A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs*

Keitaro Hashimoto
Tokyo Institute of Technology
AIST
Tokyo, Japan
hashimoto.k.au@m.titech.ac.jp

Shuichi Katsumata†
AIST
Tokyo, Japan
shuichi.katsumata@aist.go.jp

Eamonn Postlethwaite‡
CWI
Amsterdam, The Netherlands
ewp@cwi.nl

Thomas Prest§
PQShield SAS
Paris, France
thomas.prest@pqshield.com

Bas Westerbaan¶
Cloudflare
Amsterdam, The Netherlands
bas@westerbaan.name

ABSTRACT

Continuous group key agreements (CGKAs) are a class of protocols that can provide strong security guarantees to secure group messaging protocols such as Signal and MLS. Protection against device compromise is provided by *commit messages*: at a regular rate, each group member may refresh their key material by uploading a commit message, which is then downloaded and processed by all the other members. In practice, propagating commit messages dominates the bandwidth consumption of existing CGKAs.

We propose Chained CmPKE, a CGKA with an asymmetric bandwidth cost: in a group of N members, a commit message costs $O(N)$ to upload and $O(1)$ to download, for a total bandwidth cost of $O(N)$. In contrast, TreeKEM [19, 24, 76] costs $\Omega(\log N)$ in both directions, for a total cost $\Omega(N \log N)$. Our protocol relies on generic primitives, and is therefore readily post-quantum.

We go one step further and propose post-quantum primitives that are tailored to Chained CmPKE, which allows us to cut the growth rate of *uploaded* commit messages by two or three orders of magnitude compared to naive instantiations. Finally, we realize a software implementation of Chained CmPKE. Our experiments show that even for groups with a size as large as $N = 2^{10}$, commit messages can be computed and processed in less than 100 ms.

KEYWORDS

secure messaging; continuous group key agreement; post-quantum assumptions; (committing) multi-recipient PKE

1 INTRODUCTION

Secure messaging applications have seen an exponential growth in use over the last decade. For example, WhatsApp reports a user base of two billion [27]. From a security point of view, secure (group) messaging is subject to some specific constraints: end-to-end encryption, asynchrony, long sessions and – in the group setting – a number of users as large as $N \leq 50000$ [76, §2.4].

End-to-end encryption (E2EE) informally requires that no entity besides the participants in a conversation can access in the clear the contents of said conversation. The use of E2EE can be concretely motivated by the documented attempts of government agencies to access conversations of Lavabit [48] and Signal [79] users by issuing subpoenas to the providers. A common abstraction for secure (group) messaging is to model the delivery service as a public bulletin board, hence minimizing the level of trust and interactivity that users expect from it. As we will discuss in this paper, making the server slightly more active in a controlled manner can benefit efficiency, while maintaining the same level of (dis)trust.

In a secure (group) conversation over e.g., Signal, the session may last years, there may be hundreds of users, and they may not be online simultaneously. This stands in stark contrast to a TLS session, which is bounded in time and deals with two online users (server and client). It also raises new security issues. For a crude example, consider a conversation involving N participants over a span of t units of time. If each participant has an independent probability ϵ of being compromised over a unit of time, this conversation will have its contents compromised with probability $1 - (1 - \epsilon)^{Nt}$, which becomes significant as soon as $Nt = \Omega(1/\epsilon)$. This issue can be resolved by having each participant refresh their key material at a regular pace, thus limiting the scope of a compromise. This practice, called *ratcheting*, provides post-compromise security (PCS) and forward secrecy (FS) [10, 35, 38]. It also forms the basis for more sophisticated techniques [11, 12, 24] providing various levels of a stronger notion called post-compromise forward security (PCFS) [11–13].

Continuous Group Key Agreement. The notions of continuous (group) key agreement (CKA and CGKA) were put forward [9–13] to capture the particular setting that secure (group) messaging contends with, e.g., asynchrony and large groups, and achieve the security notions it requires, e.g., PCFS. In addition to representing a clean abstraction, CGKAs also include the complex cryptographic machinery of secure group messaging, and are therefore convenient objects to reason on.

The most widely academically discussed CGKA is TreeKEM [24]. It underlies the IETF draft standard for secure messaging, MLS [19, 76]. TreeKEM derives its name from its use of *ratchet trees* (bottom left of Fig. 1, p. 2), and a significant amount of research and

*This is the full version of the paper presented at ACM CCS 2021 [59].
†The author was supported by JST CREST Grant Number JPMJCR19F6.
‡The author was partially supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1). This research was started during an internship at PQShield.
§The author was partially supported by the UKRI Research Grant 104423.
¶This research was done while being employed by PQShield.

engineering effort has been undertaken to study the efficiency and security implications of this signature feature [9, 11–13, 25, 83].

The most recent iterations of TreeKEM (i.e., after version 8 on MLS) follow a "*propose-and-commit*" flow, in which members of a group may propose to add new members, remove existing ones or update their own keys, by sending *proposal messages*. These proposals only take effect when a group member initiates a new epoch by transmitting a *commit message*, which simultaneously validates a list of indicated proposals.

Bandwidth and Commit Messages. In order to realize PCFS, commit messages in TreeKEM include $\lceil \log N \rceil$ encryption keys and at least as many ciphertexts[1] (see Fig. 1), where $\log x$ denotes the logarithm in base 2 of x. As group members are arranged as the leaves of a binary tree, these encryption keys and ciphertexts allow all recipients to derive a fresh common group secret comSecret (*commit secret*), which is the root of the tree.

Let us discuss bandwidth consumption through three metrics: the cost of an upload and download, and the total cost. We focus on the bandwidth cost of the commit messages of TreeKEM, as they are the dominant term. Indeed, commit messages are the only cryptographically-heavy messages that need to be uploaded and downloaded at a regular rate, and each of them has a size of $\Omega(\log N)$. This therefore represents both the *upload* and *download* cost. If each member of a group sends a single commit message in a given time span, then they each must also download $(N-1)$ commit messages, for a *total* bandwidth cost of $\Omega(N \log N)$ per user.[2] For large groups, this can become significant. Ironically, large groups are also those that need the PCFS provided by commit messages the most, since their likelihood of compromise during a time span is higher.

This tension between security and bandwidth efficiency can be amplified by two factors: post-quantum cryptography, and the fact that secure messaging applications target mobile devices. In general, post-quantum cryptographic primitives consume more bandwidth than their classical counterparts by at least an order of magnitude, if not more: for example, all parameter sets of Classic McEliece entail encapsulation keys of at least 255 kibibytes (KiB). On the other hand, bandwidth can be a scarce resource over mobile devices, especially for users with limited mobile plans that charge an extra fee or block access to the network once a data cap has been reached.[3] To give a concrete example, instantiating TreeKEM with Classic McEliece in a group of $N = 256$ members will deplete a 1 GiB mobile plan once each user has sent *two* commit messages. This motivates the need for CGKA protocols and post-quantum primitives that remain efficient and secure for large groups. We note that mobile plan providers typically calculate data usage by treating uploaded and downloaded data as equal, and that being temporarily blocked from, or asked to pay more to continue to access, the mobile infrastructure is perhaps the most significant way in which bandwidth usage affects user experience. Hence our bandwidth cost model: *downloading one byte costs as much as uploading one byte*.

One could argue that assigning different weights to uploaded and downloaded data would be more appropriate, since uploading speed may be lower than downloading speed [81]. We believe this speed-based distinction is not necessary, for two reasons. First, the bandwidth bottleneck of our CGKA resides in commit messages, which are uploaded and downloaded in a manner that is invisible to end users. Second, all our instantiations of our protocol achieve uploaded commit messages of less than 50KiB for groups of at most 1024 users (see Fig. 6), which, even in countries with low uploading speed (as of July 2021, the slowest is Afghanistan, with 2.90 Mbps [81]), can be uploaded in less than 0.2 second. Both facts point to a minimal impact of uploading and downloading speeds on user experience.

1.1 Our Contributions

We propose a new CGKA called Chained CmPKE along with a formal security proof (Sec. 4). The main technical tools we leverage are the existence of very efficient post-quantum multi-recipient PKEs (mPKE, Sec. 5), and the notion of a committing mPKE (CmPKE, Sec. 3). We believe these tools may be of independent interest.

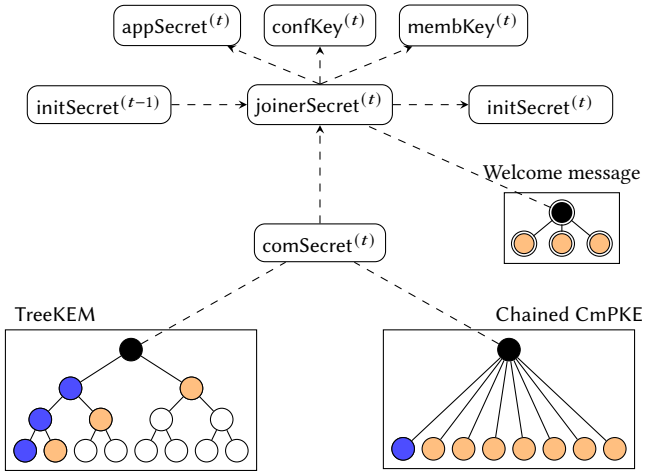


Figure 1: Initialization of a new epoch t, here with a group of $N = 8$ members. A dashed arrow from X to Y means that Y is computed by passing X (and possibly other values) to a HKDF, a dashed line means that $X = Y$.
Here, the leftmost user in the TreeKEM (resp. Chained CmPKE) box initiates a new epoch by issuing a commit message, which contains one encryption key for each ● node, and one PKE (resp. CmPKE) ciphertext for each ● node. Each recipient in the current group is able to compute comSecret$^{(t)}$, which corresponds to the root ●.
A commit message may include a welcome message, which contains ciphertexts (◎) encrypting joinerSecret$^{(t)}$ (●) under the encryption key of each newly added member.

[1] A documented property [9] of TreeKEM is that the number n_c of ciphertexts depends on the topology of the ratcheting tree, which might contain *blank* nodes. This number is $\lceil \log N \rceil$ in the best case, but may degrade to $N - 1$ for heavily blanked trees.

[2] Downloading and processing commit messages is important for security *and* functionality: a member refusing to download a commit message will be unable to decrypt subsequent messages.

[3] Surveys on mobile data pricing [31] are interesting in that regard. The median cost of 1 GiB of mobile data is on average (across all countries) $4.07. Mobile plans that cost more than $20.00 / GiB are reported in 89 countries and, in expensive countries, "*People are often buying data packages of just a tens of megabytes at a time*" [31]. This illustrates that mobile data can be a limited and expensive resource.

1.1.1 The Chained CmPKE Protocol. At a very high level, our protocol is inspired by the Chained mKEM protocol [23, 25]. One way of interpreting Chained mKEM is as TreeKEM with a tree of arity N and depth 1. This makes the size of *uploaded* commit messages scale as $O(N)$, see the bottom right of Fig. 1. The main conceptual difference between our Chained CmPKE[4] and variations of TreeKEM (including Chained mKEM) is that we no longer consider the delivery service as a public bulletin board, and instead allow it to sanitize commit messages in a straightforward manner by delivering to each group member the strict amount of data they need, while maintaining the same level of (dis)trust. In our case, this means that a member i may only receive the ciphertext ct_i that they can decrypt.

Our first line of research realizes this sanitizability by authenticating all ciphertexts with a single signature. To achieve this we rely on the notion of a CmPKE, essentially a multi-recipient PKE augmented with a commitment T. (Sec. 1.1.2). CmPKEs allow us to reduce the size of *downloaded* commit messages from $O(N)$ to $O(1)$. Effectively, this also reduces the *total bandwidth* cost of transmitting a commit message to $O(N)$, instead of $O(N^2)$ for Chained mKEM and $\Omega(N \log N)$ for TreeKEM. Alternatively, one could use a Merkle tree to authenticate all ciphertexts, as in Certificate Transparency [69]. However, each downloaded commit message would need to include a membership proof of size $O(\log N)$, in contrast to our $O(1)$ solution.

In a second line of research, we minimize the concrete cost of *uploaded* commit messages, which is $O(N)$ and larger than $\Omega(\log N)$ as for TreeKEM, by proposing and analyzing new efficient post-quantum mPKEs. (Sec. 1.1.3). As we show in Sec. 3, we can generically transform any mPKEs into CmPKEs with minimal overhead, thus we simply focus on mPKEs.

Compared to a naive instantiation of mPKEs using standard single-recipient PKEs, our mPKEs make the commit messages asymptotically smaller (in N) by factors of between 16 (Kyber512 vs. Ilum512) and 71 (Frodo640 vs. Bilbo640). In fact, while our *uploading* cost scales asymptotically as $O(N)$, it still compares favorably to the $\Omega(\log N)$ solution of TreeKEM in *concrete* efficiency, even for groups with hundreds of users.

Our bandwidth savings are summarized in an asymptotic manner in (Tab. 1, p. 10), and in a concrete manner in (Fig. 6, p. 12) and (Fig. 7, p. 13). While Fig. 6 illustrates the upload and download cost, Fig. 7 illustrates the total bandwidth cost. Compared to TreeKEM-based equivalents, our instantiations of Chained CmPKE have consistently better *upload* costs for groups of less than 200 users indicating that $O(N)$ solutions can be practically efficient. In addition, our *download* and *total* costs are better by factors of $\Omega(\log N)$ and performs well for any number of users.

1.1.2 Committing mPKE*s.* We introduce the notion of a *committing* mPKE, or CmPKE. First, a (decomposable) multi-recipient PKE (mPKE) [66] takes as input a message M and a list of N encryption keys, and outputs a multi-recipient ciphertext $(\mathsf{ct}_0, (\widehat{\mathsf{ct}}_i)_{i\in[N]})$. Each recipient $i \in [N]$ is able to recover M by decrypting $(\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$.

[4]We consciously use the term Chained CmPKE rather than Chained CmKEM since we believe PKE better reflects the protocol description.

The syntax of a CmPKE is mostly similar to that of an mPKE, however it requires one additional component. The encryption procedure of a CmPKE outputs $(\mathsf{T}, (\mathsf{ct}_i)_{i\in[N]})$, where T is called a *commitment*. Decryption then works by taking the commitment-ciphertext pair $(\mathsf{T}, \mathsf{ct}_i)$. We require T to (a) have a size *independent* of the number of recipients N, and (b) be *commitment-binding*, which means informally that T is bound to a unique message. This notion resembles committing AEADs [50], however we operate in a different setting (multi-user vs. single-user) and with a different motivation (bandwidth efficiency vs. abuse reporting).

We show how to build a CmPKE from an mPKE [66] and a *key-committing* SKE [2, 44, 46, 50], which can itself be built using standard symmetric primitives [2]. Compared to the base mPKE, the overhead is minimal: $\mathsf{ct}_i = \widehat{\mathsf{ct}}_i$, and T is formed of ct_0 and a term of size 2κ bits, which is no larger than a hash digest.

In our protocol, after computing a CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]})$, the sender of a commit message does not authenticate the whole ciphertext, only T. The server sends $(\mathsf{T}, \mathsf{ct}_i)$ to each recipient i, and the commitment-binding property allows i to indirectly verify the authenticity of the message encrypted in ct_i. As a result, the download cost of a commit message is $O(1)$ for all recipients.

1.1.3 More Efficient mPKE*s.* An mPKE allows one to encrypt a common message to N recipients more efficiently than the naive solution of computing and sending N individual ciphertexts in parallel. Indeed, as each recipient receives $(\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$, mPKEs provide asymptotic bandwidth savings if $\widehat{\mathsf{ct}}_i$ is smaller than a regular, single-recipient ciphertext ct would be.

While mPKEs based on classical assumptions [20, 68] realize $|\widehat{\mathsf{ct}}_i|/|\mathsf{ct}| = 1/2$, existing PKEs based on the post-quantum problems LWE, LWR, SIDH and CSIDH were recently adapted to the mPKE setting in [66]. These mPKEs achieve ratios $|\widehat{\mathsf{ct}}_i|/|\mathsf{ct}|$ between $1/5$ and $1/169$, which could potentially translate into inversely proportional bandwidth savings. The work of [66] has two shortcomings; (a) their mPKEs are direct transpositions of existing PKEs, which were not necessarily designed to minimize $|\widehat{\mathsf{ct}}_i|$, and (b) it does not study the concrete impact of the mPKE setting on cryptanalysis. We address these two shortcomings via a two-pronged approach.

On the constructive side, we note that minimizing the size of uploaded commit messages gives a different optimization target to that of PKEs, specifically we wish to minimize $|\widehat{\mathsf{ct}}_i|$, even at the expense of some controlled growth of $|\mathsf{ct}_0|$. We therefore attempt to improve upon the efficiency gains already reported in [66] by revisiting the designs of the NIST submissions [21, 74, 77] with our new optimization target in mind. To achieve this we rely on well known techniques such as coefficient dropping and modulus rounding. We arrive at three new parametrizations; a variant of Frodo640 [74] called Bilbo640, a variant of Kyber512 [77] called Ilum512, and a variant of LPRime653 [21] called LPRime757. Compared to using the NIST submissions as mPKEs we reduce $|\widehat{\mathsf{ct}}_i|$ by 60–80%, which translates to an identical asymptotic reduction in the size of uploaded commit messages. These parametrizations are close to optimal in the sense that $|\widehat{\mathsf{ct}}_i| \in (\kappa, 3\kappa]$ bits. Since in the Lindner–Peikert framework, $\widehat{\mathsf{ct}}_i$ encodes all the information about the message (in our case, a κ-bit symmetric key), it seems difficult to beat the κ-bit threshold without new techniques.

On the cryptanalytic side we must consider the effect of the mPKE setting on the attack surface. In [66] theoretical, reduction based, assurances for the security of the mPKE construction are given. However, the concrete security of the mPKEs dervied from NIST submissions is assumed to follow from their concrete security analyses as PKEs. As an example of differences between the two settings, variants of the Arora–Ge [5, 15] and BKW [28] attacks are typically irrelevant to lattice-based PKEs, since they require more 'samples' than provided by the single ciphertext of the PKE, ct. However, in the mPKE setting, the *per recipient* $\widehat{\mathsf{ct}}_i$ ciphertext components each provide samples for an adversary. Therefore the Arora–Ge and BKW attacks should be considered in a concrete security analysis of mPKE parameters. In App. G, we describe these attacks in more detail, and provide estimates for the concrete security of our reparametrizations in a cryptanalytic model tailored to the mPKE setting. This model targets NIST Security Level I. Schemes satisfying this are conjectured to have comparable security to AES-128 against classical and quantum adversaries. Interestingly, our attempts to improve the efficiency of our mPKEs via reparametrizing NIST submissions, specifically our use of heavy modulus rounding on the $\widehat{\mathsf{ct}}_i$, naturally *hardens* our parametrizations against these sample heavy attacks. To display the importance of an mPKE-focused cryptanalysis, we provide an artificial 'Kyber like' parameter set that is *almost* secure as a PKE, but insecure in our mPKE cryptanalytic model.

1.1.4 Security of Chained CmPKE. Finally, we provide a formal proof establishing that our Chained CmPKE is as secure as TreeKEM. We adopt the state-of-the-art UC security model presented by Alwen et al. [13] that was used to analyze the TreeKEM version 10 in MLS, which is itself an extension of [12]. In addition to party corruptions (i.e., compromise party's secret and group secrets), the model captures *active adversaries* who may tamper with or inject messages and deliver messages in an arbitrary order, and *malicious insiders* who may interact with the PKI on behalf of the corrupted parties. On a technical front, to model the sanitizing of the commit messages by the delivery service, we extend the ideal functionality in [13] and modify how the ideal functionality maintains the so-called *history graph*. Our security model is a strict generalization of prior models as it captures them as special cases.

1.2 Related Works

Secure Group Messaging. TreeKEM [24] originates from *Asynchronous Ratcheting Trees* (ART) introduced by Cohn-Gordon et al. [37]. To date, the TreeKEM discussed in MLS has gone through 11 versions, some of which have undergone formal security analyses. For example, Alwen et al. [11] and Bhargavan et al. [25] analyzed the security of TreeKEM version 7. The former proved its security based on a game-based security model for CGKA, and the latter presented a mechanized security proof. Recently, Alwen et al. [13] analyzed TreeKEM version 10, which adopts the 'parent hash' and 'tree-signing' mechanisms and showed that it is secure against active and insider adversaries.

In addition to the standard TreeKEM discussed in MLS, variants of TreeKEM have been proposed. *Tainted* TreeKEM [9] enjoys efficiency advantages for large groups maintained by a small number of 'administrators'. Re-randomized TreeKEM [11] and TreeKEM with active security [12] improve the PCFS property against passive and active adversaries, respectively, but require relatively heavy cryptographic primitives. Finally, *Causal* TreeKEM [83] supports concurrent changes to the group state but currently has no accompanying formal security proof.

Secure Two-Party Messaging. Secure messaging in the simpler two-party setting has also been an active area of research, motivated by the Signal protocol. The first full security analysis of the Signal protocol is provided in [35, 36]. The notion of *Continuous Key Agreement* (CKA) (that is, CGKA in the two-party setting) is studied in [10]. This generalizes the public-key ratchet of Signal's Double Ratchet protocol [72]. The X3DH protocol [73] of the Signal protocol, used to establish the initial secret key required for CKA, is studied in [29, 58]. As these works provide a generic construction of each building blocks from post-quantum assumptions, this results in a post-quantum secure messaging for the two-party setting.

Other Real-World Post-Quantum Protocols. In the context of post-quantum protocols, an ongoing trend is to propose protocols that are tailored to the performance profiles of post-quantum schemes. For example, KEMTLS [78] posits that post-quantum signatures are generally less efficient than post-quantum KEMs. Similarly, McTiny [22] and Post-Quantum WireGuard [62] exploit the strengths of Classic McEliece [3] (long security track record, short ciphertexts) while mitigating its main weakness (large public keys). Our construction follows the same principles by harnessing the existence of very efficient post-quantum mPKEs.

2 PRELIMINARIES

2.1 One-Time IND-CCA SKE

We use the standard syntax for SKE. Let $\mathcal{K}$ and $\mathcal{M}$ denote the key and message space, respectively. We denote by Enc_s and Dec_s the encryption and decryption algorithms, respectively, and as standard, we assume perfect correctness. Details are provided in App. A.2. We only require *one-time* IND-CCA security for SKEs in this work, formally defined as follows.

Definition 2.1 (One-Time IND-CCA). A SKE is *one-time* IND-CCA *secure* if for all PPT adversary $\mathcal{A}$, we have $|\Pr[(b,\mathsf{k}) \hookleftarrow_\$ \{0,1\}\times\mathcal{K}, (\mathsf{M}_0,\mathsf{M}_1) \leftarrow \mathcal{A}(1^\kappa), \mathsf{ct}^* \leftarrow \mathsf{Enc}_\mathsf{s}(\mathsf{k},\mathsf{M}_b), b' \leftarrow \mathcal{A}^{\mathcal{C}(\cdot)}(\mathsf{ct}^*) : b=b'] - 1/2| \leq \mathsf{negl}(\kappa)$, where $\mathcal{C}(\mathsf{ct})$ returns $\mathsf{Dec}_\mathsf{s}(\mathsf{k},\mathsf{ct})$ if $\mathsf{ct} \neq \mathsf{ct}^*$ and $\perp$ otherwise.

We also define key commitment for a SKE [46] which roughly states that it is difficult to find two secret keys that correctly decrypt the same ciphertext (to possibly different messages). As in prior works [2, 44, 46, 50], we define this notion by providing the (non-uniform) adversary oracle access to Enc_s and Dec_s, where we implicitly assume these two algorithms are implemented using an internal hash function modeled as a random oracle.

Definition 2.2 (Key Commitment). A SKE has *key commitment* if for all PPT adversary $\mathcal{A}$, we have $\Pr[(\mathsf{k}_0,\mathsf{k}_1,\mathsf{ct}) \leftarrow \mathcal{A}^{\mathsf{Enc}_\mathsf{s},\mathsf{Dec}_\mathsf{s}}(1^\kappa), (\mathsf{M}_b \leftarrow \mathsf{Dec}_\mathsf{s}(\mathsf{k}_b,\mathsf{ct}))_{b\in\{0,1\}} : \mathsf{M}_0 \neq \perp \wedge \mathsf{M}_1 \neq \perp] \leq \mathsf{negl}(\kappa)$.

Viewing SKE as (a weakened version of) AEAD, we can use [2, Sec. 5.2.] to generically transform any IND-CCA SKE, regardless of it being one-time secure or not, to one with key commitment. The transform only adds an additional κ bits of overhead to the

original ciphertext: to encrypt, the key committing scheme expands $k_{enc} \leftarrow H_{enc}(key)$ and $k_{com} \leftarrow H_{com}(key)$, runs $Enc_s(k_{enc}, M)$ and outputs the ciphertext as (ct, k_{com}). Here H_{enc} and H_{com} are modeled as random oracles. Key committing simply follows from the collision resistance of H_{com}.

2.2 Decomposable Multi-Recipient PKE

Decomposable multi-recipient PKE (mPKE) was introduced in [66]. Similarly to a standard mPKE [17, 68, 80], a decomposable mPKE allows a user to send a message to multiple recipients more efficiently than naively running a standard PKE to the individual recipients. The main difference between a decomposable and non-decomposable mPKE is whether the encryption algorithm can be decomposed into a recipient dependent and independent part. In [66] it was shown that many assumptions known to imply PKE (e.g., DDH, LWE, SIDH) can naturally be used to construct an IND-CPA decomposable mPKE. In this work, we introduce a stronger security notion than those provided in [66] where we allow the adversary to adaptively corrupt users during the IND-CPA security game. Looking ahead, this notion will be important when we target an *adaptively* secure CGKA.

Definition 2.3 (Decomposable Multi-Recipient Public Key Encryption). A (single-message) decomposable multi-recipient public key encryption (mPKE) over a message space $\mathcal{M}$ consists of the following algorithms:

- $\mathsf{mSetup}(1^\kappa) \to \mathsf{pp}$: On input the security parameter 1^κ, it outputs a public parameter pp.
- $\mathsf{mGen}(\mathsf{pp}) \to (\mathsf{ek}, \mathsf{dk})$: On input a public parameter pp, it outputs a pair of encryption key and decryption key $(\mathsf{ek}, \mathsf{dk})$.
- $\mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M}; \mathsf{r}_0, (\mathsf{r}_i)_{i\in[N]}) \to \vec{\mathsf{ct}} = (\mathsf{ct}_0, (\widehat{\mathsf{ct}}_i)_{i\in[N]})$: The (decomposable) encryption algorithm running with randomness $(\mathsf{r}_0, \mathsf{r}_1, \cdots, \mathsf{r}_N)$, splits into a pair of algorithms $(\mathsf{mEnc}^{\mathsf{i}}, \mathsf{mEnc}^{\mathsf{d}})$:
  - $\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0) \to \mathsf{ct}_0$: On input a public parameter pp and randomness r_0, it outputs an (encryption key *independent*) ciphertext ct_0.
  - $\mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{M}; \mathsf{r}_0, \mathsf{r}_i) \to \widehat{\mathsf{ct}}_i$: On input a public parameter pp, an encryption key ek_i, a message $\mathsf{M} \in \mathcal{M}$, and randomness $(\mathsf{r}_0, \mathsf{r}_i)$, it outputs an (encryption key *dependent)* ciphertext $\widehat{\mathsf{ct}}_i$.
- $\mathsf{mDec}(\mathsf{dk}, \mathsf{ct}_i) \to \mathsf{M}$ or $\perp$: On input a decryption key dk and a ciphertext $\mathsf{ct}_i = (\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$, it outputs either $\mathsf{M} \in \mathcal{M}$ or $\perp \notin \mathcal{M}$.

Observe that any standard PKE can be used to construct a decomposable mPKE in the obvious way where $\mathsf{mEnc}^{\mathsf{i}}$ is the null-function and $\mathsf{mEnc}^{\mathsf{d}}$ is the encryption algorithm of the PKE. So naturally, the main motivation for mPKE will be to reuse a large portion of the encryption randomness r_0 for all recipients and to obtain a more efficient scheme compared to the obvious solution. The asymptotic behavior will be the same as the obvious solution (i.e., the total ciphertext size is $O(N)$) but the concrete size can be drastically reduced (see Sec. 5 for more details). We require the standard notion of correctness and ciphertext-spreadness [49], where the latter informally states that the ciphertext has high min-entropy. Due to space constraints, definitions are given in App. A.3. We also define indistinguishability of chosen plaintext attacks (IND-CPA) *with adaptive corruption* for a decomposable mPKE.

*Definition 2.4 (*IND-CPA*).* The security notion is defined by the game in Fig. 2, where we say the adversary $\mathcal{A}$ *wins* if the game outputs 1. A decomposable mPKE is IND-CPA *secure with adaptive corruption* if for all PPT adversaries $\mathcal{A}$, we have $|\Pr[\mathcal{A} \text{ wins}] - 1/2| \leq \mathsf{negl}(\kappa)$. If $\mathcal{A}$ is not given access to the corruption oracle $\mathcal{C}$, this game corresponds to standard IND-CPA security.

We show in Sec. 3.3 that any IND-CPA secure decomposable mPKE can be generically bootstrapped into one that is additionally secure against adaptive corruption with a minimal overhead.

3 COMMITTING MULTI-RECIPIENT PKE

We consider a strengthening of a standard mPKE which we coin a *committing* mPKE (CmPKE). The motivation for this is similar in spirit to those of key committing SKEs or AEADs [2, 44, 46, 50], where we ask a ciphertext to be bound to a unique key and message pair. Although it may sound like an obscure property at first glance, this property has been shown to be vital for establishing security in several practical applications such as Facebook Messenger [44], (see [2] for more examples). In a CmPKE, we extend this to the multi-user setting, which requires that if any of the recipients decrypt to a message M, then the other recipients should also decrypt either to M or to $\perp$. Informally, and unlike in the single-user setting, we allow a ciphertext to be decryptable by many recipients (i.e., many different keys) but enforce that their decryption values remain consistent if not $\perp$. Looking ahead, this is a natural property to desire when guaranteeing the weak robustness of a CGKA protocol (i.e., if a user receives a message then it should be consistent with all the other group members, provided that they can process the message).[5]

3.1 Definition

Definition 3.1 (Committing Multi-Recipient Public-Key Encryption). A (single-message) committing multi-recipient public-key encryption (CmPKE) over a message space $\mathcal{M}$ consists of the following four algorithms:

- $\mathsf{CmSetup}(1^\kappa) \to \mathsf{pp}$: On input the security parameter 1^κ, it outputs a public parameter pp.
- $\mathsf{CmGen}(\mathsf{pp}) \to (\mathsf{ek}, \mathsf{dk})$: On input a public parameter pp, it outputs a pair of encryption key and decryption key $(\mathsf{ek}, \mathsf{dk})$.
- $\mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M}) \to (\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]})$: On input a public parameter pp, N encryption keys $(\mathsf{ek}_i)_{i\in[N]}$, and a message $\mathsf{M} \in \mathcal{M}$, it outputs a commitment T and N ciphertexts $\vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]}$.
- $\mathsf{CmDec}(\mathsf{dk}, \mathsf{T}, \mathsf{ct}_i) \to \mathsf{M}$ or $\perp$: On input a decryption key dk, a commitment T, and a ciphertext ct_i, it outputs either $\mathsf{M} \in \mathcal{M}$ or $\perp \notin \mathcal{M}$.

Definition 3.2 (Correctness). A CmPKE is correct if $\Pr[\forall i \in [N], \mathsf{M} = \mathsf{CmDec}(\mathsf{dk}_i, \mathsf{T}, \mathsf{ct}_i)] \geq 1 - \mathsf{negl}(\kappa)$ holds for all $N \in \mathsf{poly}(\kappa)$ and $\mathsf{M} \in \mathcal{M}$, where the probability is taken over $\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$,

[5] Since weak robustness of the CGKA protocol is implicitly taken care of by the confirmation tag, the committing nature of mPKE is not explicitly required. However, considering the practical relevance of the "committing"-ness of SKE and AEAD, we believe this notion is worth formalizing as it may have values in other contexts.

$((\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{CmGen}(\mathsf{pp}))_{i\in[N]}$, and $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]}) \leftarrow \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M})$.[6]

Definition 3.3 (Succinctness). We say a CmPKE is *succinct* if in the above Def. 3.2, the commitment T (and all ciphertext ct_i) have size independent of the number of recipients N.

In this work, we only consider a succinct CmPKE so we omit it for simplicity. We define indistinguishability of chosen ciphertext attacks (IND-CCA) *with adaptive corruption* for CmPKE.

*Definition 3.4 (*IND-CCA *with Adaptive Corruption).* The security notion is defined by a game illustrated in Fig. 2, where we say the adversary $\mathcal{A}$ *wins* if the game outputs 1. A CmPKE is IND-CCA *secure with adaptive corruption* if for all PPT adversaries $\mathcal{A}$, we have $|\Pr[\mathcal{A}\ wins] - 1/2| \leq \mathsf{negl}(\kappa)$. If $\mathcal{A}$ is not given access to the corruption oracle C, this game corresponds to standard IND-CCA security.

Finally, we define commitment-binding which roughly says that the commitment T is implicitly bound to a unique message. The notion we consider is strong in the sense that the adversary can use an arbitrary decryption key rather a correctly generated one to break commitment-binding.

Definition 3.5 (Commitment-Binding). The security notion is defined by a game illustrated in Fig. 2, where we say the adversary $\mathcal{A}$ *wins* if the game outputs 1. A CmPKE is *commitment-binding* if for all PPT adversaries $\mathcal{A}$, we have $\Pr[\mathcal{A}\ wins] \leq \mathsf{negl}(\kappa)$.

Note that independently satisfying succinctness and commitment-binding is trivial. If we run a standard PKE in parallel for all N users and set $\mathsf{T} := \perp$, then we obtain a succinct scheme but this is clearly not commitment-binding. On the other hand, if we add a non-interactive zero-knowledge (NIZK) proof π to further prove that all the PKE ciphertexts encrypt the same message and set $\mathsf{T} := (\pi, \mathsf{ct}_1, \cdots, \mathsf{ct}_N)$ (as in the strongly robust TreeKEM variant of [12]), then we obtain a commitment-binding scheme but the commitment is no longer succinct. Therefore, the main non-triviality is making the commitment size $|\mathsf{T}|$ independent of the number of users, while simultaneously allowing the users to be convinced that if $(\mathsf{T}, \mathsf{ct}_i)$ decrypts to a valid message, then any other users' $(\mathsf{T}, \mathsf{ct}_j)$ will also decrypt to the same message (or to $\perp$).

3.2 Construction of CmPKE: IND-CCA without Adaptive Corruption

We provide a simple and efficient generic construction of an IND-CCA secure CmPKE (without adaptive corruption) from a decomposable IND-CPA secure mPKE and an one-time IND-CCA secure SKE following the Fujisaki–Okamoto transform generalized to the multi-recipient setting. This is illustrated in Fig. 3, where $\mathsf{G}_1, \mathsf{G}_2, \mathsf{H}$ are hash functions modeled as random oracles in the security proof. These oracles can be simulated by a single random oracle by using appropriate domain separation. Here, we assume the output space of H is identical to the secret key space $\mathcal{K}$ of the SKE. The correctness of this CmPKE follows immediately from the correctness of the decomposable mPKE and SKE. The following theorems assert the IND-CCA security and commitment-binding of the CmPKE. The proof for Thm. 3.6 is a standard adaptation of the KEM/DEM framework to the multi-user setting. The proof for Thm. 3.7 follows naturally from the key committing property of the underlying SKE. Both proofs are provided in Appendices B.2 and B.3, respectively.

