{2}$. Thus, we have that

$$\mathsf{Adv}_5(\mathcal{A}) = 0$$

Combing all statements above, the proof is concluded by

$$\begin{aligned}\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) &\leq q_{\mathrm{NKP}}(n^2_{\mathsf{pair}}\epsilon^{\mathsf{mm\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{KDF}} + n_{\mathsf{pair}}\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{AE}}) \\ &\leq q^5_{\mathrm{NKP}}\epsilon^{\mathsf{mm\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{KDF}} + q^2_{\mathrm{NKP}}\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{AE}}\end{aligned}$$

F Proof of Theorem 3

Proof. We give the proof by a sequence of games. Let $\mathsf{Adv}_i(\mathcal{A}) = \Pr[\mathbf{G}_i(\mathcal{A}) = \mathsf{true}]$ denote the advantage of the attacker $\mathcal{A}$ in winning game $\mathbf{G}_i$.

Game $\mathbf{G}_0$. This game is identical to the CTI-CCA security game of maKEM scheme in Figure 2. Thus, it holds that

$$\mathsf{Adv}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) = \mathsf{Adv}_0(\mathcal{A})$$

Game $\mathbf{G}_1$. This game is identical to $\mathbf{G}_0$ except that the challenger guesses one pair of indices of queries to the NewKeyPair oracle $(\mathsf{idx}, \mathsf{idx}')$ and lets the attacker $\mathcal{A}$ immediately loses if

- the attacker $\mathcal{A}$ queries Fin with inputs two identifiers id and id', and
- id and id' are respectively inputs of the idx-th and idx'-th query to the NewKeyPair oracle.

Note that the attacker $\mathcal{A}$ can query NewKeyPair in total q_{NKP} times and therefore there are in total at most q^2_{NKP} pairs of queries. Thus, the challenger can guess correctly with probability at least $\frac{1}{q^2_{\mathrm{NKP}}}$ and we have that

$$\mathsf{Adv}_0(\mathcal{A}) \leq q^2_{\mathrm{NKP}} \mathsf{Adv}_1(\mathcal{A})$$

Game $\mathbf{G}_2$. This game is identical to $\mathbf{G}_1$ except that the challenger does the following modification:

- The challenger samples a random key $\widetilde{k^{\mathsf{AE}}}$.
- Whenever the attacker sends idx-th query to the NewKeyPair oracle, the challenger records the generated $(\mathsf{pk}, \mathsf{sk})$. Whenever the attacker sends idx'-th query to the NewKeyPair oracle, the challenger records the generated $(\mathsf{pk}', \mathsf{sk}')$.
- Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}, \mathsf{sk}'))$ and to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{pk}', \mathsf{sk}))$, the challenger replaces them with $\widetilde{k^{\mathsf{AE}}}$.

We prove that $\mathbf{G}_1$ and $\mathbf{G}_2$ are indistinguishable by reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_1$ and $\mathbf{G}_2$, then we can construct an attacker $\mathcal{B}_1$ that solves the mm-prf-ODH problem over ECDH and KDF. The attacker $\mathcal{B}_1$ obtains $(g, U, V, y^\star)$ from its challenger, where g is the group generator and that there exists some unknown u and v such that $U = g^u$ and $V = g^v$. Then, $\mathcal{B}_1$ simulates $\mathbf{G}_1$ honestly to $\mathcal{A}$ except for the following modifications:

- Whenever the attacker sends idx-th query to the NewKeyPair oracle, $\mathcal{B}_1$ returns U. Whenever the attacker sends idx'-th query to the NewKeyPair oracle, $\mathcal{B}_1$ returns V.
- Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(U, v))$ and to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(V, u))$, $\mathcal{B}_1$ replaces them with $y^\star$.
- Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{aek}, u))$, the $\mathcal{B}_1$ sends aek to its ODH_u oracle and uses the received reply to replace this KDF computation.

– Whenever the challenger needs to run $\mathsf{KDF}(\mathsf{ECDH.Derive}(\mathsf{aek}, v))$, the $\mathcal{B}_1$ sends aek to its ODH_v oracle and uses the received reply to replace this KDF computation.

Note that $\mathcal{B}_1$ perfectly simulates $\mathbf{G}_1$ to $\mathcal{A}$ honestly if the challenge bit is 0 and $\mathbf{G}_2$ to $\mathcal{A}$ if the challenge bit is 1. If $\mathcal{A}$ can distinguish $\mathbf{G}_1$ and $\mathbf{G}_2$, then $\mathcal{B}_1$ can always solves the $\mathsf{mm\text{-}prf\text{-}ODH}$ problem over ECDH and KDF. Thus, we have that

$$\mathsf{Adv}_1(\mathcal{A}) \leq \mathsf{Adv}_2(\mathcal{A}) + \epsilon^{\mathsf{mm\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{KDF}}$$

Game $\mathbf{G}_3$. This game is identical to $\mathbf{G}_2$ except that the challenger always lets $\mathcal{A}$ lose in the FIN oracle.

Note that the k^{AE} key has been replaced with random keys $\widetilde{k^{\mathsf{AE}}}$ for id and id' in $\mathbf{G}_2$. We can prove that $\mathbf{G}_2$ and $\mathbf{G}_3$ are indistinguishable from the attacker $\mathcal{A}$'s view by reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_2$ and $\mathbf{G}_3$, then we can easily construct an attacker $\mathcal{B}_2$ that breaks the $\mathsf{CTI\text{-}CCA}$ security of AE by a simply reduction. Thus, we have that

$$\mathsf{Adv}_2(\mathcal{A}) \leq \mathsf{Adv}_3(\mathcal{A}) + \epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{AE}}$$

Final Analysis. In $\mathbf{G}_3$, the attacker $\mathcal{A}$ cannot win and we have that

$$\mathsf{Adv}_3(\mathcal{A}) = 0$$

Combing all statements above, the proof is concluded by

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) \leq q^2_{\mathrm{NKP}}\epsilon^{\mathsf{mm\text{-}prf\text{-}ODH}}_{\mathsf{ECDH},\mathsf{KDF}} + q^2_{\mathrm{NKP}}\epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{AE}}$$

G Proof of Theorem 4

Proof. Let n denote the number of aeks included in the inputs of $\mathsf{maKEM.Encaps}$ algorithm. The encapsulation algorithm $\mathsf{maKEM.Encaps}$ computes the encryption algorithm $\mathsf{PKE.Enc}$ by using ek_i included in the public key aek_i once and the signing algorithm sk by using sk included in the secret key $\mathsf{adk}^\star$ once, where $i \in [n]$. For every $i \in [n]$, we have that the verification algorithm $\mathsf{DS.Vrfy}$ passes except with the probability at most δ_{DS}, and that the decryption algorithm $\mathsf{PKE.Dec}$ recovers the encrypted key except with the probability at most δ_{PKE}. Thus, for every $i \in [n]$, the decapsulation is correct except with the probability at most $(\delta_{\mathsf{DS}} + \delta_{\mathsf{PKE}})$. Since the correctness of maKEM is measured for all n decapsulations, we have that

$$\delta_{\mathsf{maKEM}} \leq n(\delta_{\mathsf{DS}} + \delta_{\mathsf{PKE}})$$

H Proof of Theorem 5

Proof. We give the proof by a sequence of games. Let $\mathsf{Adv}_i(\mathcal{A}) = |\Pr[\mathbf{G}_i(\mathcal{A}) = \mathsf{true}] - \frac{1}{2}|$ denote the advantage of the attacker $\mathcal{A}$ in winning game $\mathbf{G}_i$.

Game $\mathbf{G}_0$. This game is identical to the IND-CCA security game of maKEM scheme in Figure 1. Thus, it holds that

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) = \mathsf{Adv}_0(\mathcal{A})$$

Game $\mathbf{G}_1$. This game is identical to $\mathbf{G}_0$ except that the challenger guesses at the beginning a number $n_{\texttt{CHID}}$ and let $\mathcal{A}$ loses in the FIN oracle if $n_{\texttt{CHID}}$ is unequal to the size of the set CHID at the end of the game.

Note that the attacker $\mathcal{A}$ can query NEWKEYPAIR in total q_{NKP} times and therefore the size of the set CHID is bounded by q_{NKP}. Thus, the challenger can guess correctly with probability at least $\frac{1}{q_{\mathrm{NKP}}}$ and we have that

$$\mathsf{Adv}_0(\mathcal{A}) \leq q_{\mathrm{NKP}}\mathsf{Adv}_1(\mathcal{A})$$

The number $n_{\texttt{CHID}}$ is used in the following games.

Game $\mathbf{G}_2$. This game is identical to $\mathbf{G}_1$ except that the challenger does the following modification:

– For every query to CHALLENGE oracle, the challenger simply replaces all k that will be output by random keys.

We prove that $\mathbf{G}_1$ and $\mathbf{G}_2$ are indistinguishable by hybrid games.

Game $\mathbf{G}_{2.0}$. This hybrid game is identical to $\mathbf{G}_1$. It holds that

$$\mathsf{Adv}_1(\mathcal{A}) = \mathsf{Adv}_{2.0}(\mathcal{A})$$

Game $\mathbf{G}_{2.i}$, where $1 \leq i \leq q_{\mathbf{NewKeyPair}}$. This hybrid game is identical to $\mathbf{G}_{2.i-1}$ except for the following modifications:

– At the beginning of the game, the challenger guesses i indices $\{\mathsf{idx}_i\}_i$ of queries to the NEWKEYPAIR oracle.
– Whenever the attacker sends idx_i-th query to the NEWKEYPAIR oracle, the challenger records the output PKE public key as $\mathsf{ek}^\star_i$ and the generated PKE private key as $\mathsf{dk}^\star_i$.
– Whenever the attacker $\mathcal{A}$ queries to the ENCAPS oracles, the challenger samples two random keys k_0 and k_1 from the space $\mathcal{K}$ uniformly at random. Then, the challenger does the following:
  - Whenever the challenger needs to run PKE.Enc by using a public key that is equal to $\mathsf{ek}^\star_{i'}$ for any $1 \leq i' \leq i$, then the challenger encrypts k_1.
  - Whenever the challenger needs to run PKE.Enc by using a public key that is unequal to $\mathsf{ek}^\star_{i'}$ for any $1 \leq i' \leq i$, then the challenger encrypts k_0.

- In the end, the challenger returns k_0 together with all maKEM ciphertexts.

We prove that if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_{2.i-1}$ and $\mathbf{G}_{2.i}$, then there must exist an attacker $\mathcal{B}_1$ that can breaks the IND-CCA security of the underlying PKE by reduction.

The attacker $\mathcal{B}_1$ receives a public verification key $\mathsf{ek}^\star$. Then, $\mathcal{B}_1$ simulates $\mathbf{G}_{2.i-1}$ to $\mathcal{A}$ except for the following modifications:

– Whenever $\mathcal{A}$ sends the i-th query to the NewKeyPair oracle, $\mathcal{B}_1$ runs $(\mathsf{vk}^\star, \mathsf{sk}^\star) \xleftarrow{\$} \mathsf{DS.KGen}$ and returns $(\mathsf{vk}^\star, \mathsf{ek}^\star)$. For all other queries to the NewKeyPair oracle, $\mathcal{B}_1$ simulates them honestly and stores the corresponding secret keys adk. Moreover, $\mathcal{B}_1$ records the PKE public key output by the first $(i-1)$'s queries to the NewKeyPair oracle as $\{\mathsf{ek}_{i'}\}_{1 \le i' < i}$.
– Whenever the attacker $\mathcal{A}$ queries to the Encaps oracles, $\mathcal{B}_1$ samples two random keys k_0 and k_1 from the space $\mathcal{K}$ uniformly at random. Then, $\mathcal{B}_1$ does the following:
  - Whenever $\mathcal{B}_1$ needs to run PKE.Enc by using $\mathsf{ek}_{i'}$ for any $1 \le i' < i$, then $\mathcal{B}_1$ encrypts k_1.
  - Whenever $\mathcal{B}_1$ needs to run PKE.Enc by using $\mathsf{ek}^\star$, =then $\mathcal{B}_1$ sends (k_0, k_1) to its Enc oracle. When receiving the ciphertext c, $\mathcal{B}_1$ does the following DS.Sign execution honestly, as $\mathcal{B}_1$ knows the corresponding secret signing keys.
  - Whenever $\mathcal{B}_1$ needs to run PKE.Enc by using a PKE public key that is not included in the set $\{\mathsf{ek}_{i'}\}_{1 \le i' < i} \cup \{\mathsf{ek}^\star\}$, then $\mathcal{B}_1$ encrypts k_0.
  - In the end, $\mathcal{B}_1$ returns k_0 together with all maKEM ciphertexts.
– Whenever the attacker $\mathcal{A}$ queries to the Decaps oracles with input an identifier id that is the input of the i-th query to the NewKeyPair oracle, then $\mathcal{B}_1$ forwards the PKE ciphertext to its Dec oracle to recover the encrypted message. The rest computations are honest.