Theorem 3.6. *The* CmPKE *in Fig. 3 is* IND-CCA *secure (resp. with adaptive corruption) assuming the* SKE *is one-time* IND-CCA *secure and the decomposable* mPKE *is* IND-CPA *secure (resp. with adaptive corruption) and ciphertext-spread.*

Theorem 3.7. *The* CmPKE *in Fig. 3 is commitment-binding assuming the* SKE *has key commitment.*

3.3 Construction of CmPKE: IND-CCA with Adaptive Corruption

The construction in Fig. 3 can be shown to be IND-CCA secure against adaptive corruption by allowing the reduction algorithm to guess the random choices made by the adversary. However, this results in a reduction loss as large as $2^{N \log N}$, where N is the number of recipients. This exponential reduction loss will then be inherited to the CGKA protocol. Although we are unaware of any concrete attacks that take advantage of this large reduction loss, it is natural to ask if there is an efficient *and* provably adaptively secure CmPKE (and hence CGKA) without incurring such a reduction loss.

Due to Thm. 3.6, we only need to focus on an IND-CPA secure *with adaptive corruption* decomposable mPKE. Below, we provide a simple generic transformation from any IND-CPA secure decomposable mPKE that is *not* secure against adaptive corruptions into one that is. The overhead is simply doubling the encryption key and ciphertext size, where the transform is a natural adaptation of the Katz–Wang technique [67]. The full detail of the construction is provided in App. B.4, Fig. 10. Due to the page limitation, we provide the proof of correctness and security in App. B.4.

4 OUR PROTOCOL: CHAINED CMPKE

We now present our protocol. At a conceptual level, there are two core differences with TreeKEM:

(1) Instead of being arranged as the leaves of a (binary) tree, group members are arranged in a set. This is similar to Chained mKEM [25]. Alternatively, it can be interpreted as TreeKEM using a tree of arity N and depth 1.
(2) Instead of being a passive bulletin board, the delivery service may *edit* a commit message uploaded by a member before forwarding it to any of the $(N-1)$ other group members.

The impact of the first change on *uploading* commit messages is illustrated in Fig. 1. A member may initiate a new epoch t by encrypting a commit secret $\mathsf{comSecret}^{(t)}$ directly to the $(N-1)$ encryption keys of the other group members using a CmPKE. There is no tree structure anymore and, as an immediate consequence, removing a user no longer leads to "blanking" a node.

The second change is implemented via the use of a CmPKE. Instead of signing the whole CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_i)_{i\in[N]})$ embedded in a commit message, the uploader of the message only signs T. The delivery service is expected to forward $(\mathsf{T}, \mathsf{ct}_i)$ to the recipient i. Any tampering on T by the server can be detected by a

[6] In the proof of our CGKA protocol, we require that the adversary cannot find a "bad" randomness that leads to a decryption error. Since we use a PRG modeled as a random oracle to expand the randomness, standard correctness immediately implies that no PPT adversary can find such bad randomness.

GAME IND-CPA

$C := \emptyset$
$\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$
foreach $i \in [N]$ **do**
  $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{mGen}(\mathsf{pp})$
$(\mathsf{M}_0, \mathsf{M}_1, S \subseteq [N]) \leftarrow \mathcal{A}^{C(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$
$b \leftarrow\$ \{0,1\}$
$\vec{\mathsf{ct}}^* \leftarrow \mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in S}, \mathsf{M}_b)$
$b' \leftarrow \mathcal{A}^{C(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \vec{\mathsf{ct}}^*)$
if $[C \cap S \neq \emptyset]$ **then**
  return b
return $[b = b']$

GAME IND-CCA

$C := \emptyset$
$\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$
foreach $i \in [N]$ **do**
  $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{CmGen}(\mathsf{pp})$
$(\mathsf{M}_0, \mathsf{M}_1, S \subseteq [N]) \leftarrow \mathcal{A}^{C(\cdot),\mathcal{D}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$
$b \leftarrow\$ \{0,1\}$
$(\mathsf{T}^*, \vec{\mathsf{ct}}^* := (\mathsf{ct}_i^*)_{i\in S}) \leftarrow \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in S}, \mathsf{M}_b)$
$b' \leftarrow \mathcal{A}^{C(\cdot),\mathcal{D}(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \vec{\mathsf{ct}}^*)$
if $[C \cap S \neq \emptyset]$ **then**
  return b
return $[b = b']$

GAME Commitment-Bind

$\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$
$(\mathsf{T}^*, (\mathsf{dk}_b, \mathsf{ct}_b)_{b\in\{0,1\}}) \leftarrow \mathcal{A}(\mathsf{pp})$
foreach $b \in \{0,1\}$ **do**
  $\mathsf{M}_b \leftarrow \mathsf{CmDec}(\mathsf{dk}_b, \mathsf{T}^*, \mathsf{ct}_b)$
if $\mathsf{dk}_0 = \mathsf{dk}_1$ **then**
  return $[\mathsf{ct}_0 \neq \mathsf{ct}_1] \wedge [\mathsf{M}_0 \neq \bot] \wedge [\mathsf{M}_1 \neq \bot]$
else
  return $[\mathsf{M}_0 \neq \mathsf{M}_1] \wedge [\mathsf{M}_0 \neq \bot] \wedge [\mathsf{M}_1 \neq \bot]$

Decapsulation Oracle $\mathcal{D}(i, \mathsf{T}, \mathsf{ct})$

req $(\mathsf{T}, \mathsf{ct}) \neq (\mathsf{T}^*, \mathsf{ct}_i^*)$
$\mathsf{M} \leftarrow \mathsf{CmDec}(\mathsf{dk}_i, \mathsf{T}, \mathsf{ct})$
return M

Corruption Oracle $C(i)$

$C \leftarrow C \cup \{i\}$
return dk_i

Figure 2: IND-CPA **with adaptive corruption of** mPKE**, and** IND-CCA **with adaptive corruption and commitment-binding of** CmPKE**. If the condition following req does not hold, the game terminates by returning a random bit.**

$\mathsf{CmSetup}(1^\kappa)$

$\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$
return pp

$\mathsf{CmGen}(\mathsf{pp})$

$(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{mGen}(\mathsf{pp})$
return $(\mathsf{ek}, \mathsf{dk})$

$\mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M})$

$\overline{\mathsf{M}} \leftarrow\$ \mathcal{M}$
$\mathsf{ct}_0 := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{M}}))$
foreach $i \in [N]$ **do**
  $\widehat{\mathsf{ct}}_i := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{M}}; \mathsf{G}_1(\overline{\mathsf{M}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{M}}))$
$\mathsf{k} := \mathsf{H}(\overline{\mathsf{M}})$
$\mathsf{ct}_s \leftarrow \mathsf{Enc}_s(\mathsf{k}, \mathsf{M})$
return $(\mathsf{T} := (\mathsf{ct}_0, \mathsf{ct}_s), \vec{\mathsf{ct}} := (\widehat{\mathsf{ct}}_i)_{i\in[N]})$

$\mathsf{CmDec}(\mathsf{dk}, \mathsf{T}, \mathsf{ct})$

$(\mathsf{ct}_0, \mathsf{ct}_s) \leftarrow \mathsf{T}$
$\overline{\mathsf{M}} := \mathsf{mDec}(\mathsf{dk}, (\mathsf{ct}_0, \mathsf{ct}))$
if $\overline{\mathsf{M}} = \bot$ **then return** $\bot$
$\mathsf{ct}_0' := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{M}}))$
$\widehat{\mathsf{ct}}' := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}, \overline{\mathsf{M}}; \mathsf{G}_1(\overline{\mathsf{M}}), \mathsf{G}_2(\mathsf{ek}, \overline{\mathsf{M}}))$
if $(\mathsf{ct}_0, \mathsf{ct}) \neq (\mathsf{ct}_0', \widehat{\mathsf{ct}}')$ **then return** $\bot$
return $\mathsf{Dec}_s(\mathsf{H}(\overline{\mathsf{M}}), \mathsf{ct}_s)$

Figure 3: An IND-CCA **secure** CmPKE **from an** IND-CPA **secure decomposable** mPKE **and a one-time** IND-CCA **secure** SKE**.**

recipient by checking the signature, and any tampering on ct_i can be detected during the CmPKE decryption procedure. In particular, it achieves the same level of security as provided by TreeKEM.

4.1 Description of Our Protocol

We reuse most of the terminology and function names used by [12, 13]. Due to space constraints, we only provide a high-level description of our protocol in Fig. 4, and highlight the major algorithmic changes below. A complete description is given in App. D.

Low-Level Primitives. The main changes relate to two classes of low-level primitives.

The first class captures procedures related to (left-balanced binary) trees: simple ones such as computing the parent or children of a node, determining whether it is the root, an internal node or a leaf, etc., or more complex ones such as computing its path, co-path or resolution. A list of 27 such procedures is given in [13, Tab. 1 and 3]. Removing binary trees trivializes or removes these procedures.

The second class relates to public-key encryption. As we replace a standard PKE by a CmPKE, the main effects are that the encryption procedure now takes as input a list of encryption keys $(\mathsf{ek}_i)_i$ instead of a single key, and the presence of a commitment T as an additional output (resp. input) of the encryption (resp. decryption) procedure.

Ripple Effects on Mid-Level Procedures. More notions and procedures related to trees are heavily simplified. For example, treeHash becomes memberHash, and its computation now entails hashing a set in lexicographical order, instead of a binary tree (`*set-tree-hash` becomes `*set-member-hash`). As there is no longer an internal node to authenticate, parentHash and its computation (`*set-parent-hash` and `*parent-hash`) are no longer necessary.

Impact at the Top Level. Since the group is no longer arranged in a binary tree structure but in a set, each user now possess a single encryption keypair instead of $\lceil \log N \rceil$. This simplifies top level procedures (`Commit`, `Process`, `Join`), which refresh these keypairs.

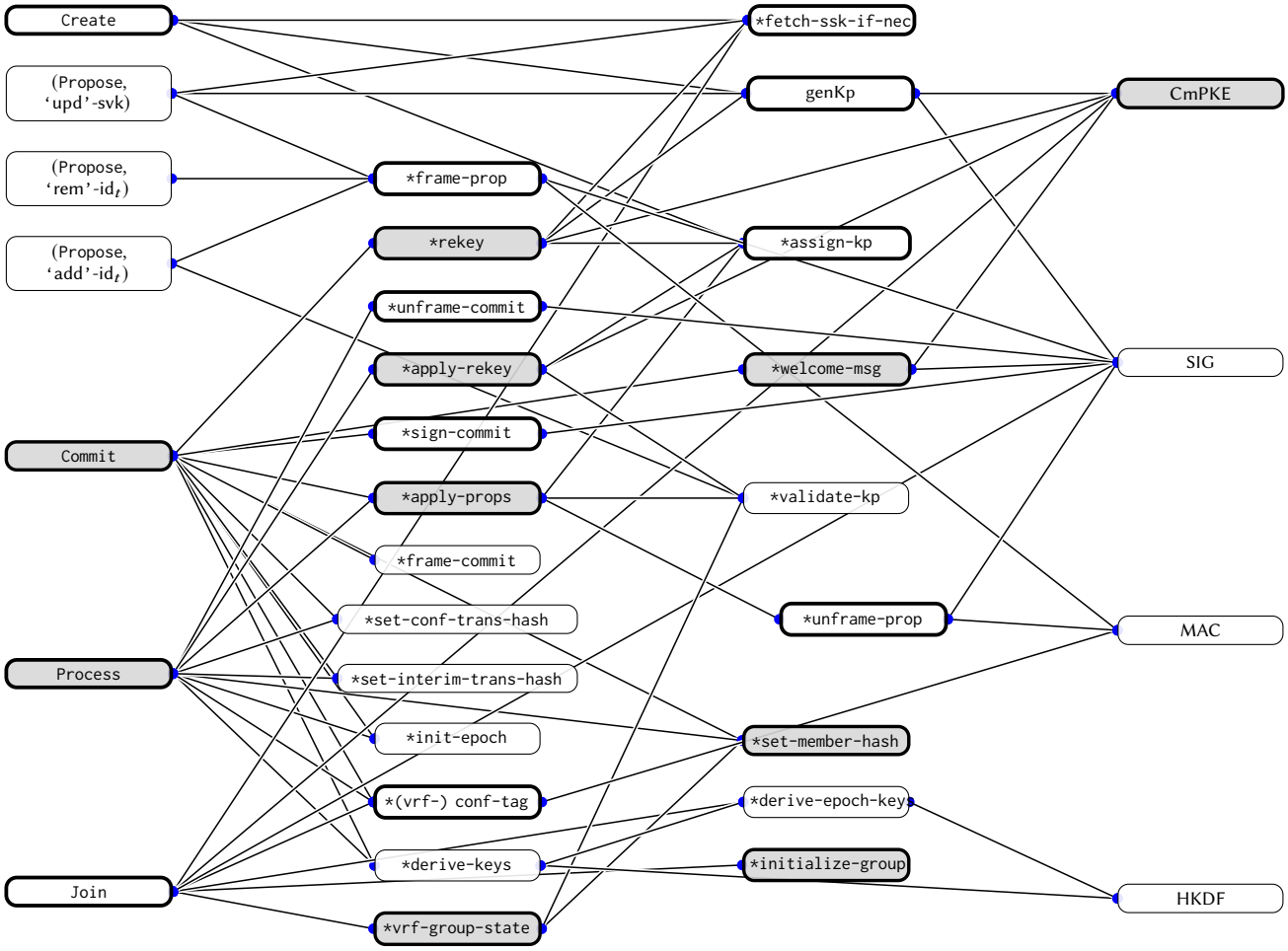


Figure 4: Call graph of Chained CmPKE. We use the notations `function`, `function` and `function` to denote functions that undergo respectively minimal, moderate and strong changes compared to [12, 13].

In TreeKEM, commit messages may contain encryptions of *path secrets* (to the resolution of the sibling of each concerned node, via `*rekey-path`) or a path secret on the least common ancestor node of the sender and each new group member (a common *joiner secret* is also sent to new group members, via `*welcome-msg`). Encryption of path secrets produces $\Omega(\log N)$ ciphertexts, see Fig. 1 and Footnote 1.

In Chained CmPKE, there is no path secret; instead, a common comSecret is encrypted to all recipients via a single call to CmEnc, producing one multi-recipient ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}'})_{\mathsf{id}' \in \mathsf{receivers}})$, see Fig. 1. Similarly, a common joinerSecret may be encrypted to newly added members. In each case, the sender of the commit message signs data that includes T, but not $\vec{\mathsf{ct}}$.

As input to `Process` and `Join`, receivers of a commit message will not receive the full package. Precisely, instead of including a full CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}'})_{\mathsf{id}' \in \mathsf{receivers}})$, the recipient id only downloads $(\mathsf{T}, \widehat{\mathsf{ct}}_{\mathsf{id}})$ from the server. We call this *selective* (or designated) downloading as the recipient only needs to download a part of the commit message it requires. Since the data signed by the sender includes T but not $\vec{\mathsf{ct}}$, each recipient can verify the signature. Intuitively, the commitment-binding property (Def. 3.5) then guarantees the authenticity of $\widehat{\mathsf{ct}}_{\mathsf{id}}$ despite it not being directly signed.

4.2 Asymptotic Bandwidth Efficiency

We now discuss the bandwidth efficiency of our protocols. We leave out elements that reflect logical group operations (e.g., a bitstring encoding "id *has been added to G*") or symmetric key cryptography (e.g., hashes or MAC tags), as they add negligible overheads (compared to public key cryptography) to all solutions.

The bottleneck of both TreeKEM and our solution resides in commit messages, as these are processed on a daily basis (as the output of `Commit`, and the input of `Process`) and contain a significant amount of public key material. We recall that we note ek an encryption key, ct_0 the (ek-independent) part of an mPKE ciphertext, $\widehat{\mathsf{ct}}_i$ the part of a ciphertext dependent of ek_i and sig a signature, and note $|x|$ the bytesize of x. We consider a group of N members, in a epoch with no new member.

TreeKEM. The size of an uploaded commit message is dominated by $2 \cdot |\mathsf{sig}| + \lceil \log N \rceil \cdot |\mathsf{ek}| + \Omega(\log N) \cdot (|\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|)$.[7] Since all ciphertexts in the commit message are signed jointly by a single signature, recipients need to download all ciphertexts to verify the signature.

Chained CmPKE. The size of an uploaded commit message is dominated by $2 \cdot |\mathsf{sig}| + |\mathsf{ek}| + |\mathsf{ct}_0| + N \cdot |\widehat{\mathsf{ct}}_i| + 2\kappa$. The term 2κ stems from our construction of a CmPKE instead of a mPKE (Thm. 3.6). This is no larger than a hash digest, and we henceforth ignore it. Since each user performs a selective downloading, the size of a *downloaded* commit message is reduced by a factor $O(N)$, as it is now dominated by $2 \cdot |\mathsf{sig}| + |\mathsf{ek}| + |\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|$.

New Members. In both TreeKEM and our protocol, newly added members use the `Join` procedure to process *welcome messages*. These contain all encryption keys $\mathsf{ek}_{\mathsf{id}}$: N in our case (included in memberPublicInfo), and at most $(2N-1)$ in TreeKEM due to the use of a binary tree. In both cases, the size of a welcome message is dominated by these keys and is $O(N)$. Overall, it seems unlikely that joining a group will be a bandwidth bottleneck, as each member of a group typically performs this operation once, whereas the number of commit messages may be unbounded.

We note that welcome messages encrypt-then-sign a common joinerSecret to the (public) encryption keys of all new members. If an epoch contains k new members, this entails an overhead $|\mathsf{sig}| + k \cdot (|\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|)$ for TreeKEM. In our protocol, this is done via a CmPKE, which entails a smaller overhead $|\mathsf{sig}| + |\mathsf{ct}_0| + k \cdot |\widehat{\mathsf{ct}}_i|$.

Two Alternative Protocols. We briefly present two protocols that also achieve a bandwidth complexity $O(N)$ and $O(1)$ for uploading and downloading commit messages, using only generic primitives.

The first protocol, that we refer to as a *Parallel* KEM, encrypts the same comSecret to all group members using $(N-1)$ parallel (non-committing, single-recipient) PKEs. A distinct signature $\mathsf{sig}_{\mathsf{id}}$ is computed for each distinct $\mathsf{ct}_{\mathsf{id}}$. The cost of an upload is $|\mathsf{ek}| + N(|\mathsf{ct}| + 2 \cdot |\mathsf{sig}|) = O(N)$ and, since each ciphertext is individually authenticated, the cost of a download is $|\mathsf{ek}| + |\mathsf{ct}| + 2 \cdot |\mathsf{sig}| = O(1)$. See P. KEMs in Tab. 1.

Since any PKE is also a decomposable mPKE for $\mathsf{ct}_0 = \perp$, a slightly more involved solution is to build a CmPKE from any single-recipient PKE as a special case of Thm. 3.6. Once we have a CmPKE, the construction, which we refer to as *Committing* PKEs, is identical to ours. The cost of an upload is now $|\mathsf{ek}| + N \cdot |\mathsf{ct}| + 2 \cdot |\mathsf{sig}| = O(N)$, and the cost of a download remains $|\mathsf{ek}| + |\mathsf{ct}| + 2 \cdot |\mathsf{sig}| = O(1)$, see C. PKE in Tab. 1.

Applying Our Techniques to TreeKEM. We can apply to the TreeKEM protocol the two techniques leveraged here: selective downloading and mPKEs.

Thanks to the tree-based structure of TreeKEM, each user can perform selective downloading to retrieve only one ciphertext per commit message. Indeed a similar idea to selective downloading was proposed for TreeKEM [18], but to the best of our knowledge it has never been implemented or formally analyzed. One possible reason for this is because unlike in Chained CmPKE, TreeKEM has the added complexity of maintaining the public keys associated to the internal nodes of the tree. Specifically, a user only needs to know the public keys associated to the internal nodes along its path to the root in order to process commit message, however, it may need to know more if it wants to upload commit messages. Notice the nodes that the user needs to know is not fixed in advance since add/remove/update proposals may adaptively change the topology of the tree. Consequently, a user may need to download additional key materials when performing a commit (which we call *on-the-fly* downloading). Hence, although we believe it is possible to further lower the download cost for TreeKEM using similar ideas, this would entail more server-side bookkeeping of the tree structure and the associating public keys for each internal nodes, which would likely add complexity to the protocol description and security proof. We leave it as an interesting future research to assess the full benefit of such an approach.

Combining TreeKEM with mPKEs/mKEMs was done in [66], which considered a variant of TreeKEM with trees of arity m. This reduces the number of encryption keys per commit message to $\lceil \log_m N \rceil$ in the best case (unblanked tree), which is still $\Omega(\log N)$ for any constant value of m. Note that setting $m = N$ results in a flat tree, which yields a protocol similar to Chained CmPKE. So while it is possible to apply our techniques to TreeKEM, we found that doing so with the goal of minimizing the total bandwidth cost leads to a protocol very similar to ours, which a posteriori validates our design choices.

Why Efficient mPKEs Matter. It may not be obvious that our solution represents an improvement upon Parallel KEMs and Committing PKEs, since all three achieve the same asymptotic bandwidth efficiency: $O(N)$ in upload, $O(1)$ in download. However, a perk of post-quantum cryptography is its ability to provide mPKEs for which the ek_i-dependent part $\widehat{\mathsf{ct}}_i$ of ciphertexts are extremely compact, as illustrated in Tab. 4. Our protocol directly benefits from this fact, since the size of uploaded commit messages is $\sim |\widehat{\mathsf{ct}}_i| \cdot N$. In Sec. 5, we propose lattice-based mPKEs inspired by the (possibly alternative) finalists to standardization by NIST Kyber [77], NTRU LPRime [21] and FrodoKEM [74]. Our mPKEs make $\widehat{\mathsf{ct}}_i$ as small as $\{48, 32, 24\}$ bytes. Concretely, this allows our protocol to reduce the *upload* bandwidth cost by two to three orders of magnitude compared to Parallel KEMs and Committing PKEs.

4.3 Provable Security

We prove our Chained CmPKE to be secure in an extended variant of the UC security model that was recently used to analyze TreeKEM version 10 in MLS by Alwen et al. [13]. The security model presented by [13] is an extension of [12] that further considers *insider* security, allowing the adversary to maliciously inject messages, deliver messages in an arbitrary order, and interact maliciously with the PKI. In addition, it formalizes the PCFS guarantee using the *safe predicate*, which decides whether the epoch key is secure. In our work, since the uploaded and downloaded commits are in different forms, we modify the ideal functionality in [13] accordingly. Effectively, this creates a subtle difference in how the *history graph* is maintained by the ideal functionality. We note that prior constructions can be handled within our new extended model,

[7] In both TreeKEM and Chained CmPKE, a commit message contains *two* signatures: one authenticates ciphertexts, and one signs the committer's new encryption key(s) ("*tree signing*" in [13]). A commit message may contain an optional welcome message, which is then signed by a third signature. Our improvements target the first signature (ciphertexts), and are orthogonal to the other two.

Table 1: Bandwidth cost of a commit message to a group of N members (with no newly added member) in terms of public key cryptography. For schemes that use single-recipient PKEs/KEMs, we assume $|\mathsf{ct}| = |\mathsf{ct}_0| + |\mathsf{ct}_i|$. All logarithms are in base 2. The notation $\lceil \log N \rceil$ expresses that for the row labelled [13] the best-case complexity is $\lceil \log N \rceil$, and the worst-case is N.

Scheme	Upload				Download (per recipient)				Total (1 upload, then $(N-1)$ downloads)			
	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$
[13]	$\lceil \log N \rceil$	$\lceil \log N \rceil$	$\lceil \log N \rceil$	2	$\lceil \log N \rceil$	$\lceil \log N \rceil$	$\lceil \log N \rceil$	2	$N\lceil \log N \rceil$	$N\lceil \log N \rceil$	$N\lceil \log N \rceil$	$2N$
Ours	1	1	$(N-1)$	2	1	1	1	2	N	N	$2(N-1)$	$2N$
P. KEMs	1	$(N-1)$	$(N-1)$	N	1	1	1	2	N	$2(N-1)$	$2(N-1)$	$3N-2$
C. PKEs	1	$(N-1)$	$(N-1)$	2	1	1	1	2	N	$2(N-1)$	$2(N-1)$	$2N$

Table 2: Bandwidth costs of mPKEs derived from existing parametrizations (gray background) and new ones (white background), for $\kappa = 128$ bits of classical security. Standard (single-recipient) PKE instantiations of existing schemes may include a seed in the encryption key or a confirmation hash in the ciphertext (in parentheses).

Scheme	Reference	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$
Kyber512	[77]	768 (+32)	640	128
Ilum512	Sec. 5	768	704	48
LPRime653	[21]	865 (+32)	865 (+32)	128
LPRime757	Sec. 5	1076	1076	32
Frodo640	[74]	9600 (+16)	9600	120
Bilbo640	Sec. 5	10240	10240	24
SIKEp434	[63]	330	330	16

thus our model is a strict generalization of prior models. Full details on our security model are provided in App. C. The security of Chained CmPKE is established by Thm. 4.1. The proof is provided in App. E.

Theorem 4.1. *Assuming that* CmPKE *is* IND-CCA *secure (resp. with adaptive corruption) and* SIG *is* sEUF-CMA *secure, the Chained CmPKE protocol selectively (resp. adaptively) securely realizes the ideal functionality $\mathcal{F}_{CGKA}$, where $\mathcal{F}_{CGKA}$ uses the predicate **safe** from Fig. 28, in the $(\mathcal{F}_{AS}, \mathcal{F}_{KS}, \mathcal{G}_{RO})$-hybrid model, where calls to the hash function* H, HKDF, *and* MAC *are replaced by calls to the global random oracle $\mathcal{G}_{RO}$.*

5 MORE EFFICIENT LATTICE-BASED mPKES

To maximize the bandwidth savings of Chained CmPKE we must reduce $|\widehat{\mathsf{ct}}_i|$ as much as possible. Indeed, see Tab. 1, where the "Ours" row is only less performant than another in one column, namely Upload $|\widehat{\mathsf{ct}}_i|$. Therefore, in this section we outline the methods employed to achieve this. We adapt several PKEs from the literature to mPKEs, specifically PKEs which underly KEMs that are either finalists or alternative finalists of the final round of the NIST PQC process [1]. Throughout this section we only consider IND-CPA mPKEs, and use the notation of Def. 2.3. For the needs of the protocol in Sec. 4, these can be converted into IND-CCA CmPKEs with a small overhead using Thm. 3.6.

We start from the construction of [66], reproduced in Fig. 5, which adapts the Lindner–Peikert framework [70] to the mPKE setting. As observed by [66], Fig. 5 can be readily applied to the (possibly alternative) finalists FrodoKEM [74], Kyber [77], NTRU LPRime [21] and Saber [41]. We take this one step further and propose new parametrizations of [21, 74, 77] that are tailored to the mPKE setting. At the cost of less than a 20% increase in $|\mathsf{ek}| + |\mathsf{ct}_0|$, we reduce $|\widehat{\mathsf{ct}}_i|$ by 60–80%. Since the size of an uploaded package is asymptotically $\sim |\widehat{\mathsf{ct}}_i| \cdot N$, we view this trade-off as favorable.

This section is arranged as follows. In Sec. 5.1, we review the techniques that one can leverage to minimize $|\widehat{\mathsf{ct}}_i|$. Then in Sec. 5.2 we provide new parametrizations of [21, 74, 77]. Finally, App. G details our cryptanalytic model, and provides security estimates for our parameter sets in this model.

5.1 Our Toolkit for Improving Efficiency

We review the known techniques at our disposal to minimize the size of the $(\widehat{\mathsf{ct}}_i)_i$ while increasing as little as possible the sizes of ek and ct_0, and maintaining security against known attacks. The coefficient dropping and modulus rounding techniques are already present in [21] and [77] respectively. Concretely, for modulus rounding we will focus on the Compress and Decompress functions of [77]. By *more* or *less* rounding, we mean a *smaller* or *larger* d in the definition of those functions, respectively. We note that modulus rounding techniques can be applied to the original parametrizations of [74], but save little in the $|\mathsf{ek}| + |\mathsf{ct}_0| + |\widehat{\mathsf{ct}}_i|$ (i.e., single recipient) metric. We revisit these techniques in light of the new constraints imposed by the mPKE setting, which in turn leads to new parameter sets. Throughout we reference Fig. 5.

We note that the ciphertexts of some PKEs and mPKEs based on lattices have a small probability of decrypting to a different message than was initially encrypted. The probability of this occuring is called the decryption failure rate, or DFR. Keeping the DFR low, specifically $O(2^{-\kappa})$, is important for both correctness and security; we discuss it more in App. G.

Coefficient Dropping. When trying to decode a message M from $(\mathbf{U}, \mathbf{V}_i)$ using S, not all of $\mathbf{V}_i$ may be necessary. Indeed let $R = \mathbb{Z}[x]/(f)$, $d = \deg(f)$, $I < d$, and $\bar{n} = \bar{m} = 1$. If $\mathsf{Encode}(\mathsf{M}) = \alpha_{I-1}x^{I-1} + \cdots + \alpha_0$ then only the I lower order coefficients of $\mathbf{V}_i$ are useful for decoding. In general, if f is any degree d polynomial and one requires $I < d$ coefficients to encode any M, then $\mathbf{V}_i$ may consist of only low degree coefficients of a single $v \in R_q$. This technique does not affect the DFR, improves efficiency, and cannot be worse for security.

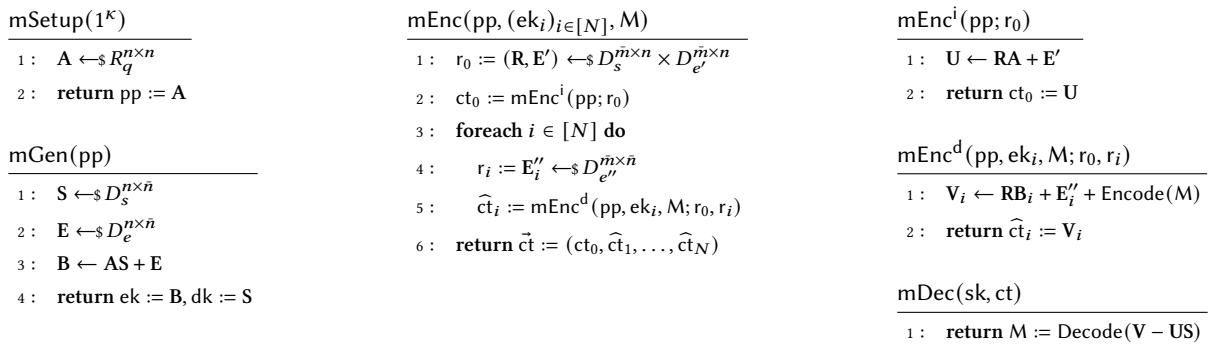


Figure 5: Lattice-based mPKE construction of [66]. R **is the base ring,** $D_s, D_e, D_{e'}, D_{e''}$ **are distributions over** R**.**

Modulus Rounding. Rounding away the least significant bits of $\mathbf{B}$, $\mathbf{U}$, and $\mathbf{V}_i$ provides more compact ek, ct_0 and $\widehat{\mathsf{ct}}_i$ (respectively), but mechanically raises the DFR. Our goal is to minimize the size of $\widehat{\mathsf{ct}}_i$, so we will maximize the rounding on $\mathbf{V}_i$, while upper bounding the DFR. To do so we may round *fewer* bits from $\mathbf{B}$ or $\mathbf{U}$ to give us more DFR headroom. Thankfully, all else being equal, rounding $\mathbf{V}_i$ incurs a milder increase in the DFR than on $\mathbf{B}$ or $\mathbf{U}$. It also makes the numerous samples introduced by the $(\mathbf{V}_i)_i$ noisy enough to nullify Arora–Ge and BKW attacks, see App. G.

Increasing the Modulus. All else being equal, increasing the modulus q reduces the DFR and therefore allows one to perform more rounding. If this extra rounding is concentrated on the $(\mathbf{V}_i)_i$, the net effect on the size of each $\widehat{\mathsf{ct}}_i$ is to decrease it. On the other hand, it slightly increases the size of ct_0 and ek and, more importantly, decreases the error rate, making lattice attacks more efficient.

Error-Correcting Codes (ECCs). Whenever in Fig. 5 we want to encrypt κ bits, for $\kappa < |\mathsf{M}|$, we can use an ECC, i.e. $\mathsf{Encode}(\mathsf{M}) = \mathsf{Encode}(\mathsf{ECC}(\kappa))$, and lower the DFR. However, this method can lead to attacks when improperly implemented [40] or analyzed [43, 55]. In addition, if the goal is to minimize $|\widehat{\mathsf{ct}}_i|$, then coefficient dropping seems to always be a safer and more efficient alternative. Hence we will not employ ECCs.

5.2 New Parametrizations

Given the methods outlined in Sec. 5.1, we make a number of alterations to the NIST Level I parameters of FrodoKEM, Kyber, and NTRU LPRime. In each case we maintain the *spirit* of the original design by e.g. keeping unique features. In all cases the new schemes satisfy the cryptanalytic model specified in App. G, see also Fig. 31 for concrete security estimates against a number of attacks.

Note that the number of bits of shared secret encoded in $\mathbf{V}$ differs in these KEMs; Frodo640 encodes 128, whereas all parameter sets of Kyber and NTRU LPRime encode 256. For the purpose of fair comparison, in all cases we encode 128 bits. We note that in the case of Ilum512 and LPRime757, encoding 128 bits rather than 256 automatically reduces $|\widehat{\mathsf{ct}}_i|$ from 128 bytes to 64. Reductions below this size are a result of the techniques outlined in Sec. 5.1.

More subtle changes are discussed in App. G, we briefly present them in this paragraph. For each scheme we give a table comparing (in the notation of the original scheme) the old and new parameter sets. We also give a dictionary of the form {`Figure: value`}, where `Figure` is a parameter from Fig. 5 and `value` either comes from the relevant table or is defined in prose. The tables and descriptions of $D_{e'}$ and $D_{e''}$ in this section do not reflect wider error distributions implied by modulus rounding. We also discuss the effect of modulus rounding on security and decryption failures in App. G. The savings achieved by our new parametrizations are given in Tab. 2.

Kyber. We introduce a new parameter set, Ilum512. We apply one less bit of rounding to $\mathbf{U}$, and one more to $\mathbf{V}$. We also drop coefficients from $\mathbf{V}$, see Tab. 3. Although altering q allowed other parametrizations, ring arithmetic over R_q consistently represents a significant fraction of the effort involved in providing embedded implementations of Kyber [8, 84]. Keeping the same ring R_q as Kyber helps make Ilum512 fast and easy to deploy. Letting B_η be the binomial distribution over R defined in [77], we have $\{R : \mathbb{Z}[x]/(x^{256}+1), n : k, q : q, \bar{n} = \bar{m} : 1, D_s = D_e : B_{\eta_1}, D_{e'} = D_{e''} : B_{\eta_2}\}$.

Table 3: Parameter sets of Kyber512 and Ilum512, using the notation of [77], we drop $n - I$ coefficients.

Scheme	n	k	q	η_1	η_2	d_u	d_v	I
Kyber512	256	2	3329	3	2	10	4	256
Ilum512	256	2	3329	3	2	11	3	128

FrodoKEM. We introduce a new parameter set, Bilbo640. Compared to Frodo640, Bilbo640 introduces aggressive rounding on $\mathbf{V}$, which has a positive effect on both the bandwidth cost and the security. To mitigate the effect on the DFR, we increase q to 2^{16}. We use a slightly larger new error distribution, χ_{Bilbo640}, which requires 32 bits of randomness per sample, see Tab. 4. We have $\{R : \mathbb{Z}, n : n, q : 2^{16}, \bar{n} : \bar{n}, \bar{m} : \bar{m}, D_s = D_e = D_{e'} = D_{e''} = \chi_{\mathsf{Bilbo640}}\}$.

NTRU LPRime. We introduce a new parameter set, LPRime757. We reduce the number of bits per entry of $\mathbf{V}$ from 4 to 2, and must increase the modulus, and decrease the weight, to account for this, see Tab. 5. The authors of NTRU LPRime [21] place a great emphasis on having $(x^p - x - 1)$ irreducible in $\mathbb{Z}_q$ and a DFR equal to zero. This is also the case for LPRime757.

Table 4: Parameter sets of Frodo640 and Bilbo640, using the notation of [74], plus b/s to denote the random bits needed to sample an integer coefficient, and $\{D_B, D_U, D_V\}$ to denote the bits/coefficient in $\{B, U, V\}$ (instead of a common D in [74]).

Scheme	n	D_B	D_U	D_V	σ	B	I	$\bar{m}$	$\bar{n}$	b/s
Frodo640	640	15	15	15	2.8	2	128	8	8	16
Bilbo640	640	16	16	3	2.9	2	128	8	8	32

We slightly alter the rounding function Top to Top′ which maintains perfect correctness while allowing us a larger weight than otherwise, see App. G. We keep the original Right. As NTRU LPRime uses rounding for its errors the syntax of Fig. 5 is not strictly correct, and we will report the errors induced by rounding. Let Short define the distribution that samples uniformly from the set Short of [21], let X assign probability $(q-1)/3q$ to ± 1 and $(q+2)/3q$ to 0, and let Y denote the probability mass function for a particular error value $\mathsf{Right}(\mathsf{Top}'(C)) - C$ over all $C \in \mathbb{Z}_q$. We have $\{R : \mathbb{Z}[x]/(x^p - x - 1), n = \bar{n} = \bar{m} : 1, q : q, D_s : \mathsf{Short}, D_e = D_{e'} : X, D_{e''} : Y\}$.

Table 5: Parameter sets of LPRime653 and LPRime757, using the notation of [21]. We drop $p - I$ coefficients from V.

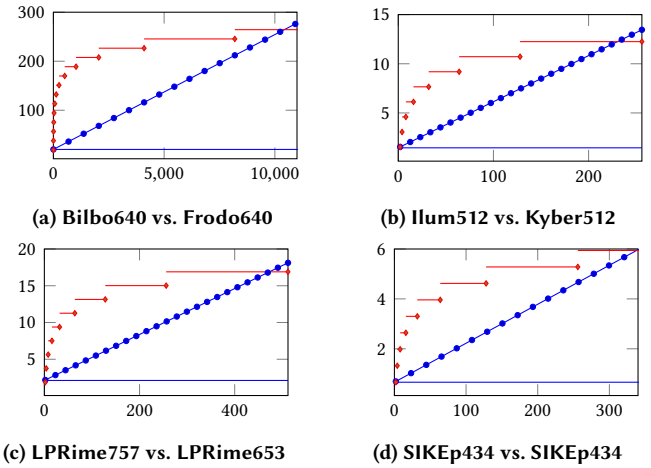
Scheme	p	q	w	δ	τ	I
LPRime653	653	4621	252	289	16	256
LPRime757	757	7879	242	2001	4	128

A Note on Isogeny-Based mPKEs**.** One of our instantiations of Chained CmPKE uses a mPKE variant of SIKE proposed in [66]. Bandwidth-wise, it seems asymptotically optimal, as $\widehat{\mathsf{ct}}_i$ is κ bits. Security-wise, [66] provides a security reduction to the SSDDH problem [47], with a loss of $1/N$ in the advantage. This security loss is minimal: concretely, it means that using mPKE-SIKE with N recipient loses at most $\lceil \log N \rceil$ bits of security compared to one recipient, which is small even for large groups. A downside of using SIKE is its slower running time, see Fig. 8.

6 INSTANTIATION AND IMPLEMENTATION

We instantiate Chained CmPKE as follows:

- *One-time* IND-CCA SKE. Since the message to be encrypted has $\kappa = 128$ bits, we may take plain AES-128 without a need for a mode. If we model plain AES as a pseudorandom permutation (PRP), then it satisfies Def. 2.1. We then obtain key-commitment by applying [2, Sec. 5.2.].
- *Signature scheme.* We choose Dilithium for two reasons: (a) its performances are well-balanced, (b) it claims sEUF-CMA security from standard lattice assumptions [71].
- mPKE. If we choose to rely on isogeny-based assumptions, we may use the SIKE mPKE from [66]. If we rely on lattice-based assumptions, we may use one of our three lattice-based mPKEs from Sec. 5: Bilbo640, Ilum512, LPRime757.



(a) Bilbo640 vs. Frodo640 (b) Ilum512 vs. Kyber512

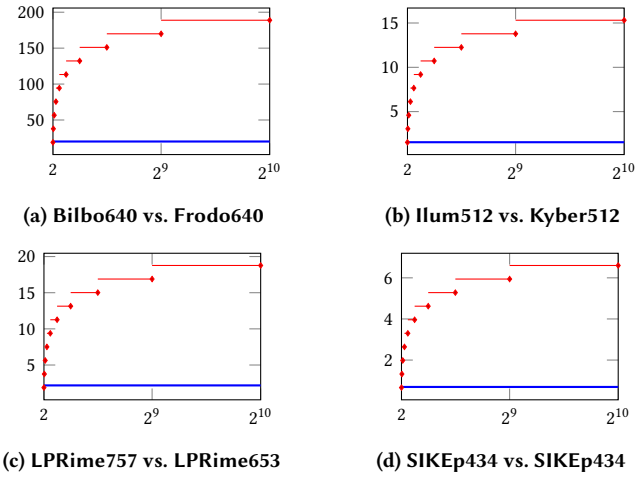
(c) LPRime757 vs. LPRime653 (d) SIKEp434 vs. SIKEp434

Figure 6: The graphs "*X vs Y*" give the bandwidth overhead (in term of encryption keys and ciphertexts) of commit messages when using Chained CmPKE with the CmPKE **X (—●— when uploaded, —— when downloaded), compared to TreeKEM with the** KEM **Y (—♦— *both* when uploaded and downloaded). The x-axis is the group size N, the y-axis is the overhead in KiB.**

The mKEMs which are at the core of the mPKEs are implemented in C, starting from the optimized public platform-independent implementations of [21, 63, 74, 77]. For Ilum512 and SIKEp434, the changes are straightforward. The modifications for Bilbo640 are only slightly more involved due to the new distribution and the Kyber-style compression. Finally, LPRime757 required most work: all encoding/decoding routines, rounding, Top and Barrett reduction had to be modified. We also improved polynomial multiplication performance, by computing them in the larger ring $GF(q')[x]/_{\langle 2^{p'}+1\rangle}$, with $q' = 1907713 > w(q-1)$ and $p' = 1536 = 3 \cdot 2^9$, which admits fast NTT-based multiplication as $3 \cdot 2^8 \mid q' - 1$. We do not use a full NTT, but leave out the layer corresponding to the factor 3 and multiply degree 2 polynomials in the NTT-domain, which is slightly more efficient than a full NTT. Chained CmPKE and the mPKEs are implemented in Go, using C bindings for the mKEMs.

Bandwidth Consumption. In Fig. 7, we compare the total bandwidth overheads of TreeKEM and Chained CmPKE in terms of ciphertexts and encryption keys. For a better comparison, terms that are identical between both protocols, such as signatures, MACs, etc, are ignored. For readability, the bandwidth cost of each graph is normalized by the group size N. As predicted by the theory, our protocol performs better than TreeKEM by factors $\Omega(\log N)$ for similar instantiations. In addition, while the size of our *uploaded* commit messages is asymptotically worse compared to TreeKEM ($O(N)$ vs $\Omega(\log N)$), in practice we compare favourably against comparable post-quantum instantiations of TreeKEM, even for groups of hundreds of users, see Fig. 6.[8]

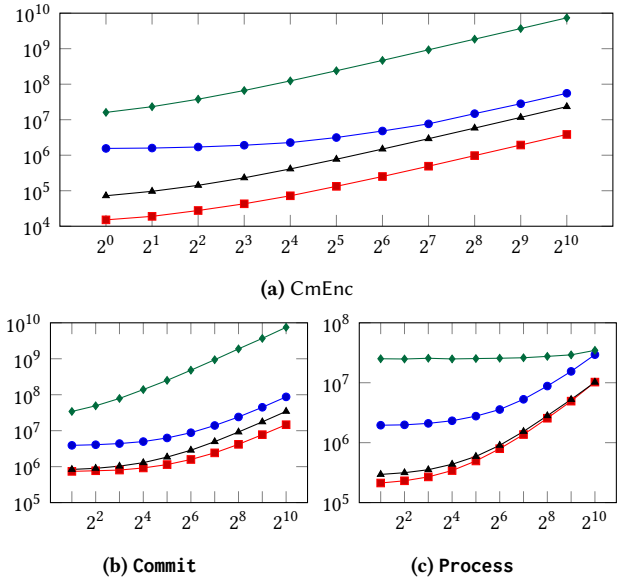
[8] In the absence of post-quantum parameter sets for TreeKEM in MLS, we came up with our own parameter sets relying on NIST PQC KEMs (finalists or alternate).



(a) Bilbo640 vs. Frodo640 (b) Ilum512 vs. Kyber512

(c) LPRime757 vs. LPRime653 (d) SIKEp434 vs. SIKEp434

Figure 7: The graphs "*X vs. Y*" (Figs. 7a to 7d) give the *normalized* total bandwidth overhead (in term of encryption keys and ciphertexts) of a commit message with Chained CmPKE using the CmPKE X (——), compared to TreeKEM using the KEM Y (—♦—). The x-axis is the group size N, the y-axis is the total bandwidth cost in KiB normalized by N. Graphs are computed using Tabs. 1 and 2.



(a) CmEnc

(b) `Commit` (c) `Process`

Figure 8: Running time in nanoseconds of some procedures as functions of the group size N, for Ilum512 (—■—), Bilbo640 (—●—), LPRime757 (—▲—) and SIKEp434 (—◆—). All measurements were obtained on an Apple M1 CPU @3.2 GHz (single-threaded).

Computational Efficiency. In Fig. 8, we provide timings for what we expect to be the two computational bottlenecks of our protocol: `Commit` (Fig. 8b) and `Process` (Fig. 8c). We also provide timings for CmEnc (Fig. 8a).

Even for group of 2^{10} members, lattice-based CmPKEs perform a multi-recipient encryption in less than 100 ms. This operation – and by extension, `Commit`– may take significantly longer when instantiating Chained CmPKE with SIKEp434 (about 7.5 s for 2^{10} recipients). Note however that `Commit` is a transparent operation for end users, and can be performed even when the end device is locked. We conclude from our measurements that the computational efficiency of Chained CmPKE is likely to have a minimal impact on the user experience.

Note that large groups also provide an amortization effect on the *computational* efficiency of CmPKEs. For example, encrypting a message to 2^{10} recipients with Bilbo640 (resp. Ilum512, LPRime757, SIKEp434) is about 29 (resp. 4, 3, 2) times faster than to perform 2^{10} encryptions. Finally, even though `Process` only entails a constant number of public-key operations, its running time eventually gets linear in N (Fig. 8c), due to the hashing of N encryption keys when verifying the group state. This is also the case in TreeKEM, and can be mitigated to some extent by storing the hashes of the encryption keys.

Code. Our code is available at the following repository:

https://github.com/PQShield/chained-cmpke

REFERENCES

[1] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. 2020. NISTIR 8309 - Status Report on the Secind Round of the NIST Post-Quantum Cryptography Standardization Process. https://csrc.nist.gov/publications/detail/nistir/8309/final.

[2] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. 2020. How to Abuse and Fix Authenticated Encryption Without Key Commitment. Cryptology ePrint Archive, Report 2020/1456. https://eprint.iacr.org/2020/1456.

[3] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. 2020. *Classic McEliece.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[4] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. 2015. On the Complexity of the BKW Algorithm on LWE. *Des. Codes Cryptography* 74, 2 (Feb. 2015), 325–354. https://doi.org/10.1007/s10623-013-9864-x

[5] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, and Ludovic Perret. 2014. Algebraic Algorithms for LWE. Cryptology ePrint Archive, Report 2014/1018. https://eprint.iacr.org/2014/1018.

[6] Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck. 2020. Estimating Quantum Speedups for Lattice Sieves. In *ASIACRYPT 2020, Part II (LNCS, Vol. 12492)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer, Heidelberg, 583–613. https://doi.org/10.1007/978-3-030-64834-3_20

[7] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9, 3 (2015), 169–203. https://doi.org/doi:10.1515/jmc-2015-0016

[8] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. 2020. ISA Extensions for Finite Field Arithmetic. *IACR TCHES* 2020, 3 (2020), 219–242. https://doi.org/10.13154/tches.v2020.i3.219-242 https://tches.iacr.org/index.php/TCHES/article/view/8589.

[9] Joë Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. 2021. Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement. In *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 596–612. https://doi.org/10.1109/SP40001.2021.00035

[10] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT 2019, Part I (LNCS, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.).

Springer, Heidelberg, 129–158. https://doi.org/10.1007/978-3-030-17653-2_5
[11] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2020. Security Analysis and Improvements for the IETF MLS Standard for Group Messaging. In *CRYPTO 2020, Part I (LNCS, Vol. 12170)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Heidelberg, 248–277. https://doi.org/10.1007/978-3-030-56784-2_9
[12] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. 2020. Continuous Group Key Agreement with Active Security. In *TCC 2020, Part II (LNCS, Vol. 12551)*, Rafael Pass and Krzysztof Pietrzak (Eds.). Springer, Heidelberg, 261–290. https://doi.org/10.1007/978-3-030-64378-2_10
[13] Joël Alwen, Daniel Jost, and Marta Mularczyk. 2020. On The Insider Security of MLS. Cryptology ePrint Archive, Report 2020/1327. https://eprint.iacr.org/2020/1327.
[14] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. 2009. Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems. In *CRYPTO 2009 (LNCS, Vol. 5677)*, Shai Halevi (Ed.). Springer, Heidelberg, 595–618. https://doi.org/10.1007/978-3-642-03356-8_35
[15] Sanjeev Arora and Rong Ge. 2011. New Algorithms for Learning in Presence of Errors. In *ICALP 2011, Part I (LNCS, Vol. 6755)*, Luca Aceto, Monika Henzinger, and Jiri Sgall (Eds.). Springer, Heidelberg, 403–415. https://doi.org/10.1007/978-3-642-22006-7_34
[16] Shi Bai and Steven D. Galbraith. 2014. Lattice Decoding Attacks on Binary LWE. In *ACISP 14 (LNCS, Vol. 8544)*, Willy Susilo and Yi Mu (Eds.). Springer, Heidelberg, 322–337. https://doi.org/10.1007/978-3-319-08344-5_21
[17] Manuel Barbosa and Pooya Farshim. 2007. Randomness reuse: Extensions and improvements. In *IMA International Conference on Cryptography and Coding*. Springer, 257–276.
[18] Richard Barnes. 2018. [MLS] Efficiency and "Ampelmann trees". IETF Mail Archive. https://mailarchive.ietf.org/arch/msg/mls/INcV28Jth25m_l__NMmQIYp13Po/.
[19] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. 2020. *The Messaging Layer Security (MLS) Protocol.* Internet-Draft draft-ietf-mls-protocol-11. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-11 Work in Progress.
[20] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. 2003. Randomness Re-use in Multi-recipient Encryption Schemeas. In *PKC 2003 (LNCS, Vol. 2567)*, Yvo Desmedt (Ed.). Springer, Heidelberg, 85–99. https://doi.org/10.1007/3-540-36288-6_7
[21] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. 2020. *NTRU Prime.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[22] Daniel J. Bernstein and Tanja Lange. 2020. McTiny: Fast High-Confidence Post-Quantum Key Erasure for Tiny Network Servers. In *USENIX Security 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1731–1748.
[23] Benjamin Beurdouche. 2020. *Formal Verification for High Assurance Security Software in F*.* Ph.D. Dissertation.
[24] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. 2018. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS).* Research Report. Inria Paris. https://hal.inria.fr/hal-02425247
[25] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. 2019. *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS.* Research Report. Inria Paris. https://hal.inria.fr/hal-02425229
[26] Nina Bindel and John M. Schanck. 2020. Decryption Failure Is More Likely After Success. In *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, Jintai Ding and Jean-Pierre Tillich (Eds.). Springer, Heidelberg, 206–225. https://doi.org/10.1007/978-3-030-44223-1_12
[27] WhatsApp Blog. 2020. Two Billion Users – Connecting the World Privately. WhatsApp Blog. https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately/.
[28] Avrim Blum, Adam Kalai, and Hal Wasserman. 2000. Noise-tolerant learning, the parity problem, and the statistical query model. In *32nd ACM STOC*. ACM Press, 435–440. https://doi.org/10.1145/335305.335355
[29] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. 2020. Towards Post-Quantum Security for Signal's X3DH Handshake. In *SAC 2020*. https://eprint.iacr.org/2019/1356.
[30] Alessandro Budroni, Qian Guo, Thomas Johansson, Erik Mårtensson, and Paul Stankovski Wagner. 2020. Making the BKW Algorithm Practical for LWE. In *INDOCRYPT 2020 (LNCS, Vol. 12578)*, Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran (Eds.). Springer, Heidelberg, 417–439. https://doi.org/10.1007/978-3-030-65277-7_19
[31] Cable.co.uk. 2021. Worldwide Mobile Data Pricing 2021 | 1GB Cost in 230 Countries. https://www.cable.co.uk/mobiles/worldwide-data-pricing/.
[32] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2016. Universal Composition with Responsive Environments. In *ASIACRYPT 2016, Part II (LNCS, Vol. 10032)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Springer, Heidelberg, 807–840. https://doi.org/10.1007/978-3-662-53890-6_27
[33] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*. IEEE Computer Society Press, 136–145. https://doi.org/10.1109/SFCS.2001.959888
[34] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *TCC 2007 (LNCS, Vol. 4392)*, Salil P. Vadhan (Ed.). Springer, Heidelberg, 61–85. https://doi.org/10.1007/978-3-540-70936-7_4
[35] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 451–466. https://doi.org/10.1109/EuroSP.2017.27
[36] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2020. A formal security analysis of the signal messaging protocol. *Journal of Cryptology* (2020), 1–70. https://doi.org/10.1007/s00145-020-09360-1
[37] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 2018. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 1802–1819. https://doi.org/10.1145/3243734.3243747
[38] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. 2016. On Post-compromise Security. In *CSF 2016Computer Security Foundations Symposium*, Michael Hicks and Boris Köpf (Eds.). IEEE Computer Society Press, 164–178. https://doi.org/10.1109/CSF.2016.19
[39] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. 2020. LWE with Side Information: Attacks and Concrete Security Estimation. In *CRYPTO 2020, Part II (LNCS, Vol. 12171)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Heidelberg, 329–358. https://doi.org/10.1007/978-3-030-56880-1_12
[40] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. Timing Attacks on Error Correcting Codes in Post-Quantum Schemes. In *TIS@CCS*, Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen (Eds.). ACM, 2–9. https://doi.org/10.1145/3338467.3358948
[41] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. 2020. *SABER.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[42] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. 2020. (One) Failure Is Not an Option: Bootstrapping the Search for Failures in Lattice-Based Encryption Schemes. In *EUROCRYPT 2020, Part III (LNCS, Vol. 12107)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 3–33. https://doi.org/10.1007/978-3-030-45727-3_1
[43] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. The Impact of Error Dependencies on Ring/Mod-LWE/LWR Based Schemes. In *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*, Jintai Ding and Rainer Steinwandt (Eds.). Springer, Heidelberg, 103–115. https://doi.org/10.1007/978-3-030-25510-7_6
[44] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. 2018. Fast Message Franking: From Invisible Salamanders to Encryptment. In *CRYPTO 2018, Part I (LNCS, Vol. 10991)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 155–186. https://doi.org/10.1007/978-3-319-96884-1_6
[45] Léo Ducas. 2018. Shortest Vector from Lattice Sieving: A Few Dimensions for Free. In *EUROCRYPT 2018, Part I (LNCS, Vol. 10820)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Heidelberg, 125–145. https://doi.org/10.1007/978-3-319-78381-9_5
[46] Pooya Farshim, Claudio Orlandi, and Razvan Rosie. 2017. Security of symmetric primitives under incorrect usage of keys. *IACR Transactions on Symmetric Cryptology* (2017), 449–473.
[47] Luca De Feo, David Jao, and Jérôme Plû. 2014. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology* 8, 3 (2014), 209–247. https://doi.org/doi:10.1515/jmc-2012-0015
[48] Electronic Frontier Foundation. 2021. Lavabit. EFF. https://www.eff.org/fr/cases/lavabit.
[49] Eiichiro Fujisaki and Tatsuaki Okamoto. 1999. Secure Integration of Asymmetric and Symmetric Encryption Schemes. In *CRYPTO'99 (LNCS, Vol. 1666)*, Michael J. Wiener (Ed.). Springer, Heidelberg, 537–554. https://doi.org/10.1007/3-540-48405-1_34
[50] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. 2017. Message Franking via Committing Authenticated Encryption. In *CRYPTO 2017, Part III (LNCS, Vol. 10403)*, Jonathan Katz and Hovav Shacham (Eds.). Springer, Heidelberg, 66–97. https://doi.org/10.1007/978-3-319-63697-9_3
[51] Qian Guo, Thomas Johansson, Erik Mårtensson, and Paul Stankovski. 2017. Coded-BKW with Sieving. In *ASIACRYPT 2017, Part I (LNCS, Vol. 10624)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer, Heidelberg, 323–346. https://doi.org/10.1007/978-3-319-70694-8_12
[52] Qian Guo, Thomas Johansson, Erik Mårtensson, and Paul Stankovski Wagner. 2019. On the Asymptotics of Solving the LWE Problem Using Coded-BKW

With Sieving. *IEEE Transactions on Information Theory* 65, 8 (2019), 5243–5259. https://doi.org/10.1109/TIT.2019.2906233
[53] Qian Guo, Thomas Johansson, and Paul Stankovski. 2015. Coded-BKW: Solving LWE Using Lattice Codes. In *CRYPTO 2015, Part I (LNCS, Vol. 9215)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.). Springer, Heidelberg, 23–42. https://doi.org/10.1007/978-3-662-47989-6_2
[54] Qian Guo, Thomas Johansson, and Paul Stankovski. 2016. A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors. In *ASIACRYPT 2016, Part I (LNCS, Vol. 10031)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Springer, Heidelberg, 789–815. https://doi.org/10.1007/978-3-662-53887-6_29
[55] Qian Guo, Thomas Johansson, and Jing Yang. 2019. A Novel CCA Attack Using Decryption Errors Against LAC. In *ASIACRYPT 2019, Part I (LNCS, Vol. 11921)*, Steven D. Galbraith and Shiho Moriai (Eds.). Springer, Heidelberg, 82–111. https://doi.org/10.1007/978-3-030-34578-5_4
[56] Qian Guo, Erik Mårtensson, and Paul Stankovski Wagner. 2021. On the Sample Complexity of solving LWE using BKW-Style Algorithms. arXiv:2102.02126 [cs.CR]
[57] Chris Hall, Ian Goldberg, and Bruce Schneier. 1999. Reaction Attacks against several Public-Key Cryptosystems. In *ICICS 99 (LNCS, Vol. 1726)*, Vijay Varadharajan and Yi Mu (Eds.). Springer, Heidelberg, 2–12.
[58] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. 2021. An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable. In *Public-Key Cryptography – PKC 2021*, Juan A. Garay (Ed.). Springer International Publishing, Cham, 410–440. https://doi.org/10.1007/978-3-030-75248-4_15
[59] Keitaro Hashimoto, Shuichi Katsumata, Eamonn W. Postlethwaite, Thomas Prest, and Bas Westerbaan. 2021. A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs. In *To appear at ACM CCS 2021*.
[60] Gottfried Herold, Elena Kirshanova, and Alexander May. 2018. On the Asymptotic Complexity of Solving LWE. *Des. Codes Cryptography* 86, 1 (Jan. 2018), 55–83. https://doi.org/10.1007/s10623-016-0326-0
[61] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. 2020. Isochronous Gaussian Sampling: From Inception to Implementation. In *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, Jintai Ding and Jean-Pierre Tillich (Eds.). Springer, Heidelberg, 53–71. https://doi.org/10.1007/978-3-030-44223-1_5
[62] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. 2021. Post-Quantum WireGuard. In *2021 2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 511–528. https://doi.org/10.1109/SP40001.2021.00030
[63] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. 2020. *SIKE*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[64] Samuel Jaques, Michael Naehrig, Martin Roetteler, and Fernando Virdia. 2020. Implementing Grover Oracles for Quantum Key Search on AES and LowMC. In *EUROCRYPT 2020, Part II (LNCS, Vol. 12106)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 280–310. https://doi.org/10.1007/978-3-030-45724-2_10
[65] Ravi Kannan. 1987. Minkowski's Convex Body Theorem and Integer Programming. *Math. Oper. Res.* 12, 3 (Aug. 1987), 415–440.
[66] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. 2020. Scalable Ciphertext Compression Techniques for Post-quantum KEMs and Their Applications. In *ASIACRYPT 2020, Part I (LNCS, Vol. 12491)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer, Heidelberg, 289–320. https://doi.org/10.1007/978-3-030-64837-4_10
[67] Jonathan Katz and Nan Wang. 2003. Efficiency Improvements for Signature Schemes with Tight Security Reductions. In *ACM CCS 2003*, Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger (Eds.). ACM Press, 155–164. https://doi.org/10.1145/948109.948132
[68] Kaoru Kurosawa. 2002. Multi-recipient Public-Key Encryption with Shortened Ciphertext. In *PKC 2002 (LNCS, Vol. 2274)*, David Naccache and Pascal Paillier (Eds.). Springer, Heidelberg, 48–63. https://doi.org/10.1007/3-540-45664-3_4
[69] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. 2021. *Certificate Transparency Version 2.0*. Internet-Draft draft-ietf-trans-rfc6962-bis-35. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-trans-rfc6962-bis-35 Work in Progress.
[70] Richard Lindner and Chris Peikert. 2011. Better Key Sizes (and Attacks) for LWE-Based Encryption. In *CT-RSA 2011 (LNCS, Vol. 6558)*, Aggelos Kiayias (Ed.). Springer, Heidelberg, 319–339. https://doi.org/10.1007/978-3-642-19074-2_21
[71] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. 2020. *CRYSTALS-DILITHIUM*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[72] Moxie Marlinspike and Trevor Perrin. 2016. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/ https://signal.org/docs/specifications/doubleratchet/.
[73] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH key agreement protocol. https://signal.org/docs/specifications/x3dh/ https://signal.org/docs/specifications/x3dh/.
[74] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. 2020. *FrodoKEM*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[75] NIST. 2017. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. Accessed: 2021-04-16.
[76] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. 2021. *The Messaging Layer Security (MLS) Architecture*. Internet-Draft draft-ietf-mls-architecture-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-06 Work in Progress.
[77] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. 2020. *CRYSTALS-KYBER*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.
[78] Peter Schwabe, Douglas Stebila, and Thom Wiggers. 2020. Post-Quantum TLS Without Handshake Signatures. In *ACM CCS 20*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 1461–1480. https://doi.org/10.1145/3372297.3423350
[79] Signal. 2021. Grand jury subpoena for Signal user data, Central District of California. Signal Blog. https://signal.org/bigbrother/central-california-grand-jury/.
[80] Nigel P. Smart. 2005. Efficient Key Encapsulation to Multiple Parties. In *SCN 04 (LNCS, Vol. 3352)*, Carlo Blundo and Stelvio Cimato (Eds.). Springer, Heidelberg, 208–219. https://doi.org/10.1007/978-3-540-30598-9_15
[81] Speedtest. 2021. Speedtest Global Index — Internet Speed around the world. https://www.speedtest.net/global-index.
[82] Katherine E. Stange. 2020. Algebraic aspects of solving Ring-LWE, including ring-based improvements in the Blum-Kalai-Wasserman algorithm. arXiv:1902.07140 [cs.CR]
[83] Matthew Weidner. 2019. *Group messaging for secure asynchronous collaboration*. MPhil dissertation. University of Cambridge, Cambridge, UK.
[84] Yufei Xing and Shuguo Li. 2021. A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. *IACR TCHES* 2021, 2 (2021), 328–356. https://doi.org/10.46586/tches.v2021.i2.328-356 https://tches.iacr.org/index.php/TCHES/article/view/8797.

A OMITTED PRELIMINARIES

A.1 Notation

We denote the set of natural numbers (non-negative integers) by $\mathbb{N}$ and the security parameter by $\kappa \in \mathbb{N}$. For an algorithm A, we write $A(\cdot; \mathsf{r})$ to denote that A is run with the explicit randomness r. For $n \in \mathbb{N}$, we write $[n]$ to denote the set $[n] := \{1, \ldots, n\}$. We use $v \leftarrow x$ and $v := x$ to denote assigning the value x to the variable v, and use $v \leftarrow\!\!\$\ S$ to denote sampling an element v uniformly and randomly from a set S. We denote by $[cond]$ the bit that is 1 if the boolean statement $cond$ is true, and 0 otherwise.

Data structure. If V is a set, we write $V \xleftarrow{+} x$ and $V \xleftarrow{-} x$ as shorthands for $V \leftarrow V \cup \{x\}$ and $V \leftarrow V \setminus \{x\}$, respectively. For another set W, we write $V \xleftarrow{+} W$ and $V \xleftarrow{-} W$ as shorthands for $V \leftarrow V \cup W$ and $V \leftarrow V \setminus W$, respectively. For lists (vectors) $x := (x_1, \ldots, x_n)$ and $y := (y_1, \ldots, y_m)$, we denote the concatenation by $x \| y = (x_1, \ldots, x_n, y_1, \ldots, y_m)$ and use $x \xleftarrow{\shortparallel} v$ as a shorthand for $x \leftarrow x \| (v)$. We further use associative arrays and use $A[i] \leftarrow x$ and $y \leftarrow A[i]$ to assignment and retrieval of element i, respectively. We denote by $A[*] \leftarrow v$ the Initialization of the array to the default value v. For simplicity, we use the wildcard notation when dealing with sets of tuples and multi-argument associative arrays. For example, for an array with domain $\mathcal{I} \times \mathcal{J}$, we write $A[*, j] := \{A[i, j] \mid i \in \mathcal{I}\}$ and for a set $S \subseteq \mathcal{I} \times \mathcal{J}$, we write $(i, *) \in S$ as a shorthand for the condition $\exists j \in \mathcal{J} : (i, j) \in S$.

Keywords. We use the following keywords:

- **req** *cond* denotes that if the condition *cond* is false, then the current function unwinds all state changes and immediately returns $\perp$.
- **parse** $(m_1, \ldots, m_n) \leftarrow m$ denotes an attempt to parse a message m as a tuple. If m is not of the correct format, the current function unwinds all state changes and immediately returns $\perp$.
- **try** $y \leftarrow *\mathsf{func}(x)$ is a shorthand notation for calling a helper function $*\mathsf{func}$ and executing **req** $y \neq \perp$.
- **assert** *cond* is only used to describe functionalities. It denotes that if *cond* is false, then the functionality permanently halts, making the real and ideal worlds trivially distinguishable (this is used to validate inputs of the simulator).

A.2 Secret Key Encryption

We provide the formal syntax and correctness definition of SKEs.

Definition A.1 (Secret-Key Encryption). A secret-key encryption (SKE) over a key space $\mathcal{K}$ and message space $\mathcal{M}$ consists of the following two algorithms:

- $\mathsf{Enc_s}(\mathsf{k}, \mathsf{M}) \rightarrow \mathsf{ct}$: On input a secret key $\mathsf{k} \in \mathcal{K}$ and a message $\mathsf{M} \in \mathcal{M}$, it outputs a ciphertext ct.
- $\mathsf{Dec_s}(\mathsf{k}, \mathsf{ct}) \rightarrow \mathsf{M}$ or $\perp$: On input a secret key k and a ciphertext ct, it (deterministically) outputs either $\mathsf{M} \in \mathcal{M}$ or $\perp \notin \mathcal{M}$.

Definition A.2 (Correctness). A SKE is correct if $\Pr[\mathsf{Dec_s}(\mathsf{k}, \mathsf{Enc_s}(\mathsf{k}, \mathsf{M})) = \mathsf{M}] = 1$ holds for all $\mathsf{M} \in \mathcal{M}$ and $\mathsf{k} \in \mathcal{K}$.

A.3 Decomposable mPKE

We provide the definition of correctness and ciphertext-spreadness for a decomposable mPKE. The later roughly states that the probability of generating an identical ciphertext is negligibly small if we use proper randomness.

Definition A.3 (Correctness). A mPKE is correct if

$$\mathbb{E}\left[\max_{\mathsf{M}\in\mathcal{M}} \Pr\left[\begin{array}{c} \mathsf{ct}_0 \leftarrow \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}), \\ \widehat{\mathsf{ct}} \leftarrow \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}, \mathsf{M}) : \\ \mathsf{M} = \mathsf{mDec}(\mathsf{dk}, (\mathsf{ct}_0, \widehat{\mathsf{ct}})) \end{array}\right]\right] \geq 1 - \mathsf{negl}(\kappa), \quad (1)$$

where the expectation is taken over $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$ and $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{mGen}(\mathsf{pp})$.

Definition A.4 (Ciphertext-Spreadness). For all $\mathsf{pp} \in \mathsf{mSetup}(1^\kappa)$, and $(\mathsf{ek}, \mathsf{dk}) \in \mathsf{mGen}(\mathsf{pp})$, define $\Gamma(\mathsf{pp}, \mathsf{ek})$ as

$$\max_{\mathsf{ct},\mathsf{M}\in\mathcal{M}} \Pr_{\mathsf{r}_0,\mathsf{r}}[\mathsf{ct} = (\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0), \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}, \mathsf{M}; \mathsf{r}_0, \mathsf{r}))].$$

We say mPKE is ciphertext-spread if $\mathbb{E}[\Gamma(\mathsf{pp}, \mathsf{ek})] \leq \mathsf{negl}(\kappa)$, where the expectation is taken over $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$ and $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{mGen}(\mathsf{pp})$.

A.4 Digital Signatures

We provide the standard notion of digital signatures.

Definition A.5 (Signature Scheme). A signature scheme SIG over a message space $\mathcal{M}$ consists of the following algorithms:

- $\mathsf{Setup}(1^\kappa) \rightarrow \mathsf{pp}$: On input the security parameter 1^κ, it outputs a public parameter pp.
- $\mathsf{KeyGen}(\mathsf{pp}) \rightarrow (\mathsf{svk}, \mathsf{ssk})$: On input a public parameter pp it outputs a pair of verification key and signing key $(\mathsf{svk}, \mathsf{ssk})$.
- $\mathsf{Sign}(\mathsf{pp}, \mathsf{ssk}, \mathsf{m}) \rightarrow \mathsf{sig}$: On input a public parameter pp, a signing key ssk and a message m, it outputs a signature sig.
- $\mathsf{Verify}(\mathsf{pp}, \mathsf{svk}, \mathsf{m}, \mathsf{sig}) \rightarrow \top/\perp$: On input a public parameter pp, a verification key ssk, a message m and a signature sig, it outputs $\top$ or $\perp$.

Definition A.6 (Correctness). A signature scheme SIG is correct if for all $\kappa \in \mathbb{N}$, all messages $\mathsf{m} \in \mathcal{M}$ and all $\mathsf{pp} \in \mathsf{Setup}(1^\kappa)$,

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \mathsf{svk}, \mathsf{m}, \mathsf{sig}) = \top : \begin{array}{c} (\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}); \\ \mathsf{sig} \leftarrow \mathsf{Sign}(\mathsf{pp}, \mathsf{ssk}, \mathsf{m}) \end{array}\right] \geq 1 - \mathsf{negl}(\kappa).$$

*Definition A.7 (*sEUF-CMA*).* A signature scheme is sEUF-CMA secure if for all PPT adversary $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{l} \mathsf{Verify}(\mathsf{pp}, \mathsf{svk}, \mathsf{m}^*, \mathsf{sig}^*) = \top \\ \wedge (\mathsf{m}^*, \mathsf{sig}^*) \notin L^* \end{array} : \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa); \\ (\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}); \\ (\mathsf{m}^*, \mathsf{sig}^*) \leftarrow \mathcal{A}^{\mathcal{S}(\cdot)}(\mathsf{pp}, \mathsf{svk}) \end{array}\right] \leq \mathsf{negl}(\kappa),$$

where $\mathcal{S}$ is the signing oracle which on input m returns $\mathsf{Sign}(\mathsf{ssk}, \mathsf{m})$, and L^* is the set of pairs of message and signature generated by the signing oracle.