Note that $\mathcal{B}_1$ perfectly simulates $\mathbf{G}_{2.i-1}$ honestly if the challenge bit $\mathsf{b} = 0$ and $\mathbf{G}_{2.i}$ honestly if the challenge bit $\mathsf{b} = 1$. Thus, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_{2.i-1}$ and $\mathbf{G}_{2.i}$, then $\mathcal{B}_1$ can always distinguish the challenge bit and win in the IND-CCA security game of the underlying PKE. Thus,it holds that

$$\mathsf{Adv}_{2.i-1}(\mathcal{A}) \le \mathsf{Adv}_{2.i}(\mathcal{A}) + \epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{PKE}}$$

Game $\mathbf{G}_{2.n}$, where $n = n_{\mathtt{CHID}}$. In this hybrid game, whenever the attacker $\mathcal{A}$ sends queries to the Encaps oracles, the challenger samples two random keys k_0 and k_1 from the space $\mathcal{K}$ uniformly at random. Then, the challenger does the following:

– Whenever the challenger needs to run PKE.Enc, then the challenger encrypts k_1.
– In the end, the challenger returns k_0 together with all maKEM ciphertexts.

It is easy to observe that this hybrid game is identical to $\mathbf{G}_2$.

Note that there are in total $n_{\mathtt{CHID}}$ hybrid games and all these hybrid games are independent. Combing the arguments above, it holds that

$$\mathsf{Adv}_1(\mathcal{A}) \le \mathsf{Adv}_2(\mathcal{A}) + n_{\mathtt{CHID}}\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{PKE}}$$

Final Analysis. In $\mathbf{G}_2$, the attacker $\mathcal{A}$ obtains no information about the challenge bit, and therefore can win only by randomly guessing the challenge bit. This leads the probability of $\mathcal{A}$'s winning to be $\frac{1}{2}$. Thus, we have that

$$\mathsf{Adv}_2(\mathcal{A}) = 0$$

Combing all statements above, the proof is concluded by

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) \leq q^2_{\mathrm{NKP}} \epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{PKE}}$$

I Proof of Theorem 6

Proof. We give the proof by a sequence of games. Let $\mathsf{Adv}_i(\mathcal{A}) = \Pr[\mathbf{G}_i(\mathcal{A}) = \mathsf{true}]$ denote the advantage of the attacker $\mathcal{A}$ in winning game $\mathbf{G}_i$.

Game $\mathbf{G}_0$. This game is identical to the CTI-CCA security game of maKEM scheme in Figure 2. Thus, it holds that

$$\mathsf{Adv}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) = \mathsf{Adv}_0(\mathcal{A})$$

Game $\mathbf{G}_1$. This game is identical to $\mathbf{G}_0$ except that the challenger guesses an index idx to the NewKeyPair oracle at the beginning of the game and let $\mathcal{A}$ loses in the Fin oracle the input of this query to the NewKeyPair oracle is equal to the second inputs of the Fin at the end of the game.

Note that the attacker $\mathcal{A}$ can query NewKeyPair in total q_{NKP}. Thus, the challenger can guess correctly with probability at least $\frac{1}{q_{\mathrm{NKP}}}$ and we have that

$$\mathsf{Adv}_0(\mathcal{A}) \leq q_{\mathrm{NKP}} \mathsf{Adv}_1(\mathcal{A})$$

Game $\mathbf{G}_2$. This game is identical to $\mathbf{G}_1$ except that the challenger always lets $\mathcal{A}$ lose in the Fin oracle.

We can prove that $\mathbf{G}_1$ and $\mathbf{G}_2$ are indistinguishable from the attacker $\mathcal{A}$'s view by reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_1$ and $\mathbf{G}_2$, then we can easily construct an attacker $\mathcal{B}_1$ that breaks the SUF-CMA security of AE by a simply reduction. The attacker $\mathcal{B}_1$ receives a DS verification key $\mathsf{vk}^\star$. Then, $\mathcal{B}_1$ simulates $\mathbf{G}_1$ honestly to $\mathcal{A}$ except for replacing the DS public key output by idx-th query to the NewKeyPair oracle by $\mathsf{vk}^\star$. Whenever $\mathcal{B}_1$ needs to sign messages by using the private key of $\mathsf{vk}^\star$, $\mathcal{B}_1$ queries its singing oracle. If the attacker $\mathcal{A}$ can send a query $(*, \mathsf{id}', \mathsf{ct})$ to the Fin oracle such that the included maKEM ciphertext ct is never signed by id', which the challenger guesses correctly in $\mathbf{G}_1$, then $\mathcal{B}_1$ simply returns this ct to its challenger and wins.

Thus, we have that

$$\mathsf{Adv}_1(\mathcal{A}) \leq \mathsf{Adv}_2(\mathcal{A}) + \epsilon^{\mathsf{suf\text{-}cma}}_{\mathsf{DS}}$$

Final Analysis. In $\mathbf{G}_2$, the attacker $\mathcal{A}$ cannot win and we have that

$$\mathsf{Adv}_2(\mathcal{A}) = 0$$

Combing all statements above, the proof is concluded by

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{maKEM}}(\mathcal{A}) \leq q_{\mathrm{NKP}} \epsilon^{\mathsf{suf\text{-}cma}}_{\mathsf{DS}}$$

J Proof of Theorem 7

Proof. We provide the proof by a sequence of games. Without loss of generality, we assume that if the attacker $\mathcal{A}$ can trigger $\mathsf{win} \leftarrow \mathsf{true}$ then $\mathcal{A}$ immediately terminates the game.

Game $\mathbf{G}_0$. This game is identical to the $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD}}$ game in Definition 7. Thus, it holds that

$$\mathsf{Adv}^{\mathsf{Sec}}_{\mathsf{CGKD}}(\mathcal{A}) = \Pr[\mathbf{G}_0(\mathcal{A}) = \mathsf{true}] - \frac{1}{2}$$

Game $\mathbf{G}_1$. This game is identical to $\mathbf{G}_0$ except that the session identifier sid of all honestly generated sessions (i.e., via InitGroup and RegParticipant oracle are distinct). Recall that all sids are random sampled at Line 327 in Figure 9. Thus, $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable from any attacker $\mathcal{A}$'s view unless any two randomly sampled sids collide, which happens with probability at most $\binom{q_{\mathrm{IG}}+q_{\mathrm{RegP}}}{2} \cdot 2^{-\lambda} \leq (q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 2^{-\lambda}$. It holds that

$$\Pr[\mathbf{G}_0(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_1(\mathcal{A}) = \mathsf{true}] + (q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 2^{-\lambda}$$

Game $\mathbf{G}_2$. This game is identical to $\mathbf{G}_1$ except that the challenger guesses an index $\widetilde{\mathsf{idx}}_{\mathrm{NG}}$ of all queries to the NewGroup such that (1) the query input is some $(\widetilde{\mathsf{gid}}, \widetilde{\mathcal{D}})$, and (2) the attacker can trigger $\mathsf{win} \leftarrow \mathsf{true}$ only via the group $\widetilde{\mathsf{gid}}$, and let $\mathcal{A}$ win only with probability $\frac{1}{2}$.

More precisely, we say that the attacker can trigger $\mathsf{win} \leftarrow \mathsf{true}$ via the group $\widetilde{\mathsf{gid}}$ if and only if the attacker wins

- via $*\mathsf{winAthrz}(\widetilde{\mathsf{gid}}, *)$ at Line 133 in the JoinLeader oracle, or
- via $*\mathsf{winAthrz}(\widetilde{\mathsf{gid}}, *)$ at Line 142 in the JoinParticipant oracle, or
- via $*\mathsf{winGKRosAthn}(\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}})$ at Line 163 in the EvolveParticipant oracle for some $\widetilde{\mathsf{id}}$ and $\widetilde{\mathsf{sid}}$ such that $\mathrm{RG}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}] = \widetilde{\mathsf{gid}}$, or
- by testing challenge bit at Line 83 in the Fin oracle and it holds that either $\mathsf{tested} = \perp$ or $\mathsf{tested} = (\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, *, *)$ such that $\mathrm{RG}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}] = \widetilde{\mathsf{gid}}$.

Note that the attacker can trigger $\mathsf{win} \leftarrow \mathsf{true}$ only in four oracles: JoinLeader, JoinParticipant, EvolveParticipant, and Fin. If the attacker win can win $\mathbf{G}_1$ with non-negligible probability, then it must trigger $\mathsf{win} \leftarrow \mathsf{true}$ in one of

these four oracles, which means that the challenger can guess correctly with probability at least $\frac{1}{q_{\mathrm{NG}}}$.

$$\Pr[\mathbf{G}_1(\mathcal{A}) = \mathsf{true}] \leq q_{\mathrm{NG}} \Pr[\mathbf{G}_2(\mathcal{A}) = \mathsf{true}]$$

Case Split. Below, we split the proof into three cases C1, C2, and C3, according to how the attacker can trigger $\mathsf{win} \leftarrow \mathsf{true}$. More precisely,

- Case C1 analyzes the case that attacker triggers $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winAthrz}$ at Line 133 in the JoinLeader oracle or via $*\mathsf{winAthrz}$ at Line 142 in the JoinParticipant oracle,
- Case C2 analyzes the case that attacker triggers $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winGKRosAthn}$ at Line 163 in the EvolveParticipant oracle, and
- Case C3 analyzes the case that the attacker triggers $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winChl}$ at Line 83 in the Fin oracle.

Note that the above three cases are disjoint and can cover all scenarios that the attacker can trigger $\mathsf{win} \leftarrow \mathsf{true}$. Thus, we have that

$$\Pr[\mathbf{G}_2(\mathcal{A}) = \mathsf{true}] = \max\Big(\Pr[\mathbf{G}_2^{C1}(\mathcal{A}) = \mathsf{true}], \Pr[\mathbf{G}_2^{C2}(\mathcal{A}) = \mathsf{true}], \Pr[\mathbf{G}_2^{C3}(\mathcal{A}) = \mathsf{true}]\Big)$$

J.1 Case 1: Attacker wins via $*$winAuthrz.

In this case, we analyze the probability that the attacker triggers $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winAthrz}$ at Line 133 in the JoinLeader oracle or via $*\mathsf{winAthrz}$ at Line 142 in the JoinParticipant oracle. From $\mathbf{G}_2^{\mathrm{C1}}$, the challenger has already guessed the correct index $\widetilde{\mathsf{idx}_{\mathrm{NG}}}$ of queries to the NewGroup oracle such that (1) the query input is some $(\widetilde{\mathsf{gid}}, \widetilde{\mathcal{D}})$, and (2) the attacker triggers $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winAthrz}(\widetilde{\mathsf{gid}}, *)$ at Line 133 in the JoinLeader oracle or via $*\mathsf{winAthrz}(\widetilde{\mathsf{gid}}, *)$ at Line 142 in the JoinParticipant oracle. This means that

- $\widetilde{\mathsf{gid}} \notin \mathtt{CGrp}[\widetilde{\mathsf{gid}}]$, which further means that the attacker $\mathcal{A}$ has never queried Reveal$(\widetilde{\mathsf{gid}})$ before, and
- $\alpha_{\widetilde{\mathcal{D}}} \geq \alpha_{\mathsf{min}}$.

Game $\mathbf{G}_3^{\mathrm{C1}}$. This game is identical to $\mathbf{G}_2^{\mathrm{C1}}$ except for the following modifications:

- Whenever the attacker queries the JoinLeader oracle (resp. JoinParticipant oracle) for some $(\mathsf{id}, \mathsf{sid}, \mathsf{m})$ such that $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] = \widetilde{\mathsf{gid}}$, the challenger does not execute $\mathsf{key} \leftarrow \pi^{\mathsf{pake}}.\mathsf{key}$ at Line 236 in the Join-L algorithm in Figure 8 (resp. at Line 284 in the Join-P algorithm in Figure 8). Instead, the challenger does the following:

- If the challenger has recorded any $(\mathsf{tr}^\star, \mathsf{key}^\star)$ such that $\pi^{\mathsf{pake}}.\mathsf{tr} = \mathsf{tr}^\star$, then the challenger replaces $\pi^{\mathsf{pake}}.\mathsf{key}$ with $\mathsf{key}^\star$.
- Otherwise, the challenger replaces $\pi^{\mathsf{pake}}.\mathsf{key}$ with a random key key and records $(\pi^{\mathsf{pake}}.\mathsf{tr}, \mathsf{key})$.

We prove that $\mathbf{G}_2^{\mathrm{C1}}$ and $\mathbf{G}_3^{\mathrm{C1}}$ are indistinguishable from the attacker $\mathcal{A}$'s view by reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_2^{\mathrm{C1}}$ and $\mathbf{G}_3^{\mathrm{C1}}$, then we can construct an attacker $\mathcal{B}_1$ that breaks the wBPR security of PAKE. The attacker $\mathcal{B}_1$ simulates $\mathbf{G}_2^{\mathrm{C1}}$ honestly, except for the following oracles:

– NewGroup$(\mathsf{gid}, \mathcal{D})$: When $\mathcal{B}_1$ receives the $\widetilde{\mathsf{idx}_{\mathrm{NG}}}$-th query to the NewGroup oracle, $\mathcal{B}_1$ records the input as $(\widetilde{\mathsf{gid}}, \widetilde{\mathcal{D}})$. Then $\mathcal{B}_1$ queries its Init oracle with input $(\widetilde{\mathsf{gid}} \parallel \text{“}\mathsf{joiner}\text{”}, \widetilde{\mathsf{gid}} \parallel \text{“}\mathsf{leader}\text{”})$ and ends this query afterwards.
For all other queries to this oracle, $\mathcal{B}_1$ simulates them honestly.
– JoinLeader$(\mathsf{id}, \mathsf{sid}, \mathsf{m})$: When $\mathcal{B}_1$ receives this query, $\mathcal{B}_1$ first does the following checks:
  - whether $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] = \widetilde{\mathsf{gid}}$ and $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{role} = \mathsf{leader}$,
  - whether some $(\mathsf{id}', \mathsf{sid}', \mathsf{rec}, \mathsf{m}^{\mathsf{pake}}, \sigma)$ can be parsed from m,
  - check whether no records in $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ros}$ corresponds to party id''s session sid',
  - check whether $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathrm{Temp}[\mathsf{id}', \mathsf{sid}'] = \bot$, and
  - check whether $\mathsf{rec} = \mathbf{0}$, $\mathsf{m}^{\mathsf{pake}} \neq \mathbf{0}$, and $\sigma = \mathbf{0}$.

  If any of the above check fails, then $\mathcal{B}_1$ simulates this query honestly. Otherwise, $\mathcal{B}_1$ queries its SendResp oracle with input $\mathsf{m}^{\mathsf{pake}}$ and receives an index $\mathsf{idx}^{\mathsf{pake}}$, a message $\mathsf{m}^{\mathsf{resp}}$, and an accepted flag acc.
  Next, if $\mathsf{acc} \neq \mathsf{true}$, then $\mathcal{B}_1$ undoes all above execution and ends this query. Otherwise, $\mathcal{B}_1$ checks whether it has recorded a PAKE transcript and a PAKE key $(\mathsf{tr}^\star, \mathsf{key}^\star)$ in the JoinLeader or JoinParticipant oracle, where $\mathsf{tr}^\star$ equals to the combination of $\mathsf{m}^{\mathsf{pake}}$ and $\mathsf{m}^{\mathsf{resp}}$. If such record does not exists, then $\mathcal{B}_1$ queries its Test oracle with input $(\widetilde{\mathsf{gid}} \parallel \text{“}\mathsf{leader}\text{”}, \mathsf{idx}^{\mathsf{pake}})$. Note that the check on $*\mathsf{isFresh}$ in the Test oracle will always pass, because (1) $\mathsf{acc} = \mathsf{true}$, (2) $(\widetilde{\mathsf{gid}} \parallel \text{“}\mathsf{leader}\text{”}, \mathsf{idx}^{\mathsf{pake}})$ has not been queried to Test oracle, and (3) there is no partnered PAKE session that shares the same PAKE transcript. This means, $\mathcal{B}_1$ can always receive a key $\mathsf{key}^\star$.
  Afterwards, $\mathcal{B}_1$ replaces the PAKE key $\pi^{\mathsf{pake}}.\mathsf{key}$ at Line 236 in the Join-L algorithm in Figure 8 with $\mathsf{key}^\star$. Finally, $\mathcal{B}_1$ executes all remaining actions in this query honestly and records $(\mathsf{tr}^\star, \mathsf{key}^\star)$, where $\mathsf{tr}^\star$ denotes the combination of $\mathsf{m}^{\mathsf{pake}}$ and $\mathsf{m}^{\mathsf{resp}}$.
– JoinParticipant$(\mathsf{id}, \mathsf{sid}, \mathsf{m})$: When $\mathcal{B}_1$ receives this query, $\mathcal{B}_1$ first checks whether $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] = \widetilde{\mathsf{gid}}$ and $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{role} = \mathsf{joiner}$. If the check fails, then $\mathcal{B}_1$ simulates this query honestly. Next, $\mathcal{B}_1$ executes differently according to whether $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{temp} = \bot$ or not.
  In the case that $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{temp} = \bot$**:** $\mathcal{B}_1$ first does the following checks:
  - check whether $(\mathsf{id}', \mathsf{sid}')$ can be parsed from m, and
  - check whether $(\mathsf{id}, \mathsf{sid}) \neq (\mathsf{id}', \mathsf{sid}')$.

If any check above fails, then $\mathcal{B}_1$ simply ends this query. Otherwise, $\mathcal{B}_1$ queries its SendInit oracle and receives an index $\mathsf{idx}^{\mathsf{pake}}$ and a message $\mathsf{m}^{\mathsf{resp}}$.
Afterwards, $\mathcal{B}_1$ runs the remaining execution honestly, notably, except for replacing $\mathsf{m}^{\mathsf{pake}}_{\mathsf{out}}$ in the Join-P algorithm with $\mathsf{m}^{\mathsf{resp}}$.
Finally $\mathcal{B}_1$ records this $\mathsf{idx}^{\mathsf{pake}}$ and the message $\mathsf{m}^{\mathsf{resp}}$ for $(\mathsf{id}, \mathsf{sid})$.
In the case that $\mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{temp} \neq \bot$**:** $\mathcal{B}_1$ first does the following check:

- whether $(\mathsf{m}^{\mathsf{pake}}, \mathsf{rec}, \mathsf{stg}, \mathsf{tag})$ can be parsed from the input message m.

If the above check fails, then $\mathcal{B}_1$ simply ends this query. Otherwise, $\mathcal{B}_1$ must have recorded an index $\mathsf{idx}^{\mathsf{pake}}$ and a message $\mathsf{m}^{\mathsf{resp}}$ for $(\mathsf{id}, \mathsf{sid})$.
Next, $\mathcal{B}_1$ queries its SendTerInit oracle with input $(\mathsf{idx}^{\mathsf{pake}}, \mathsf{m}^{\mathsf{pake}})$ and receives an accepted flag acc.
Then, if $\mathsf{acc} \neq \mathsf{true}$, then $\mathcal{B}_1$ undoes all above execution and ends this query. Otherwise, $\mathcal{B}_1$ checks whether it has recorded a PAKE transcript and a PAKE key $(\mathsf{tr}^\star, \mathsf{key}^\star)$ in the JoinLeader or JoinParticipant oracle, where $\mathsf{tr}^\star$ equals to the combination of $\mathsf{m}^{\mathsf{resp}}$ and $\mathsf{m}^{\mathsf{pake}}$. If such record does not exist, then $\mathcal{B}_1$ queries its Test oracle with input $(\widetilde{\mathsf{gid}} \parallel \text{“}\mathsf{joiner}\text{”}, \mathsf{idx}^{\mathsf{pake}})$. Note that the check on $*\mathsf{isFresh}$ in the Test oracle will always pass, because (1) $\mathsf{acc} = \mathsf{true}$, (2) $(\widetilde{\mathsf{gid}} \parallel \text{“}\mathsf{joiner}\text{”}, \mathsf{idx}^{\mathsf{pake}})$ has not been queried to Test oracle, and (3) there is no partnered PAKE session that shares the same PAKE transcript. This means, $\mathcal{B}_1$ can always receive a key $\mathsf{key}^\star$.
Afterwards, $\mathcal{B}_1$ replaces the PAKE key $\pi^{\mathsf{pake}}.\mathsf{key}$ at Line 284 in the Join-P algorithm in Figure 8 with $\mathsf{key}^\star$. Finally, $\mathcal{B}_1$ executes all remaining actions in this query honestly and records $(\mathsf{tr}^\star, \mathsf{key}^\star)$, where $\mathsf{tr}^\star$ denotes the combination of $\mathsf{m}^{\mathsf{pake}}$ and $\mathsf{m}^{\mathsf{resp}}$.

– Reveal(gid): When $\mathcal{B}_1$ receives this query, $\mathcal{B}_1$ checks whether $\mathsf{gid} = \widetilde{\mathsf{gid}}$. If the check fails, then $\mathcal{B}_1$ simulates this query honestly. Otherwise, $\mathcal{B}_1$ terminates the game and let $\mathcal{A}$ immediately lose.

It is easy to observe that $\mathcal{B}_1$ perfectly simulates $\mathbf{G}^{\mathrm{C1}}_2$ if the challenge bit $\mathsf{b} = 0$, and $\mathbf{G}^{\mathrm{C1}}_3$ if the challenge bit $\mathsf{b} = 1$. This means, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}^{\mathrm{C1}}_2$ and $\mathbf{G}^{\mathrm{C1}}_3$, then $\mathcal{B}_1$ can easily break the wBPR security of the underlying PAKE. Thus, it holds that

$$\Pr[\mathbf{G}^{\mathrm{C1}}_2(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}^{\mathrm{C1}}_3(\mathcal{A}) = \mathsf{true}] + \epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha}$$

Game $\mathbf{G}^{\mathrm{C1}}_4$. This game is identical to $\mathbf{G}^{\mathrm{C1}}_3$ except for the following modification:

– Whenever the attacker queries JoinLeader oracle (resp. JoinParticipant oracle) for some $(\mathsf{id}, \mathsf{sid}, \mathsf{m})$ such that $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] = \widetilde{\mathsf{gid}}$, the challenger does not directly derive k^{MAC} and r^{DS} by running $(k^{\mathsf{MAC}}, r^{\mathsf{DS}}) \leftarrow \mathsf{KDF}(\mathsf{key}, \text{“}\mathsf{Join}\text{”})$ at Line 237 in the Join-L algorithm (resp. at Line 285 in the Join-P algorithm). Instead, the challenger does the following:

- If the challenger has not derived k^{MAC} and r^{DS} for the key key, then the challenger samples k^{MAC} and r^{DS} uniformly at random.
- Otherwise, the challenger has already derived k^{MAC} and r^{DS} for the key key, then the challenger simply reuses them.

Recall that all keys are chosen uniformly at random in $\mathbf{G}_3^{\mathrm{C1}}$. It is straightforward that if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_3^{\mathrm{C1}}$ and $\mathbf{G}_4^{\mathrm{C1}}$, we can easily construct an attacker $\mathcal{B}_2$ that breaks the PRF security of KDF. Thus, it holds that

$$\Pr[\mathbf{G}_3^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_4^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] + (q_{\mathrm{JL}} + q_{\mathrm{JP}})\epsilon_{\mathsf{KDF}}^{\mathsf{prf}}$$

Game $\mathbf{G}_5^{\mathrm{C1}}$. This game is identical to $\mathbf{G}_4^{\mathrm{C1}}$ except for the following modification:

– Whenever the attacker $\mathcal{A}$ sends queries to JoinParticipant with input $(\mathsf{id}, \mathsf{sid}, \mathsf{m})$ of the following form, then the challenger immediately ends this query:
  - $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] = \widetilde{\mathsf{gid}}$,
  - $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{temp} \neq \bot$,
  - the message m cannot be parsed as $(\mathsf{m}^{\mathsf{pake}}, \mathsf{rec}, \mathsf{stg}, \mathsf{tag})$ (see Line 279), or the tuple $(\mathsf{rec}, \mathsf{stg})$ is forged by the attacker $\mathcal{A}$.