A.5 Message Authentication Codes

We provide the standard notion of (deterministic) message authentication codes (MAC).

*Definition A.8 (*MAC*).* A (deterministic) message authentication code MAC over a key space $\mathcal{K}$ and a message space $\mathcal{M}$ consists of the following algorithms:

- $\mathsf{TagGen}(\mathsf{k}, \mathsf{m}) \rightarrow \mathsf{tag}$: On input a key $\mathsf{k} \in \mathcal{K}$ and a message $\mathsf{m} \in \mathcal{M}$, it (deterministically) outputs a tag tag.
- $\mathsf{TagVerify}(\mathsf{k}, \mathsf{m}, \mathsf{tag}) \rightarrow \perp/\top$: On input a key k, a message m and a tag tag, it (deterministically) outputs $\top$ or $\perp$.

Since the TagGen algorithm is deterministic, we can simply define TagVerify to run TagGen on (k, M) and check if the generated tag' is identical to the provided tag.

Definition A.9 (Correctness). A MAC is correct if for all keys $\mathsf{k} \in \mathcal{K}$ and all messages $\mathsf{m} \in \mathcal{M}$,

$$\Pr[\mathsf{TagVerify}(\mathsf{k}, \mathsf{m}, \mathsf{TagGen}(\mathsf{k}, \mathsf{m})) = \top] = 1.$$

We define collision resistance of MAC by providing the (non-uniform) adversary oracle access to TagGen and TagVerify, where we implicitly assume these two algorithms are implemented using an internal hash function modeled as a random oracle. We note that natural and practical constructions of a MAC based on a hash function modeled as a random oracle possesses this property.

Definition A.10 (Collision Resistant). A MAC is *collision resistant* if for all PPT adversary $\mathcal{A}$, we have

$$\Pr\left[(\mathsf{k}, \mathsf{m}, \mathsf{k}', \mathsf{m}', \mathsf{tag}) \leftarrow \mathcal{A}^{\mathsf{TagGen},\mathsf{TagVerify}}(1^\kappa) : \begin{array}{c} (\mathsf{k}, \mathsf{m}) \neq (\mathsf{k}', \mathsf{m}') \wedge \\ \mathsf{TagVerify}(\mathsf{k}, \mathsf{m}, \mathsf{tag}) = \\ \mathsf{TagVerify}(\mathsf{k}', \mathsf{m}', \mathsf{tag}) \end{array}\right] \leq \mathsf{negl}(\kappa).$$

A.6 HKDF

HKDF is the key derivation function (KDF) based on HMAC. It consists of the two algorithms HKDF.Extract and HKDF.Expand. The extraction algorithm $\mathsf{k} \leftarrow \mathsf{HKDF.Extract}(s_0, s_1)$ outputs an uniform and random key k if either s_0 or s_1 has high min-entropy. The expansion algorithm $\mathsf{k}_{\mathsf{lbl}} \leftarrow \mathsf{HKDF.Expand}(\mathsf{k}, \mathsf{lbl})$, on input a key k, outputs a random key $\mathsf{k}_{\mathsf{lbl}}$ for (public) label lbl. In the security proof of our Chained CmPKE, we model both HKDF.Extract and HKDF.Expand as a random oracle.

B OMITTED DETAILS FROM SEC. 3

In this section, we provide the omitted details from Sec. 3.

B.1 Omitted Property of CmPKE

Definition B.1 (Ciphertext-Spreadness). For all $\mathsf{pp} \in \mathsf{CmSetup}(1^\kappa)$, and $(\mathsf{ek}_i, \mathsf{dk}_i) \in \mathsf{CmGen}(\mathsf{pp})$ for all $i \in [N]$, define $\Gamma(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$ as

$$\max_{\mathsf{T},i,\mathsf{ct},\mathsf{M}\in\mathcal{M}} \Pr_{\mathsf{r}}[\mathsf{ct} = \mathsf{ct}_i \wedge (\mathsf{T}, (\mathsf{ct}_i)_{i\in[N]}) = \mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M}; \mathsf{r})].$$

We say CmPKE is ciphertext-spread if $\mathbb{E}[\Gamma(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})] \le \mathsf{negl}(\kappa)$, where the expectation is taken over $\mathsf{pp} \leftarrow \mathsf{mSetup}(1^\kappa)$ and $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{mGen}(\mathsf{pp})$ for all $i \in [N]$.

B.2 Proof of Thm. 3.6: IND-CCA Security

We provide the proof of Thm. 3.6. For reference, we restate the statement below.

Theorem B.2. *The* CmPKE *in Fig. 3 is* IND-CCA *secure (resp. with adaptive corruption) assuming the* SKE *is one-time* IND-CCA *secure and the decomposable* mPKE *is* IND-CCA *secure (resp. with adaptive corruption) and ciphertext-spread.*

Proof of Thm. 3.6. Let $\mathcal{A}$ be an adversary against the IND-CCA security of CmPKE with advantage ϵ. Without loss of generality, we make a simplifying argument that $\mathcal{A}$'s random oracle queries to G_1 and G_2 are answered as $(\mathsf{G}_1(\mathsf{M}), \mathsf{G}_2(\mathsf{ek}_1, \mathsf{M}), \cdots, \mathsf{G}_2(\mathsf{ek}_N, \mathsf{M}))$, where $(\mathsf{ek}_i)_{i\in[N]}$ are the encryption keys generated by the security game. It is clear that this modification does not weaken $\mathcal{A}$. Moreover, we can always transform an adversary $\mathcal{A}$ that does not conform to this style to a one that does. Below, we upper bound $\mathcal{A}$'s advantage ϵ by considering a sequence of games. We denote by E_i the event $\mathcal{A}$ wins in Game i.

- Game 1: This is the real IND-CCA security game. By definition $|\Pr[\mathsf{E}_1] - 1/2| = \epsilon$. We assume without loss of generality that the random message $\overline{\mathsf{M}}^* \leftarrow \mathcal{M}$ used to generate the challenge ciphertext is sampled at the beginning of the game.

- Game 2: In this game, we modify the random oracle G so that the output is distributed randomly over the space of randomness for which the decomposable mPKE does not fail decryption. That is, we require $\overline{\mathsf{M}} = \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \widehat{\mathsf{ct}}_i))$ for all $i \in [N]$ and $\overline{\mathsf{M}} \in \mathcal{M}$, where $\mathsf{ct}_0 := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{M}}))$ and $\widehat{\mathsf{ct}}_i := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{M}}; \mathsf{G}_1(\overline{\mathsf{M}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{M}}))$. Due to correctness of the decomposable mPKE, for any $\mathcal{A}$ making at most polynomial random oracle queries, we have $|\Pr[\mathsf{E}_1] - \Pr[\mathsf{E}_2]| \le \mathsf{negl}(\kappa)$.

Game 4 : Decryption. Oracle $\mathcal{D}(i, \mathsf{T}, \mathsf{ct})$

1 : **req** $(\mathsf{T}, \mathsf{ct}) \neq (\mathsf{T}^*, \widehat{\mathsf{ct}}_i^*)$
2 : $(\mathsf{ct}_0, \mathsf{ct}_s) \leftarrow \mathsf{T}$
3 : **if** $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$ **then**
4 : **return** $\mathsf{Dec}_\mathsf{s}(\mathsf{H}(\overline{\mathsf{M}}^*), \mathsf{ct}_s)$
5 : $\overline{\mathsf{M}} := \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \mathsf{ct}))$
6 : **if** $\overline{\mathsf{M}} \notin L_\mathsf{G} \vee \overline{\mathsf{M}} = \perp$ **then**
7 : **return** $\perp$
8 : $\mathsf{ct}_0' := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{M}}))$
9 : $\widehat{\mathsf{ct}}_i' := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{M}}; \mathsf{G}_1(\overline{\mathsf{M}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{M}}))$
10 : **if** $(\mathsf{ct}_0, \mathsf{ct}) \neq (\mathsf{ct}_0', \widehat{\mathsf{ct}}')$ **then**
11 : **return** $\perp$
12 : **return** $\mathsf{Dec}_\mathsf{s}(\mathsf{H}(\overline{\mathsf{M}}), \mathsf{ct}_s)$

Game 5 : Decryption. Oracle $\mathcal{D}(i, \mathsf{T}, \mathsf{ct})$

1 : **req** $(\mathsf{T}, \mathsf{ct}) \neq (\mathsf{T}^*, \widehat{\mathsf{ct}}_i^*)$
2 : $(\mathsf{ct}_0, \mathsf{ct}_s) \leftarrow \mathsf{T}$
3 : **if** $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$ **then**
4 : **return** $\mathsf{Dec}_\mathsf{s}(\mathsf{H}(\overline{\mathsf{M}}^*), \mathsf{ct}_s)$
5 : **foreach** $\overline{\mathsf{M}} \in L_\mathsf{G}$ **do**
6 : $\mathsf{ct}_0' := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{M}}))$
7 : $\widehat{\mathsf{ct}}_i' := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{M}; \mathsf{G}_1(\overline{\mathsf{M}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{M}}))$
8 : **if** $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0', \widehat{\mathsf{ct}}_i')$ **then**
9 : **return** $\mathsf{Dec}_\mathsf{s}(\mathsf{H}(\overline{\mathsf{M}}), \mathsf{ct}_s)$
10 : **return** $\perp$

Figure 9: decryption oracles of Game 4 **and** Game 5**.**

(The next Game 3*,* Game 4 *and* Game 5 *aim to get rid of the secret keys* ek_i *to answer* $\mathcal{A}$*'s decryption oracle queries.)*

- Game 3: In this game, the challenger modifies how it answers the decryption oracle query. When $\mathcal{A}$ queries $(i, \mathsf{T} = (\mathsf{ct}_0, \mathsf{ct}_s), \mathsf{ct})$ such that $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$, the challenger simply returns $\mathsf{Dec}_\mathsf{s}(\mathsf{H}(\overline{\mathsf{M}}^*), \mathsf{ct}_s)$. This is in contrast to the previous game where the challenger decrypted $(\mathsf{ct}_0, \mathsf{ct})$ using mDec. Nonetheless, since the decomposable mPKE is perfect correct due to the modification we made in Game 2, this modification does not alter the view of the adversary. In particular, we have $\Pr[\mathsf{E}_2] = \Pr[\mathsf{E}_3]$.
- Game 4: In this game, the challenger adds an additional check when answering the decryption oracle query. This is illustrated in Fig. 9, where the red underline indicates the modification. Here, L_G is a list that stores the random oracle queries made to G_1 and G_2 by the adversary. We have $\mathsf{M} \in L_\mathsf{G}$ if G_1 was queried on M and G_2 was queried on $(\mathsf{ek}, \mathsf{M})$ for any ek. Note that due to our assumption on $\mathcal{A}$, if one of the oracles G_1 or G_2 was queried on M, then so would have the other.

The only difference occurs when $\mathcal{A}$ queries $(i, \mathsf{T} = (\mathsf{ct}_0, \mathsf{ct}_s), \mathsf{ct})$ such that $\overline{\mathsf{M}} := \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \mathsf{ct}))$ has not been queried to the

random oracles G_1 and G_2 but $\mathsf{ct}_0 = \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{M}}))$ and $\mathsf{ct} = \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{M}}; \mathsf{G}_1(\overline{\mathsf{M}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{M}}))$. Notice $\mathsf{G}_1(\overline{\mathsf{M}})$ and $\mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{M}})$ are information theoretically hidden from $\mathcal{A}$ unless $\mathcal{A}$ queries them. Therefore, due to ciphertext-spreadness of the decomposable mPKE, we must have had $(\mathsf{ct}_0, \mathsf{ct}) \neq (\mathsf{ct}'_0, \widehat{\mathsf{ct}}'_i)$ in the previous game as well. Hence, we have $|\Pr[\mathsf{E}_3] - \Pr[\mathsf{E}_4]| \leq \mathsf{negl}(\kappa)$.

- Game 5: In this game, the challenger further modifies how it answers the decryption-oracle query. This is illustrated in Fig. 9, where notice that the challenger no longer requires the secret keys dk_i to answer the queries.

We check the output of the decryption oracles in Game 4 and Game 5 are identical. Since the two oracles run identically in case $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}^*_0, \widehat{\mathsf{ct}}^*_i)$, we only focus on the case that this does not hold. Assume the decryption oracle in Game 4 outputs a non-$\perp$ message M (i.e., $\mathsf{M} = \mathsf{Dec}_{\mathsf{s}}(\mathsf{H}(\overline{\mathsf{M}}), \mathsf{ct}_s)$). Then $\overline{\mathsf{M}} \in L_{\mathsf{G}}$ and $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}'_0, \widehat{\mathsf{ct}}'_i)$ hold, where $\overline{\mathsf{M}} := \mathsf{mDec}(\mathsf{dk}_i, (\mathsf{ct}_0, \mathsf{ct}))$. Therefore, the decryption oracle in Game 5 outputs the same non-$\perp$ message M. On the other hand, assume the decryption oracle in Game 5 outputs a non-$\perp$ message M. Then, there exists a $\overline{\mathsf{M}} \in L_{\mathsf{G}}$ such that $\mathsf{ct}'_0 := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{G}_1(\overline{\mathsf{M}}))$ and $\widehat{\mathsf{ct}}'_i := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \overline{\mathsf{M}}; \mathsf{G}_1(\overline{\mathsf{M}}), \mathsf{G}_2(\mathsf{ek}_i, \overline{\mathsf{M}}))$ such that $(\mathsf{ct}_0, \mathsf{ct}) = (\mathsf{ct}'_0, \widehat{\mathsf{ct}}'_i)$. Conditioning on no correctness error occurring, $(\mathsf{ct}_0, \mathsf{ct})$ decrypts to $\overline{\mathsf{M}}$. Therefore, this implies that the decryption oracle in Game 4 outputs the same non-$\perp$ message M. Combining the arguments together, we have $\Pr[\mathsf{E}_4] = \Pr[\mathsf{E}_5]$.

- Game 6: In this game, we undo the change we made in Game 2 and alter the output of the random oracles G_1 and G_2 to be over all the randomness space. Due to the same argument we made before, we have $|\Pr[\mathsf{E}_5] - \Pr[\mathsf{E}_6]| \leq \mathsf{negl}(\kappa)$

(We are now ready to invoke IND-CPA *security of the decomposable* mPKE *and* IND-CCA *security of the* SKE*.)*

- Game 7: Let us define QUERY as the event that $\mathcal{A}$ queries the random oracles $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$ on input $\overline{\mathsf{M}}^*$, where $\star$ denotes an arbitrary element. (Recall the change we made in Game 1 for $\overline{\mathsf{M}}^*$.) In this game, the challenger aborts the game and forces $\mathcal{A}$ to output a random bit when QUERY occurs. We show in Lem. B.3 that we have $|\Pr[\mathsf{E}_6] - \Pr[\mathsf{E}_7]| \leq \mathsf{negl}(\kappa)$ assuming the decomposable mPKE is IND-CPA secure (with adaptive corruption) and the message space $\mathcal{M}$ is sufficiently large. So as not to interrupt the main proof, we postpone the proof of Lem. B.3 to the end.

We finally show in Lem. B.4 that assuming the IND-CCA security of the SKE, we have

$$\Pr[\mathsf{E}_7] = \frac{1}{2} + \mathsf{negl}(\kappa).$$

Combining all the bounds together, we obtain the statement in Thm. 3.6.

It remains to prove Lems. B.3 and B.4 below.

Lemma B.3. *We have* $|\Pr[\mathsf{E}_6] - \Pr[\mathsf{E}_7]| \leq \mathsf{negl}(\kappa)$ *assuming the decomposable* mPKE *is* IND-CPA *secure (with adaptive corruption) and the message space* $\mathcal{M}$ *is super-polynomially large.*

Proof. Since the two games are identical unless QUERY occurs, we have $|\Pr[\mathsf{E}_6] - \Pr[\mathsf{E}_7]| \leq \Pr[\mathsf{QUERY}]$. In the following, we upper bound $\Pr[\mathsf{QUERY}]$. Let us construct an IND-CPA adversary $\mathcal{B}$ which runs $\mathcal{A}$ as a subroutine: On input $(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$, $\mathcal{B}$ samples two random messages $\overline{\mathsf{M}}^*_0, \overline{\mathsf{M}}^*_1 \leftarrow \mathcal{M}$ and a random SKE key $\mathsf{k}^* \leftarrow \mathcal{K}$. It then invokes $\mathcal{A}$ on input $(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$. $\mathcal{B}$ can simulate the decryption queries as it no longer requires knowledge of the secret key. When $\mathcal{A}$ corrupts a user, $\mathcal{B}$ simply relays the corruption to its own challenger. Finally, when $\mathcal{A}$ submits $(\mathsf{M}_0, \mathsf{M}_1, S \subseteq [N])$ as its challenge, $\mathcal{B}$ submits $(\overline{\mathsf{M}}^*_0, \overline{\mathsf{M}}^*_1, S)$ to its challenger and receives $(\mathsf{ct}^*_0, (\widehat{\mathsf{ct}}^*_i)_{i\in[S]}) \leftarrow \mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in S}, \overline{\mathsf{M}}^*_b)$ for an unknown randomly chosen bit b. $\mathcal{B}$ then samples a random challenge bit $b' \leftarrow \{0,1\}$ and generates $\mathsf{ct}^*_s \leftarrow \mathsf{Enc}_{\mathsf{s}}(\mathsf{k}^*, \mathsf{M}_{b'})$. It finally provides the challenge ciphertext $(\mathsf{T}^* := (\mathsf{ct}^*_0, \mathsf{ct}^*_s), \vec{\mathsf{ct}}^* := (\widehat{\mathsf{ct}}^*_i)_{i\in S})$ to $\mathcal{A}$. $\mathcal{B}$ outputs $\hat{b} := 0$, if $\overline{\mathsf{M}}^*_0$ is queried to $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$ before $\overline{\mathsf{M}}^*_1$ is; outputs $\hat{b} := 1$, if $\overline{\mathsf{M}}^*_1$ is queried to $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$ before $\overline{\mathsf{M}}^*_0$ is; and a random $\hat{b}$ when neither $\overline{\mathsf{M}}^*_0$ nor $\overline{\mathsf{M}}^*_1$ are queried. Let us denote GOOD (resp. BAD) the event that $\mathcal{A}$ queries $\overline{\mathsf{M}}^*_b$ (resp. $\overline{\mathsf{M}}^*_{1-b}$) before $\overline{\mathsf{M}}^*_{1-b}$ (resp. $\overline{\mathsf{M}}^*_b$) to $\mathsf{H}(\cdot)$, $\mathsf{G}_1(\cdot)$, or $\mathsf{G}_2(\star, \cdot)$. Moreover, let us denote RAND the event that neither $\overline{\mathsf{M}}^*_0$ nor $\overline{\mathsf{M}}^*_1$ are queried. Observe that until either GOOD or BAD occurs, $\mathcal{B}$ simulates the view of Game 6 and Game 7 perfectly to $\mathcal{A}$. Since QUERY is the event that $\overline{\mathsf{M}}^*_b$ is ever queried throughout the game, we have $\Pr[\mathsf{QUERY}] \leq \Pr[\mathsf{GOOD}] + \Pr[\mathsf{BAD}]$. Moreover, since M^*_{1-b} is completely hidden from $\mathcal{A}$, we have $\Pr[\mathsf{BAD}] \leq \mathsf{negl}(\kappa)$ assuming the message size is super-polynomially large and $\mathcal{A}$ only makes polynomially many random oracle queries.

Using these observation, we can rewrite the advantage of $\mathcal{B}$ as follows:

$$\begin{aligned}
&\left|\Pr[\hat{b} = b] - \frac{1}{2}\right| \\
&= \left|\Pr[\hat{b} = b \wedge \mathsf{GOOD}] + \Pr[\hat{b} = b \wedge \mathsf{BAD}] + \Pr[\hat{b} = b \wedge \mathsf{RAND}] - \frac{1}{2}\right| \\
&= \left|\Pr[\mathsf{GOOD}] + \frac{1}{2} \cdot \Pr[\mathsf{RAND}] - \frac{1}{2}\right| \\
&= \left|\Pr[\mathsf{GOOD}] + \frac{1}{2} \cdot (1 - \Pr[\mathsf{GOOD}] - \Pr[\mathsf{BAD}]) - \frac{1}{2}\right| \\
&= \left|\frac{1}{2}(\Pr[\mathsf{GOOD}] - \Pr[\mathsf{BAD}])\right| \\
&\geq \frac{1}{2}(\Pr[\mathsf{QUERY}] - 2\Pr[\mathsf{BAD}]).
\end{aligned}$$

Assuming the hardness of the IND-CPA security of the decomposable mPKE, we have $\left|\Pr[\hat{b} = b] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa)$. Thus, rewriting the inequality and plugging in $\Pr[\mathsf{BAD}] \leq \mathsf{negl}(\kappa)$, we obtain $\Pr[\mathsf{QUERY}] \leq \mathsf{negl}(\kappa)$ as desired. This concludes the proof. □

Lemma B.4. *We have* $\Pr[\mathsf{E}_7] = \frac{1}{2} + \mathsf{negl}(\kappa)$ *assuming the* SKE *is* IND-CCA *secure.*

Proof. Assume $\mathcal{A}$ has advantage ϵ in Game 7. We construct an adversary $\mathcal{B}$ that breaks the IND-CCA security of the SKE with the same advantage by internally running $\mathcal{A}$ as follows:

$\mathcal{B}$ generates $(\mathsf{pp}, (\mathsf{ek}_i, \mathsf{dk}_i)_{i\in[N]})$ and samples a random $\overline{\mathsf{M}}^* \leftarrow \mathcal{M}$ used to generate the challenge ciphertext. $\mathcal{B}$ then invokes $\mathcal{A}$ on input $(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]})$ as in Game 7. When $\mathcal{A}$ queries any of the random oracles on input $\overline{\mathsf{M}}^*$ (i.e., when event QUERY occurs), then abort as specified by the modification we made in Game 6. When

$\mathcal{A}$ queries for a challenge ciphertext on input $(\mathsf{M}_0, \mathsf{M}_1, S \subseteq [N])$, $\mathcal{B}$ first generates $(\mathsf{ct}_0^*, (\widehat{\mathsf{ct}}_i^*))_{i\in[N]}$. It then queries its SKE-challenger for a challenge ciphertext on challenge messages $(\mathsf{M}_0, \mathsf{M}_1)$ and receives back ct_s^*. Finally, $\mathcal{B}$ returns $(\mathsf{T}^* := (\mathsf{ct}_0^*, \mathsf{ct}_s^*), \vec{\mathsf{ct}}^* := (\widehat{\mathsf{ct}}_i^*)_{i\in[N]})$ to $\mathcal{A}$. Here, notice that $\mathcal{B}$ implicitly sets $\mathsf{H}(\overline{\mathsf{M}}^*) = \mathsf{k}^*$, where k^* is the secret key used by the SKE-challenger. When $\mathcal{A}$ queries the decryption oracle on input $(i, \mathsf{T}, \mathsf{ct})$, if $(\mathsf{ct}_0, \mathsf{ct}) \neq (\mathsf{ct}_0^*, \widehat{\mathsf{ct}}_i^*)$, then $\mathcal{B}$ proceeds exactly as in Game 7. Otherwise, it queries its own SKE-decryption oracle on input ct_s (which is guaranteed to be different from ct_s^*) and outputs $\mathcal{A}$ the decryption result. Corruption queries made by $\mathcal{A}$ can be handled as in the real game since $\mathcal{B}$ knows all the user secrets. Finally, when $\mathcal{A}$ outputs b' at the end of the game, $\mathcal{B}$ outputs b' as its guess.

Conditioning on event QUERY not occurring, $\mathcal{B}$ perfectly simulates Game 7 to $\mathcal{A}$. Therefore, $\mathcal{B}$ has the same advantage of winning the IND-CCA security game of SKE as $\mathcal{A}$ does in winning Game 7. Hence, assuming the IND-CCA security of SKE, we conclude $\Pr[\mathsf{E}_7] = \frac{1}{2} + \mathsf{negl}(\kappa)$. □

□

B.3 Proof of Thm. 3.7: Commitment-Binding

We provide the proof of Thm. 3.7. For reference, we restate the statement below.

Theorem B.5. *The* CmPKE *in Fig. 3 is commitment-binding assuming the* SKE *has key commitment.*

Proof. Assume by contradiction that $\mathcal{A}$ breaks commitment-binding of CmPKE. By assumption $\mathcal{A}$ outputs $(\mathsf{T}^*, (\mathsf{dk}_b, \mathsf{ct}_b)_{i\in b})$. Let $\mathsf{T}^* := (\mathsf{ct}_0^*, \mathsf{ct}_s^*)$ and $\mathsf{M}_b \leftarrow \mathsf{CmDec}(\mathsf{dk}_b, \mathsf{T}^*, \mathsf{ct}_b)$ for $b \in \{0, 1\}$. Moreover, let $\overline{\mathsf{M}}_b \leftarrow \mathsf{mDec}(\mathsf{dk}_b, \mathsf{ct}_0^*, \mathsf{ct}_b)$ for $b \in \{0, 1\}$ be the internal random message decrypted while running CmDec.

We first show that we must have $\overline{\mathsf{M}}_0 \neq \overline{\mathsf{M}}_1$. If this does not hold, then in case $\mathsf{dk}_0 = \mathsf{dk}_1$, we must have $(\mathsf{ct}_0^*, \mathsf{ct}_0) = (\mathsf{ct}_0^*, \mathsf{ct}_1)$ due to the re-encryption check during decryption.[9] However, this does not constitute a valid attack. On the other hand, in case $\mathsf{dk}_0 \neq \mathsf{dk}_1$, we have $\mathsf{Dec}_s(\mathsf{H}(\overline{\mathsf{M}}_0), \mathsf{ct}_s^*) = \mathsf{Dec}_s(\mathsf{H}(\overline{\mathsf{M}}_1), \mathsf{ct}_s^*) = \mathsf{M}$. Therefore, this too does not constitute a valid attack either.

Next, conditioning on $\overline{\mathsf{M}}_0 \neq \overline{\mathsf{M}}_1$, we can further assume $\mathsf{H}(\overline{\mathsf{M}}_0) \neq \mathsf{H}(\overline{\mathsf{M}}_1)$ with making negligible difference in the advantage of the adversary since H is modeled as a random oracle. This implies that the adversary implicitly outputs two keys $\mathsf{k}_0 := \mathsf{H}(\overline{\mathsf{M}}_0)$ and $\mathsf{k}_1 := \mathsf{H}(\overline{\mathsf{M}}_1)$ such that $\mathsf{Dec}_s(\mathsf{k}_0, \mathsf{ct}_s^*) = \mathsf{M}_0$ and $\mathsf{Dec}_s(\mathsf{k}_1, \mathsf{ct}_s^*) = \mathsf{M}_1$. However, this contradicts the key commitment property of SKE (regardless of M_0 being the same or different from M_1). This concludes the proof.[10] □

B.4 mPKE with Adaptive Corruption Security

We provide in Fig. 10 the simple generic transformation from any IND-CPA secure decomposable mPKE that is *not* secure against adaptive corruptions into a one that is.

[9] Here, we assume that a decryption key dk implicitly includes the encryption key ek required for reencryption.

[10] To be precise, we will provide the adversary $\mathcal{A}$ against the commitment-binding game oracle access to Enc_s and Dec_s, which are both instantiated using the random oracle to formally invoke the key commitment property of SKE.

It is clear that the construction satisfies correctness. We provide the proof of Lem. B.6, which establishes the IND-CPA security with adaptive corruption.

Lemma B.6. *The decomposable* mPKE *in Fig. 10 is* IND-CPA *secure with adaptive corruption assuming the decomposable* mPKE′ *is* IND-CPA *secure.*

Proof. Let $\mathcal{A}$ be an adversary against the IND-CPA security with adaptive corruption. Consider the following game sequence where the first and last correspond to the case where the challenger uses $b = 0$ and 1 as the challenge, respectively. We denote E_i as the event that $\mathcal{A}$ wins in game Game i.

Game 0 : This is the real security game where the challenger uses $b = 0$ as its challenge. That is, the challenge ciphertext $\vec{\mathsf{ct}}^*$ encrypts the message M_0.

Game 1 : We modify how the challenger creates the challenge ciphertext. Let $\mathbf{b} \in \{0, 1\}^N$ be the random string associated to the decryption keys of each user. That is, let user i's encryption and decryption keys be $\mathsf{ek}_i := (\mathsf{ek}_{i,0}, \mathsf{ek}_{i,1})$ and $\mathsf{dk}_i := (\mathbf{b}_i, \mathsf{dk}_{i,\mathbf{b}_i})$. Then, when $\mathcal{A}$ outputs $(\mathsf{M}_0, \mathsf{M}_1, S \subseteq [N])$, the challenger creates the challenge ciphertext as

$$(\mathsf{ct}_{0,0}, (\widehat{\mathsf{ct}}_{i,\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,\mathbf{b}_i})_{i\in[N]}, \mathsf{M}_0)$$
$$(\mathsf{ct}_{0,1}, (\widehat{\mathsf{ct}}_{i,1-\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,1-\mathbf{b}_i})_{i\in[N]}, \mathsf{M}_0).$$

Recall in the previous game, the challenger sampled a random string $\mathbf{w} \neq \mathbf{b}$ to answer the challenge ciphertext. Due to the winning condition, $\mathcal{A}$ never queries a user $i \in S$ to the corruption oracle. Therefore, $\mathbf{b}_i$ is information theoretically hidden to $\mathcal{A}$ and the challenge ciphertexts are distributed identically in both games. Therefore, we have $\Pr[\mathsf{E}_0] = \Pr[\mathsf{E}_1]$.

Game 2 : We further modify how the challenger creates the challenge ciphertext. When $\mathcal{A}$ outputs $(\mathsf{M}_0, \mathsf{M}_1, S \subseteq [N])$, the challenger creates the challenge ciphertext as

$$(\mathsf{ct}_{0,0}, (\widehat{\mathsf{ct}}_{i,\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,\mathbf{b}_i})_{i\in[N]}, \mathsf{M}_0)$$
$$(\mathsf{ct}_{0,1}, (\widehat{\mathsf{ct}}_{i,1-\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,1-\mathbf{b}_i})_{i\in[N]}, \mathsf{M}_1).$$

We have $|\Pr[\mathsf{E}_1] - \Pr[\mathsf{E}_2]| \leq \mathsf{negl}(\kappa)$ assuming mPKE′ is IND-CPA secure. This can be checked in a straightforward fashion by observing that the only secret information in Game 2 is $(\mathsf{dk}_i := (\mathbf{b}_i, \mathsf{dk}_{i,\mathbf{b}_i}))_{i\in[N]}$, which the reduction can simulate on its own. Namely, the reduction embeds the given encryption keys into $\mathsf{ek}_{i,1-\mathbf{b}_i}$.

Game 3: We further modify how the challenger creates the challenge ciphertext. When $\mathcal{A}$ outputs $(\mathsf{M}_0, \mathsf{M}_1, S \subseteq [N])$, the challenger creates the challenge ciphertext as

$$(\mathsf{ct}_{0,0}, (\widehat{\mathsf{ct}}_{i,\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,\mathbf{b}_i})_{i\in[N]}, \mathsf{M}_1)$$
$$(\mathsf{ct}_{0,1}, (\widehat{\mathsf{ct}}_{i,1-\mathbf{b}_i})_{i\in[S]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,1-\mathbf{b}_i})_{i\in[N]}, \mathsf{M}_0).$$

Swapping the message M_0 and M_1 keeps the distribution of the challenge ciphertext identical following the same argument we made to jump between Game 0 and Game 1. Hence, $\Pr[\mathsf{E}_2] = \Pr[\mathsf{E}_3]$.

At this point, we simply undo the changes we made. For completeness, we explain the games.

$\mathsf{mSetup}(1^\kappa)$

1 : $\mathsf{pp} \leftarrow \mathsf{mSetup}'(1^\kappa)$
2 : **return** pp

$\mathsf{mGen}(\mathsf{pp})$

1 : **foreach** $b' \in \{0,1\}$ **do**
2 : $\quad(\mathsf{ek}_{b'}, \mathsf{dk}_{b'}) \leftarrow \mathsf{mGen}'(\mathsf{pp})$
3 : $b \leftarrow\$ \{0,1\}$
4 : **return** $(\mathsf{ek} := (\mathsf{ek}_0, \mathsf{ek}_1), \mathsf{dk} := (b, \mathsf{dk}_b))$

$\mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M})$

1 : $((\mathsf{ek}_{i,0}, \mathsf{ek}_{i,1}))_{i\in[N]} \leftarrow (\mathsf{ek}_i)_{i\in[N]}$
2 : $\mathbf{w} \leftarrow\$ \{0,1\}^N$
3 : $(\mathsf{ct}_{0,0}, (\widehat{\mathsf{ct}}_{i,\mathbf{w}_i})_{i\in[N]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,\mathbf{w}_i})_{i\in[N]}, \mathsf{M})$
4 : $(\mathsf{ct}_{0,1}, (\widehat{\mathsf{ct}}_{i,1-\mathbf{w}_i})_{i\in[N]}) \leftarrow \mathsf{mEnc}'(\mathsf{pp}, (\mathsf{ek}_{i,1-\mathbf{w}_i})_{i\in[N]}, \mathsf{M})$
5 : **return** $\vec{\mathsf{ct}} := (\mathsf{ct}_0 := (\mathsf{ct}_{0,0}, \mathsf{ct}_{0,1}), (\widehat{\mathsf{ct}}_i := (\widehat{\mathsf{ct}}_{i,0}, \widehat{\mathsf{ct}}_{i,1}))_{i\in[N]})$

$\mathsf{mDec}(\mathsf{dk}, \mathsf{ct})$

1 : $(b, \mathsf{dk}_b) \leftarrow \mathsf{dk}$
2 : $(\mathsf{ct}_0, \widehat{\mathsf{ct}}) \leftarrow \mathsf{ct}$
3 : $(\mathsf{ct}_{0,0}, \mathsf{ct}_{0,1}) \leftarrow \mathsf{ct}_0$
4 : $(\widehat{\mathsf{ct}}_0, \widehat{\mathsf{ct}}_1) \leftarrow \widehat{\mathsf{ct}}$
5 : **foreach** $b' \in \{0,1\}$ **do**
6 : $\quad\mathsf{M}_{b'} := \mathsf{mDec}'(\mathsf{dk}_b, (\mathsf{ct}_{0,b'}, \widehat{\mathsf{ct}}_{b'}))$
7 : $\quad$**if** $\mathsf{M}_{b'} \neq \perp$ **then return** $\mathsf{M}_{b'}$
8 : **return** $\perp$

Figure 10: An IND-CPA **secure with adaptive corruption decomposable** mPKE **from an** IND-CPA **secure decomposable** mPKE′**.**

Game 4: We make the same change we made in Game 2 and swap M_0 to M_1. We have $|\Pr[\mathsf{E}_3] - \Pr[\mathsf{E}_4]| \leq \mathsf{negl}(\kappa)$ assuming mPKE′ is IND-CPA secure.

Game 5: We make the same change we made in Game 1 and use $\mathbf{w}$ instead of $\mathbf{b}$ to answer the challenge ciphertext. We have $\Pr[\mathsf{E}_4] = \Pr[\mathsf{E}_5]$. This corresponds to the real game where $b = 1$ is chosen.

Collecting all the bounds, we conclude $|\Pr[\mathsf{E}_1] - \Pr[\mathsf{E}_5]| \leq \mathsf{negl}(\kappa)$. This completes the proof. □

C CONTINUOUS GROUP KEY AGREEMENT

In this section, we define the syntax and security of continuous group key agreement (CGKA) protocols. We adopt the state-of-the-art syntax and (UC) security model presented in [13], which was used to analyze TreeKEM in MLS version 10. The model presented by [13] is an extension of those presented by [12] that further considers *insider* security.