When $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{temp} \neq \bot$, the query will continue (i.e., not end) at Line 286 in the Join-P algorithm only if the tuple $(\mathsf{rec}, \mathsf{stg})$ and the tag tag can pass the MAC verification. Note that the MAC key k^{MAC} is chosen uniformly at random in $\mathbf{G}_4^{\mathrm{C1}}$. Note also that this MAC key is either shared by a unique PAKE partner or such partner does not exist (see $\mathbf{G}_3^{\mathrm{C1}}$). If the tuple $(\mathsf{rec}, \mathsf{stg})$ is forged by the attacker $\mathcal{A}$, which means that this tuple is not signed by the unique PAKE partner by using the same k^{MAC}, then there must be an attacker $\mathcal{B}_3$ that can break the EUF-CMA security of the underlying MAC. Since the MAC key generation and verification are occur in the JoinParticipant algorithm, there are at most q_{JP} distinct MAC keys. Thus, $\mathbf{G}_5^{\mathrm{C1}}$ and $\mathbf{G}_4^{\mathrm{C1}}$ are indistinguishable except for the probability at most $\epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}}$. It holds that

$$\Pr[\mathbf{G}_4^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_5^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] + q_{\mathrm{JP}}\epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}}$$

In particular, this game ensures that the record rec is not forged by the attacker (and therefore sent by the honest partner). Note that all honest participants will not run the Join-L algorithm. This means, the owner of the record rec must be the leader of the same group as the joiner id's session sid, which means that $\mathrm{RG}[\mathsf{id}', \mathsf{sid}'] = \mathrm{RG}[\mathsf{id}, \mathsf{sid}]$, where $(\mathsf{id}', \mathsf{sid}', *) \leftarrow \mathsf{rec.getIds}$. Moreover, recall that all honest parties can join a group only if they are authorized. This means, the leader id' must be authorized by group $\widetilde{\mathsf{gid}} = \mathrm{RG}[\mathsf{id}', \mathsf{sid}'] = \mathrm{RG}[\mathsf{id}, \mathsf{sid}]$.

Combing the statements above, we know that the attacker $\mathcal{A}$ can never trigger $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winAthrz}$ at Line 142 in the JoinParticipant oracle in $\mathbf{G}_5^{\mathrm{C1}}$.

Game $\mathbf{G}_6^{\mathrm{C1}}$. This game is identical to $\mathbf{G}_5^{\mathrm{C1}}$ except for the following modification:

– Whenever the attacker sends queries to JoinLeader with input $(\mathsf{id}, \mathsf{sid}, \mathsf{m})$ of the following form, then the challenger immediately ends this query:
  - $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] = \widetilde{\mathsf{gid}}$,
  - the message m cannot be parsed as $(\mathsf{id}', \mathsf{sid}', \mathsf{rec}, \mathsf{m}^{\mathsf{pake}}, \sigma)$ (see Line 228), where $\mathsf{rec} \neq \mathbf{0}$,
  - $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathrm{Temp}[\mathsf{id}', \mathsf{sid}'] \neq \bot$, and

- the record rec is forged by the attacker $\mathcal{A}$.

When $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathrm{Temp}[\mathsf{id}', \mathsf{sid}'] = \mathsf{vk}$ is not $\perp$, the query will continue (i.e., not end) at Line 246 in the Join-L algorithm only if the the record rec can pass the DS verification under the DS verification key vk and signature σ. Recall that the DS key pair $(\mathsf{vk}, \mathsf{sk})$ is generated by running $\mathsf{DS.KGen}(r^{\mathsf{DS}})$ at Line 238, where r^{DS} is chosen uniformly at random in $\mathbf{G}_4^{\mathrm{C1}}$.

Note also that this DS key generation randomness r^{DS} is either shared by a unique PAKE partner or such partner does not exist (see $\mathbf{G}_3^{\mathrm{C1}}$). If the record rec is forged by the attacker $\mathcal{A}$, meaning that the record is not signed by the unique PAKE partner by using the corresponding DS secret signing key sk of the DS public verification key vk, then there must be an attacker $\mathcal{B}_4$ that can break the $\mathsf{EUF\text{-}CMA}$ security of the DS. Since the DS key generation and verification are occur only in the JoinLeader algorithm, there are at most q_{JL} distinct DS public-secret key pairs. Thus, $\mathbf{G}_6^{\mathrm{C1}}$ and $\mathbf{G}_5^{\mathrm{C1}}$ are indistinguishable except for the probability at most $\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$. It holds that

$$\Pr[\mathbf{G}_5^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_6^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] + q_{\mathrm{JL}}\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$$

In particular, this game ensures that the record rec derived and output by the leader must not be forged by the attacker. This means that the the rec must be sent by the honest joiner id''s session sid' of the same group, i.e., $\mathrm{RG}[\mathsf{id}', \mathsf{sid}'] = \mathrm{RG}[\mathsf{id}, \mathsf{sid}]$. Moreover, recall that all honest parties can join a group only if they are authorized. This means, the leader id' must be authorized by group $\mathsf{gid} = \mathrm{RG}[\mathsf{id}', \mathsf{sid}'] = \mathrm{RG}[\mathsf{id}, \mathsf{sid}]$.

Recall that the gs is not leaked by the Reveal oracle, as we have ensured in $\mathbf{G}_2^{\mathrm{C1}}$. This means, the attacker can never trigger $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winAthrz}$ at Line 133 in the JoinLeader oracle in $\mathbf{G}_6^{\mathrm{C1}}$.

Final Case Analysis. Now, the attacker can neither trigger $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winAthrz}$ at Line 142 in the JoinParticipant oracle, as we have ensured in $\mathbf{G}_5^{\mathrm{C1}}$, nor trigger $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winAthrz}$ at Line 133 in the JoinLeader oracle in $\mathbf{G}_6^{\mathrm{C1}}$. Thus, the attacker can never win in this case and we have that

$$\Pr[\mathbf{G}_6^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] = 0$$

Combing all statements above, it holds that

$$\Pr[\mathbf{G}_2^{\mathrm{C1}}(\mathcal{A}) = \mathsf{true}] \leq \epsilon_{\mathsf{PAKE},\alpha}^{\mathsf{wBPR}} + \epsilon_{\mathsf{KDF}}^{\mathsf{prf}} + q_{\mathrm{JP}}\epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}} + q_{\mathrm{JL}}\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$$

J.2 Case 2: Attacker wins via $*\mathsf{winGKRosAuthn}$.

In this case, we analyze the probability that the attacker $\mathcal{A}$ triggers $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winGKRosAthn}(\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}})$ at Line 163 in the EvolveParticipant oracle for some $\widetilde{\mathsf{id}}$ and $\widetilde{\mathsf{sid}}$ such that $\mathrm{RG}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}] = \widetilde{\mathsf{gid}}$, where $\widetilde{\mathsf{gid}}$ is the input of $\widetilde{\mathsf{idx}}_{\mathrm{NG}}$-th query to the NewGroup oracle.

Game $\mathbf{G}_3^{\mathrm{C2}}$. This game is identical to $\mathbf{G}_2^{\mathrm{C2}}$ except that the challenger immediately uses false as the output of $*\mathsf{winGKRosAthn}$ algorithm with inputs $(\mathsf{id}, \mathsf{sid})$ if

- $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, where $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) = *\mathsf{leader}(\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ros})$, and
- $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ros} \neq \mathrm{Ros}[\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}, \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{stg}]$.

If the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_2^{\mathrm{C2}}$ and $\mathbf{G}_3^{\mathrm{C2}}$, then $\mathcal{A}$ must trigger the above event. Due to the second result in our Lemma 1, the probability of the occurrence of the above event is bounded by

$$\epsilon_{\mathsf{PAKE},\alpha}^{\mathsf{wBPR}} + \epsilon_{\mathsf{KDF}}^{\mathsf{prf}} + q_{\mathrm{JP}}\epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}} + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$$

Thus, it holds that

$$\begin{aligned}\Pr[\mathbf{G}_2^{\mathrm{C2}}(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_3^{\mathrm{C2}}(\mathcal{A}) = \mathsf{true}] + \epsilon_{\mathsf{PAKE},\alpha}^{\mathsf{wBPR}} + \epsilon_{\mathsf{KDF}}^{\mathsf{prf}} + q_{\mathrm{JP}}\epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}} \\ + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}\end{aligned}$$

Game $\mathbf{G}_4^{\mathrm{C2}}$. This game is identical to $\mathbf{G}_3^{\mathrm{C2}}$ except that the challenger immediately uses false as the output of $*\mathsf{winGKRosAthn}$ algorithm with inputs $(\mathsf{id}, \mathsf{sid})$ if

- $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) \notin \mathtt{CEp}$, where $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) = *\mathsf{leader}(\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ros})$,
- $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathtt{CEp}$, and
- $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{gk} \neq \mathrm{GK}[\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}, \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{stg}]$.

We prove that $\mathbf{G}_3^{\mathrm{C2}}$ and $\mathbf{G}_4^{\mathrm{C2}}$ are indistinguishable from the attacker $\mathcal{A}$'s view by reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_3^{\mathrm{C2}}$ and $\mathbf{G}_4^{\mathrm{C2}}$, then we can construct an attacker $\mathcal{B}_5$ that breaks the $\mathsf{CTI\text{-}CCA}$ security of the underlying maKEM. The attacker $\mathcal{B}_5$ simulates $\mathbf{G}_3^{\mathrm{C2}}$ honestly to $\mathcal{A}$ except for the following modifications for any party id' and session sid' that is registered for $\widetilde{\mathsf{gid}}$:

- Whenever $\mathcal{B}_5$ needs to run $\mathsf{maKEM.KGen}$, $\mathcal{B}_5$ queries its NewKeyPair oracle with some unique input (e.g., the current party id's session sid's epoch ep), and use the received aek as the generated public key.
- Whenever $\mathcal{B}_5$ needs to run $\mathsf{maKEM.Encaps}$, if the private key adk has been corrupted, then $\mathcal{B}_5$ computes simulates $\mathsf{maKEM.Encaps}$ by itself. Otherwise, $\mathcal{B}_5$ queries its Encaps oracle for a key and a ciphertext.
- Whenever $\mathcal{B}_5$ needs to run $\mathsf{maKEM.Decaps}$, $\mathcal{B}_5$ queries its Decaps oracle for a key.
- Whenever $\mathcal{B}_5$ needs to return a private adk, $\mathcal{B}_5$ queries Corrupt oracle with the corresponding previously picked unique identifier as input, the returns the received adk.

If $\mathcal{A}$ can distinguish $\mathbf{G}_3^{\mathrm{C2}}$ and $\mathbf{G}_4^{\mathrm{C2}}$, then $\mathcal{A}$ must trigger $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{gk} \neq \mathrm{GK}[\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}, \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{stg}]$ for some $(\mathsf{id}, \mathsf{sid})$ and $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep})$ at some time point.

Note that we have ensured in $\mathbf{G}_3^{\mathrm{C2}}$ that the party-session $(\mathsf{id}, \mathsf{sid})$ must agree on the roster with the leader-session $(\mathsf{lid}, \mathsf{lsid})$ at epoch ep. It means that maKEM

public keys included in the records with identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep})$ and with identifier $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep})$ must both be generated by query NewKeyPair oracle.

Moreover, $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) \notin \mathtt{CEp}$ and $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathtt{CEp}$ imply that their maKEM private keys are not leaked by querying Corrupt oracle in the CTI-CCA game.

From $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{gk} \neq \mathrm{GK}[\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}, \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{stg}]$, we know that the group key that is generated by the leader is unequal to the one recovered by the party id's session sid, which further means that the ciphertext must be forged, i.e., not output by the Encaps oracle.

Thus, if $\mathcal{A}$ can distinguish $\mathbf{G}_3^{\mathrm{C2}}$ and $\mathbf{G}_4^{\mathrm{C2}}$, then $\mathcal{B}_5$ can always easily win by return its Fin with the identifiers and ciphertext that triggered the above event. It holds that

$$\Pr[\mathbf{G}_3^{\mathrm{C2}}(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_4^{\mathrm{C2}}(\mathcal{A}) = \mathsf{true}] + \epsilon_{\mathsf{maKEM}}^{\mathsf{cti\text{-}cca}}$$

Final Case Analysis. Now, the attacker $\mathcal{A}$ can trigger $\mathsf{win} \leftarrow *\mathsf{winGKRosAthn}$ neither at Line 190 due to $\mathbf{G}_3^{\mathrm{C2}}$ nor at Line 188 due to $\mathbf{G}_4^{\mathrm{C2}}$. Thus, the attacker can never win in this case and we have that

$$\Pr[\mathbf{G}_4^{\mathrm{C2}}(\mathcal{A}) = \mathsf{true}] = 0$$

Combing all statements above, it holds that

$$\begin{aligned}\Pr[\mathbf{G}_2^{\mathrm{C2}}(\mathcal{A}) = \mathsf{true}] \leq & \epsilon_{\mathsf{PAKE},\alpha}^{\mathsf{wBPR}} + \epsilon_{\mathsf{KDF}}^{\mathsf{prf}} + q_{\mathrm{JP}}\epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}} \\ & + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}} + \epsilon_{\mathsf{maKEM}}^{\mathsf{cti\text{-}cca}}\end{aligned}$$

J.3 Case 3: Attacker wins via *winChl.