C.1 Syntax

Proposal and Commit. As in the TreeKEM discussed by the current MLS group, we follow a 'propose-and-commit' flow where current group members propose to add new members, remove existing ones, or update their own keys by sending *proposal* messages. These proposals only take effect when a group member initiates a new epoch by issuing a commit message, i.e., a special message that commits to the (subset of) proposals. Upon receiving such commit message, a party applies the now committed proposals and transitions to the new epoch.

Akin to the recent specification of TreeKEM and also considered in [13], we require the proposals to be ordered. Namely, proposals are structured as a vector where it contains all update, then all removes, and finally all adds in this order. As done by prior work, we delegate the buffering of proposals to the high-level protocol.

Formal Syntax. We extend the syntax presented in [13] so that the commit and welcome messages can be divided into two parts. Commit (and welcome) message consists of a party *independent* message and a party *dependent* message. When the server receives a request from a party id, it constructs the necessary packets for id from the stored commit message (which is initially created for all the group member) and only sends the portion of the commit message necessary for id. In other words, during a process operation, each receiving party takes a party independent message and the receiver dependent message.

We consider a stateful protocol for a single group that takes the following inputs. Below, we assume the protocol knows the party's identity id running the protocol:

Group Creation ($\texttt{Create}$, svk)**:** It initializes a new group state. Only the party id using the verification key svk belongs to this group. In our model, Group Creation is only allowed once.

Add Proposals ($\texttt{Propose}$, 'add'-id_t) $\rightarrow p$**:** It outputs a message p proposing to add a party id_t, or $\perp$ if either id is not in the group or it tries to add an id_t that already belongs to the group.

Remove Proposals ($\texttt{Propose}$, 'rem'-id_t) $\rightarrow p$**:** It outputs a message p proposing to remove a party id_t, or $\perp$if either id is not in the group or it tries to remove an id_t that is not in the group.

Update proposals ($\texttt{Propose}$, 'upd'-svk) $\rightarrow p$**:** It outputs a message p proposing to update the party id's key material, and optionally the signature key svk, or $\perp$ if id is not in the group.

Commit ($\texttt{Commit}$, $\vec{p}$, svk) $\rightarrow (c_0, \vec{c}, w_0, \vec{w})$**:** It commits a vector of proposals $\vec{p}$ and outputs a commit message $(c_0, \vec{c})$. c_0 is a party independent message while $\vec{c} = (\widehat{c}_{\mathsf{id}'})_{\mathsf{id}'}$ is a vector of party dependent messages $\widehat{c}_{\mathsf{id}'}$ designated to id'. If $\vec{p}$ contains at least one add proposal, then it outputs a welcome message $(w_0, \vec{w})$. As in a commit message, w_0 is a party independent message and $\vec{w} = (\widehat{w}_{\mathsf{id}_t})_{\mathsf{id}_t}$ is a vector of party dependent messages. The operation optionally updates the committer's signature key svk.

Process ($\texttt{Process}$, $c_0, \widehat{c}_{\mathsf{id}}, \vec{p}$) $\rightarrow (\mathsf{id}_c, \mathsf{propSem})$**:** It processes a message $(c_0, \widehat{c}_{\mathsf{id}})$ and committed proposals $\vec{p}$, and advances id to the next epoch. It outputs the committer's identity id_c and the semantics of the applied proposals.

Join ($\texttt{Join}$, $w_0, \widehat{w}_{\mathsf{id}}$) $\rightarrow (\mathsf{id}_c, \mathsf{mem})$**:** It allows id (who is not yet a group member) to join the group using the welcome message $(w_0, \widehat{w}_{\mathsf{id}})$. It outputs the committer's identity id_c and mem, the set of a pair of identity and signature key of all group members.

Key $\texttt{Key} \rightarrow \mathbf{k}$**:** It outputs the current group key. This can be queried once every epoch by any group member (otherwise retuning $\perp$).

We note that we omit 'add-only' mode of commits in MLS for simplicity. This allows for a special commit where the proposals $\vec{p}$ consist of all add proposals. In such case, MLS allows to permit skipping the implicit update performed by the committer. Our construction naturally handles this 'add-only' mode as well.

C.2 Security Model

We adopt the universally composable (UC) security model presented in [13] with some modifications. The model of [13] is an extension of the UC model presented in [12] that captures the strong notion of *insider security*. Here, a corrupted party can not only send arbitrary network packets but can further interact with the PKI to inject maliciously generated long-term keys and key packages. In our model, since we allow the delivery service (i.e., environment) to sanitize commit messages by delivering to each group member the strict amount of data they need, we further extend [12, 13] in the following way:

- We extend the input and output interface of the ideal functionality according to the new format of commit message and welcome message. The commit function outputs a commit and welcome messages for all the receivers while the process and join functions only take the relevant part of the commit and welcome messages as input. Accordingly, we modify what to store in the commit node of the history graph. (In previous works, since the receiver downloads the same content as those uploaded by the sender, the commit node was simply defined by the uploaded content.) We note that prior constructions can be handled within our new extended model, thus our model is as general as the previous ones.
- We separate the **inj-allowed** predicate into two predicates **sig-inj-allowed** and **mac-inj-allowed**. This is only a conceptual modification but we believe it allows for a more modular proof since we can differentiate between different types of injected messages, i.e., injected by forging a MAC or a signature.

C.2.1 Universal Composable Security. The following description in this sub-section is taken almost verbatim from [13, Sec.2.2 and Sec.3.2]. For further details we refer the readers to [12, 13].

We formalize security in the generalized universal composability (GUC) framework [34], an extension to the UC framework [33]. We moreover use the modification of responsive environments introduced by Camenisch et al. [32] to avoid artifacts arising from seemingly local operations (such as sampling randomness or producing a ciphertext) to involve the adversary.

The (G)UC framework requires a real-world execution of the protocol to be indistinguishable from an ideal world, to an interactive environment. The real-world experiment consists of the group members executing the protocol (and interacting with the PKI setup). In the ideal world, on the other hand, the protocol is replaced by dummy instances that just forward all inputs and outputs to an ideal functionality characterizing the appropriate guarantees. The functionality interacts with a so-called simulator, that translates the real-world adversary's actions into corresponding ones in the ideal world. Since the ideal functionality is secure by definition, this implies that the real-world execution cannot exhibit any attacks either.

The Corruption Model. We use the — standard for CGKA/SGM but non-standard for UC — corruption model of continuous state leakage (transient passive corruptions) and adversarially chosen randomness of [12]. This corruption model allows the adversary to repeatedly corrupt parties by sending them two types of corruption messages: (1) a message `Expose` causes the party to send its current state to the adversary (once), (2) a message $(\texttt{CorrRand}, b)$ sets the party's rand-corrupted flag to b. If b is set, the party's randomness-sampling algorithm is replaced by the adversary providing the coins instead. Ideal functionalities are activated upon corruptions and can adjust their behavior accordingly.

Restricted Environments. In order to avoid the so-called commitment problem, caused by adaptive corruptions in simulation-based frameworks, we restrict the environment not to corrupt parties at certain times. We consider a weakened variant of UC security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment is admissible or not is defined by the ideal functionality $\mathcal{F}$ with statements of the form **restrict** *cond* and an environment is called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such *cond* when interacting with $\mathcal{F}$.

Security via Idealized Services. We consider an ideal CGKA functionality that represents an idealized "CGKA service" agnostic to the usage of the protocol. That is, whenever a party performs a certain group operation (e.g., creating a proposal or commit) the functionality simply hands back an idealized protocol message to that party — it is then up to the environment to deliver those protocol messages to the other group members, thus not making any assumptions on the underlying network or the architecture of the delivery service. Additionally, this also allows us to consider correctness and robustness guarantees, in contrast to more "classical" UC treatments that let the adversary deliver the messages. (Such models typically permit trivial protocols that just reject all messages with the simulator just not delivering them in the ideal world.)

The Real-World Experiment. In the real-world experiment, the parties execute the protocol that furthermore interacts with the Authentication Service (AS) and Key Service (KS) PKI functionalities. For instance, the environment can instruct the Authentication Service (via the party's protocol) to register a new key for a party. As a result, the AS generates a new key pair for the party and hands the public key to the environment, making the secret key available to the party's protocol upon request. The PKI is defined in detail in the next section.

The Ideal World. The ideal world formalizes the security guarantees via the ideal functionality $\mathcal{F}_{\text{CGKA}}$, which internally maintains a so-called history graph. The history graph is a labeled directed graph that acts as a symbolic representation of a group's evolution. It has two types of nodes: commit and proposal nodes, representing all executed commit and propose operations, respectively. Note that each commit node represents an epoch. The nodes' labels, furthermore, keep track of all the additional information relevant for defining security. For instance, proposal nodes have a label that stores the proposed action, and commit nodes have labels that store the epoch's application secret and the set of parties corrupted in the given epoch. Security of the application secrets is then formalized by the functionality choosing a random and independent key

for each commit node whenever security is guaranteed; otherwise the simulator gets to choose the key. Whether security is guaranteed in given node, is determined via an explicit safe predicate on the node and the history graph. In addition to the secrecy of the keys, the functionality also formalizes authenticity by appropriately disallowing injections. As the PKI management is exposed to the environment in the real world, we consider "ideal-world variants" of the AS and KS interacting with $\mathcal{F}_{\text{CGKA}}$. Those variants essentially record which keys have been exposed, which in turn is then used to define the safe predicate. The actual keys in the ideal world do not convey any particular meaning beyond serving as identifiers — thus in the ideal world we can leak all secret keys to the simulator (they are necessary to simulate signatures on protocol messages). We note that this roughly corresponds to treating the PKI setup as local rather than global (in the sense UC versus GUC).

C.2.2 PKI functionality. We model untrusted PKI, where the adversary can register arbitrary signature keys for any party. This models insider adversary.

Authentication Service (AS). The AS certifies the ownership of a signature key. The functionality $\mathcal{F}_{\text{AS}}$ is defined in Fig. 12. $\mathcal{F}_{\text{AS}}$ allows a party, identified by id, to register a fresh signature key pair via `register-svk` query and verifies whether a verification key svk is registered by another party via certSvks query. On registration, the new key pair for a party id is generated by $\mathcal{F}_{\text{AS}}$ using genSsk() algorithm (see Fig. 11). If id's current randomness is corrupted (i.e., RandCor[id] = 'bad'), $\mathcal{F}_{\text{AS}}$ asks the adversary to provide the randomness. After registration, the party id receives the new verification key svk. A party id can retrieve its signing keys via `get-ssk` query and delete signing keys via `del-ssk` query.

The adversary can register arbitrary verification keys in the name of any party. Moreover, when a party is corrupted, all signing keys except for the deleted ones are leaked to the adversary. Security is modeled by the ideal-world variant of $\mathcal{F}_{\text{AS}}$, called $\mathcal{F}_{\text{AS}}^{\text{IW}}$. It marks leaked and adversarially registered keys as exposed (see boxes in Fig. 12).

$\mathcal{F}_{\text{AS}}$ allows the Key Service functionality $\mathcal{F}_{\text{KS}}$ to signal that a certain ssk is leaked. $\mathcal{F}_{\text{KS}}$ sends this signal when the signature key is leaked due to the leakage of key packages.

Finally, $\mathcal{F}_{\text{AS}}^{\text{IW}}$ always leaks all signing keys to the simulator.

Key Service (KS). The KS allows parties to upload one-time key package used to add them to groups while they are offline.

The KS is formalized by the functionality $\mathcal{F}_{\text{KS}}$ defined in Fig. 13. Similar to $\mathcal{F}_{\text{AS}}$, a party id can register a key package via `register-kp` query. Upon receiving `register-kp` query, $\mathcal{F}_{\text{KS}}$ generates a new key package using genKp(id, svk, ssk) algorithm (see Fig. 11), which takes on input the party's identity id and its signature key pair (svk, ssk) and outputs a key package and the corresponding decryption key. If id's randomness is corrupted, $\mathcal{F}_{\text{KS}}$ uses the randomness provided from the adversary. Moreover, signatures generated with bad randomness may leak the signing key ssk. Hence, $\mathcal{F}_{\text{KS}}$ signals to $\mathcal{F}_{\text{AS}}$ that svk is exposed and sends ssk to the adversary.

Parties can request another party's key package via `get-kp` query. The returned key package is specified by the adversary reflecting that we allow the adversary to maliciously inject key packages that were not registered by honest parties. Finally, parties can retrieve all their (not yet deleted) decryption keys alongside the respective key package via `get-keys` query. The other queries are analogous to $\mathcal{F}_{\text{AS}}$.

C.2.3 History Graph. As in [13], we use the history graph to manage sent or received messages. A history graph contains proposal nodes and commit nodes. All nodes in the history graph stores the following values:

- orig: the identity of the party who created the node, i.e., the message sender.
- par: the parent commit node, representing the sender's current epoch.
- stat ∈ { 'good', 'bad', 'adv' }: the status flag indicating whether the secrets corresponding to the node is known to the adversary. 'good' means this node is secure, 'bad' means this node is created with adversarial randomness (hence it is well-formed but the adversary knows the secret), and 'adv' means this node is created by the injected message from the adversary.

Proposal nodes further store the following values:

- act ∈ { 'upd'-svk, 'add'-id_t-svk_t, 'rem'-id_t }: the proposal action. The history graph also stores the signature verification key svk. 'add'-id_t-svk_t means id_t is added with the verification key svk_t.

Commit nodes further store the following values:

- pro: the ordered list of committed proposals.
- mem: the list of a pair of group member's identity and its signature verification key.
- key: the group (application) secret key.
- chall: the flag indicating whether the group key is challenged. That is, chall = `true` if a random group key was generated for this node, and `false` if the key was set by the adversary (or not generated).
- exp: the set keeping track of corrupted parties in this node. It includes a flag whether only their secret state is leaked (the flag is false), or also the current group key is leaked (the flag is true).

C.2.4 CGKA Functionality. Using the history graph and the PKI functionality, we introduce the ideal functionality $\mathcal{F}_{\text{CGKA}}$, formally defined in Figs. 14 to 16 with the helper functions in Figs. 17 to 19. $\mathcal{F}_{\text{CGKA}}$ is parameterized by the predicates **safe**, **sig-inj-allowed** and **mac-inj-allowed**, which specify which epoch secrets are secure and when authenticity is guaranteed. The predicates are defined in Fig. 28. In previous works [12, 13], **sig-inj-allowed** and **mac-inj-allowed** were handled by a single predicate **inj-allowed** that checked any injection regardless of it being a forgery of the MAC or signature. We intentionally divide the **inj-allowed** predicate into two predicates to make the proof more accessible. This modification is merely conceptual. Moreover, we add an explicit additional check for **sig-inj-allowed** in case id is assigned to a detached root (highlighted in Fig. 19). This was implicitly checked by the simulator in previous works and we only made it explicit. Namely, the inclusion of this check is aimed to improve the readability of the ideal functionality, and has no effect on the security.

Below, we extend the input and output interface of the functionality according to the new format of commit message and welcome message.

genSsk()

1 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{SIG.KeyGen}(\mathsf{pp}_{\mathsf{SIG}})$
2 : **return** $(\mathsf{svk}, \mathsf{ssk})$

genKp(id, svk, ssk)

1 : $s \leftarrow\!\$\ \{0,1\}^{\kappa}$
2 : $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{CmGen}(\mathsf{pp}_{\mathsf{CmPKE}}; \mathsf{H}(s))$
3 : $\mathsf{sig} \leftarrow \mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{ssk}, (\mathsf{id}, \mathsf{ek}, \mathsf{svk}))$
4 : $\mathsf{kp} \leftarrow (\mathsf{id}, \mathsf{ek}, \mathsf{svk}, \mathsf{sig})$
5 : **return** $(\mathsf{kp}, \mathsf{dk})$

Figure 11: Key generation algorithms.

Initialization

1 : Registered $\leftarrow \emptyset$; $\boxed{\text{Exposed} \leftarrow \emptyset}$
2 : $\mathsf{SSK}[*,*] \leftarrow \perp$
3 : $\mathsf{RandCor}[*] \leftarrow$ 'good'

Inputs from a party id

Input (`register-svk`)

1 : **if** $\mathsf{RandCor}[\mathsf{id}] =$ 'good' **then**
2 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{genSsk}()$
3 : **else**
4 : Send $(\mathsf{rnd}, \mathsf{id})$ to the adversary and receive r
5 : $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{genSsk}(r)$
6 : $\boxed{\text{Exposed} \mathrel{+}\!\leftarrow \mathsf{svk}}$
7 : Registered $\mathrel{+}\!\leftarrow (\mathsf{id}, \mathsf{svk})$
8 : $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \leftarrow \mathsf{ssk}$
9 : Send $(\texttt{register-svk}, \mathsf{id}, \mathsf{svk}, \boxed{\mathsf{ssk}})$ to the adversary
10 : Send svk to the party id

Input (`get-ssk`, svk) from id

1 : Send $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}]$ to id

Input (`del-ssk`, svk)

1 : $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \leftarrow \perp$

Input (`verify-cert`, id′, svk) from id

1 : Send $(\mathsf{id}', \mathsf{svk}) \in$ Registered to id

Inputs from the adversary

Input (`register-svk`, id, svk)

1 : $\boxed{\textbf{if } (*, \mathsf{svk}) \notin \text{Registered } \textbf{then}}$
2 : $\boxed{\text{Exposed} \mathrel{+}\!\leftarrow \mathsf{svk}}$
3 : Registered $\mathrel{+}\!\leftarrow (\mathsf{id}, \mathsf{svk})$

Input (`expose-ssk`, id)

1 : $\boxed{\text{Exposed} \mathrel{+}\!\leftarrow \{\ \mathsf{svk} \mid \mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \neq \perp\ \}}$
2 : Send $\mathsf{SSK}[\mathsf{id}, *]$ to the adversary

Input (`CorrRand`, id, b), $b \in \{$ 'good', 'bad' $\}$

1 : $\mathsf{RandCor}[\mathsf{id}] \leftarrow b$

Inputs from $\mathcal{F}_{\mathbf{CGKA}}$ and $\mathcal{F}_{\mathbf{KS}}$

Input (`exposed`, id, svk)

1 : $\boxed{\text{Exposed} \mathrel{+}\!\leftarrow \mathsf{svk}}$
2 : Send $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}]$ to the adversary

Inputs from $\mathcal{F}_{\mathrm{CGKA}}$

Input (`has-ssk`, id, svk)

1 : Send $\mathsf{SSK}[\mathsf{id}, \mathsf{svk}] \neq \perp$ to $\mathcal{F}_{\mathrm{CGKA}}$

Figure 12: The ideal authentication service functionality $\mathcal{F}_{\mathbf{AS}}$ and its variant $\boxed{\mathcal{F}_{\mathbf{AS}}^{\mathbf{IW}}}$ used during the security proof.

States. $\mathcal{F}_{\mathrm{CGKA}}$ maintains the history graph. It addresses proposal nodes by (idealized) proposal message p and non-root commit nodes by (idealized) proposal messages c_0. We consider the single main group. The root node corresponding to the main group is addressed by the special label root_0. Moreover, other roots may be created without a commit message (e.g., when a party uses an injected welcome message to an adversarially created epoch, which is not directly related to the main group). Such roots are addressed by the special labels root_{rt} for $rt \in \mathbb{N}$ and their tree are called *detached*.

$\mathcal{F}_{\mathrm{CGKA}}$ also stores a pointer $\mathsf{Ptr}[\mathsf{id}]$ for each party id. $\mathsf{Ptr}[\mathsf{id}]$ indicates id's current commit node (i.e., current epoch). If id currently is not in the group, $\mathsf{Ptr}[\mathsf{id}] = \perp$.

Interfaces. $\mathcal{F}_{\mathrm{CGKA}}$ offers interfaces for creating group, creating a proposal, committing a list of proposals, processing a commit, joining a group, and retrieving the current group key. We assume the main group is created by the designated party $\mathsf{id}_{\mathsf{creator}}$. Initially, the main group has a single party $\mathsf{id}_{\mathsf{creator}}$, and it can invite additional members. All interface except create and join are for group members only (i.e., parties for which $\mathsf{Ptr}[\mathsf{id}] \neq \perp$).

Proposals. When a party id create a proposal, $\mathcal{F}_{\mathrm{CGKA}}$ notifies the adversary. Then it returns a flag *ack*, a node identifier p (i.e., a message) and a signature verification key svk_t. $\mathcal{F}_{\mathrm{CGKA}}$ allows the adversary to send $ack = \texttt{false}$ to report that the protocol fails, i.e., the output is $p = \perp$. If the protocol succeeds, and if no node with identifier p exists, $\mathcal{F}_{\mathrm{CGKA}}$ creates a new proposal node $\mathsf{Prop}[p]$. For add proposals, it extends the action by the verification key svk_t (specified by the adversary) of the added party id_t.

Initialization

1 : DK[∗, ∗] ← ⊥; SVK[∗, ∗] ← ⊥
2 : RandCor[∗] ← 'good'

Inputs from a party id

Input (register-kp, svk, ssk)

1 : **if** RandCor[id] = 'good' **then**
2 : (kp, dk) ← genKp(id, svk, ssk)
3 : **if** kp = ⊥ **then return**
4 : **else**
5 : Send (rnd, id) to the adversary and receive r
6 : (kp, dk) ← genKp(id, svk, ssk; r)
7 : **if** kp = ⊥ **then return**
8 : Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$
9 : DK[id, kp] ← dk; SVK[id, kp] ← svk
10 : Send (register-kp, id, svk, kp, $\boxed{\text{dk}}$) to the adversary
11 : Send kp to the party id

Input (get-dks)

1 : Send { (kp, DK[id, kp]) | DK[id, kp] ≠ ⊥ } to id

Input (get-kp, id′)

1 : Send (get-kp, id, id′) to the adversary and receive kp
2 : Send kp to id

Input (del-dk, kp)

1 : DK[id, kp] ← ⊥

Inputs from the adversary

Input (CorrRand, id, b), $b \in$ { 'good', 'bad' }

1 : RandCor[id] ← b

Inputs from the adversary and $\mathcal{F}_{\text{CGKA}}$

Input (exposed, id)

1 : Send DK[id, ∗] to the adversary
2 : **foreach** svk ∈ SVK[id, ∗] **s.t.** svk ≠ ⊥ **do**
3 : Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$

Figure 13: The ideal key service functionality $\mathcal{F}_{\text{KS}}$ and and its variant $\boxed{\mathcal{F}_{\text{KS}}^{\text{IW}}}$ used during the security proof.

In certain situations, $\mathcal{F}_{\text{CGKA}}$ may not create a new proposal node. For example, id proposes to remove the same party twice in the same epoch. Another such situation is that a party proposes to update using the same randomness. In these cases, the adversary can specify the preexisting p. $\mathcal{F}_{\text{CGKA}}$ enforces that the states on the existing node is consistent to the expected one using ∗consistent-prop.

Finally, $\mathcal{F}_{\text{CGKA}}$ returns the proposal identifier p to the calling party id.

Commits. When id creates a commit message, it specifies a list of proposals $\vec{p}$, a (possibly fresh) signature verification key svk. Then $\mathcal{F}_{\text{CGKA}}$ forwards the all inputs to the adversary and receives a flag *ack* and identifiers c_0 of commit node with a list $\vec{c}$ and w_0 of welcome node with a list $\widehat{w}$. $\vec{c}$ (resp. $\vec{w}$) contains the party dependent information $\widehat{c}_{\text{id}}$ (resp. $\widehat{w}_{\text{id}}$), and it is used when id processes the message c_0 (resp. w_0). The adversary sets *ack* := false to report that the protocol fails. If the commit protocol succeeds, $\mathcal{F}_{\text{CGKA}}$ first asks the adversary to interpret the injected proposals, i.e., proposal where no node has been created, by calling ∗fill-prop. It then computes the member set resulting from applying $\vec{p}$ by calling ∗members (which returns ⊥ if $\vec{p}$ is invalid).

Then $\mathcal{F}_{\text{CGKA}}$ either creates a new commit node or verifies that the existing node is consistent (cf. ∗consistent-com). It may happen that the existing node is the detached root. In such case, $\mathcal{F}_{\text{CGKA}}$ attaches it to id's current node calling ∗attach. This helper assigns c_0 as the proper identifier of the detached root and deletes the root. Once the detached root is attached, the root's tree achieves the same security guarantee as the main group. Since attaching a detached root changes the the history graph, $\mathcal{F}_{\text{CGKA}}$ enforces two invariants: cons-invariant enforcing the consistency of the graph; and auth-invariant enforcing the authenticity guarantee.

Finally, when add proposals are committed, $\mathcal{F}_{\text{CGKA}}$ records the welcome message that leads the new member to the created commit node. Then $\mathcal{F}_{\text{CGKA}}$ returns $(c_0, \vec{c}, w_0, \vec{w})$ to the calling party id.

Processing Commits. When id processes a commit message, it specifies the commit message $(c_0, \widehat{c})$, where $\widehat{c}$ is the id-dependent message, and a list of committed proposals $\vec{p}$. Then $\mathcal{F}_{\text{CGKA}}$ forwards all the inputs to the adversary and receives the interpreted result from $(c_0, \widehat{c})$.

If the processing succeeds, $\mathcal{F}_{\text{CGKA}}$ either creates a new commit node or verifies that the existing node is consistent. If corresponding nodes do not exist, $\mathcal{F}_{\text{CGKA}}$ checks the validity of $\vec{p}$ and creates a new commit node with the committer identity orig′ and its signature key svk′ which are interpreted by the adversary from $(c_0, \widehat{c})$. If the node Node[c_0] ≠ ⊥ exists, $\mathcal{F}_{\text{CGKA}}$ enforces that it is a valid successor of id's current node (cf. ∗valid-successor). If c_0 matches a detached root, $\mathcal{F}_{\text{CGKA}}$ attaches them.

Finally, depending on whether c_0 removes id, $\mathcal{F}_{\text{CGKA}}$ either moves id's pointer Ptr[id] to the new node or sets the pointer to ⊥. The calling party receives the committer's identity and the semantics of the applied proposals.

Joining. When a party id joins a group, it specifies the welcome message w_0 and the id-dependent message $\widehat{w}$. Then $\mathcal{F}_{\text{CGKA}}$ forwards all the inputs to the adversary and receives the interpreted result from $(w_0, \widehat{w})$. As usual, the adversary sets *ack* := false to report that the protocol fails.

If the processing succeeds, $\mathcal{F}_{\text{CGKA}}$ identifies the commit c_0 = Wel[w_0] corresponding to w_0. If this is the first time $\mathcal{F}_{\text{CGKA}}$ sees w_0, i.e., Wel[w_0] = ⊥, the adversary chooses c_0'. If the commit node for c_0' does not exist (i.e., Node[c_0'] = ⊥), $\mathcal{F}_{\text{CGKA}}$ creates a new detached root where all stored values are chosen by the adversary.

Finally, $\mathcal{F}_{\text{CGKA}}$ returns the state of the joining group (the committer's identity and the list of (id, svk)-pair) to the calling party id.

Group keys and Corruptions. Parties can fetch the current group key via Key query. The Key is random if the protocol guarantees its secrecy as identified by the **safe** predicate. Otherwise, the key is set by the adversary.

The predicate **safe** uses information which are recorded by $\mathcal{F}_{\text{CGKA}}$. When the state of a current group member id is exposed, $\mathcal{F}_{\text{CGKA}}$ records leakage of the following information.

- The current group key (if not retrieved yet) and any key materials (e.g., encryption key and signing key) to process future messages. This is recorded by adding the pair (id, HasKey[id]) to the exposed set of id's current node (cf. line 2 in (Expose, id)). The flag HasKey[id] indicates whether id currently stores the group key (if the group key has not calculated yet or was already retrieved, HasKey[id] = false).
- The key material for updates and commits created by id in the current epoch. This is recorded by setting the status of all child nodes created by id (i.e., nodes with par = Ptr[id]) to 'bad' (cf. *update-stat-after-exp function).
- The current signature signing key ssk. This is recorded by signaling to $\mathcal{F}_{\text{AS}}$ that svk is exposed and sends ssk to the adversary (cf. line 5 in (Expose, id)).

In addition, exposure of party id who is not a group member reveals key packages that will be used to process welcome messages. $\mathcal{F}_{\text{CGKA}}$ signals to $\mathcal{F}_{\text{KS}}$ that key packages (including signing key) are exposed and sends the corresponding decryption keys and signing keys to the adversary (cf. line 6 in (Expose, id)).

Adaptive corruptions become a problem if the adversary reveals a key material that can be used to compute a group key which has already been outputted as random by $\mathcal{F}_{\text{CGKA}}$, i.e., the challenge key. Hence, we restrict the environment not to corrupt key materials such that it would cause **safe** to switch to false for some commit nodes with chall = true.

Initialization

1 : $\mathsf{Ptr}[*], \mathsf{Node}[*], \mathsf{Prop}[*], \mathsf{Wel}[*] \leftarrow \bot$
2 : $\mathsf{RandCor}[*] \leftarrow$ 'good'; $\mathsf{HasKey}[*] \leftarrow \mathtt{false}$; $\mathsf{rootCtr} \leftarrow 0$

Inputs from a party $\mathsf{id}_{\mathsf{creator}}$

Input $(\mathtt{Create}, \mathsf{svk})$

1 : **req** $\mathsf{Node}[\mathsf{root}_0] = \bot$
2 : **req** $*\mathtt{valid\text{-}svk}(\mathsf{id}_{\mathsf{creator}}, \mathsf{svk})$
3 : $\mathsf{mem} \leftarrow \{ (\mathsf{id}_{\mathsf{creator}}, \mathsf{svk}) \}$
4 : $\mathsf{Node}[\mathsf{root}_0] \leftarrow *\mathtt{create\text{-}root}(\mathsf{id}_{\mathsf{creator}}, \mathsf{mem}, \mathsf{RandCor}[\mathsf{id}_{\mathsf{creator}}])$
5 : $\mathsf{HasKey}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \mathtt{true}$; $\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \mathsf{root}_0$
6 : Send $(\mathtt{Create}, \mathsf{id}_{\mathsf{creator}}, \mathsf{svk})$ to the adversary

Inputs from a party id

Input $(\mathtt{Propose}, \mathsf{act})$, $\mathsf{act} \in \{$ 'upd'-svk, 'add'-id_t, 'rem'-id_t $\}$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : Send $(\mathtt{Propose}, \mathsf{id}, \mathsf{act})$ to the adversary and receive $(\mathit{ack}, p, \mathsf{svk}_t)$
3 : **req** *ack*
4 : **if** act = 'upd'-svk **then req** $*\mathtt{valid\text{-}svk}(\mathsf{id}, \mathsf{svk})$
5 : **if** act = 'add'-id_t **then** act $\leftarrow$ 'add'-id_t-svk_t
6 : **if** $\mathsf{Prop}[p] = \bot$ **then**
7 : $\quad \mathsf{Prop}[p] \leftarrow *\mathtt{create\text{-}prop}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \mathsf{act}, \mathsf{RandCor}[\mathsf{id}])$
8 : **else**
9 : $\quad *\mathtt{consistent\text{-}prop}(p, \mathsf{id}, \mathsf{act})$
10 : **if** act = 'upd'-svk $\wedge$ $\mathsf{RandCor}[\mathsf{id}]$ = 'bad' **then**
11 : $\quad$ Send $(\mathsf{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$
12 : **return** p

Input $(\mathtt{Commit}, \vec{p}, \mathsf{svk})$

1 : **req** $\mathsf{Ptr}[\mathsf{id}] \neq \bot$
2 : Send $(\mathtt{Commit}, \mathsf{id}, \vec{p}, \mathsf{svk})$ to the adversary and receive $(\mathit{ack}, \mathit{rt}, c_0, \vec{c}, w_0, \vec{w})$
3 : **req** $*\mathtt{succeed\text{-}com}(\mathsf{id}, \vec{p}, \mathsf{svk}) \vee \mathit{ack}$
4 : $*\mathtt{fill\text{-}prop}(\mathsf{id}, \vec{p})$
5 : **req** $*\mathtt{valid\text{-}svk}(\mathsf{id}, \mathsf{svk})$
6 : $(\mathsf{mem}, *) \leftarrow *\mathtt{members}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \mathsf{svk})$
7 : **assert** $\mathsf{mem} \neq \bot \wedge (\mathsf{id}, \mathsf{svk}) \in \mathsf{mem}$
8 : **if** $\mathsf{Node}[c_0] = \bot \wedge \mathit{rt} = \bot$ **then**
9 : $\quad \mathsf{Node}[c_0] \leftarrow *\mathtt{create\text{-}child}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \mathsf{mem}, \mathsf{RandCor}[\mathsf{id}])$
10 : $\quad$ **if** $w_0 \neq \bot$ **then**
11 : $\quad\quad$ **assert** $\mathsf{Wel}[w_0] = \bot$
12 : $\quad\quad \mathsf{Wel}[w_0] \leftarrow c_0$
13 : **else**
14 : $\quad$ **if** $\mathsf{Node}[c_0] = \bot$ **then** $c_0' \leftarrow \mathsf{root}_{\mathit{rt}}$
15 : $\quad$ **else** $c_0' \leftarrow c_0$
16 : $\quad *\mathtt{consistent\text{-}com}(c_0', \mathsf{id}, \vec{p}, \mathsf{mem})$
17 : $\quad$ **if** $c_0' = \mathsf{root}_{\mathit{rt}}$ **then** $*\mathtt{attach}(c_0, c_0', \mathsf{id}, \vec{p})$
18 : $\quad$ **if** $w_0 \neq \bot$ **then**
19 : $\quad\quad$ **assert** $\mathsf{Wel}[w_0] \in \{ \bot, c_0 \}$
20 : $\quad\quad \mathsf{Wel}[w_0] \leftarrow c_0$
21 : **assert** $\mathtt{cons\text{-}invariant} \wedge \mathtt{auth\text{-}invariant}$
22 : **if** $\mathsf{RandCor}[\mathsf{id}]$ = 'bad' **then**
23 : $\quad$ Send $(\mathsf{exposed}, \mathsf{id}, \mathsf{svk})$ to $\mathcal{F}_{\mathrm{AS}}$
24 : **return** $(c_0, \vec{c}, w_0, \vec{w})$

Figure 14: The ideal CGKA functionality $\mathcal{F}_{\mathrm{CGKA}}$: `Create`, `Propose`, `Commit`.

Input (Process, $c_0, \widehat{c}, \vec{p}$)

1 : **req** Ptr[id] $\neq \perp$
2 : Send (Process, id, $c_0, \widehat{c}, \vec{p}$) to the adversary and
receive (ack, rt, orig′, svk′)
3 : **req** *succeed-proc(id, $c_0, \widehat{c}, \vec{p}$) $\vee$ ack
4 : *fill-prop(id, $\vec{p}$)
5 : **if** Node[c_0] $= \perp \wedge rt = \perp$ **then**
6 : (mem, $*$) $\leftarrow$ *members(Ptr[id], orig′, $\vec{p}$, svk′)
7 : **assert** mem $\neq \perp$
8 : Node[c_0] $\leftarrow$ *create-child(Ptr[id], orig′, $\vec{p}$, mem, 'adv')
9 : **else**
10 : **if** Node[c_0] $= \perp$ **then** $c_0' \leftarrow \text{root}_{rt}$
11 : **else** $c_0' \leftarrow c_0$
12 : $\text{id}_c \leftarrow$ Node[c_0'].orig; $\text{svk}_c \leftarrow$ Node[c_0'].mem[id_c]
13 : (mem, $*$) $\leftarrow$ *members(Ptr[id], $\text{id}_c, \vec{p}, \text{svk}_c$)
14 : **assert** mem $\neq \perp$
15 : *valid-successor($c_0', \text{id}_c, \vec{p}$, mem)
16 : **if** $c_0' = \text{root}_{rt}$ **then** *attach($c_0, c_0', \text{id}, \vec{p}$)
17 : **if** $\exists p \in \vec{p}$: Prop[p].act = 'rem'-id **then**
18 : Ptr[id] $\leftarrow \perp$
19 : **else**
20 : **assert** (id, $*$) $\in$ Node[c_0].mem
21 : Ptr[id] $\leftarrow c_0$; HasKey[id] $\leftarrow$ true
22 : **assert** cons-invariant $\wedge$ auth-invariant
23 : **return** *output-proc(c_0)

Input (Join, $w_0, \widehat{w}$)

1 : **req** Ptr[id] $= \perp$
2 : Send (Join, id, $w_0, \widehat{w}$) to the adversary and
receive (ack, c_0', orig′, mem′)
3 : **req** *succeed-wel(id, $w_0, \widehat{w}$) $\vee$ ack
4 : $c_0 \leftarrow$ Wel[w_0]
5 : **if** $c_0 = \perp$ **then**
6 : **if** Node[c_0'] $\neq \perp$ **then** $c_0 \leftarrow c_0'$
7 : **else**
8 : rootCtr++
9 : $c_0 \leftarrow \text{root}_{\text{rootCtr}}$ / Assume root_i are reserved words
10 : Node[c_0] $\leftarrow$ *create-root(orig′, mem′, 'adv')
11 : Wel[w_0] $\leftarrow c_0$
12 : **assert** (id, $*$) $\in$ Node[c_0].mem
13 : Ptr[id] $\leftarrow c_0$
14 : HasKey[id] $\leftarrow$ true
15 : **assert** cons-invariant $\wedge$ auth-invariant
16 : **return** (Node[c_0].orig, Node[c_0].mem)

Input Key

1 : **req** Ptr[id] $\neq \perp \wedge$ HasKey[id]
2 : **if** Node[Ptr[id]].key $= \perp$ **then** *set-key(Ptr[id])
3 : HasKey[id] $\leftarrow$ false
4 : **return** Node[Ptr[id]].key

Figure 15: The ideal CGKA functionality $\mathcal{F}_{\text{CGKA}}$: Process and Join.