In this case, we analyze the probability that the attacker $\mathcal{A}$ triggers $\mathsf{win} \leftarrow \mathsf{true}$ via $*\mathsf{winChl}$ at Line 83 in the EvolveParticipant oracle. Note that if the attacker $\mathcal{A}$ does not query the Test oracle, then $\mathcal{A}$ obtains no information about the challenge bit and can win only by randomly guessing. Below, we assume that the attacker must query the Test oracle. For simplicity, we use denote tested tuple by $(\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}})$ and $\mathrm{RG}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}]$ by $\widetilde{\mathsf{gid}}$. We use $(\widetilde{\mathsf{lid}}, \widetilde{\mathsf{lsid}}, *)$ to denote $*\mathsf{leader}(\mathrm{Ros}[\mathsf{tested}]).\mathsf{getIds}$. It is possible that $(\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}) = (\widetilde{\mathsf{lid}}, \widetilde{\mathsf{lsid}})$.

Game $\mathbf{G}_3^{\mathrm{C3}}$. This game is identical to $\mathbf{G}_2^{\mathrm{C3}}$ except that the challenger uses false as the output of the $*\mathsf{winChl}$ if there exists any tuple $(\mathsf{id}, \mathsf{sid})$ (possibly equal to $(\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}})$) such that either of the following event happens:

– Event 1:
  - $\mathrm{RG}[\mathsf{id}, \mathsf{sid}] = \widetilde{\mathsf{gid}}$,
  - $(\mathsf{id}, \mathsf{sid}, \widetilde{\mathsf{ep}}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$,
  - the record with identifiers $(\mathsf{id}, \mathsf{sid}, \widetilde{\mathsf{ep}})$ received by any partner session of the party id's session sid at epoch ep and any stg is forged. Recall that the partner session means that (i) the leader lid's session lsid with $(\mathsf{lid}, \mathsf{lsid}, *) = *\mathsf{leader}(\mathrm{Ros}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}]).\mathsf{getIds}$, if id's session sid takes the role of participant, or (ii) any participant id''s session sid' with $\exists \mathsf{rec} \in \mathrm{Ros}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}]$ and $(\mathsf{id}', \mathsf{sid}', *) = \mathsf{rec}.\mathsf{getIds}$.

– Event 2:
  - there exists a record $\mathsf{rec} \in \mathrm{Ros}[\widetilde{\mathsf{lid}}, \widetilde{\mathsf{lsid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$ and $(\mathsf{id}', \mathsf{sid}', *) = \mathsf{rec.getIds}$,
  - $(\mathsf{lid}, \mathsf{lsid}, \widetilde{\mathsf{ep}}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, and
  - $\mathrm{Ros}[\mathsf{id}, \mathsf{sid}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}] \neq ()$ and $\mathrm{Ros}[\mathsf{id}, \mathsf{sid}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}] \neq \mathrm{Ros}[\widetilde{\mathsf{lid}}, \widetilde{\mathsf{lsid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$.

If the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_2^{\mathrm{C3}}$ and $\mathbf{G}_3^{\mathrm{C3}}$, then $\mathcal{A}$ must trigger either of the above event. The first event is covered by the second result in our Lemma 1 and the second event is covered by the first result in our Lemma 1. Thus, the probability of the occurrence of the above event is bounded by

$$\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha} + \epsilon^{\mathsf{prf}}_{\mathsf{KDF}} + q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}} + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}$$

And we have that

$$\begin{aligned}\Pr[\mathbf{G}_2^{\mathrm{C3}}(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_3^{\mathrm{C3}}(\mathcal{A}) = \mathsf{true}] + \epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha} + \epsilon^{\mathsf{prf}}_{\mathsf{KDF}} + q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}} \\ + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}\end{aligned}$$

In particular, this game ensures that

– for all every record $\mathsf{rec} \in \mathrm{Ros}[\widetilde{\mathsf{lid}}, \widetilde{\mathsf{lsid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$ and $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}') = \mathsf{rec.getIds}$ that if $\mathsf{rec.getIds} \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, then the record rec must be honestly generated by the party id''s session sid' for epoch $\mathsf{ep}' = \widetilde{\mathsf{ep}}$.
– for every group member indicated by the roster $\mathrm{Ros}[\mathsf{lid}, \widetilde{\mathsf{lsid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$ must agree on the this roster, and therefore agree on each other's record.

Game $\mathbf{G}_4^{\mathrm{C3}}$. This game is identical to $\mathbf{G}_3^{\mathrm{C3}}$ except that the challenger replaces the output of the Test oracle with a random key.

We prove that $\mathbf{G}_3^{\mathrm{C3}}$ and $\mathbf{G}_4^{\mathrm{C3}}$ are indistinguishable by reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_3^{\mathrm{C3}}$ and $\mathbf{G}_4^{\mathrm{C3}}$, then we can construct an attacker $\mathcal{B}_6$ that breaks the IND-CCA security of the underlying maKEM. The attacker $\mathcal{B}_6$ simulates $\mathbf{G}_3^{\mathrm{C3}}$ honestly except for the following modifications:

– At the beginning, $\mathcal{B}_6$ guesses an index idx of an query to the EvolveLeader oracle such that this query will generates the group key $\mathrm{GK}[\widetilde{\mathsf{lid}}, \widetilde{\mathsf{lsid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$.
– Whenever $\mathcal{B}_6$ needs to run $\mathsf{maKEM.KGen}$, $\mathcal{B}_6$ queries its NewKeyPair oracle with some unique input (e.g., the current party id's session sid's epoch ep), and use the received aek as the generated public key.
– Whenever $\mathcal{B}_6$ needs to run $\mathsf{maKEM.Encaps}$ in the EvolveLeader oracle and this is not the idx-th query, $\mathcal{B}_6$ queries its Encaps oracle for a key and a ciphertext. If this is the idx-th query, then $\mathcal{B}_6$ queries its Challenge oracle for the key and ciphertexts, followed by recording the input identifiers (e.g., the party identifier, session identifier, and epoch index), the output key, and the output ciphertexts.
– Whenever $\mathcal{B}_6$ needs to run $\mathsf{maKEM.Decaps}$ to decapsulate a maKEM ciphertext ct, $\mathcal{B}_6$ considers the following cases:

- If ct was output by the CHALLENGE oracle for the desired sender and recipient pair, then $\mathcal{B}_6$ simply uses the key output by the CHALLENGE oracle.
- Otherwise, $\mathcal{B}_6$ queries its DECAPS oracle for a key.

– Whenever $\mathcal{B}_6$ needs to return a private adk, $\mathcal{B}_6$ queries CORRUPT oracle with the corresponding previously picked unique identifier as input, the returns the received adk. If the input identifier is included in the $\mathtt{CHID}$, then $\mathcal{B}_6$ simply aborts.
– Whenever $\mathcal{A}$ queries the LEAK oracle so that $\mathcal{B}_6$ needs to return a group key that is $\mathrm{GK}[\mathsf{id}, \mathsf{sid}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$, where $\mathsf{id}, \mathsf{sid}$ indicates a record included in $\mathrm{Ros}[\mathsf{tested}]$, $\mathcal{B}_6$ aborts.

Assume that the attacker $\mathcal{B}_6$ guesses correctly. If the attacker $\mathcal{A}$ can win, then the fresh condition must hold. In particular, this indicates that for every $\mathsf{rec} \in \mathrm{Ros}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$, it always holds that $\mathsf{rec.getIds} \notin \mathtt{CEp}$. This means the abortion in the CORRUPT oracle never happens.

Moreover, the freshness condition also indicates for every $(\mathsf{id}, \mathsf{sid})$ indicated by $\mathrm{Ros}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}]$, it always holds that $(\mathsf{id}, \mathsf{sid}, \widetilde{\mathsf{ep}}, \widetilde{\mathsf{stg}}) \notin \mathtt{CStg}$. This means the abortion in the LEAK oracle never happens.

Now, if the challenge bit $\mathsf{b} = 0$, then $\mathcal{B}_6$ perfectly simulates $\mathbf{G}_3^{\mathrm{C3}}$ to $\mathcal{A}$, and $\mathbf{G}_4^{\mathrm{C3}}$ to $\mathcal{A}$ otherwise. If $\mathcal{A}$ can distinguish $\mathbf{G}_3^{\mathrm{C3}}$ and $\mathbf{G}_4^{\mathrm{C3}}$, then $\mathcal{B}_6$ can always win. Thus, it holds that

$$\Pr[\mathbf{G}_3^{\mathrm{C3}}(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_4^{\mathrm{C3}}(\mathcal{A}) = \mathsf{true}] + q_{\mathrm{EL}} \epsilon_{\mathsf{maKEM}}^{\mathsf{ind\text{-}cca}}$$

Final Case Analysis. Now, the attacker $\mathcal{A}$ can trigger $\mathsf{win} \leftarrow *\mathsf{winChl}$ only by randomly guessing, as $\mathcal{A}$ receives no information about the challenge bit. Thus, the attacker $\mathcal{A}$ can win in this case with probability at most $\frac{1}{2}$ and it holds that

$$\Pr[\mathbf{G}_4^{\mathrm{C3}}(\mathcal{A}) = \mathsf{true}] \leq \frac{1}{2}$$

Combing all statements above, it holds that

$$\begin{aligned}\Pr[\mathbf{G}_2^{\mathrm{C3}}(\mathcal{A}) = \mathsf{true}] \leq & \epsilon_{\mathsf{PAKE},\alpha}^{\mathsf{wBPR}} + \epsilon_{\mathsf{KDF}}^{\mathsf{prf}} + q_{\mathrm{JP}} \epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}} \\ & + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}}) \epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}} + q_{\mathrm{EL}} \epsilon_{\mathsf{maKEM}}^{\mathsf{ind\text{-}cca}}\end{aligned}$$

Final Proof Analysis. Combing all statements above, it holds that

$$\begin{aligned}\Pr[\mathbf{G}_2(\mathcal{A}) = \mathsf{true}] \leq & q_{\mathrm{NG}} \epsilon_{\mathsf{PAKE},\alpha}^{\mathsf{wBPR}} + q_{\mathrm{NG}} \epsilon_{\mathsf{KDF}}^{\mathsf{prf}} + q_{\mathrm{NG}} q_{\mathrm{JP}} \epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}} \\ & + q_{\mathrm{NG}} (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}}) \epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}} + q_{\mathrm{NG}} \epsilon_{\mathsf{maKEM}}^{\mathsf{cti\text{-}cca}} \\ & + q_{\mathrm{NG}} q_{\mathrm{EL}} \epsilon_{\mathsf{maKEM}}^{\mathsf{ind\text{-}cca}} + \frac{1}{2}\end{aligned}$$

The proof is concluded by

$$\begin{aligned}\mathsf{Adv}^{\mathsf{Sec}}_{\mathsf{CGKD}}(\mathcal{A}) \leq & (q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 \cdot 2^{-\lambda} + q_{\mathrm{NG}}\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha} + q_{\mathrm{NG}}(q_{\mathrm{JL}} + q_{\mathrm{JP}})\epsilon^{\mathsf{prf}}_{\mathsf{KDF}} \\ & + q_{\mathrm{NG}}q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}} + q_{\mathrm{NG}}(q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}} \\ & + q_{\mathrm{NG}}\epsilon^{\mathsf{cti\text{-}cca}}_{\mathsf{maKEM}} + q_{\mathrm{NG}}q_{\mathrm{EL}}\epsilon^{\mathsf{ind\text{-}cca}}_{\mathsf{maKEM}}\end{aligned}$$

J.4 Helping Lemma 1

Below, we present Lemma 1, which is used in the proof of Theorem 7. Informally, Lemma 1 shows that, for any target group gid in which the attacker attempts to win:

- if a group leader accepts a record rec such that $\mathsf{rec.getIds} \notin \mathrm{CanInj}[\mathsf{gid}]$, then rec must have been honestly generated by the participant id in session sid during epoch ep, where $\mathsf{rec.getIds} = (\mathsf{id}, \mathsf{sid}, \mathsf{ep})$;
- if a group participant accepts a record rec such that $\mathsf{rec.getIds} \notin \mathrm{CanInj}[\mathsf{gid}]$, then rec must have been honestly generated by the group leader id in session sid during epoch ep, where $\mathsf{rec.getIds} = (\mathsf{id}, \mathsf{sid}, \mathsf{ep})$.

Lemma 1. *Let* $\mathbf{G}'$ *denote the* $\mathbf{G}^{\mathsf{Sec}}_{\mathsf{CGKD},\alpha_{\mathsf{min}}}$ *game with the modification that*

- *all honestly generated session identifiers in* InitGroup *and* RegParticipant *oracles are distinct, i.e., the modification we made in* $\mathbf{G}_1$ *in the proof of Theorem 7 in Appendix J.*

Let $\widetilde{\mathsf{idx}_{\mathrm{NG}}}$ *denote an index of all queries to the* NewGroup *oracle that initializes a group* $\widetilde{\mathsf{gid}}$*. Let* $\widetilde{\mathsf{id}}$ *and* $\widetilde{\mathsf{sid}}$ *respectively denote the identifiers of any honest party and its session with* $\mathrm{RG}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}] = \widetilde{\mathsf{gid}}$*.*

- *Consider the event that the attacker* $\mathcal{A}$ *queries* $\textsc{RefreshLeader}(\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, \widetilde{\mathsf{m}})$ *for some message* $\widetilde{\mathsf{m}}$ *and the execution* $(\mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}, \widetilde{\mathsf{rec}}) \xleftarrow{\$} \mathsf{Refresh\text{-}L}(\mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}, \widetilde{\mathsf{m}})$ *that successfully ends (i.e., without throwing any error) and* $\widetilde{\mathsf{rec}}.\mathsf{getIds} \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$*. It must hold that* $\mathrm{Rec}[\widetilde{\mathsf{rec}}.\mathsf{getIds}] = \widetilde{\mathsf{rec}}$*.*
- *Consider the event that the attacker* $\mathcal{A}$ *queries* $\textsc{EvolveParticipant}(\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, \widetilde{\mathsf{m}})$ *for some message* $\widetilde{\mathsf{m}}$ *and the execution* $(\mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}, \mathsf{nxtEp}) \xleftarrow{\$} \mathsf{Evolve\text{-}P}(\mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}, \widetilde{\mathsf{m}})$ *that successfully ends (i.e., without throwing any error), and* $\widetilde{\mathsf{rec}}.\mathsf{getIds} \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$*, where* $\widetilde{\mathsf{rec}}$ *denotes the leader record in the local roster underlying* $\mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}$*, i.e.,* $\widetilde{\mathsf{rec}} = *\mathsf{leader}(\mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}.\mathsf{ros})$*. Let* $\widetilde{\mathsf{stg}}$ *denote the stage underlying* $\mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}$*, i.e.,* $\widetilde{\mathsf{stg}} = \mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}.\mathsf{stg}$*. It must hold that* $\mathrm{Ros}[\widetilde{\mathsf{rec}}.\mathsf{id}, \widetilde{\mathsf{rec}}.\mathsf{sid}, \widetilde{\mathsf{rec}}.\mathsf{ep}, \widetilde{\mathsf{stg}}] = \mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}.\mathsf{ros}$*.*

We have the above two results always hold except for the probability at most:

$$\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha} + (q_{\mathrm{JL}} + q_{\mathrm{JP}})\epsilon^{\mathsf{prf}}_{\mathsf{KDF}} + q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}} + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}$$

Proof. We prove this lemma by a sequence of games.

Game $\mathbf{G}_0$. This game is identical to $\mathbf{G}^{'}$ and we have that

$$\Pr[\mathbf{G}_0(\mathcal{A}) = \mathsf{true}] = \Pr[\mathbf{G}^{'}(\mathcal{A}) = \mathsf{true}]$$

Game $\mathbf{G}_1$. This game is identical to $\mathbf{G}_0$ except that the challenger lets $\mathcal{A}$ immediately lose, if there exists any past epoch-stage tuple $(\mathsf{ep}, \mathsf{stg})$, i.e., $\mathsf{ep} < \mathsf{sst}^{\mathsf{sid}}_{\mathsf{id}}.\mathsf{ep}$ or (2) $\mathsf{ep} = \mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}.\mathsf{ep}$ and $\mathsf{stg} \leq \mathsf{sst}^{\widetilde{\mathsf{sid}}}_{\widetilde{\mathsf{id}}}.\mathsf{stg}$, such that

– $\mathsf{rec}.\mathsf{getIds} \in \mathrm{CanInj}[\mathsf{gid}]$, where $\mathsf{rec} = *\mathsf{leader}(\mathrm{Ros}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, \mathsf{ep}, \mathsf{stg}])$.

Note that the above equation indicates that there exists any past leader's record with some identifiers $(\mathsf{lid}, \mathsf{lsid}) = (\mathsf{rec}.\mathsf{id}, \mathsf{rec}.\mathsf{sid})$ such that $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}) \in \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$. If this happens, then the identifiers of all records in the roster $\mathrm{Ros}[\widetilde{\mathsf{id}}, \widetilde{\mathsf{sid}}, \mathsf{ep}, \mathsf{stg}]$ will be added into $\mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, due to Lines 172 to 174 in the $*$stg-mgmt helper function.

Moreover, if any record identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep})$ is added into $\mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, then the identifiers of following records $(\mathsf{id}', \mathsf{sid}', \mathsf{ep}'')$ with $\mathsf{ep}'' \geq \mathsf{ep}'$ will also be added into the $\mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, due to Lines 193 and 194 in the $*$refr-mgmt helper function.

This means, the identifiers of the record rec must also be added into $\mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, $\mathsf{rec}.\mathsf{getIds} \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ never happens. Thus, $\mathbf{G}_0^{\mathrm{C0}}$ and $\mathbf{G}_1^{\mathrm{C0}}$ are identical from the attacker $\mathcal{A}$'s view. It holds that

$$\Pr[\mathbf{G}_0(\mathcal{A}) = \mathsf{true}] = \Pr[\mathbf{G}_1(\mathcal{A}) = \mathsf{true}]$$

Game $\mathbf{G}_2$. This game is identical to $\mathbf{G}_1$ except that the challenger lets $\mathcal{A}$ immediately lose if the group $\widetilde{\mathsf{gid}}$ is compromised $\widetilde{\mathsf{gid}} \in \mathtt{CGrp}$ before the party $\widetilde{\mathsf{id}}$'s session $\widetilde{\mathsf{sid}}$ joined the group $\widetilde{\mathsf{gid}}$.

Note that if $\widetilde{\mathsf{gid}} \in \mathtt{CGrp}$ before the party $\widetilde{\mathsf{id}}$'s session $\widetilde{\mathsf{sid}}$ joined the group $\widetilde{\mathsf{gid}}$, then the identifiers of the leader's record output by $\mathsf{Join}\text{-}\mathsf{P}$ algorithm must be added into $\mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ at Line 143. This contradicts to $\mathbf{G}_1$, which ensures that the identifiers of all past leaders of the party $\widetilde{\mathsf{id}}$'s session $\widetilde{\mathsf{sid}}$ at any past epoch and stage are not added into the dictionary $\mathrm{CanInj}[\widetilde{\mathsf{gid}}]$.

Thus, $\mathbf{G}_1^{\mathrm{C0}}$ and $\mathbf{G}_2^{\mathrm{C0}}$ are identical from the attacker $\mathcal{A}$'s view. It holds that

$$\Pr[\mathbf{G}_1(\mathcal{A}) = \mathsf{true}] = \Pr[\mathbf{G}_2(\mathcal{A}) = \mathsf{true}]$$

Game $\mathbf{G}_3$. This game is identical to $\mathbf{G}_2$ except that the challenger let $\mathcal{A}$ immediately lose if

– there exists any party id's session sid joined the group $\widetilde{\mathsf{gid}}$ at the time $\widetilde{\mathsf{gid}} \notin \mathtt{CGrp}$, but either the party id's session sid or its communicated party (if exists) obtains the partner's forged record.

Similar to the proofs from $\mathbf{G}_2^{\mathrm{C1}}$ to $\mathbf{G}_6^{\mathrm{C1}}$, we know that the above event cannot happen except with the probability at most $\epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha}+\epsilon^{\mathsf{prf}}_{\mathsf{KDF}}+q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}}+q_{\mathrm{JL}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}$. Thus, it holds that

$$\Pr[\mathbf{G}_2(\mathcal{A})=\mathsf{true}] \leq \Pr[\mathbf{G}_3(\mathcal{A})=\mathsf{true}] + \epsilon^{\mathsf{wBPR}}_{\mathsf{PAKE},\alpha} + (q_{\mathrm{JL}}+q_{\mathrm{JP}})\epsilon^{\mathsf{prf}}_{\mathsf{KDF}} + q_{\mathrm{JP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{MAC}} + q_{\mathrm{JL}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}$$

Game $\mathbf{G}_4$. This game is identical to $\mathbf{G}_3$ except that the challenger immediately lets $\mathcal{A}$ lose if there exists any party id with session sid of the group $\widetilde{\mathsf{gid}}$ at any past epoch-stage tuple $(\mathsf{ep},\mathsf{stg})$ such that

- the leader lid and session lsid are $(\mathsf{lid},\mathsf{lsid}) = (\mathsf{rec.id},\mathsf{rec.sid})$, where $\mathsf{rec} = *\mathsf{leader}(\mathrm{Ros}[\mathsf{id},\mathsf{sid},\mathsf{ep},\mathsf{stg}])$,
- either of the following events happens:
  - the leader lid's session lsid accepts id's session sid honest record at epoch ep but accepts a forged record of id's session sid at epoch $\mathsf{ep}+1$ in the RefreshLeader oracle while at that time $\mathsf{id} \notin \mathtt{CPar}$, or
  - the party id's session sid accepts the honest record of lid's session lsid at epoch ep but accepts a forged record of lid's session lsid at epoch $\mathsf{ep}+1$ in the EvolveParticipant oracle while at that time $\mathsf{lid} \notin \mathtt{CPar}$.

Note that the above event happens only if the attacker $\mathcal{A}$ can forge a record-signature pair that can pass $\mathsf{DS.Vrfy}$ verification at either Line 349. We can prove that $\mathbf{G}_3$ and $\mathbf{G}_4$ are indistinguishable from the attacker $\mathcal{A}$'s view by a simple reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_3$ and $\mathbf{G}_4$, then we can construct an attacker $\mathcal{B}_1$ that breaks the $\mathsf{EUF\text{-}CMA}$ security of the underlying DS scheme.

The attacker $\mathcal{B}_1$ first guesses the index of the query to the NewParty oracle that can create either the party id or lid above. The guess is correct with probability at least $\frac{1}{q_{\mathrm{NP}}}$. Then, $\mathcal{B}_1$ replaces the $\mathsf{lst_{id}.ipk}$ with the vk obtained from its challenger via its Init oracle. Whenever $\mathcal{B}_1$ needs to sign messages by using $\mathsf{lst_{id}.isk}$, $\mathcal{B}_1$ forwards these messages to its Sign oracle to obtain the signatures.

If the attacker $\mathcal{A}$ can trigger the above event, i.e., forging a record that is not honestly generated by its partner, then $\mathcal{B}_1$ simply returns this record and the associated signature to its challenger and wins. Thus, it holds that

$$\Pr[\mathbf{G}_3(\mathcal{A})=\mathsf{true}] \leq \Pr[\mathbf{G}_4(\mathcal{A})=\mathsf{true}] + q_{\mathrm{NP}}\epsilon^{\mathsf{euf\text{-}cma}}_{\mathsf{DS}}$$

Game $\mathbf{G}_5$. This game is identical to $\mathbf{G}_4$ except that the challenger lets $\mathcal{A}$ immediately lose if there exists any party id with session sid of the group $\widetilde{\mathsf{gid}}$ at any epoch-stage tuple $(\mathsf{ep},\mathsf{stg})$ such that

- the leader lid and session lsid are $(\mathsf{lid},\mathsf{lsid}) = (\mathsf{rec.id},\mathsf{rec.sid})$, where $\mathsf{rec} = *\mathsf{leader}(\mathrm{Ros}[\mathsf{id},\mathsf{sid},\mathsf{ep},\mathsf{stg}])$,
- either of the following events happens:

- the leader lid's session lsid accepts id's session sid honest record at epoch ep but accepts a forged record of id's session sid at epoch $\mathsf{ep}+1$ in the RefreshLeader oracle while at that time $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathtt{CEp}$, or
- the party id's session sid accepts the honest record of lid's session lsid at epoch ep but accepts a forged record of lid's session lsid at epoch $\mathsf{ep}+1$ in the EvolveParticipant oracle while at that time $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}) \notin \mathtt{CEp}$.

Note that the above event happens only if the attacker $\mathcal{A}$ can forge a record-signature pair that can pass $\mathsf{DS.Vrfy}$ verification at either Line 350. We can prove that $\mathbf{G}_4$ and $\mathbf{G}_5$ are indistinguishable from the attacker $\mathcal{A}$'s view by a simple reduction. Namely, if the attacker $\mathcal{A}$ can distinguish $\mathbf{G}_4$ and $\mathbf{G}_5$, then we can construct an attacker $\mathcal{B}_2$ that breaks the $\mathsf{EUF\text{-}CMA}$ security of the underlying DS scheme.