Input (Expose, id)

1 : **if** Ptr[id] $\neq \perp$ **then**
2 : Node[Ptr[id]].exp +$\leftarrow$ (id, HasKey[id])
3 : *update-stat-after-exp(id)
4 : svk $\leftarrow$ Node[Ptr[id]].mem[id]
5 : Send (exposed, id, svk) to $\mathcal{F}_{\text{AS}}$
6 : Send (exposed, id) to $\mathcal{F}_{\text{KS}}$
7 : **restrict** $\forall c_0$ **if** Node[c_0].chall = true **then safe**(c_0) = true

Input (CorrRand, id, b), $b \in$ { 'good', 'bad' }

1 : RandCor[id] $\leftarrow b$

Figure 16: The CGKA functionality $\mathcal{F}_{\text{CGKA}}$: Corruptions from the adversary.

*create-root(id, mem, stat)

1 : **return** new node with par ← ⊥, orig ← id, pro ← ⊥,
mem ← mem, stat ← stat.

*create-child(c_0, id, $\vec{p}$, mem, stat)

1 : **return** new node with par ← c_0, orig ← id, pro ← $\vec{p}$,
mem ← mem, stat ← stat.

*create-prop(c_0, id, act, stat)

1 : **return** new node with par ← c_0, orig ← id,
act ← act, stat ← stat.

*fill-prop(id, $\vec{p}$)

1 : **foreach** $p \in \vec{p}$ **s.t.** Prop[p] = ⊥ **do**
2 : Send (Propose, p) to the adversary and receive (orig, act)
3 : Prop[p] ← *create-prop(Ptr[id], orig, act, 'adv')

*output-proc(c_0)

1 : id_c ← Node[c_0].orig
2 : svk_c ← Node[c_0].mem[id_c]
3 : ($*$, propSem) ← *members(c_0, id_c, Node[c_0].pro, svk_c)
4 : **return** (Node[c_0].orig, propSem)

*valid-svk(id, svk′)

1 : **if** Ptr[id] ≠ ⊥ **then**
2 : svk ← Node[Ptr[id]].mem[id]
3 : **if** svk ≠ ⊥ ∧ svk = svk′ **then return** true
4 : Send (has-ssk, id, svk′) to $\mathcal{F}_{\mathrm{AS}}$ and receive *ack*
5 : **return** *ack*

*set-key(c_0)

1 : **if safe**(c_0) **then**
2 : Node[c_0].key ←$ $\mathcal{K}$; Node[c_0].chall ← true
3 : **else**
4 : Send (Key, id) to the adversary and receive k
5 : Node[c_0].key ← k; Node[c_0].chall ← false

*update-stat-after-exp(id)

1 : **foreach** p **s.t.** Prop[p] ≠ ⊥ ∧ Prop[p].par = Ptr[id]
∧ Prop[p].orig = id ∧ Prop[p].act = 'upd-$*$' **do**
2 : Prop[p].stat ← 'bad'
3 : **foreach** c_0 **s.t.** Node[c] ≠ ⊥ ∧ Node[c].par = Ptr[id]
∧ Node[c].orig = id **do**
4 : Node[c].stat ← 'bad'

Figure 17: The helper functions for creating and maintaining the history graph.

*consistent-prop(p, id, act)

1 : **assert** Prop[p].par = Ptr[id] ∧ Prop[p].orig = id
∧ Prop[p].act = act

*consistent-com(c_0, id, $\vec{p}$, mem)

1 : *valid-successor(c_0, id, $\vec{p}$, mem)
2 : **assert** RandCor[id] = 'bad' ∧ Node[c_0].orig = id

*valid-successor(c_0, id, $\vec{p}$, mem)

1 : **assert** Node[c_0] ≠ ⊥ ∧ Node[c_0].mem = mem
∧ Node[c_0].pro ∈ { ⊥, $\vec{p}$ }
∧ Node[c_0].par ∈ { ⊥, Ptr[id] }

*attach(c_0, c_0', id, $\vec{p}$)

1 : **assert** $c_0' \neq \mathrm{root}_0$
2 : Node[c_0'].par ← Ptr[id]; Node[c_0'].pro ← $\vec{p}$
3 : Node[c_0] ← Node[c_0']; Node[c_0'] ← ⊥
4 : **foreach** p **s.t.** Prop[p].par = c_0' **do**
5 : Prop[p].par ← Ptr[id]
6 : **foreach** w_0 **s.t.** Wel[w_0] = c_0' **do**
7 : Wel[w_0] ← c_0
8 : **foreach** id **s.t.** Ptr[id] = c_0' **do**
9 : Ptr[id] ← c_0

*succeed-com(id, $\vec{p}$, svk)

1 : **return** *members(Ptr[id], id, $\vec{p}$, svk) ≠ (⊥, ⊥)
∧ *valid-svk(id, svk) ∧ $\forall p \in \vec{p}$: Prop[p].stat ≠ 'adv'

*succeed-proc(id, c_0, $\widehat{c}$, $\vec{p}$)

1 : **return** Node[c_0] ≠ ⊥ ∧ Node[c_0].par = Ptr[id]
∧ Node[c_0].pro = $\vec{p}$ ∧ Node[c_0].stat ≠ 'adv'
∧ $\forall p \in \vec{p}$: Prop[p].stat ≠ 'adv'

*succeed-wel(id, w_0, $\widehat{w}$)

1 : c_0 ← Wel[w_0]
2 : **return** Ptr[id] = ⊥ ∧ c_0 ≠ ⊥
∧ Node[c_0] ≠ ⊥ ∧ Node[c_0].stat ≠ 'adv'
∧ (id, $*$) ∈ (Node[c_0].mem \ Node[Node[c_0].par].mem)

Figure 18: The helper functions for consistency and correctness.

*members($c_0, \mathsf{id}_c, \vec{p}, \mathsf{svk}_c$)

1 : **if** Node[c_0] $\neq \bot \wedge (\mathsf{id}_c, *) \in$ Node[c_0].mem
  $\wedge \forall p \in \vec{p} :$ Prop[p] $\neq \bot \wedge$ Prop[p].par $= c_0$
  $\wedge \vec{p} = \vec{p}_{\text{'upd'}} \| \vec{p}_{\text{'rem'}} \| \vec{p}_{\text{'add'}}$ for some $\vec{p}_{\text{'upd'}}, \vec{p}_{\text{'rem'}}, \vec{p}_{\text{'add'}}$
    with $\forall \text{act} (\forall p \in \vec{p}_{\text{act}} :$ Prop[p].act = act-$*$) **then**
2 : mem $\leftarrow$ Node[c_0].mem
3 : mem $-\!\!\leftarrow (\mathsf{id}_c, *)$; mem $+\!\!\leftarrow (\mathsf{id}_c, \mathsf{svk}_c)$
4 : $L \leftarrow \{ \mathsf{id}_c \}$ / set of updated parties
5 : **foreach** $p \in \vec{p}_{\text{'upd'}}$ **do**
6 : $(\mathsf{id}_s,$ 'upd'-svk) $\leftarrow$ (Prop[p].orig, Prop[p].act)
7 : **if** $\neg((\mathsf{id}_s, *) \in$ mem $\wedge\ \mathsf{id}_s \notin L)$ **then return** $(\bot, \bot)$
8 : mem $-\!\!\leftarrow (\mathsf{id}_s, *)$; mem $+\!\!\leftarrow (\mathsf{id}_s,$ svk)
9 : $L +\!\!\leftarrow \mathsf{id}_s$
10 : **foreach** $p \in \vec{p}_{\text{'rem'}}$ **do**
11 : $(\mathsf{id}_s,$ 'rem'-id_t) $\leftarrow$ (Prop[p].orig, Prop[p].act)
12 : **if** $\neg((\mathsf{id}_s, *) \in$ mem $\wedge\ (\mathsf{id}_t \in$ mem $\wedge\ \mathsf{id}_t \notin L))$ **then return** $(\bot, \bot)$
13 : mem $-\!\!\leftarrow (\mathsf{id}_t, *)$
14 : **foreach** $p \in \vec{p}_{\text{'add'}}$ **do**
15 : $(\mathsf{id}_s,$ 'add'-id_t-svk_t) $\leftarrow$ (Prop[p].orig, Prop[p].act)
16 : **if** $\neg((\mathsf{id}_s, *) \in$ mem $\wedge\ (\mathsf{id}_t, *) \notin$ mem) **then return** $(\bot, \bot)$
17 : mem $+\!\!\leftarrow (\mathsf{id}_t, \mathsf{svk}_t)$
18 : $P \leftarrow ((\text{Prop}[p].\text{orig}, \text{Prop}[p].\text{act}) : p \in \vec{p})$
19 : **return** (mem, P)
20 : **else**
21 : **return** $(\bot, \bot)$

auth-invariant

return true **iff**
(a) $\forall c_0$ with $c_p \coloneqq$ Node[c_0].par, $c_p \neq \bot$ and id $\coloneqq$ Node[c_0].orig,
  if Node[c_0].stat = 'adv' **then** **sig-inj-allowed**(c_p, id) $\wedge$ **mac-inj-allowed**(c_p) and
(b) $\forall p$ with $c_p \coloneqq$ Prop[p].par and id $\coloneqq$ Prop[p].orig,
  if Prop[p].stat = 'adv' **then** **sig-inj-allowed**(c_p, id) $\wedge$ **mac-inj-allowed**(c_p) and
(c) $\forall \text{root}_{rt} \neq \bot$ with id $\coloneqq$ Node[root_{rt}].orig, **sig-inj-allowed**(root_{rt}, id)

cons-invariant

return true **iff**
(a) $\forall c_0$ **s.t.** Node[c_0].par $\neq \bot$: (Node[c_0].pro $\neq \bot$
  $\wedge \forall p \in$ Node[c_0].pro : Prop[p].par = Node[c_0].par) and
(b) $\forall$id **s.t.** Ptr[id] $\neq \bot$: id $\in$ Node[Ptr[id]].mem and
(c) the history graph contains no cycle

Figure 19: The helper functions to determine the group state after applying a commit and the history graph invariants. We explicitly add the authentication invariant concerning welcome messages which is highlighted in yellow.

D THE CHAINED CMPKE PROTOCOL

In this section, we provide a more in-depth exposition of our Chained CmPKE protocol.

As already explained in Sec. 4, unlike TreeKEM, we no longer require to maintain a tree structure since the structure we maintain is a depth-1 tree (which is much like a comb). This makes the description of our protocol much simpler relative to TreeKEM and relieves us from "blanking" nodes when updating and removing users from the group. Effectively, the security analysis is also simpler since we no longer need to keep track of the exposed/unexposed secrets assigned to the internal nodes of the tree.

Moreover, during a commit protocol, the committer does not sign the whole ciphertext but only the part that binds the message, i.e., the commitment T in CmPKE. The delivery server is expected to parse the uploaded commit message and forward the relevant parts to the receivers.

Below we describe our Chained CmPKE protocol and provide details on the differences between TreeKEM version 10 of MLS formalized by [13].

D.1 Protocol States

Each user holds a group state G. It consists of the variables listed in Tab. 6. The G.member array stores the information of the group members. The index of G.member is specified by the party identities and each entry consists of the variables listed in Tab. 7. The member hash G.memberHash is the hash of all key packages stored in G.member.

The group state contains three hashes: *confirmation transcript hash* (confTransHash), *confirmation transcript hash without committer identifier* (confTransHash-w.o-'id_c') and *interim transcript hash* (interimTransHash). Roughly, these hashes maintain the consistency between the previous and current epoch and are used to enforce a consistent view within the group members.

If a group member issues an update proposal or commit message that did not get confirmed by the server, the corresponding secrets are stored in G.pendUpd and G.pendCom, respectively. When a member receives a message which has been created by itself, it retrieves the corresponding secrets from G.pendUpd or G.pendCom (rather than processing it from scratch).

For readability, we define the useful helper methods corresponding to the group state, listed in Tab. 8. In the security proof, G additionally stores the variables listed in Tab. 9

Differences from TreeKEM. All variables except for G.member, G.memberHash and G.confTransHash-w.o-'id_c' are defined identically to TreeKEM. G.member corresponds to the left-balanced binary tree τ considered in [13], restricted to arity N and depth 1. Namely, G.member only maintains a simply array rather than a tree. G.memberHash is a replacement of treeHash in TreeKEM. We newly define the hash value G.confTransHash-w.o-'id_c', which is used in the join protocol to confirm the sender of the welcome message.

D.2 Protocol Algorithms

The main protocol is depicted in Figs. 20 and 21. The associated helper functions are depicted in Figs. 23 to 27. In these figures, the differences from TreeKEM version 10 in MLS considered by [13] are highlighted in yellow.

(1) Group Creation. The group is created (by the designated party $\mathsf{id}_{\mathsf{creator}}$ in our model) using the input $(\texttt{Create}, \mathsf{svk})$. This input initializes the group state and creates a new group with the single member $\mathsf{id}_{\mathsf{creator}}$. The group creator fetches the corresponding signing key ssk from $\mathcal{F}_{\mathrm{AS}}$ using the helper function `*fetch-ssk-if-nec`.

Differences from TreeKEM. The group creation protocol is defined identically to TreeKEM except that party $\mathsf{id}_{\mathsf{creator}}$ maintains a simpler group protocol state G compared to TreeKEM. Note that, unlike TreeKEM, our protocol initializes a random joiner secret and derive the epoch secrets from it. Then, it computes the confirmation tag confTag for the initial group. This is because confTag is necessary to discuss the security of the protocol.

(2) Proposals. The protocol first prepares a preliminary proposal message P.

- To create an update proposal, the protocol generates a fresh key package together with the corresponding decryption key dk. The key package kp is included in the proposal and dk is stored in G.pendUpd. When a new verification key svk is used, the protocol fetches the corresponding signing key ssk from $\mathcal{F}_{\mathrm{AS}}$. (ssk is also stored in G.pendUpd.)
- To create an add proposal, the protocol fetches the key package for the added party from $\mathcal{F}_{\mathrm{KS}}$. The proposal consists of the key package which includes the added party's identity.
- The remove proposal consists of the identity of a removed party.

All proposals are framed using `*frame-prop`. It first signs the proposal P together with the string 'proposal', the group context including confTransHash, and the sender's identity. This signature prevents impersonation by another group member. In addition, to ensure the PCS security and group membership of the sender, everything including the signature is MACed using the membership key. The MAC tag ties the proposal to a specific group/epoch since the signature key may be shared across groups and is long-lived. In summary, to inject or modify messages, the adversary must corrupt both the sender's signing key and the current epoch secrets. The actual proposal message p consists of everything except the G.memberHash and G.confTransHash since the other components can be retrieved from the protocol state of the recipients.

Differences from TreeKEM. The propose issue protocol is defined identically to TreeKEM.

(3) Commits. To create a new commit message, a party id runs the protocol on input $(\texttt{Commit}, \vec{p}, \mathsf{svk})$. The protocol first initializes the next epoch's group state by copying the current one. It then applies the proposals $\vec{p}$ using `*apply-props`. It verifies the validity of the MAC tag and signature in each proposal. The protocol then derives id's new CmPKE key pair and a new *commit secret* using the helper function `*rekey`. It outputs a fresh commit secret, a fresh key package kp for the committer, and a CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}})$ encrypting the commit secret. Note that the commit secret will be shared among existing users who are not removed in the next epoch.

G.groupid	The identifier of the group.
G.epoch	The current epoch number.
G.confTransHash	The confirmed transcript hash.
G.confTransHash-w.o-'id_c'	The confirmed transcript hash without the committer identity.
G.interimTransHash	The interim transcript hash for the next epoch.
G.member[$*$]	A mapping associating party id with its state.
G.memberHash	A hash of the public part of G.member[$*$].
G.certSvks[$*$]	A mapping associating the set of validated signature verification keys to each party.
G.pendUpd[$*$]	A mapping associating the secret keys for each pending update proposal issued by id.
G.pendCom[$*$]	A mapping associating the new group state for each pending commit issued by id.
G.id	The identity of the party.
G.ssk	The current signing key.
G.appSecret	The current epoch's shared key.
G.membKey	The key used to MAC proposal packages.
G.initSecret	The next epoch's init secret.

Table 6: The protocol state.

id	The identity of the party.
ek	The encryption key of a CmPKE scheme.
dk	The corresponding decryption key.
svk	The signature verification key of a signature scheme.
sig	The signature for (id, ek, svk) under the signature singing key corresponding to svk.
kp()	Returns (id, ek, svk, sig) (if G.member[id] $\neq \perp$).

Table 7: The party id's state stored in G.member[id] and helper method.

G.clone()	Returns (independent) copy of G.
G.memberIDs()	Returns the list of party ids sorted by dictionary order.
G.memberIDsvks()	Returns the list of party ids and its associating svk sorted by dictionary order in the ids.
G.memberPublicInfo()	Returns the public part of G.member[$*$].
G.groupCont()	Returns (G.groupid, G.epoch, G.memberHash, G.confTransHash).

Table 8: The helper methods on the protocol state.

G.joinerSecret	The current epoch's joiner secret.
G.comSecret	The current epoch's commit secret.
G.confKey	The key used to MAC for commit and welcome messages.
G.confTag	The MAC tag included either in the commit or welcome message.
G.membTags	The set of MAC tags included in the proposal messages.

Table 9: The protocol state maintained only during the security proof.

The commit message consists of two parts: a party independent message c_0 and a party dependent message $\widehat{c}$. The protocol first prepares a preliminary commit message C_0 including the list of the hash of all the applied proposals propIDs, the key package kp, and the commitment T. This commit message is signed alongside the group context using `*sign-commit`. Afterwards, the protocol derives the epoch secrets using `*derive-keys` and computes the confirmation tag (see `*gen-conf-tag`). c_0 is constructed from C_0, the signature, and the confirmation tag. Then, the protocol prepares the party dependent message $\widehat{c}$. It is set as $(\mathsf{id}, \widehat{\mathsf{ct}}_{\mathsf{id}})$, or $(\mathsf{id}, \perp)$ if the party id is removed in the next epoch. (Here, $\vec{c}$ is the list of $\widehat{c}$.)

If new members are added, the protocol creates a welcome message using the function `*welcome-msg`. The welcome message also consists of two parts: a party independent message w_0 and a party dependent message $\widehat{w}$. It first encrypts the joiner secret (which will be used to derive epoch secrets) with the added members' encryption keys, and obtains a CmPKE ciphertext $(\mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}_t})_{\mathsf{id}_t \in \mathsf{addedMem}})$. Then the protocol composes a group information groupInfo which contains the public part of the group state, the confirmation tag, and the sender's identity. groupInfo and T are signed by the sender's signing key and w_0 is set as

(groupInfo, T, sig). Then, the protocol prepares the party dependent message $\widehat{w}$. It is set as (id, kphash, $\widehat{\mathsf{ct}}_{\mathsf{id}}$) where kphash is the hash of the used key package. (Here, $\vec{w}$ is the list of $\widehat{w}$.)

Finally, the protocol computes the interim transcript hash for the next epoch by hashing the current confirmation hash and the newly generated confirmation tag. The next epoch's state is stored in G.pendCom.

Differences from TreeKEM. The following summarizes the differences between Chained CmPKE and TreeKEM.

(1) Our `*apply-props` simply rewrites entries in G.member: if id is deleted, it sets G.member[id] to $\perp$; if id is added, it stores its key package in a new entry; if id is updated, it replaces the old key package with the new one. In contrast, TreeKEM additionally runs the 'blank node' operation after updating the leaf nodes. That is, the committer blanks the nodes on the path from the updated or removed leaf to the root.
(2) Our `*rekey` operation simply encrypts a new comSecret with the recipients' CmPKE encryption keys. In contrast, TreeKEM runs a 'path update' operation to derive comSecret. It refreshes all PKE keys along the path from the committer's leaf to the root. Each path secret is then encrypted to the resolution of the sibling of the concerned node. Here, the secret on the root is used as comSecret.
(3) Chained CmPKE signs only T, rather than T *and* $(\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id}\in\mathsf{receivers}}$. This allows the delivery server to send only the message needed for each user, and effectively lowers the downloaded package size from $O(N)$ to $O(1)$. In contrast, in TreeKEM, all the ciphertexts (each encrypting a path secret) is signed. The size of the downloaded package is therefore $O(\log N)$ in the best case (i.e., full non-blanked tree) and $O(N)$ in the worst case (i.e., heavily blanked tree).
(4) Our commit message consists of two parts: c_0 is a party independent message and will be sent to all the recipients. $\widehat{c}_{\mathsf{id}}$ is a party dependent message that contains the identity of a single recipient id and the ciphertext its corresponding $\widehat{\mathsf{ct}}_{\mathsf{id}}$. This is only sent to the specific party id. In contrast, in TreeKEM, a commit message is viewed as a monolithic bloc and the commit message is sent to all the recipients without any modification. This corresponds to setting $c_0 = \perp$ and $\widehat{\mathsf{ct}}_{\mathsf{id}} = \widehat{\mathsf{ct}}_{\mathsf{id}'}$ for all $\mathsf{id}, \mathsf{id}' \in$ receivers in our new ideal functionality.
(5) Our welcome message only encrypts a new joinerSecret with the added members' CmPKE encryption keys. There is no need to send the secrets assigned to the internal nodes of a tree as in TreeKEM. Analogous to the commit message, the welcome message also consists of two parts.

The other process (e.g., generating hash values, re-keying) are identical.

(4) Process. Consider the input (`Process`, $c_0, \widehat{c}, \vec{p}$). If the party id is the creator of the received commit message c_0, then the protocol simply retrieves the new epoch state from G.pendCom; otherwise, it proceeds as follows.

First, the protocol unframes the message, i.e., verifies the signature and checks that it belongs to the correct group and epoch (cf. `*unframe-commit` in Fig. 27). Next, it verifies whether $\vec{p}$ matches the committed proposals in c_0. If so, it applies them using `*apply-props`.

If id is not removed, the protocol derives a new epoch secret. It decrypts the ciphertext using `*apply-rekey` (it also applies the committer's new key package) and computes the epoch secret using `*derive-keys`. Finally, it verifies the confirmation tag in c_0 and derives a new interim transcript hash.

Differences from TreeKEM. There are two differences. First is input message. Chained CmPKE allows the sever to sanitize commit messages by delivering to each group member the strict amount of data they need. Namely, the server only sends $(c_0, \widehat{c}_{\mathsf{id}})$ to the party id, and hence, party id only receives the ciphertext it needs to update its protocol state. This reduces the party's download cost and the server's bandwidth.

Second is the `*apply-rekey` function. To obtain comSecret, Chained CmPKE simply decrypts the ciphertext. In contrast, TreeKEM decrypts the ciphertext, which contains the secret on the least common ancestor of the committer and the recipient, and then runs the 'path update' operation to recover comSecret (i.e., root secret).

(5) Join. Upon receiving an input $(w_0, \widehat{w})$, the protocol initializes a new group state and copies the public group information from w_0. Then it checks the validity of the confirmation hash and interim transcript hash by recomputing these hashes from the received information. It also verifies the signature and the validity of the member list and each group member's key package. If the information is valid, the protocol decrypts the joiner secret. To this end, it fetches all its key package and decryption key pairs from $\mathcal{F}_{\mathrm{KS}}$ and determines the one that has been used for the welcome message by checking the hash of the key package.

Finally, it derives the epoch secrets from the joiner secret and verifies the confirmation tag.

Differences from TreeKEM. As for the commit message, new member receives the sanitized message $(w_0, \widehat{w}_{\mathsf{id}})$. Chained CmPKE simply decrypts the ciphertext and derives the epoch secret from the decrypted joinerSecret. In contrast, in TreeKEM, the welcome message contains the secret on the least common ancestor of the committer and the recipient. The receiver then runs the 'path update' operation in order to derive the decryption keys of its parents. This process does not appear in Chained CmPKE.

Chained CmPKE checks the validity of the confirmation hash in the welcome message by using confTransHash-w.o-'id_c' and id_c. This allows the recipient of the welcome message to verify that id_c has computed the confirmation hash.

(6) Key. Upon input (`Key`), the protocol outputs the application secret of the current epoch and deletes it form the local state.

Differences from TreeKEM. This key protocol is the same as TreeKEM.

Input (Create, svk)

1 : **req** $G = \bot \wedge \mathsf{id} = \mathsf{id}_{\mathsf{creator}}$
2 : $G.\mathsf{groupid} \leftarrow\!\!\$\ \{0,1\}^\kappa$; $G.\mathsf{joinerSecret} \leftarrow\!\!\$\ \{0,1\}^\kappa$
3 : $G.\mathsf{epoch} \leftarrow 0$
4 : $G.\mathsf{member}[*] \leftarrow \bot$; $G.\mathsf{memberHash} \leftarrow \bot$
5 : $G.\mathsf{confTransHash\text{-}w.o\text{-}`id_c'} \leftarrow \bot$
6 : $G.\mathsf{confTransHash} \leftarrow \bot$
7 : $G.\mathsf{certSvks}[*] \leftarrow \emptyset$
8 : $G.\mathsf{pendUpd}[*] \leftarrow \bot$; $G.\mathsf{pendCom}[*] \leftarrow \bot$
9 : $G.\mathsf{id} \leftarrow \mathsf{id}$
10 : **try** $\mathsf{ssk} \leftarrow$ *fetch-ssk-if-nec(G, svk)
11 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKp}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
12 : $G \leftarrow$ *assign-kp$(G, \mathsf{id}, \mathsf{kp})$
13 : $G.\mathsf{ssk} \leftarrow \mathsf{ssk}$
14 : $G.\mathsf{member}[\mathsf{id}].\mathsf{dk} \leftarrow \mathsf{dk}$
15 : $G.\mathsf{memberHash} \leftarrow$ *derive-member-hash(G)
16 : $(G, \mathsf{confKey}) \leftarrow$ *derive-epoch-keys$(G, G.\mathsf{joinerSecret})$
17 : $\mathsf{confTag} \leftarrow$ *gen-conf-tag$(G, \mathsf{confKey})$
18 : $G \leftarrow$ *set-interim-trans-hash$(G, \mathsf{confTag})$

Input (Propose, 'upd'-svk)

1 : **req** $G \neq \bot$
2 : **try** $\mathsf{ssk} \leftarrow$ *fetch-ssk-if-nec(G, svk)
3 : $(\mathsf{kp}, \mathsf{dk}) \leftarrow \mathsf{genKp}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk})$
4 : $P \leftarrow (\text{'upd'}, \mathsf{kp})$
5 : $p \leftarrow$ *frame-prop(G, P)
6 : $G.\mathsf{pendUpd}[p] \leftarrow (\mathsf{ssk}, \mathsf{dk})$
7 : **return** p

Input (Propose, 'add'-id_t)

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \notin G.\mathsf{memberIDs}()$
2 : Send $(\mathsf{get\text{-}kp}, \mathsf{id}_t)$ to $\mathcal{F}_{\mathrm{KS}}$ and receive kp_t
3 : **req** $\mathsf{kp}_t \neq \bot$
4 : **try** $G \leftarrow$ *validate-kp$(G, \mathsf{kp}_t, \mathsf{id}_t)$
5 : $P \leftarrow (\text{'add'}, \mathsf{kp}_t)$
6 : $p \leftarrow$ *frame-prop(G, P)
7 : **return** p

Input (Propose, 'rem'-id_t)

1 : **req** $G \neq \bot \wedge \mathsf{id}_t \in G.\mathsf{memberIDs}()$
2 : $P \leftarrow (\text{'rem'}, \mathsf{id}_t)$
3 : $p \leftarrow$ *frame-prop(G, P)
4 : **return** p

Input (Commit, $\vec{p}$, svk)

1 : **req** $G \neq \bot$
2 : $G' \leftarrow$ *init-epoch(G)
3 : **try** $(G', upd, rem, add) \leftarrow$ *apply-props$(G, G', \vec{p})$
4 : **req** $(*, \text{'rem'-id}) \notin rem \wedge (\mathsf{id}, *) \notin upd$
5 : $\mathsf{addedMem} \leftarrow \{\, \mathsf{id}_t \mid (*, \text{'add'-}\mathsf{id}_t\text{-}*) \in add \,\}$ / Recipients of the welcome message
6 : $\mathsf{receivers} \leftarrow G'.\mathsf{memberIDs}() \setminus \mathsf{addedMem}$ / Recipients of the new commit secret
7 : **try** $(G', \mathsf{comSecret}, \mathsf{kp}, \mathsf{T}, \vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id} \in \mathsf{receivers}}) \leftarrow$ *rekey$(G', \mathsf{receivers}, \mathsf{id}, \mathsf{svk})$
8 : $G' \leftarrow$ *set-member-hash(G')
9 : $\mathsf{propIDs} \leftarrow ()$
10 : **foreach** $p \in \vec{p}$ **do** $\mathsf{propIDs}$ +$\leftarrow \mathsf{H}(p)$
11 : $C_0 \leftarrow (\mathsf{propIDs}, \mathsf{kp}, \mathsf{T})$
12 : $\mathsf{sig} \leftarrow$ *sign-commit(G, C_0)
13 : $G' \leftarrow$ *set-conf-trans-hash$(G, G', \mathsf{id}, C_0, \mathsf{sig})$
14 : $(G', \mathsf{confKey}, \mathsf{joinerSecret}) \leftarrow$ *derive-keys$(G, G', \mathsf{comSecret})$
15 : $\mathsf{confTag} \leftarrow$ *gen-conf-tag$(G', \mathsf{confKey})$
16 : $c_0 \leftarrow$ *frame-commit$(G, C_0, \mathsf{sig}, \mathsf{confTag})$
17 : $G' \leftarrow$ *set-interim-trans-hash$(G', \mathsf{confTag})$
18 : $\vec{c} \leftarrow \emptyset$
19 : **foreach** $\mathsf{id} \in G.\mathsf{memberIDs}()$ **do**
20 : **if** $\mathsf{id} \in \mathsf{receivers}$ **then** $\vec{c}$ +$\leftarrow \widehat{c}_{\mathsf{id}} = (\mathsf{id}, \widehat{\mathsf{ct}}_{\mathsf{id}})$
21 : **else** $\vec{c}$ +$\leftarrow \widehat{c}_{\mathsf{id}} = (\mathsf{id}, \bot)$
22 : **if** $add \neq ()$ **then**
23 : $(G', w_0, \vec{w}) \leftarrow$ *welcome-msg$(G', \mathsf{addedMem}, \mathsf{joinerSecret}, \mathsf{confTag})$
24 : **else**
25 : $w_0 \leftarrow \bot$; $\vec{w} \leftarrow \emptyset$
26 : $G.\mathsf{pendCom}[c_0] \leftarrow (G', \vec{p}, upd, rem, add)$
27 : **return** $(c_0, \vec{c}, w_0, \vec{w})$

Input Key

1 : **req** $G \neq \bot$
2 : $\mathsf{k} \leftarrow G.\mathsf{appSecret}$
3 : $G.\mathsf{appSecret} \leftarrow \bot$
4 : **return** k

Figure 20: Main protocol: Create, Propose, and Commit. The major changes from [13] are highlighted in yellow.

Input (Process, $c_0, \widehat{c}, \vec{p}$)

1: **req** $G \neq \bot$
2: $(\mathrm{id}_c, C_0, \mathrm{sig}, \mathrm{confTag}) \leftarrow$ *unframe-commit(G, c_0)
3: **if** $\mathrm{id}_c = \mathrm{id}$ **then**
4: **parse** $(G', \vec{p}', upd, rem, add) \leftarrow G.\mathrm{pendCom}[c_0]$
5: **req** $\vec{p} = \vec{p}'$
6: **return** $(\mathrm{id}_c, upd \| rem \| add)$
7: **parse** $(\mathrm{propIDs}, \mathrm{kp}_c, \mathsf{T}) \leftarrow C_0$
8: **parse** $(\mathrm{id}', \widehat{\mathrm{ct}}_{\mathrm{id}'}) \leftarrow \widehat{c}$
9: **req** $\mathrm{id} = \mathrm{id}'$
10: **for** $i \in 1, \ldots, |\vec{p}|$ **do**
11: **req** $\mathsf{H}(\vec{p}[i]) = \mathrm{propIDs}[i]$
12: $G' \leftarrow$ *init-epoch(G)
13: **try** $(G', upd, rem, add) \leftarrow$ *apply-props$(G, G', \vec{p})$
14: **req** $(*, \mathrm{id}_c) \notin rem \wedge (\mathrm{id}_c, *) \notin upd$
15: **if** $(*, \text{'rem'-id}) \in rem$ **then**
16: $G' \leftarrow \bot$
17: **else**
18: $G' \leftarrow$ *set-conf-trans-hash$(G, G', \mathrm{id}_c, C_0, \mathrm{sig})$
19: $(G', \mathrm{comSecret}) \leftarrow$ *apply-rekey$(G', \mathrm{id}_c, \mathrm{kp}_c, \mathsf{T}, \widehat{\mathrm{ct}}_{\mathrm{id}})$
20: $G' \leftarrow$ *set-member-hash(G')
21: $(G', \mathrm{confKey}, \mathrm{joinerSecret}) \leftarrow$ *derive-keys$(G, G', \mathrm{comSecret})$
22: **req** *vrf-conf-tag$(G', \mathrm{confKey}, \mathrm{confTag})$
23: $G' \leftarrow$ *set-interim-trans-hash$(G', \mathrm{confTag})$
24: **return** $(\mathrm{id}_c, upd \| rem \| add)$

Input (Join, $w_0, \widehat{w}$)

1: **req** $G = \bot$
2: **parse** $(\mathrm{groupInfo}, \mathsf{T}, \mathrm{sig}) \leftarrow w_0$
3: **parse** $(\mathrm{id}', \mathrm{kphash}, \widehat{\mathrm{ct}}_{\mathrm{id}'}) \leftarrow \widehat{w}$
4: **req** $\mathrm{id} = \mathrm{id}'$
5: **try** $(G, \mathrm{confTag}, \mathrm{id}_c) \leftarrow$ *initialize-group$(G, \mathrm{id}, \mathrm{groupInfo})$
6: **req** $G.\mathrm{confTransHash} = \mathsf{H}(G.\mathrm{confTransHash}\text{-w.o-'}\mathrm{id}_c\text{'}, \mathrm{id}_c)$
7: **req** $G.\mathrm{interimTransHash} = \mathsf{H}(G.\mathrm{confTransHash}, \mathrm{confTag})$
8: **req** $\mathsf{SIG.Verify}(G.\mathrm{member}[\mathrm{id}_c].\mathrm{svk}, \mathrm{sig}, (\mathrm{groupInfo}, \mathrm{ct}_0))$
9: **try** $G \leftarrow$ *vrf-group-state(G)
10: $\mathrm{svk} \leftarrow G.\mathrm{member}[\mathrm{id}].\mathrm{svk}$
11: **try** $G.\mathrm{ssk} \leftarrow$ *fetch-ssk-if-nec(G, svk)
12: Send get-dks to $\mathcal{F}_{\mathrm{KS}}$ and receive kbs
13: $\mathrm{joinerSecret} \leftarrow \bot$
14: **foreach** $(\mathrm{kp}, \mathrm{dk}) \in \mathrm{kbs}$ **do**
15: **if** $\mathsf{H}(\mathrm{kp}) = \mathrm{kphash}$ **then**
16: **req** $G.\mathrm{member}[\mathrm{id}].\mathrm{kp}() = \mathrm{kp}$
17: $G.\mathrm{member}[\mathrm{id}].\mathrm{dk} \leftarrow \mathrm{dk}$
18: $\mathrm{joinerSecret} \leftarrow \mathsf{CmDec}(\mathrm{dk}, \mathsf{T}, \widehat{\mathrm{ct}}_{\mathrm{id}})$
19: **req** $\mathrm{joinerSecret} \neq \bot$
20: $(G, \mathrm{confKey}) \leftarrow$ *derive-epoch-keys$(G, \mathrm{joinerSecret})$
21: **req** *vrf-conf-tag$(G, \mathrm{confKey}, \mathrm{confTag})$
22: **return** $(\mathrm{id}_c, G.\mathrm{memberIDsvks}())$

Figure 21: Main protocol: Process and Join. The major changes from [13] are highlighted in yellow. The orange highlights components that are missing from prior works, which we believe is required to satisfy the UC functionality. Please see the proof for more detail.