The attacker $\mathcal{B}_2$ first guesses the index of the query to the RefreshParticipant and EvolveLeader oracles that can create the record of either the party id's session sid's at epoch ep' or the leader lid's session lsid's at epoch ep. The guess is correct with probability at least $\frac{1}{q_{\mathrm{RefP}}+q_{\mathrm{EL}}}$. Then, when $\mathcal{B}_2$ needs to generate a vk included in the spk in the above query, $\mathcal{B}_2$ replaces it with the one obtained from its challenger. Afterwards, whenever $\mathcal{B}_2$ needs to sign messages by using sk, $\mathcal{B}_2$ forwards these messages to its Sign oracle to obtain the signatures.

If the attacker $\mathcal{A}$ can trigger the above event, i.e., forging a record that is not honestly generated by its partner, then $\mathcal{B}_2$ simply returns this record and the associated signature to its challenger and wins. Thus, it holds that

$$\Pr[\mathbf{G}_4(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_5(\mathcal{A}) = \mathsf{true}] + (q_{\mathrm{RefP}} + q_{\mathrm{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$$

Game $\mathbf{G}_6$. This game is identical to $\mathbf{G}_5$ except that the challenger lets $\mathcal{A}$ immediately lose if there exists any participant id with session sid of the group $\widetilde{\mathsf{gid}}$ at any epoch-stage tuple $(\mathsf{ep}, \mathsf{stg})$ such that

– The participant id with session sid receives the leader lid's session lsid's honest records at epoch-stage $(\mathsf{ep}, \mathsf{stg})$, and
– $\mathrm{Ros}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}] \neq \mathrm{Ros}[\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}, \mathsf{stg}]$.

We prove that $\mathbf{G}_5$ and $\mathbf{G}_6$ are indistinguishable from the attacker $\mathcal{A}$'s view by reduction. Namely, if the attacker can distinguish $\mathbf{G}_5$ and $\mathbf{G}_6$, then we can construct an attacker $\mathcal{B}_3$ that breaks the $\mathsf{EUF\text{-}CMA}$ security of the underlying DS. Recall that the roster is always signed by using the leader's DS signing key included in the session secret key before sent out, and is always verified by using the leader's DS's public key included in the record. If the attacker $\mathcal{A}$ can forge a roster from the leader to participants, under the condition that the participant receives the honest record of the leader, then the attacker $\mathcal{A}$ must forge a new $(\mathsf{ros}, \mathsf{stg})$ pair and a signature that is not signed before. Then, $\mathcal{B}_3$ can always wins. Note that the new records can be honestly generated either in RefreshParticipant oracle or EvolveLeader oracle. Thus, it holds that

$$\Pr[\mathbf{G}_5(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_6(\mathcal{A}) = \mathsf{true}] + (q_{\mathrm{RefP}} + q_{\mathrm{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}$$

Game $\mathbf{G}_7$. This game is identical to $\mathbf{G}_6$ except that the challenger lets $\mathcal{A}$ immediately lose if there exists any party id with session sid of the group $\widetilde{\mathsf{gid}}$ at any epoch-stage tuple $(\mathsf{ep}, \mathsf{stg})$ such that

- $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, and
- the record with identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep})$ received by any partner session of the party id's session sid at epoch ep and stg is forged.
  Here, the partner session means the leader lid's session lsid with $(\mathsf{lid}, \mathsf{lsid}) = (\mathsf{rec.id}, \mathsf{rec.sid})$ where $\mathsf{rec} = *\mathsf{leader}(\mathrm{Ros}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}])$, if id's session sid takes the role of participant; and any participant id''s session sid' with $\exists \mathsf{rec} \in \mathrm{Ros}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}, \mathsf{stg}]$ and $(\mathsf{id}', \mathsf{sid}') = (\mathsf{rec.id}, \mathsf{rec.sid})$, if id's session sid takes the role of leader.

We prove that the attacker $\mathcal{A}$ can never trigger the above event by contradiction. Suppose that the attacker $\mathcal{A}$ can trigger the above event, then there exists a party id with session sid of the group $\widetilde{\mathsf{gid}}$ at any epoch-stage tuple $(\mathsf{ep}, \mathsf{stg})$ such that

- $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, and
- the record with identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep})$ received by any partner session of the party id's session sid at epoch ep and stg is forged.

We consider the above event happens for the first time. Now, we consider the following two scenarios depending one whether id' session sid acts as either participant or leader.

(1) the party id*'s session* sid *acts as a participant.* First, we exclude the case that partner lid's session lsid obtains a forged record of id's session sid when id's session sid joins the group $\widetilde{\mathsf{gid}}$. From $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ it is easy to know that the group $\widetilde{\mathsf{gid}}$ must not be compromised before id's session sid joins the group $\widetilde{\mathsf{gid}}$. Note that we have ensured in $\mathbf{G}_3$ that if a participant joins the group $\widetilde{\mathsf{gid}}$ before the group is compromised, then the leader and the participant must exchange the honest records. This contradicts to that the record with identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep})$ received by any partner session of the party id's session sid at epoch ep and stg is forged.

Second, we excludes the case that the partner lid's session lsid obtains a forged record of id's session sid via the $\mathsf{Leave\text{-}P}$ algorithm. This means, the leader lid's session lsid accepts a forged record of the party id's session sid only by inherit that record from its leader lid''s session lsid', when lid's session lsid acted as a participant.

By $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}') \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ we know that it must also hold $(\mathsf{lid}', \mathsf{lsid}', \mathsf{ep}') \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$. Furthermore, it must also hold that

- lid's session lsid and lid''s session lsid' must receive each other's honest record.
- id's session sid and lid''s session lsid' must receive each other's honest record.

Moreover, due to $\mathbf{G}_6$, we know that $\mathsf{Ros}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}', \mathsf{stg}'] = \mathsf{Ros}[\mathsf{lid}, \mathsf{lid}, \mathsf{ep}', \mathsf{stg}'] = \mathsf{Ros}[\mathsf{lid}', \mathsf{lid}', \mathsf{ep}', \mathsf{stg}']$, in particular, which includes both the honest records of id's session sid at any past epoch ep' and stage stg'. Thus, the leader lid's session lsid cannot a forged record of the party id's session sid only by inherit that record from its leader lid''s session lsid' neither.

Third, we excludes the case that the partner lid's session lsid obtains a forged record of id's session sid via the $\mathsf{Refresh\text{-}L}$ algorithm. By $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, we know that $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}') \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ for all past $\mathsf{ep}' \leq \mathsf{ep}$, due to Lines 193 and 194. Moreover, we also know that the for every all past epoch-stage (ep', stg'), for any leader lid's session lsid with $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}') = *\mathsf{leader}(\mathsf{Ros}[\mathsf{id}, \mathsf{sid}, \mathsf{ep}', \mathsf{stg}']).\mathsf{getIds}$, it must hold that $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}') \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, due to Lines 172 to 174. This further means that

- the record with identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}')$ received by any leader session of any participant id's session sid at epoch ep' and stg' must be honest.
- the record with identifiers $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}')$ received by any participant session of any leader lid's session lsid at epoch ep' and stg' must be honest.

Furthermore, from $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}') \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ and $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}') \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, we know it must hold before their records are delivered to partner sessions that

- $\mathsf{id} \notin \mathtt{CPar}$ or $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}' - 1) \notin \mathtt{CPar}$, and
- $\mathsf{lid} \notin \mathtt{CPar}$ or $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}' - 1) \notin \mathtt{CPar}$, and

Note that the above second condition $\mathsf{lid} \notin \mathtt{CPar}$ or $(\mathsf{lid}, \mathsf{lsid}, \mathsf{ep}' - 1) \notin \mathtt{CPar}$ indicates that the partner leader lid's session lsid cannot accept a forged record of the party id''s session sid' by $\mathsf{Refresh\text{-}L}$ algorithm, due to $\mathbf{G}_4$ and $\mathbf{G}_5$.

(2) the party id*'s session* sid *acts as a leader.* First, we excludes the case that id's session sid initializes the group $\widetilde{\mathsf{gid}}$ by InitGroup oracle, because if so, there are no partner session and the contradiction is trivial.

Second, we excludes the case that the partner id''s session sid' obtains a forged record of id's session sid via the $\mathsf{Join\text{-}P}$ algorithm. From $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ it is easy to know that the group $\widetilde{\mathsf{gid}}$ must not be compromised before id's session sid joins the group $\widetilde{\mathsf{gid}}$. Recall that we have ensured in $\mathbf{G}_3$ that if a participant joins the group $\widetilde{\mathsf{gid}}$ before the group is compromised, then the leader and the participant must exchange the honest records. This contradicts to that the record with identifiers $(\mathsf{id}, \mathsf{sid}, \mathsf{ep})$ received by any partner session of the party id's session sid at epoch ep and stg is forged.

Third, we excludes the case that the partner id''s session sid' obtains a forged record of id's session sid by $\mathsf{Leave\text{-}P}$ algorithm. This means, the party id''s session sid' accepts a forged record of the party id's session sid only by inherit that record from its leader lid''s session lsid', when id's session sid acted as a participant. This is impossible, similar to the arguments in the second case in the scenario (1).

Fourth, we excludes the case that the partner id''s session sid' obtains a forged record of id's session sid by $\mathsf{Evolve\text{-}P}$ algorithm. This means, the partner id''s

session sid' obtains an honest record of id's session sid at $\mathsf{ep}-1$ but a forged record at epoch ep. From $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}') \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, we know that $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}'-1) \notin \mathtt{CPar}$ must hold, which further indicates that either $\mathsf{id} \notin \mathtt{CPar}$ or $(\mathsf{id}, \mathsf{sid}, \mathsf{ep}'-1) \notin \mathtt{CPar}$. Due to $\mathbf{G}_4^{\mathrm{C2}}$ and $\mathbf{G}_5^{\mathrm{C2}}$, the partner id''s session sid' must accept the honest record of the leader id's session sid at epoch ep

To sum up, the above event cannot happen, leading $\mathbf{G}_6^{\mathrm{C0}}$ and $\mathbf{G}_7^{\mathrm{C0}}$ are identical from the attacker $\mathcal{A}$'s view. Thus, it holds that

$$\Pr[\mathbf{G}_6(\mathcal{A}) = \mathsf{true}] = \Pr[\mathbf{G}_7(\mathcal{A}) = \mathsf{true}]$$

Notably, this game ensures that the first result in Lemma 1 must hold.

Game $\mathbf{G}_8$. This game is identical to $\mathbf{G}_7$ except that the challenger lets $\mathcal{A}$ immediately loses if

- $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$, where $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) = *\mathsf{leader}(\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ros})$, and
- $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ros} \neq \mathrm{Ros}[\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}, \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{stg}]$.

Note the in the Evolve-P algorithm, every participant always check whether the epoch included in the leader's record matches the participant's epoch whenever the epoch steps forward. Thus, it hold that $\mathsf{lep} = \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ep}$. Due to $\mathbf{G}_7$, we know that $(\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}) \notin \mathrm{CanInj}[\widetilde{\mathsf{gid}}]$ indicates that the party id's session sid must receive the honest record of the leader lid's session lsid at epoch $\mathsf{lep} = \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ep}$. Due to $\mathbf{G}_6$, it always holds that $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{ros} = \mathrm{Ros}[\mathsf{lid}, \mathsf{lsid}, \mathsf{lep}, \mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{stg}]$, meaning that the attacker $\mathcal{A}$ can never trigger the above event. This means, $\mathbf{G}_7^{\mathrm{C0}}$ and $\mathbf{G}_8^{\mathrm{C0}}$ are identical from the attacker $\mathcal{A}$'s view, and it holds that

$$\Pr[\mathbf{G}_7(\mathcal{A}) = \mathsf{true}] = \Pr[\mathbf{G}_8(\mathcal{A}) = \mathsf{true}]$$

Notably, this game ensures that the second result in Lemma 1 must hold.

Final Analysis. Now, the two results in Lemma 1 both hold. Combing all statements above, our Lemma 1 always holds except for the probability at most

$$\begin{aligned}&\Pr[\mathbf{G}_8(\mathcal{A}) = \mathsf{true}] - \Pr[\mathbf{G}^{'}(\mathcal{A}) = \mathsf{true}]\\ \leq&\epsilon_{\mathsf{PAKE},\alpha}^{\mathsf{wBPR}} + (q_{\mathrm{JL}} + q_{\mathrm{JP}})\epsilon_{\mathsf{KDF}}^{\mathsf{prf}} + q_{\mathrm{JP}}\epsilon_{\mathsf{MAC}}^{\mathsf{euf\text{-}cma}} + (q_{\mathrm{JL}} + q_{\mathrm{NP}} + 2q_{\mathrm{RefP}} + 2q_{\mathrm{EL}})\epsilon_{\mathsf{DS}}^{\mathsf{euf\text{-}cma}}\end{aligned}$$

K Proof of Theorem 8

Proof. We provide the proof by a sequence of games. Without loss of generality, we assume that if the attacker $\mathcal{A}$ can trigger $\mathsf{win} \leftarrow \mathsf{true}$ then $\mathcal{A}$ immediately terminates the game.

Game $\mathbf{G}_0$. This game is identical to the $\mathbf{G}_{\mathsf{CGKD}}^{\mathsf{Corr}}$ game in Definition 30. Thus, it holds that

$$\mathsf{Adv}_{\mathsf{CGKD}}^{\mathsf{Corr}}(\mathcal{A}) = \Pr[\mathbf{G}_0(\mathcal{A}) = \mathsf{true}]$$

Game $\mathbf{G}_1$. This game is identical to $\mathbf{G}_0$ except that the challenger lets $\mathcal{A}$ immediately lose if there exists any newly generated session state in the InitGroup and RegParticipant oracles that shares the same $(\mathsf{id}, \mathsf{sid})$ with any previous session state. Recall that all sids are random sampled at Line 327 in Figure 9. Thus, $\mathbf{G}_0$ and $\mathbf{G}_1$ are indistinguishable from any attacker $\mathcal{A}$'s view unless any two randomly sampled sids collide, which happens with probability at most $\binom{q_{\mathrm{IG}}+q_{\mathrm{RegP}}}{2} \cdot 2^{-\lambda}$. It holds that

$$\begin{aligned}\Pr[\mathbf{G}_0(\mathcal{A}) = \mathsf{true}] &\leq \Pr[\mathbf{G}_1(\mathcal{A}) = \mathsf{true}] + \binom{q_{\mathrm{IG}} + q_{\mathrm{RegP}}}{2} \cdot 2^{-\lambda} \\ &\leq \Pr[\mathbf{G}_1(\mathcal{A}) = \mathsf{true}] + (q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 \cdot 2^{-\lambda}\end{aligned}$$

Game $\mathbf{G}_2$. This game is identical to $\mathbf{G}_1$ except that the attacker $\mathcal{A}$ can never win via $*\mathsf{winRecAgr}$ helper function at Line 499 in the JoinGroup oracle. Note that $*\mathsf{winRecAgr}$ outputs true only if

- either the leader or the joiner cannot output the partner's record, or
- the leader's record is unequal to the record output by the joiner, or
- the joiner's epoch is unequal to the record output by the leader, or
- the leader's group is equal to the joiner's group, or
- the leader's epoch is unequal to the joiner's epoch, or
- the leader's stage is unequal to the joiner's stage, or
- the joiner's role is not changed to the participant role.

Note that the joiner always changes its role to parti after a successful join phase at Line 293 in Figure 8. These events happen only if

(1) either the $\mathsf{PAKE.Resp}$ algorithm is not accepted at Line 235, or
(2) the $\mathsf{PAKE.TerInit}$ algorithm is not accepted at Line 283, or
(3) the id_I and id_R that are respectively specified by the leader at Line 233 and by the joiner at Line 274 do not match, or
(4) the keys output by the PAKE of the leader at Line 236 and of the joiner at Line 284 do not match, or
(5) the $\mathsf{MAC.Vrfy}$ execution at Line 286 in the $\mathsf{Join\text{-}P}$ algorithm fails, or
(6) the $\mathsf{DS.Vrfy}$ execution at Line 246 in the $\mathsf{Join\text{-}L}$ algorithm fails,

For every query to the JoinGroup oracle, the above conditions (1) - (4) happen with probability at most δ_{PAKE}. If conditions (1) - (4) do not happen, then condition (5) and (6) respectively happen with probability at most δ_{MAC} and δ_{DS}. Thus, $\mathbf{G}_1$ and $\mathbf{G}_2$ are indistinguishable from any attacker $\mathcal{A}$'s view unless conditions (1) - (6) happen with probability at most $q_{\mathrm{JG}}(\delta_{\mathsf{PAKE}} + \delta_{\mathsf{MAC}} + \delta_{\mathsf{DS}})$. It holds that

$$\Pr[\mathbf{G}_1(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_2(\mathcal{A}) = \mathsf{true}] + q_{\mathrm{JG}}(\delta_{\mathsf{PAKE}} + \delta_{\mathsf{MAC}} + \delta_{\mathsf{DS}})$$

Note also that every party id's session sid can be added into $\mathrm{Ros}[\mathsf{gid}]$ and $\mathrm{ToAdd}[\mathsf{gid}]$ only at Line 500 in the JoinGroup oracle. This further means that

in this game $\mathbf{G}_2$ for any group gid and any record in Ros[gid] and ToAdd[gid] that indicates a party id's session sid, it must always hold that $\mathsf{sst}_{\mathsf{id}}^{\mathsf{sid}}.\mathsf{gid} = \mathsf{gid}$.

Game $\mathbf{G}_3$. This game is identical to $\mathbf{G}_2$ except that the attacker $\mathcal{A}$ can never win via $*\mathsf{winEvolve}$ helper function at Line 505 in the EVOLVE oracle.

If $\mathsf{nxtEp} = \mathsf{true}$, then the group first run $\mathsf{Refresh}$ phase. Every participant in the roster Ros[gid] refresh its states by running $\mathsf{Refresh\text{-}P}$ algorithm and sends the message to the leader, the leader runs $\mathsf{Refresh\text{-}L}$ algorithm. The leader can recover each participant's new record if the two $\mathsf{DS.Vrfy}$ executions at Lines 349 and 350 pass, yielding error probability at most $2\delta_{\mathsf{DS}}$. The leader then also refreshes its state and every participant can recover the next record correctly except for error probability at most $2\delta_{\mathsf{DS}}$. In every group, there are at most q_{JG} participants.

Afterwards, the leader and all participants respectively run $\mathsf{Evolve\text{-}L}$ and $\mathsf{Evolve\text{-}P}$ algorithm. Every participant can recover the roster and stage from the leader's message only if the DS verification at Line 305 passes, which happens except probability at most δ_{DS}. Every participant can recover the group key only if the maKEM is correct, which happens except probability at most δ_{maKEM}.

Thus, $\mathbf{G}_2$ and $\mathbf{G}_3$ are indistinguishable from any attacker $\mathcal{A}$'s view unless conditions (1) - (6) happen with probability at most $q_{\mathrm{E}}(q_{\mathrm{JG}}+1)\delta_{\mathsf{DS}} + q_{\mathrm{E}}\delta_{\mathsf{maKEM}}$. It holds that

$$\Pr[\mathbf{G}_2(\mathcal{A}) = \mathsf{true}] \leq \Pr[\mathbf{G}_3(\mathcal{A}) = \mathsf{true}] + q_{\mathrm{E}}(q_{\mathrm{JG}}+1)\delta_{\mathsf{DS}} + q_{\mathrm{E}}\delta_{\mathsf{maKEM}}$$

Game $\mathbf{G}_4$. This game is identical to $\mathbf{G}_3$ except that the attacker $\mathcal{A}$ can never win via $*\mathsf{winLdrLeave}$ helper function at Line 491 in the LEADERLEAVE oracle.

Note that whenever a leader itself leaves the group, its session state will always be erased at Line 372. This means that the attacker $\mathcal{A}$ can never trigger win at Line 579.

Note that every newly joined participant that has not been announced to the group has a roster that only includes the leader's record. When the leader leaves, the participant's roster will be empty, which leads its session state also be erased at Line 384. This means that the attacker $\mathcal{A}$ can never trigger win at Line 582.

Note also that every other participants will removes the leader's record from the roster at the same time at Line 383, leading their rosters also same. In particular, this means, they also agree on the same leaders. This means that the attacker $\mathcal{A}$ can never trigger win at Line 582.

Thus, $\mathbf{G}_3$ and $\mathbf{G}_4$ are indistinguishable from any attacker $\mathcal{A}$'s view unless conditions (1) - (6) happen with probability at most $q_{\mathrm{E}}(q_{\mathrm{JG}}+1)\delta_{\mathsf{DS}} + q_{\mathrm{E}}\delta_{\mathsf{maKEM}}$. It holds that

$$\Pr[\mathbf{G}_3(\mathcal{A}) = \mathsf{true}] = \Pr[\mathbf{G}_4(\mathcal{A}) = \mathsf{true}]$$

Game $\mathbf{G}_5$. This game is identical to $\mathbf{G}_4$ except that the attacker $\mathcal{A}$ can never win via $*\mathsf{winPartiLeave}$ helper function at Line 516 in the PARTICIPANTLEAVE oracle.

Note that whenever a participant itself leaves the group, its session state will always be erased at Line 379. This means that the attacker $\mathcal{A}$ can never trigger win at Line 587.

Note also that the leader will also remove this participant's record at Line 375. Due to the uniqueness of sid that we ensured in $\mathbf{G}_2$, we know that the the attacker $\mathcal{A}$ can never trigger win at Line 590.

Thus, $\mathbf{G}_4$ and $\mathbf{G}_5$ are identical from any attacker $\mathcal{A}$'s view. It holds that

$$\Pr[\mathbf{G}_4(\mathcal{A}) = \mathsf{true}] = \Pr[\mathbf{G}_5(\mathcal{A}) = \mathsf{true}]$$

Final Analysis. In the final $\mathbf{G}_5$, the attacker cannot trigger $\mathsf{win} \leftarrow \mathsf{true}$ throughout the game. Thus, it holds that

$$\Pr[\mathbf{G}_5(\mathcal{A}) = \mathsf{true}] = 0$$

Combing all statements above, the proof is concluded by:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{Corr}}_{\mathsf{CGKD}}(\mathcal{A}) \leq & (q_{\mathrm{IG}} + q_{\mathrm{RegP}})^2 \cdot 2^{-\lambda} + q_{\mathrm{JG}}\delta_{\mathsf{PAKE}} + q_{\mathrm{JG}}\delta_{\mathsf{MAC}} + q_{\mathrm{E}}\delta_{\mathsf{maKEM}} \\ & + (q_{\mathrm{E}}q_{\mathrm{JG}} + q_{\mathrm{E}} + q_{\mathrm{JG}})\delta_{\mathsf{DS}}\end{aligned}$$

Here, λ denotes the security parameter.

L Changelog

– Version 1.0: June 4, 2026.
– Version 1.1: June 15, 2026:
  - Added changelog.
  - Fixed typos and improved editorial quality.
  - Added intuition for Lemma 1.

Table of Contents

On the State-Compromise Security of End-to-End Real-Time Group Communication . . . 1
Mang Zhao[0009−0002−4720−6551] and Qian Wang ✉[0000−0002−8967−8525]
1 Introduction . . . 1
2 Background . . . 4
2.1 Real-Time Group Communication . . . 4
2.2 Asynchronous Group Communication . . . 5
2.3 Multi-Recipient Cryptography . . . 6
3 Multi-Recipient Authenticated Key Encapsulation Mechanism . . . 7
3.1 Syntax . . . 7
3.2 Security Notions . . . 8
3.3 Instantiations . . . 9
4 Continuous Group Key Distribution . . . 11
5 CGKD Security Model . . . 15
5.1 Strong Security Guarantees . . . 15
5.2 Security Model . . . 17
6 Our Generic Construction . . . 23
7 Summary . . . 29
Acknowledgements. . . . 30
A Preliminaries . . . 33
A.1 Message Authentication Code . . . 33
A.2 (Randomized) Authenticated Encryption . . . 33
A.3 Digital Signature . . . 35
A.4 Public Key Encryption . . . 36
A.5 Hash Functions and Pseudorandom Functions . . . 37
A.6 lr-prf-ODH Assumption . . . 37
A.7 Password-Authenticated Key Exchange . . . 38
B Correctness Analysis for CGKD . . . 40
B.1 Detailed Correctness Model for CGKD . . . 40
B.2 Correctness of Our CGKD Construction . . . 44
C Detailed Description for Our CGKD Construction . . . 44
D Proof of Theorem 1 . . . 48
E Proof of Theorem 2 . . . 49
F Proof of Theorem 3 . . . 51
G Proof of Theorem 4 . . . 53
H Proof of Theorem 5 . . . 54
I Proof of Theorem 6 . . . 56
J Proof of Theorem 7 . . . 57
J.1 Case 1: Attacker wins via ∗winAuthrz. . . . 58
J.2 Case 2: Attacker wins via ∗winGKRosAuthn. . . . 62
J.3 Case 3: Attacker wins via ∗winChl. . . . 64
J.4 Helping Lemma 1 . . . 67
K Proof of Theorem 8 . . . 73
L Changelog . . . 76