*fetch-ssk-if-nec(G, svk)

1: **if** $G.\mathrm{member}[G.\mathrm{id}].\mathrm{svk} \neq \mathrm{svk}$ **then**
2: Send (get-ssk, svk) to $\mathcal{F}_{\mathrm{AS}}$ and receive ssk
3: **else**
4: $\mathrm{ssk} \leftarrow G.\mathrm{ssk}$
5: **return** ssk

*validate-kp$(G, \mathrm{kp}, \mathrm{id})$

1: **parse** $(\mathrm{id}', \mathrm{ek}, \mathrm{svk}, \mathrm{sig}) \leftarrow \mathrm{kp}$
2: **req** $\mathrm{id} = \mathrm{id}'$
3: **if** $\mathrm{svk} \notin G.\mathrm{certSvks}[\mathrm{id}]$ **then**
4: Send $(\mathrm{verify\text{-}cert}, \mathrm{id}', \mathrm{svk})$ to $\mathcal{F}_{\mathrm{AS}}$ and receive *succ*
5: **req** *succ*
6: $G.\mathrm{certSvks}[\mathrm{id}] \mathrel{+}\!\!\leftarrow \mathrm{svk}$
7: **req** $\mathsf{SIG.Verify}(\mathrm{pp}_{\mathrm{SIG}}, \mathrm{svk}, \mathrm{sig}, (\mathrm{id}, \mathrm{ek}, \mathrm{svk}))$
8: **return** G

*assign-kp(G, kp)

1: **parse** $(\mathrm{id}, \mathrm{ek}, \mathrm{svk}, \mathrm{sig}) \leftarrow \mathrm{kp}$
2: $G.\mathrm{member}[\mathrm{id}].\mathrm{ek} \leftarrow \mathrm{ek}$
3: $G.\mathrm{member}[\mathrm{id}].\mathrm{svk} \leftarrow \mathrm{svk}$
4: $G.\mathrm{member}[\mathrm{id}].\mathrm{sig} \leftarrow \mathrm{sig}$
5: **return** G

Figure 22: Helper functions: key material related.

*init-epoch(G)

$G' \leftarrow G$.clone()
G'.epoch $\leftarrow G$.epoch + 1
G'.pendUpd[$*$] $\leftarrow \bot$; G'.pendCom[$*$] $\leftarrow \bot$
return G'

*rekey(G', receivers, id, svk)

try ssk $\leftarrow$ *fetch-ssk-if-nec(G', svk)
(kp, dk) $\leftarrow$ genKp(id, svk, ssk)
$G' \leftarrow$ *assign-kp(G', kp)
G'.ssk $\leftarrow$ ssk; G'.member[id].dk $\leftarrow$ dk
comSecret $\leftarrow\!\!\$$ $\{0,1\}^\kappa$
$\vec{\text{ek}} \leftarrow (G.\text{member}[\text{id}'].\text{ek})_{\text{id}' \in \text{receivers}}$
  / receivers is non-empty because it always contains the committer
$(\mathsf{T}, \vec{\text{ct}} = (\widehat{\text{ct}}_{\text{id}'})_{\text{id}' \in \text{receivers}}) \leftarrow \text{CmEnc}(\text{pp}_{\text{CmPKE}}, \vec{\text{ek}}, \text{comSecret})$
return $(G', \text{comSecret}, \text{kp}, \mathsf{T}, \vec{\text{ct}})$

*apply-rekey(G', id_c, kp_c, T, ct)

dk $\leftarrow G'$.member[G'.id].dk
comSecret $\leftarrow$ CmDec(dk, T, ct)
try $G' \leftarrow$ *validate-kp(G', kp_c, id_c)
$G' \leftarrow$ *assign-kp(G', kp_c)
return $(G', \text{comSecret})$

*welcome-msg(G', addedMem, joinerSecret, confTag)

$\vec{\text{ek}} \leftarrow (G'.\text{member}[\text{id}_t].\text{ek})_{\text{id}_t \in \text{addedMem}}$ **do**
$(\mathsf{T}, \vec{\text{ct}} = (\widehat{\text{ct}}_{\text{id}_t})_{\text{id}_t \in \text{addedMem}}) \leftarrow \text{CmEnc}(\text{pp}_{\text{CmPKE}}, \vec{\text{ek}}, \text{joinerSecret})$
groupInfo $\leftarrow$ (G'.groupid, G'.epoch,
  G'.memberPublicInfo(), G'.memberHash,
  G'.confTransHash-w.o-'id_c', G'.confTransHash,
  G'.interimTransHash, confTag, G'.id)
sig $\leftarrow$ SIG.Sign(pp_{SIG}, G'.ssk, (groupInfo, T))
$w_0 \leftarrow$ (groupInfo, T, sig)
$\vec{w} \leftarrow \emptyset$
foreach $\text{id}_t \in$ addedMem **do**
  $\text{kphash}_t \leftarrow \mathsf{H}(G'.\text{member}[\text{id}_t].\text{kp}())$
  $\vec{w} \mathrel{+}\!\!\leftarrow \widehat{w}_{\text{id}_t} = (\text{id}_t, \text{kphash}_t, \widehat{\text{ct}}_{\text{id}_t})$
return $(G', w_0, \vec{w})$

*vrf-group-state(G)

req G.memberHash = *derive-member-hash(G)
mem $\leftarrow G$.memberIDs()
foreach id $\in$ mem **do**
  kp $\leftarrow G$.member[id].kp()
  try $G \leftarrow$ *validate-kp(G, kp, id)
return G

*apply-props(G, G', $\vec{p}$)

upd, *rem*, *add* $\leftarrow$ ()
foreach $p \in \vec{p}$ **do**
  try $(\text{id}_s, P) \leftarrow$ *unframe-prop(G, p)
  parse (type, val) $\leftarrow P$
  if type = 'upd' **then**
    req $\text{id}_s \in G$.memberIDs()
    req $(\text{id}_s, *) \notin upd \wedge rem = () \wedge add = ()$
    try $G' \leftarrow$ *validate-kp(G', val, id_s)
    $G' \leftarrow$ *assign-kp(G', val)
    if $\text{id}_s = G$.id **then**
      parse (ssk, dk) $\leftarrow G$.pendUpd[p]
      G'.ssk $\leftarrow$ ssk
      G'.member[G.id].dk $\leftarrow$ dk
    svk $\leftarrow G'$.member[id_s].svk
    upd $\mathrel{+}\!\!\leftarrow$ (id_s, 'upd'-svk)
  elseif type = 'rem' **then**
    parse $\text{id}_t \leftarrow$ val
    req $\text{id}_t \neq \text{id}_s \wedge \text{id}_t \in G'$.memberIDs()
    req $(\text{id}_t, *) \notin upd \wedge add = ()$
    G'.member[id_t] $\leftarrow \bot$
    rem $\mathrel{+}\!\!\leftarrow$ (id_s, 'rem'-id_t)
  elseif type = 'add' **then**
    parse $(\text{id}_t, *, \text{svk}_t, *, *) \leftarrow$ val
    req $\text{id}_t \notin G'$.memberIDs()
    try $G' \leftarrow$ *validate-kp(G', val, id_t)
    $G' \leftarrow$ *assign-kp(G', val)
    add $\mathrel{+}\!\!\leftarrow$ (id_s, 'add'-id_t-svk_t)
  else
    return $\bot$
return (G', upd, rem, add)

*initialize-group(G, id, groupInfo)

parse (groupid, epoch, member, memberHash, confTransHash-w.o-'id_c',
  confTransHash, interimTransHash, confTag, id_c) $\leftarrow$ groupInfo
(G.groupid, G.epoch, G.member, G.memberHash,
  G.confTransHash-w.o-'id_c', G.confTransHash, G.interimTransHash)
  $\leftarrow$ (groupid, epoch, member, memberHash,
  confTransHash-w.o-'id_c', confTransHash, interimTransHash)
G.certSvks[$*$] $\leftarrow \emptyset$
G.pendUpd[$*$] $\leftarrow \bot$; G.pendCom[$*$] $\leftarrow \bot$
G.id $\leftarrow$ id
return $(G, \text{confTag}, \text{id}_c)$

Figure 23: Helper functions: Commit, Process and Join. The major changes from [13] are highlighted in yellow. The orange highlights components that are missing from prior works, which we believe is required to satisfy the UC functionality. Please see the proof for more detail.

*gen-conf-tag(G, confKey)

1 : **return** MAC.TagGen(confKey, G.confTransHash)

*vrf-conf-tag(G, confKey, confTag)

1 : **return** MAC.TagVerify(confKey, confTag, G.confTransHash)

Figure 24: Helper function: Confirmation tag.

*set-member-hash(G)

1 : G.memberHash ← *derive-member-hash(G)
2 : **return** G

*derive-member-hash(G)

1 : KP ← (); mem ← G.memberIDs() / mem is sorted by dictionary order
2 : **foreach** id ∈ mem **do**
3 : KP ∥← G.member[id].kp()
4 : **return** H(KP)

*set-conf-trans-hash(G, G', id_c, C_0, sig)

1 : comCont ← (G.groupid, G.epoch, 'commit', C_0, sig)
2 : G'.confTransHash-w.o-'id_c' ← H(G.interimTransHash, comCont)
3 : G'.confTransHash ← H(G'.confTransHash-w.o-'id_c', id_c)
4 : **return** G'

*set-interim-trans-hash(G', confTag)

1 : G'.interimTransHash ← H(G'.confTransHash, confTag)
2 : **return** G'

Figure 25: Helper function: Member hash and Transcript hash. The major changes from [13] are highlighted in yellow. The orange highlights components that are missing from prior works, which we believe is required to satisfy the UC functionality. Please see the proof for more detail.

*derive-keys(G, G', comSecret)

1 : s ← HKDF.Extract(G.initSecret, comSecret)
2 : joinerSecret ← HKDF.Expand(s, 'joi')
3 : (G', confKey) ← *derive-epoch-keys(G', joinerSecret)
4 : **return** (G', confKey, joinerSecret)

*derive-epoch-keys(G', joinerSecret)

1 : confKey ← HKDF.Expand(joinerSecret, G'.groupCont() ∥'conf')
2 : G'.appSecret ← HKDF.Expand(joinerSecret, G'.groupCont() ∥'app')
3 : G'.membKey ← HKDF.Expand(joinerSecret, G'.groupCont() ∥'memb')
4 : G'.initSecret ← HKDF.Expand(joinerSecret, G'.groupCont() ∥'init')
5 : **return** (G', confKey)

Figure 26: Helper function: Key scheduling.

*frame-prop(G, P)

1 : propCont ← (G.groupCont(), G.id, 'proposal', P)
2 : sig ← SIG.Sign($\mathrm{pp}_{\mathrm{SIG}}$, G.ssk, propCont)
3 : membTag ← MAC.TagGen(G.membKey, (propCont, sig))
4 : **return** (G.groupid, G.epoch, G.id, 'proposal', P, sig, membTag)

*unframe-prop(G, p)

1 : **parse** (groupid, epoch, id_s, contType, P, sig, membTag) ← p
2 : **req** contType = 'proposal' ∧ groupid = G.groupid
∧ epoch = G.epoch
3 : propCont ← (G.groupCont(), id_s, 'proposal', P)
4 : **req** G.member[id_s] ≠ ⊥
∧ SIG.Verify($\mathrm{pp}_{\mathrm{SIG}}$, G.member[id_s].svk, sig, propCont)
∧ MAC.TagVerify(G.membKey, membTag, (propCont, sig))
5 : **return** (id_s, P)

*sign-commit(G, C_0)

1 : comCont ← (G.groupCont(), G.id, 'commit', C_0)
2 : sig ← SIG.Sign($\mathrm{pp}_{\mathrm{SIG}}$, G.ssk, comCont)
3 : **return** sig

*frame-commit(G, C_0, sig, confTag)

1 : **return** (G.groupid, G.epoch, G.id, 'commit', C_0, sig, confTag)

*unframe-commit(G, c_0)

1 : **parse** (groupid, epoch, id_c, contType, C_0, sig, confTag) ← c_0
2 : **req** contType = 'commit' ∧ groupid = G.groupid
∧ epoch = G.epoch
3 : comCont ← (G.groupCont-wInterim(), id_c, 'commit', C_0)
4 : svk_c ← G.member[id_c].svk
5 : **req** G.member[id_c] ≠ ⊥
∧ SIG.Verify($\mathrm{pp}_{\mathrm{SIG}}$, svk_c, sig, comCont)
6 : **return** (id_c, C_0, sig, confTag)

Figure 27: Helper function: Frame and unframe packets.

E SECURITY OF CHAINED CMPKE

In this section we provide the full security proof of our proposed protocol Chained CmPKE in App. D. We first explain the **safe** predicate used within the ideal functionality $\mathcal{F}_{\text{CGKA}}$ to exclude trivial attacks. The full security proof is provided subsequently.

E.1 Safety Predicates

Whether security is guaranteed in a given node (i.e, epoch) is determined via an explicit **safe** predicate on the node and the state of the history graph. This is the same approach taken by prior works [12, 13]. Here, in addition to the secrecy of the keys, the functionality also implicitly formalizes authenticity by appropriately disallowing injections.

The safety predicate, depicted in Fig. 28, is defined using recursive deducing rules **know**(c, id) and **know**$(c, \text{'epoch'})$.

know(c, id)**:** It indicates that the adversary knows id's key materials (e.g., decryption key) at epoch c. It consists of four conditions. Conditions (a) or (b) is true if id's key materials at epoch c are known to the adversary because they are exposed at c (Condition (a)) or injected by the adversary at c (Condition (b)). Conditions (c) and (d) reflect the fact that id's key materials will not change unless id commits, updates, is added, or is removed. If c does not change id's key materials, **know**(c, id) implies **know**$(\mathsf{Node}[c].\mathsf{par}, \mathsf{id})$ (Condition (c)). If a child c' does not change id's key materials, **know**(c, id) implies **know**(c', id) (Condition (d)).

know$(c, \text{'epoch'})$**:** It indicates that the adversary knows the epoch secrets (e.g., confirmation key) except for the application secret at epoch c. The adversary knows the epoch secrets if it corrupts a party at c, or if it computes them from the corrupted information. The latter is formalized by the ***can-traverse** predicate, which consists of three conditions. First three conditions of ***can-traverse** reflect the fact that the epoch secrets (or the joiner secret to be more precise) can be computed from welcome messages: Condition (a) is true if a committer processes an injecting add proposals at c; Condition (b) is true if the adversary corrupts the decryption key in the key package used at c; and Condition (c) reflects the fact that the joiner secret is leaked if its ciphertext is generated with bad randomness. The last Condition (d) reflects the fact that the epoch secrets are derived from the initial secret at c's parent node and the commit secret.

The **safe** predicate indicates that whether the adversary knows the application secret at epoch c. Since the application secret is leaked via a corruption query only if $\mathsf{HasKey}[\mathsf{id}] = \mathsf{true}$ (i.e., a party did not output the application secret via Key query), **safe** checks whether $(*, \mathsf{true}) \in \mathsf{Node}[c].\mathsf{exp}$. On the other hand, the other epoch secrets are always leaked when a party at c is corrupted. Thus, **know**$(c, \text{'epoch'})$ simply checks $\mathsf{Node}[c].\mathsf{exp} \neq \emptyset$.

The **sig-inj-allowed** and **mac-inj-allowed** predicates concern the authenticity of the signature and MAC, respectively. Since MAC keys (i.e., membership key and confirmation key) are a part of the epoch secrets, **mac-inj-allowed** is implied by **know**$(c, \text{'epoch'})$. These two predicates are used in `auth-invariant` (see Fig. 19). Condition (a) and (b) of `auth-invariant` reflect the fact that injecting commit or proposal messages needs both a signing key of the sender and the MAC key. Condition (c) of `auth-invariant`, which was previously missing in [13], reflects the fact that injecting welcome messages needs a signing key of the sender. Condition (c) was implicitly handled by the simulator within the security proof in [13], but we believe explicitly including this condition makes the intuition of disallowing injection more clear in the ideal functionality.

E.2 Security Statement

We restate our main theorem Thm. 4.1 that establishes the security of Chained CmPKE. We provide an overview of the proof before diving into the formal proof. Below, if we assume the CmPKE to be only IND-CCA secure, then it satisfies adaptive security with an exponential security loss, while if we assume the CmPKE to be IND-CCA secure with adaptive corruption, then it satisfies adaptive security with only a polynomial security loss.

Theorem E.1. *Assuming that* CmPKE *is* IND-CCA *secure (resp. with adaptive corruption) and* SIG *is* sEUF-CMA *secure, the Chained CmPKE protocol selectively (resp. adaptively) securely realizes the ideal functionality $\mathcal{F}_{CGKA}$, where $\mathcal{F}_{CGKA}$ uses the safety predicate from Fig. 28, in the $(\mathcal{F}_{AS}, \mathcal{F}_{KS}, \mathcal{G}_{RO})$-hybrid model, where calls to the hash function* H, HKDF, *and* MAC *are replaced by a call to the global random oracle $\mathcal{G}_{RO}$.*

Remark 1 (Modeling HKDF and MAC as Random Oracle). *Our proof relies on a variant of the* generalized selective decryption *(GSD) security as in the prior works [9, 12, 13], and it requires that* HKDF.Expand *and* HKDF.Extract *are modeled as a random oracle. More precisely, the reduction is expected to be able to extract a valid MAC secret key from the signature. To this end, we also model* MAC *as a random oracle to incorporate the* MAC *function into the GSD security.*[11] *We consider that the MAC* tag *is the hash value of the MAC key* k *and the message* m*, that is,* $\mathsf{tag} := \mathsf{RO}(\mathsf{k}, \mathsf{m})$ *where* RO *is a random oracle.*

Proof Overview. The high level structure of the proof is similar to [12, 13] who considered the UC security of TreeKEM. The main difference is due to the new **safe** predicate we introduce in order to differentiate between two types of injection attacks: one using signature schemes (see **sig-inj-allowed** in Fig. 19) and the other using MAC (see **mac-inj-allowed** in Fig. 19). Previously, these two types of injection attacks were handled within one hybrid but we differentiate them in hope to make the proof more clear.

Below, we provide an overview of the six hybrids we consider to establish security. We first consider the real world, denoted as Hybrid 1, where the environment $\mathcal{Z}$ is interacting with the real parties and the adversary $\mathcal{A}$. (To be more precise, $\mathcal{Z}$ is interacting with a simulator that internally runs all the real parties and adversary as in the real world).

In Hybrid 2, we swap the ideal authentication and key service $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}})$ to their "ideal world" variant $(\mathcal{F}_{\text{AS}}^{\text{IW}}, \mathcal{F}_{\text{KS}}^{\text{IW}})$, which provides all the secret keys (i.e., secret keys of the signature scheme and CmPKE scheme) to the simulator. Since these functionalities

[11] Previous work [13] assumes the standard EUF-CMA security of MAC, but did not provide a concrete proof. It seems it would be difficult to prove UC security by only assuming the standard EUF-CMA security of MAC.

Knowledge of party's secrets.

know$(c, \mathrm{id}) \iff$
(a) $(\mathrm{id}, *) \in \mathrm{Node}[c].\mathrm{exp} \vee$
(b) ***secrets-injected**$(c, \mathrm{id}) \vee$
(c) $(\mathrm{Node}[c].\mathrm{par} \neq \bot \wedge \neg\, ^*$**secrets-replaced**$(c, \mathrm{id}) \wedge$ **know**$(\mathrm{Node}[c].\mathrm{par}, \mathrm{id})) \vee$
(d) $\exists c' : (\mathrm{Node}[c'].\mathrm{par} = c \wedge \neg\, ^*$**secrets-replaced**$(c', \mathrm{id}) \wedge$ **know**$(c', \mathrm{id}))$

***secrets-injected**$(c, \mathrm{id}) \iff$
(a) $(\mathrm{Node}[c].\mathrm{orig} = \mathrm{id} \wedge \mathrm{Node}[c].\mathrm{stat} \neq \text{'good'}) \vee$
(b) $\exists p \in \mathrm{Node}[c].\mathrm{pro}$:
$(\mathrm{Prop}[p].\mathrm{act} = \text{'upd'-}* \wedge \mathrm{Prop}[p].\mathrm{orig} = \mathrm{id} \wedge \mathrm{Prop}[p].\mathrm{stat} \neq \text{'good'}) \vee$
(c) $\exists p \in \mathrm{Node}[c].\mathrm{pro} : (\mathrm{Prop}[p].\mathrm{act} = \text{'add'-id-svk} \wedge \mathrm{svk} \in \mathrm{Exposed})$

***secrets-replaced**$(c, \mathrm{id}) \iff$
$\mathrm{Node}[c].\mathrm{orig} = \mathrm{id} \vee$
$\exists p \in \mathrm{Node}[c].\mathrm{pro} : \mathrm{Prop}[p].\mathrm{act} \in \{\text{'add'-id-}*, \text{'rem'-id}\} \vee$
$\exists p \in \mathrm{Node}[c].\mathrm{pro} : (\mathrm{Prop}[p].\mathrm{act} = \text{'upd'-}* \wedge \mathrm{Prop}[p].\mathrm{orig} = \mathrm{id})$

Knowledge of epoch secrets.

know$(c, \text{'epoch'}) \iff \mathrm{Node}[c].\mathrm{exp} \neq \emptyset \vee ^*$**can-traverse**$(c)$
***can-traverse**$(c) \iff$
(a) $\exists p \in \mathrm{Node}[c].\mathrm{pro} : (\mathrm{Prop}[p].\mathrm{act} = \text{'add'-id-svk} \wedge \mathrm{svk} \in \mathrm{Exposed}) \vee$
(b) ***reused-welcome-key-leaks**$(c) \vee$
(c) $\mathrm{Node}[c].\mathrm{stat} = \text{'bad'} \wedge \exists p \in \mathrm{Node}[c].\mathrm{pro} : \mathrm{Prop}[p].\mathrm{act} = \text{'add'-}* \vee$
(d) $(c = \mathrm{root}_* \vee$ **know**$(\mathrm{Node}[c].\mathrm{par}, \text{'epoch'})) \wedge$
$\exists (\mathrm{id}, *) \in \mathrm{Node}[c].\mathrm{mem} :$ **know**(c, id)

***reused-welcome-key-leaks**$(c) \iff$
$\exists \mathrm{id}, p \in \mathrm{Node}[c].\mathrm{pro} : \mathrm{Prop}[p].\mathrm{act} = \text{'add'-id-}* \wedge$
$\exists c_d : c_d$ is a descendant of $c \wedge (\mathrm{id}, *) \in \mathrm{Node}[c_d].\mathrm{exp} \wedge$
no node c_h exists on c–c_d path s.t. ***secrets-replaced**$(c_h, \mathrm{id}) = \mathtt{true}$

Safe and can-inject.

safe$(c) \iff \neg\, ((*, \mathtt{true}) \in \mathrm{Node}[c].\mathrm{exp} \vee ^*$**can-traverse**$(c))$
sig-inj-allowed$(c, \mathrm{id}) \iff \mathrm{Node}[c].\mathrm{mem}[\mathrm{id}] \in \mathrm{Exposed}$
mac-inj-allowed$(c) \iff$ **know**$(c, \text{'epoch'})$

Figure 28: The safety predicate for the Chained CmPKE.

are not accessible from $\mathcal{Z}$, one can think of these functions as being simulated by $\mathcal{S}$. In particular, this modification is only conceptual.

In Hybrid 3, we plug in a variant of the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$ in between $\mathcal{Z}$ and the simulator, where the secrets are always set by the simulator and injections are always allowed (i.e., whether `auth-invariant` hold is never checked). This modification concerns the consistency between the protocol states of each user id and the history graph generated by the ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. For instance, if id_1 and id_2 are assigned to the same node in the history graph, that is $\mathsf{Ptr}[\mathsf{id}_1] = \mathsf{Ptr}[\mathsf{id}_2]$, then we want their views in the real protocol to be identical, e.g., they agree on the same group member and group secret. Moreover, this hybrid establishes the correctness of the protocol.

In Hybrid 4, we modify the **sig-inj-allowed** predicate to be those used by the actual ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. This establishes that an adversary cannot inject a malicious message that amounts to breaking the security of the signature scheme.

In Hybrid 5, we modify the **mac-inj-allowed** predicate to be consistent with those used by the actual ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. This establishes that an adversary cannot inject a malicious message without knowing the MAC keys. As in most previous proofs [9, 12, 13], we rely on a variant of the *generalized selective decryption* (GSD) security, which we formally introduce as the *Chained CmPKE conforming GSD* security in App. F. At a high level, the GSD security extracts the essence of the secrecy guarantee of the group secret and simplifies the proof. In this part, we first prove that if $\mathcal{Z}$ can distinguish between Hybrids 4 and 5, then it can be used to break the Chained CmPKE conforming GSD security. We then show in App. F that no efficient adversary can break the Chained CmPKE conforming GSD security assuming the security of CmPKE, which proves that Hybrids 4 and 5 remain the same in the view of $\mathcal{Z}$. We note that the variant of GSD security we introduce in this work is much more tailored to the CGKA setting than those previously considered and allows for a much simpler proof.

In Hybrid 6, we use the original **safe** predicate to be those used by the actual ideal functionality $\mathcal{F}_{\mathrm{CGKA}}$. This establishes that the application secret looks random as long as **safe** is true for the epoch. We prove that if $\mathcal{Z}$ can distinguish between Hybrids 5 and 6, then it can be used to break the Chained CmPKE conforming GSD security of CmPKE. At this point, the functionality that sits between $\mathcal{Z}$ and the simulator is exactly $\mathcal{F}_{\mathrm{CGKA}}$, thus we complete the proof. □

Proof. We now provide a more formal proof of the above overview. Below, we use a sequence of hybrids explained above. We gradually modify the behavior of the simulator and denote the simulator in Hybrid i as $\mathcal{S}_i$. The first (resp. last) hybrid provides the environment $\mathcal{Z}$ the view of the real (resp. ideal) world. Below, when we say "the simulator aborts", we mean that the simulator terminates the simulation and does not respond to further queries made by the environment $\mathcal{Z}$.

Hybrid 1. This is the real world, where we make a syntactic change. We consider a simulator $\mathcal{S}_1$ that interacts with a dummy functionality $\mathcal{F}_{\mathrm{dummy}}$ and $(\mathcal{F}_{\mathrm{AS}}, \mathcal{F}_{\mathrm{KS}})$. $\mathcal{F}_{\mathrm{dummy}}$ sits between the environment $\mathcal{Z}$ and $\mathcal{S}_1$, and simply routs all messages without any modification. $\mathcal{S}_1$ internally runs the real-world parties and adversary $\mathcal{A}$ by routing all messages sent from $\mathcal{F}_{\mathrm{dummy}}$, which corresponds to those from $\mathcal{Z}$.

Hybrid 2. This change concerns the authentication and key service. In this world, $(\mathcal{F}_{\mathrm{AS}}, \mathcal{F}_{\mathrm{KS}})$ is replaced with $(\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}, \mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}})$. Since these functions are not accessible by $\mathcal{Z}$, this modification is undetectable from $\mathcal{Z}$. Hence, the view of $\mathcal{Z}$ in Hybrid 1 and Hybrid 2 are identical.[12]

[12] Since $(\mathcal{F}_{\mathrm{AS}}, \mathcal{F}_{\mathrm{KS}})$ are local functions, we can instead simply assume that the simulator simulates these functionalities rather than replacing them. We use $(\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}, \mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}})$ to be consistent with the presentation provided in [13].

Hybrid 3. This change concerns consistency guarantees. We replace $\mathcal{F}_{\text{dummy}}$ with a variant of $\mathcal{F}_{\text{CGKA}}$, denoted as $\mathcal{F}_{\text{CGKA},3}$, where **safe** (resp. **sig-inj-allowed** and **mac-inj-allowed**) always returns false (resp. true). In other words, all application secrets are set by the simulator and injections are always allowed. The simulator $\mathcal{S}_3$ sets all messages and keys according to the protocol.

Hybrid 4. This change concerns the security of the signature scheme. We further modify $\mathcal{F}_{\text{CGKA},3}$ to use the original **sig-inj-allowed** predicate, denoted as $\mathcal{F}_{\text{CGKA},4}$. $\mathcal{F}_{\text{CGKA},4}$ halts if a message is injected even if the sender's signing key is not exposed. The simulator $\mathcal{S}_4$ is identical to $\mathcal{S}_3$.

Hybrid 5. This change concerns the security of the MAC. We further modify $\mathcal{F}_{\text{CGKA},4}$ to use the original **mac-inj-allowed** predicate, denoted as $\mathcal{F}_{\text{CGKA},5}$. $\mathcal{F}_{\text{CGKA},5}$ halts if a proposal or commit message is injected even if the corresponding MAC key is not exposed. The simulator $\mathcal{S}_5$ is identical to $\mathcal{S}_4$.

Hybrid 6. This change concerns the confidentiality of application secrets. We further modify $\mathcal{F}_{\text{CGKA},5}$ where it uses the original **safe** predicate, denoted as $\mathcal{F}_{\text{CGKA},6}$. The simulator $\mathcal{S}_6$ is identical to $\mathcal{S}_5$ except that it sets only those application secrets for which **safe** is false. This functionality corresponds to the ideal functionality $\mathcal{F}_{\text{CGKA}}$.

We show indistinguishability of Hybrids 2 to 6 in Lems. E.2, E.15, E.19 and E.29. This completes the proof of the main theorem. □

E.3 From Hybrid 2 to 3: Lem. E.2

To show Lem. E.2, we first consider additional hybrids (Hybrids 2-1 to 2-7) in between Hybrids 2 and 3 and show that each adjacent hybrids are indistinguishable. The most technically involved proof is showing the indistinguishability of Hybrids 2-4 and 2-5. All other hybrids are simply provided to make the proof between Hybrids 2-4 and 2-5 readable by taking care of subtleties such as decryption error, collisions in random oracles, and so on. Namely, for those interested readers, we believe it would be informative to check the proof between Hybrids 2-4 and 2-5 (Lem. E.5) before checking the other hybrids.

E.3.1 Intermediate Hybrids. Here, we first provide the additional hybrids.

Hybrid 2-0 := Hybrid 2. This is identical to Hybrid 2.

Hybrid 2-1. [No collision in RO] This change concerns the collision resistance of the random oracle. Recall that all queries regarding the hash function H is simulated using the (global) random oracle. In this hybrid, we consider a simulator $\mathcal{S}_{2\text{-}1}$ that aborts when a collision ever occurs in the random oracle. Since $\mathcal{Z}$ only makes at most polynomially many queries, this makes negligible change to the view of $\mathcal{Z}$. Hence, the view of $\mathcal{Z}$ in Hybrid 2-0 and Hybrid 2-1 are negligibly different.

Hybrid 2-2. [Unique confTag/membTag **in** $L_{\text{prop}}/L_{\text{com}}/L_{\text{wel}}$ **]** This concerns the uniqueness of membTag included in a proposal message and the uniqueness of confTag included in commit and welcome messages. We consider a simulator $\mathcal{S}_{2\text{-}2}$ defined exactly as $\mathcal{S}_{2\text{-}1}$ except that it maintains three lists $L_{\text{prop}}, L_{\text{com}}$, and L_{wel} all initially set to $\emptyset$ and performs the following additional checks.

Checks regarding L_{prop}. Let G be the protocol state of party id *before* being invoked by $\mathcal{S}_{2\text{-}2}$. There are three checks $\mathcal{S}_{2\text{-}2}$ performs. First, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input (Propose, act), if id outputs a proposal message p, then $\mathcal{S}_{2\text{-}2}$ extracts membTag included in p (which is guaranteed to exist) and checks if there exists an entry $(p', \text{membKey}, \text{membTag}) \in L_{\text{prop}}$ such that $(p', \text{membKey}) \neq (p, G.\text{membKey})$. If so $\mathcal{S}_{2\text{-}2}$ aborts. Otherwise it updates the list $L_{\text{prop}} \leftarrow\!\!+ (p, G.\text{membKey}, \text{membTag})$. Second, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input (Commit, $\vec{p}$, svk), if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ extracts membTag included in each $p \in \vec{p}$ (which is guaranteed to exist) and performs the same procedure above for each p. Finally, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input (Propose, $c_0, \widehat{c}, \vec{p}$), if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ extracts membTag included in each $p \in \vec{p}$ (which is guaranteed to exist) and performs the same procedure above for each p.

Checks regarding L_{com}. Let G be the protocol state of party id *after* being invoked by $\mathcal{S}_{2\text{-}2}$. There are two checks $\mathcal{S}_{2\text{-}2}$ performs. First, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input (Commit, $\vec{p}$, svk), if id outputs $(c_0, \vec{c}, w_0, \vec{w})$, then $\mathcal{S}_{2\text{-}2}$ extracts confTag included in c_0 (which is guaranteed to exist). Moreover, let $G' = G.\text{pendCom}[c_0]$. Then, $\mathcal{S}_{2\text{-}2}$ checks if there exists an entry $(c_0', \text{confKey}, \text{confTransHash}, \text{confTag}) \in L_{\text{com}}$ where we have $(c_0', \text{confKey}, \text{confTransHash}) \neq (c_0, G'.\text{confKey}, G'.\text{confTransHash})$. If so $\mathcal{S}_{2\text{-}2}$ aborts, and otherwise it updates the list $L_{\text{com}} \leftarrow\!\!+ (c_0, G'.\text{confKey}, G'.\text{confTransHash}, \text{confTag})$. Second, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input (Propose, $c_0, \widehat{c}, \vec{p}$), if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ extracts confTag included in c_0 (which is guaranteed to exist) and checks if there exists an entry $(c_0', \text{confKey}, \text{confTransHash}, \text{confTag}) \in L_{\text{com}}$ where $(c_0', \text{confKey}, \text{confTransHash}) \neq (c_0, G.\text{confKey}, G.\text{confTransHash})$. If so $\mathcal{S}_{2\text{-}2}$ aborts. Otherwise it updates $L_{\text{com}} \leftarrow\!\!+ (c_0, G.\text{confKey}, G.\text{confTransHash}, \text{confTag})$.

Checks regarding L_{wel}. Let G be the protocol state of party id *after* being invoked by $\mathcal{S}_{2\text{-}2}$. There are two checks $\mathcal{S}_{2\text{-}2}$ performs. First, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input (Commit, $\vec{p}$, svk), if id outputs $(c_0, \vec{c}, w_0, \vec{w})$, then $\mathcal{S}_{2\text{-}2}$ parses w_0 as (groupInfo, T, sig) and extracts confTag included in groupInfo (which is guaranteed to exist). Moreover, let $G' = G.\text{pendCom}[c_0]$. It then checks if there exists an entry (groupInfo, confKey, confTransHash, confTag) $\in L_{\text{wel}}$ where (groupInfo′, confKey, confTransHash) $\neq$ (groupInfo, G'.confKey, G'.confTransHash). If so $\mathcal{S}_{2\text{-}2}$ aborts and otherwise it updates the list $L_{\text{wel}} \leftarrow\!\!+$ (groupInfo, G'.confKey, G'.confTransHash, confTag). Second, when $\mathcal{S}_{2\text{-}2}$ invokes party id on input (Join, $w_0, \widehat{w}$), if id outputs non-$\perp$, then $\mathcal{S}_{2\text{-}2}$ parses w_0 as (groupInfo, T, sig) and extracts confTag included in groupInfo and checks if there exists an entry (groupInfo, confKey, confTransHash, confTag) $\in$ L_{wel} where (groupInfo′, confKey, confTransHash) $\neq$ (groupInfo, G.confKey, G.confTransHash). If so $\mathcal{S}_{2\text{-}2}$ aborts. Otherwise it updates $L_{\text{wel}} \leftarrow\!\!+$ (groupInfo, G.confKey, G.confTransHash, confTag).

Checks regarding L_{com} and L_{wel}. Every time $\mathcal{S}_{2\text{-}2}$ updates L_{com} or L_{wel}, it checks if there exists (c_0, confKey, confTransHash, confTag) $\in L_{\text{com}}$ and (groupInfo, confKey′, confTransHash′, confTag′) $\in L_{\text{wel}}$ such that confTag = confTag′ but (id_c, confKey,

confTransHash) ≠ (id′_c, confKey′, confTransHash′), where id_c and id′_c are the (purported) user identity included in c_0 and groupInfo, respectively. If so, $\mathcal{S}_{2\text{-}2}$ aborts.

We show in Lem. E.3 that Hybrid 2-1 and Hybrid 2-2 are indistinguishable to $\mathcal{Z}$ assuming collision resistance of MAC.

Hybrid 2-3. [Unique c_0 with good randomness] This concerns the uniqueness of a commit message with good randomness. Assume party id outputs $(c_0, \vec{c}, w_0, \widehat{w})$ (where possibly $(w_0, \widehat{w}) = (\perp, \perp)$) on input (Commit, $\vec{p}$, svk), where RandCor[id] = 'good'. Let $G' = G$.pendCom[c_0] and G be the protocol state after being invoked by $\mathcal{S}_{2\text{-}2}$. We consider a simulator $\mathcal{S}_{2\text{-}3}$ that aborts if the same (c_0, G'.confKey, G'.confTransHash) is already included in L_{com}. Recall that in the previous hybrid, we did not abort in case the same entry was found. $\mathcal{S}_{2\text{-}3}$ is otherwise defined identically to $\mathcal{S}_{2\text{-}2}$.

Now, in case RandCor[id] = 'good', due to the ciphertext-spreadness of CmPKE (see Def. B.1), c_0 has high min-entropy. Therefore, the probability of c_0 already being in L_{com} is negligibly small. Hence, Hybrid 2-2 and Hybrid 2-3 are indistinguishable to $\mathcal{Z}$.

Hybrid 2-4. [Consistency of no-join] This concerns the consistency of confTag included in commit and welcome messages. Assume party id outputs $(c_0, \vec{c}, \perp, \perp)$ on input (Commit, $\vec{p}$, svk). That is, there are no newly added members to the group. If any party id′ ever correctly processes (Join, w_0, $\widehat{w}$) (i.e., id′ outputs (id_c, mem)) and w_0 includes the same confTag as the one included in c_0, then $\mathcal{S}_{2\text{-}4}$ aborts. Otherwise, $\mathcal{S}_{2\text{-}4}$ is identical to the previous simulator. Informally, this implies that confTag implicitly commits to the information of the group members and if confTag was generated as a result of no new additions, then confTag cannot be used as a welcome message. We show in Lem. E.4 that Hybrid 2-3 and Hybrid 2-4 are indistinguishable to $\mathcal{Z}$ assuming collision resistance of MAC.

Hybrid 2-5. [Adding consistency checks] This change concerns consistency guarantees. We replace $\mathcal{F}_{\text{dummy}}$ with a variant of $\mathcal{F}_{\text{CGKA}}$, denoted as $\mathcal{F}_{\text{CGKA,2-5}}$, where **safe** (resp. **sig-inj-allowed** and **mac-inj-allowed**) always returns false (resp. true), and the correctness conditions *succeed-com, *succeed-proc, and *succeed-wel always output false. In other words, all application secrets are set by the simulator, injections are always allowed, and the protocol does not need to satisfy correctness. However, the simulator $\mathcal{S}_{2\text{-}5}$ does set all messages and keys according to the protocol. We show in Lem. E.5 that Hybrid 2-4 and Hybrid 2-5 are identical.

Hybrid 2-6. [No correctness error] This change concerns the correctness of the signature scheme and encryption scheme. We replace $\mathcal{F}_{\text{CGKA,2-5}}$ with $\mathcal{F}_{\text{CGKA,2-5}}$, where the only difference is that the correctness conditions *succeed-com, *succeed-proc, and *succeed-wel defined as those in the ideal functionality $\mathcal{F}_{\text{CGKA}}$. At a high level, these correctness conditions guarantee that if the real protocol is run as expected, then there should be no correctness error. Moreover, this should hold true even if bad randomness is used.[13]

We show in Lem. E.14 that Hybrid 2-5 and Hybrid 2-6 are identical.

Hybrid 2-7 := Hybrid 3. [Removing abort conditions] This is identical to Hybrid 3. The only difference between Hybrid 2-6 is that the simulator $\mathcal{S}_{2\text{-}7} = \mathcal{S}_3$ no longer aborts the simulation. Specifically, we remove all the abort condition checked by the simulator that was introduced from moving to Hybrid 2-0 to 2-5. Using the same arguments to move through Hybrid 2-0 to Hybrid 2-5, Hybrid 2-6 and Hybrid 2-7 remains indistinguishable.

E.3.2 From Hybrid 2 to 3: Proof of main Lem. E.2. The following is the main lemma of this section which proves indistinguishability between Hybrid 2 and 3. The proof is a direct consequence of the argument made in App. E.3.1 and the subsequent Lems. E.3 to E.5.

Lemma E.2. *Hybrid 2 and Hybrid 3 are indistinguishable assuming the collision resistance of* MAC*, the correctness of* CmPKE *and* SIG*, and the ciphertext-spreadness of* CmPKE*.*

E.3.3 From Hybrid 2-1 to 2-2: Proof of Lem. E.3.

Lemma E.3. *Hybrid 2-1 and Hybrid 2-2 are indistinguishable assuming* MAC *is collision resistant.*

Proof. We first consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking the list L_{prop}. $\mathcal{S}_{2\text{-}2}$ checks the list during either a propose, commit, or process query. Assume $\mathcal{S}_{2\text{-}2}$ was invoking party id on a propose query. By correctness of the propose protocol, if id outputs p = (groupid, epoch, id, 'proposal', P, sig, membTag), then we have the following

- G.groupid = groupid;
- G.epoch = epoch;
- G.groupCont() = (G.groupid, G.epoch, G.memberHash, G.confTransHash);
- membTag = MAC.TagGen(G.membKey, (G.groupCont(), id, 'proposal', P, sig)),

where recall G is the protocol state of id. Notice that the entire description of p is included as a message of membTag. Then if there exists $(p', \text{membKey}') \neq (p, G.\text{membKey})$ then it can be used to break collision resistance of MAC. The proof for the other cases where $\mathcal{S}_{2\text{-}2}$ was invoking party id on a commit or process query is identical to the above. Therefore, assuming collision resistance of MAC, the abort condition regarding L_{prop} cannot occur.

We next consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking the list L_{com}. $\mathcal{S}_{2\text{-}2}$ checks the list during either a commit or process query. Assume $\mathcal{S}_{2\text{-}2}$ was invoking party id on a commit query. By correctness of the commit protocol, if id outputs c_0 = (groupid, epoch, id_c, 'commit', C_0 = (propIDs, kp, T), sig, confTag), then we have the following

- G'.groupid = G.groupid = groupid;
- G'.epoch = G.epoch + 1 = epoch + 1;
- G'.confTransHash-w.o-'id_c' = H(G.interimTransHash, (G.groupid, epoch, 'commit', C_0, sig));
- G'.confTransHash = H(G'.confTransHash-w.o-'id_c', id_c);

[13] Unlike classical schemes (e.g., ElGamal encryption), there are correctness errors in post-quantum schemes such as those based on lattices. Looking ahead, we argue that no adversary can find a bad randomness that leads to a correctness error by requiring the underlying cryptographic primitives to expand the randomness through a hash function model as a random oracle. Concretely, the adversary can only control the random seed, which is then expanded via a hash function (or more precisely a PRG) modeled as a random oracle.

- G'.groupCont() = (G'.groupid, G'.epoch, G'.memberHash, G'.confTransHash);
- G'.confKey = H(G'.joinerSecret, G'.groupCont(), 'conf');
- confTag = MAC.TagGen(G'.confKey, G'.confTransHash);

where recall G' is the pending protocol state of id included in G.pendCom[c_0]. Due to the modification we made in Hybrid 2-1 [*No collision in RO*], c_0 is the unique commitment that leads to G'.confKey. Namely, for any (c_0', confKey, confTransHash, confTag) $\in L_{\mathsf{com}}$, we have confKey $\neq G'$.confKey if $c_0' \neq c_0$. Then, regardless of $c_0' \neq c_0$ or $c_0' = c_0$, we would have (confKey, confTransHash) $\neq$ (G'.confKey, G'.confTransHash). Hence, the abort condition in L_{com} does not occur. The proof for the other case where $\mathcal{S}_{2\text{-}2}$ was invoking party id on a process query is identical to the above, where the only difference is that G' is the updated protocol state of id rather then the pending protocol state. Therefore, assuming collision resistance of MAC, the abort condition regarding L_{com} cannot occur.

We next consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking the list L_{wel}. $\mathcal{S}_{2\text{-}2}$ checks the list during either a commit or join query. Assume $\mathcal{S}_{2\text{-}2}$ was invoking party id on a commit query. Then, by correctness of the commit protocol, if id outputs w_0 = (groupInfo = (groupid, epoch, memberPublicInfo, memberHash, confTransHash-w.o-'id_c', confTransHash, interimTransHash, confTag, id_c), T, sig) then we have the above listed relations considered during L_{com}. Notice due to the modification we made in Hybrid 2-1 [*No collision in RO*], (groupid, epoch, memberHash, confTransHash-w.o-'id_c', id_c) is the unique pair that leads to a valid memberPublicInfo and interimTransHash[14], where recall memberHash is set via the helper function `*derive-member-hash` (see Fig. 25). This in particular implies for any (groupInfo$'$, confKey, confTransHash, confTag) $\in L_{\mathsf{wel}}$, we have confKey $\neq G'$.confKey if groupInfo$'$ $\neq$ groupInfo. Then, regardless of groupInfo$'$ $\neq$ groupInfo or groupInfo$'$ = groupInfo, we have (confKey, confTransHash) $\neq$ (G'.confKey, G'.confTransHash). Hence, the abort condition in L_{wel} does not occur. The proof for the other case where $\mathcal{S}_{2\text{-}2}$ was invoking party id on a join query is identical to the above. Therefore, assuming collision resistance of MAC, the abort condition regarding L_{wel} cannot occur.

We finally consider the case $\mathcal{S}_{2\text{-}2}$ aborts while checking both of the lists L_{com} and L_{wel}. It is clear that we cannot have confTag = confTag$'$ while (confKey, confTransHash) $\neq$ (confKey$'$, confTransHash$'$) since this can be directly used to break collision resistance of MAC. However, recall confTransHash is created by hashing confTransHash-w.o-'id_c' and id_c. Therefore, due to the modification we made in Hybrid 2-1 [*No collision in RO*], id_c and id_c' must be the same as well. This establishes (id_c, confKey, confTransHash) = (id_c', confKey$'$, confTransHash$'$).

This completes the proof. □

E.3.4 From Hybrid 2-3 to 2-4: Proof of Lem. E.4.

Lemma E.4. *Hybrid 2-3 and Hybrid 2-4 are indistinguishable assuming* MAC *is collision resistant.*

Proof. Let G_{id} and $G'_{\mathsf{id}'}$ be the protocol states of id and id$'$ after they execute the commit and join query, respectively. Moreover, let G'_{id} be the pending protocol state stored in G_{id}.pendCom[c_0]. Then, by the correctness of the protocol, since the commit c_0 created by id does not include new parties, we must have G'_{id}.memberIDsvks() $\neq$ $G'_{\mathsf{id}'}$.memberIDsvks(). Then, due to the modification we made in Hybrid 2-1 [*No collision in RO*] and taking into consideration of how confKey is generated, we must have G'_{id}.confKey $\neq$ $G'_{\mathsf{id}'}$.confKey. However, this cannot happen since otherwise $\mathcal{S}_{2\text{-}4}$ can break collision resistance of MAC by outputting (G'_{id}.confKey, G'_{id}.confTransHash, $G'_{\mathsf{id}'}$.confKey, $G'_{\mathsf{id}'}$.confTransHash, confTag).

This completes the proof. □

E.3.5 From Hybrid 2-4 to 2-5: Proof of Lem. E.5.

This is the technically most involved lemma which checks the consistency between the real protocol and the ideal protocol. The proof consists of three parts: we first formally define the behavior of simulator $\mathcal{S}_{2\text{-}5}$ in Hybrid 2-5 (see *Part 1*); we then provide supporting propositions that establish consistencies between the protocol states and the history graph (see *Part 2*); finally, using the supporting propositions, we analyze the simulation provided by $\mathcal{S}_{2\text{-}5}$ provides an identical view to $\mathcal{Z}$ as in Hybrid 2-4 (see *Part 3*).

Part 1. Description of the Simulator $\mathcal{S}_{2\text{-}5}$. Throughout the hybrid, $\mathcal{S}_{2\text{-}5}$ creates the same history graph created within $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$. That is, it initializes Ptr[$*$], Node[$*$], Prop[$*$], and so on and maintains the same view as $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$. Moreover, throughout this hybrid, we augment the protocol state G of party id to also maintain the values presented in Tab. 9. Although these values are deleted once the protocol state is updated in the real protocol, e.g., after processing a process query, we can keep these without loss of generality as they are never provided to the environment $\mathcal{Z}$ or the adversary $\mathcal{A}$. In particular, they will simply be helpful objects to discuss the consistency of the simulation.

The description of $\mathcal{S}_{2\text{-}5}$ consists of how it answers each queries made by the ideal functionality $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$. Here, note that any queries made by $\mathcal{Z}$ to $\mathcal{S}_{2\text{-}5}$ will be simply relayed to the internally simulated $\mathcal{A}$. Moreover, $\mathcal{S}_{2\text{-}5}$ aborts the simulation whenever any of the checks we included in Hybrids 2-1 to 2-5 are triggered.

(1) Create query from $\mathsf{id}_{\mathsf{creator}}$. This concerns the case when $\mathcal{Z}$ queries (Create, svk) to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$. If $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$ outputs (Create, $\mathsf{id}_{\mathsf{creator}}$, svk) to $\mathcal{S}_{2\text{-}5}$, $\mathcal{S}_{2\text{-}5}$ simply runs the simulated party $\mathsf{id}_{\mathsf{creator}}$ on input (Create, svk).

(2) Propose query from id. This concerns the case when $\mathcal{Z}$ queries (Propose, act) for some act $\in$ { 'upd'-svk, 'add'-id_t, 'rem'-id_t } to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$. If Ptr[id] $\neq \perp$, then $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$ outputs (Propose, id, act) to $\mathcal{S}_{2\text{-}5}$. $\mathcal{S}_{2\text{-}5}$ then runs the simulated party id on input (Propose, act), where it asks $\mathcal{A}$ to provide the randomness to run party id if RandCor[id] = 'bad' and act = 'upd'-svk. Here, recall randomness is only used to generate a new key package (kp, dk).[15] If party id returns $\perp$, then $\mathcal{S}_{2\text{-}5}$ returns (ack := false, $\perp$, $\perp$) to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$. Otherwise, if id returns p, then $\mathcal{S}_{2\text{-}5}$ returns (ack := true, p, svk_t), where a long-term key $\mathsf{svk}_t \neq \perp$ is extracted from p only when act = 'add'-id_t.

(3) Commit query from id. This concerns the case when $\mathcal{Z}$ queries (Commit, $\vec{p}$, svk) to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$. If Ptr[id] $\neq \perp$, then $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$

[14] Here, we explicitly rely on the new confTransHash-w.o-'id_c' satisfying confTransHash = H(confTransHash-w.o-'id_c', id_c).

[15] As in prior works, note that $\mathcal{A}$ is only allowed to control the output of party id via randomness corruption. This is to capture *post-compromise security* in a meaningful way. We impose the same restriction when id is invoked on a commit query. See [12, 13] for more details.

outputs (Commit, id, $\vec{p}$, svk) to $\mathcal{S}_{2\text{-}5}$. $\mathcal{S}_{2\text{-}5}$ then runs the simulated party id on input (Commit, $\vec{p}$, svk), where it asks $\mathcal{A}$ to provide the randomness to run party id if RandCor[id] = 'bad'. If party id returns $\perp$, then $\mathcal{S}_{2\text{-}5}$ returns (ack := false, $\perp, \perp, \perp, \perp, \perp$) to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, if party id returns ($c_0, \vec{c}, w_0, \vec{w}$), then it checks if Node[c_0] = $\perp$, $w_0 \neq \perp$, and if there exists some $rt' \in \mathbb{N}$ and w_0' such that Wel[w_0'] = root$_{rt'}$ and w_0' includes the same confTag as w_0. If so, $\mathcal{S}_{2\text{-}5}$ chooses any such (w_0', rt') and returns (ack := true, $rt := rt', c_0, \vec{c}, w_0, \vec{w}$) to $\mathcal{F}_{\text{CGKA,2-5}}$. As we show in Proposition E.6 below, such for any such pair, the value of rt' is unique. Otherwise, if either Node[c_0] $\neq \perp$; or $w_0 = \perp$; or there does not exist w_0' such that Wel[w_0'] = root$_{rt'}$ for some $rt' \in \mathbb{N}$ and w_0' includes the same confTag as w_0, then $\mathcal{S}_{2\text{-}5}$ returns (ack := true, $rt := \perp, c_0, \vec{c}, w_0, \vec{w}$) to $\mathcal{F}_{\text{CGKA,2-5}}$. Finally, when $\mathcal{F}_{\text{CGKA,2-5}}$ queries (Propose, p) to $\mathcal{S}_{2\text{-}5}$ during the *fill-prop check, $\mathcal{S}_{2\text{-}5}$ extracts the unique orig = id and act included in p (which are guaranteed to exist when commit succeeds in the real protocol) and returns them to $\mathcal{F}_{\text{CGKA,2-5}}$.

(4) Process query from id**.** This concerns the case when $\mathcal{Z}$ queries (Process, $c_0, \widehat{c}, \widehat{p}$) to $\mathcal{F}_{\text{CGKA,2-5}}$. If Ptr[id] $\neq \perp$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs (Process, id, $c_0, \widehat{c}, \widehat{p}$) to $\mathcal{S}_{2\text{-}5}$. $\mathcal{S}_{2\text{-}5}$ then (deterministically) runs the simulated party id on input (Process, $c_0, \widehat{c}, \widehat{p}$). If party id returns $\perp$, then $\mathcal{S}_{2\text{-}5}$ returns (ack := false, $\perp, \perp, \perp$) to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, if party id returns (id$_c$, $upd \| rem \| add$), then $\mathcal{S}_{2\text{-}5}$ checks if Node[c_0] = $\perp$ and if there exists w_0 that includes the same confTag as c_0 such that Wel[w_0] = root$_{rt'}$ for some $rt' \in \mathbb{N}$. If so, $\mathcal{S}_{2\text{-}5}$ chooses any such (w_0, rt') and returns (ack := true, $rt := rt', \perp, \perp$) to $\mathcal{F}_{\text{CGKA,2-5}}$. As we show in Proposition E.6 below, such for any such pair, the value of rt' is unique. If Node[c_0] = $\perp$ and no such w_0 exists, then $\mathcal{S}_{2\text{-}5}$ retrieves the associating long-term public key svk$_c$ of id$_c$ (which is guaranteed to exist when process succeeds in the real protocol) and returns (ack := true, $\perp$, orig$'$:= id$_c$, svk$'$:= svk$_c$). Finally, if Node[c_0] $\neq \perp$, then $\mathcal{S}_{2\text{-}5}$ simply returns (ack := true, $\perp, \perp, \perp$). *fill-prop queries from $\mathcal{F}_{\text{CGKA,2-5}}$ to $\mathcal{S}_{2\text{-}5}$ are answered exactly as in commit queries described above.

(5) Join query from id**.** This concerns the case when $\mathcal{Z}$ queries (Join, $w_0, \widehat{w}$) to $\mathcal{F}_{\text{CGKA,2-5}}$. If Ptr[id] = $\perp$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs (Join, id, $w_0, \widehat{w}$) to $\mathcal{S}_{2\text{-}5}$. $\mathcal{S}_{2\text{-}5}$ then (deterministically) runs the simulated party id on input (Join, $w_0, \widehat{w}$). If party id returns $\perp$, then $\mathcal{S}_{2\text{-}5}$ returns (ack := false, $\perp, \perp, \perp$) to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, if party id returns (id$_c$, mem), where mem is a list of (id, svk)-tuples, then $\mathcal{S}_{2\text{-}5}$ checks if Wel[w_0] $\neq \perp$. If so, $\mathcal{S}_{2\text{-}5}$ returns (ack := true, $\perp, \perp, \perp$) to $\mathcal{F}_{\text{CGKA,2-5}}$. Otherwise, it checks if there exists a non-root c_0 such that Node[c_0] $\neq \perp$ and c_0 includes the same confTag as the one included in w_0. Due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}] and by how $\mathcal{S}_{2\text{-}5}$ simulates the commit and process query (see above **(4)** and **(5)**), such c_0 is unique if it exists. Now, if such c_0 exists, then $\mathcal{S}_{2\text{-}5}$ returns (ack := true, $c_0' := c_0, \perp, \perp$). Otherwise, if no such c_0 exists, then $\mathcal{S}_{2\text{-}5}$ further checks if there exists w_0' such that Wel[w_0'] $\neq \perp$ that includes the same confTag as the one included in w_0. If so, $\mathcal{S}_{2\text{-}5}$ chooses any such w_0' and returns (ack := true, $c_0' :=$ Wel[w_0'], $\perp, \perp$). As we show in Proposition E.6 below, the value of Wel[w_0'] (which can either be a non-root or a detached root) is the same for all such w_0'. Finally, if no such c_0 or w_0' exist, then $\mathcal{S}_{2\text{-}5}$ returns (ack := true, $\perp$, orig$'$:= id$_c$, mem$'$:= mem). This corresponds to the case Wel[w_0] is initialized by root$_{rt}$ for a new $rt \in \mathbb{N}$.

(6) Key query from id**.** This concerns the case when $\mathcal{Z}$ queries Key to $\mathcal{F}_{\text{CGKA,2-5}}$. If Ptr[id] = $c_0 \neq \perp$, HasKey[id] = true, and Node[Ptr[id]].key = $\perp$, then $\mathcal{F}_{\text{CGKA,2-5}}$ outputs (Key, id) to $\mathcal{S}_{2\text{-}5}$. (Recall that **safe** is always set to false in this hybrid.) If HasKey[id] = true, $\mathcal{S}_{2\text{-}5}$ must have invoked party id on input either a valid ($c_0, \widehat{c}, \widehat{p}$) corresponding to a process query or ($w_0, \widehat{w}$) corresponding to a join query. In either case, the simulated party id is guaranteed to have computed a valid appSecret which is stored in its protocol state G (i.e., G.appSecret). Thus, $\mathcal{S}_{2\text{-}5}$ returns key := G.appSecret to $\mathcal{F}_{\text{CGKA,2-5}}$.

With the simulator $\mathcal{S}_{2\text{-}5}$ formally defined, we are now ready to prove the following lemma.

Lemma E.5. *Hybrid 2-4 and Hybrid 2-5 are identical.*

Proof. *Part 2. Supporting Propositions.* Directly proving that the simulation provided by $\mathcal{S}_{2\text{-}5}$ creates an identical view to $\mathcal{Z}$ as in the previous hybrid is quite complex and possibly unreadable. To this end, we provide several supporting propositions that check the consistency within and between the history graph and protocol states maintained by $\mathcal{S}_{2\text{-}5}$ (and the ideal functionality $\mathcal{F}_{\text{CGKA,2-5}}$). The interested readers may first skim through Part 3 to check how the supporting propositions are used. Looking ahead, we are able to prove that cons-invariant in Fig. 19 as a simple corollary of the propositions we prove in *Part 2.*

Part 2-1. Basic checks within/between history graphs and protocol states. The following shows that informally, if two w_0 and w_0' share the same confTag, then their corresponding nodes Wel[w_0] and Wel[w_0'] must be assigned to the same non-root or a detached root. This shows that the confTag included in the welcome message w_0 commits the added users to be in the same group state (or equivalently to the same node in the history graph). Roughly, if this does not hold, then the environment can distinguish between the previous hybrid by causing an inconsistency in the history graph between parties that should belong to the same node Node[c_0].

Proposition E.6 (Uniqueness of confTag in welcome message). *If two distinct w_0 and w_0' that include the same* confTag *satisfy* Wel[w_0] $\neq \perp$ *and* Wel[w_0'] $\neq \perp$*, then we must have* Wel[w_0] = Wel[w_0'].

Proof. Let us prove by contradiction. Assume we have two distinct w_0 and w_0' that include the same confTag but either of the following four cases hold:

(1) (Wel[w_0], Wel[w_0']) = (root$_{rt}$, root$_{rt'}$) for some distinct rt and $rt' \in \mathbb{N}$;
(2) (Wel[w_0], Wel[w_0']) = (c_0, root$_{rt'}$) for some $rt' \in \mathbb{N}$ and non-root c_0;
(3) (Wel[w_0], Wel[w_0']) = (root$_{rt}$, c_0') for some $rt \in \mathbb{N}$ and non-root c_0';
(4) (Wel[w_0], Wel[w_0']) = (c_0, c_0') for some distinct non-roots c_0 and c_0'.

Note that by construction Wel[w_0] or Wel[w_0'] is never attached to the main root root$_0$ so we can safely discard this case. Below we assume Wel[w_0'] is set before Wel[w_0] and further assume that

all other w_0'' with the same confTag as w_0' satisfies $\mathsf{Wel}[w_0'] = \mathsf{Wel}[w_0'']$ and are set *after* $\mathsf{Wel}[w_0']$ is set. Namely, we assume without loss of generality that $\mathsf{Wel}[w_0']$ is the first to be created and w_0 to be the first welcome message that forms the contradiction.

Case (1) and *(2)*: $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$. Observe $\mathsf{Wel}[w_0']$ is only set to a detached root $\mathsf{root}_{rt'}$ during a join query. Moreover, by how $\mathcal{S}_{2\text{-}5}$ simulates the join query, at the point when $\mathsf{Wel}[w_0']$ is set, there does not exist a non-root c_0 such that $\mathsf{Node}[c_0] \neq \bot$ and c_0 includes the same confTag as w_0'. Below we consider the timing that $\mathsf{Wel}[w_0]$ is set, which can be either during a commit or a join query.

Let us consider the former case. Notice $\mathsf{Wel}[w_0]$ cannot be set to a detached root during a commit query due to the `*attach` function in the ideal commit protocol. Hence, since *Case (1)* cannot occur, we only consider *Case (2)*, that is, $\mathsf{Wel}[w_0] = c_0$. We have two cases, $\mathsf{Node}[c_0] = \bot$ or not right before $\mathsf{Wel}[w_0] = c_0$ is set. In the former case, due to how $\mathcal{S}_{2\text{-}5}$ simulates the commit query, $\mathsf{Wel}[w_0]$ and $\mathsf{Wel}[w_0']$ are assigned to the same node c_0. Hence, *Case (2)* cannot occur. In the latter case, if $\mathsf{Node}[c_0]$ was already set, then due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}], $\mathsf{Node}[c_0]$ must have been set during a process query. However, due to `*attach` function in the ideal process protocol, if this happends, then $\mathsf{Wel}[w_0']$ will be reattached to c_0. Hence, *Case (2)* cannot occur either. Summarizing so far, $\mathsf{Wel}[w_0]$ cannot be set during a commit query.

Let us consider the latter case. We first consider *Case (1)*, where $\mathsf{Wel}[w_0]$ is set to root_{rt}. By observing how $\mathcal{S}_{2\text{-}5}$ simulates the join query and by our assumption, $\mathsf{Wel}[w_0]$ must be set to $\mathsf{Wel}[w_0']$. Hence, *Case (1)* cannot occur. Next, consider *Case (2)*, where $\mathsf{Wel}[w_0]$ is set to c_0. By how $\mathcal{S}_{2\text{-}5}$ simulates the join query, c_0 must contain the same confTag as w_0' and satisfy $\mathsf{Node}[c_0] \neq \bot$. However, considering that $\mathsf{Node}[c_0]$ is set only during a commit or a process query, it is clear that `*attach` function in the ideal commit or process protocols assigns $\mathsf{Wel}[w_0']$ to c_0, thus contradicting $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$. Hence, *Case (2)* cannot occur either.

Case (3). Observe $\mathsf{Wel}[w_0]$ is only set to a detached root root_{rt} during a join query. Due to how $\mathcal{S}_{2\text{-}5}$ simulates the join query and considering that $\mathsf{Wel}[w_0]$ is not set to a non-root, we must have $\mathsf{Wel}[w_0] = \mathsf{Wel}[w_0']$. However, this is a contradiction. Hence, *Case (3)* cannot occur.

Case (4). Due to how $\mathcal{S}_{2\text{-}5}$ simulates the commit, process, and join queries and by the definition of the `*attach` function in the ideal commit or process protocols, the confTag included in w_0, c_0, w_0', c_0' are identical, where we also use the fact that w_0 and w_0' include the same confTag. Moreover, due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}], we must have $c_0 = c_0'$ if they include the same confTag (and if $\mathcal{S}_{2\text{-}5}$ does not abort). However, this is a contradiction. Hence, *Case (4)* cannot occur.

This completes the proof. □

Remark 2 (Different welcome messages for the same group). *Ideally, we might want a commit message c_0 to be uniquely bound to a single welcome message w_0 (i.e., if $\mathsf{Wel}[w_0] \neq \bot$, then no other w_0' with the same* confTag *satisfies $\mathsf{Wel}[w_0'] \neq \bot$). However, due to the following concrete attack, Proposition E.6 is the best we can hope for. Namely, users can be added to the same group by different welcome messages. However, Proposition E.6 does guarantee that any different welcome messages provide a consistent view of the group to the invited users (i.e., $\mathsf{Wel}[w_0] = \mathsf{Wel}[w_0']$).*

1. *The adversary $\mathcal{A}$ corrupts party* id *to obtain all secret information and state. It then runs* id *"in the head" using randomness* rand *it generated to obtain (c_0', w_0').*
2. *$\mathcal{A}$ modifies the signature* sig′ *attached to w_0' and creates another valid signature* sig′ *on the same message, and creates a modified but valid welcome message w_0'. (Note that unforgeability does not say anything when the secret signing key* ssk *is leaked).*
3. *$\mathcal{A}$ queries $(\mathtt{Join}, w_0, *)$ on some valid party* id′*. This sets $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$.*
4. *$\mathcal{A}$ queries $(\mathtt{Process}, c_0', *)$ on some valid party* id″*. This sets attaches $\mathsf{Wel}[w_0]$ to c_0'. That is, we now have $\mathsf{Node}[c_0'] \neq \bot$ and $\mathsf{Wel}[w_0] = c_0'$.*
5. *Finally, $\mathcal{A}$ queries party* id *by setting* RandCor[id] = *'bad' and using randomness* rand*. Since $\mathsf{Node}[c_0'] \neq \bot$, $\mathsf{Wel}[w_0']$ is newly created and set to c_0'. Namely, we now have $\mathsf{Wel}[w_0'] = c_0'$.*

Next, we provide a proposition that informally states that if some party id used w_0 to join a group and $\mathsf{Wel}[w_0]$ is assigned to a detached root, then a $\mathsf{Node}[c_0]$ with the same confTag as w_0 cannot yet exist in the history graph. In other words, if such $\mathsf{Node}[c_0]$ exists, then any w_0 with the same confTag should be assigned to c_0, i.e., $\mathsf{Wel}[w_0] = c_0$.

Proposition E.7 (Consistency of confTag in commit and welcome messages). *If $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$, then any non-root c_0 such that $\mathsf{Node}[c_0] \neq \bot$ does not include the same* confTag *as w_0.*

Proof. The statement can be equally stated as, if $\mathsf{Node}[c_0] \neq \bot$, then any w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$ does not include the same confTag as c_0. Observe that the only place $\mathsf{Node}[c_0]$ for a non-root c_0 is set is either during a commit or process query. We provide the proof considering these two cases individually.

First, assume $\mathsf{Node}[c_0]$ for a non-root c_0 is set during a commit query. This means that some party id was invoked by $\mathcal{S}_{2\text{-}5}$ and output some $(c_0, \vec{c}, w_0', \vec{w}')$. There are two cases to consider: $w_0' = \bot$ or $w_0' \neq \bot$. In the former case, due to the modification we made in Hybrid 2-5 [*Consistency of no-join*], there cannot exist any w_0 that includes the same confTag as c_0 but $\mathsf{Wel}[w_0] \neq \bot$. Therefore, the statement holds as desired. In the latter case, we consider two more cases: $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ was set before or after $\mathcal{S}_{2\text{-}5}$ invoked party id. If $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$ was set before $\mathcal{S}_{2\text{-}5}$ invoked party id, then due to how $\mathcal{S}_{2\text{-}5}$ simulates the commit query, any $\mathsf{Wel}[w_0]$ attached to a detached root must have been reattached to c_0 via the `*attach` function in the ideal commit protocol. Namely, there will not exist a $\mathsf{Wel}[w_0]$ that is attached to a detached root so the statement holds as desired. On the other hand, $\mathsf{Wel}[w_0]$ cannot be set to a detached root after $\mathcal{S}_{2\text{-}5}$ invoked party id either. This is because due to how $\mathcal{S}_{2\text{-}5}$ simulates the join query (which is the only place $\mathsf{Wel}[w_0]$ can potentially be set to a detached root), if w_0 and c_0 include the same confTag, then $\mathsf{Wel}[w_0]$ is attached to c_0. Therefore, the statement holds in case $\mathsf{Node}[c_0]$ is set during a commit query.

Next, assume Node$[c_0]$ for a non-root c_0 is set during a process query. This means that some party id was invoked by $\mathcal{S}_{2\text{-}5}$ on input $(\texttt{Process}, c_0, \widehat{c}, \vec{p})$. Again, we consider two cases: Wel$[w_0]$ = root$_{rt}$ was set before or after $\mathcal{S}_{2\text{-}5}$ invoked party id. If Wel$[w_0]$ = root$_{rt}$ was already set before $\mathcal{S}_{2\text{-}5}$ invoked party id, then due to how $\mathcal{S}_{2\text{-}5}$ simulates the process query, any Wel$[w_0]$ attached to a detached root must have been reattached to c_0 via the *attach function in the ideal process protocol. Hence, the statement holds as desired. The other case when Wel$[w_0]$ was not set to a detached root before $\mathcal{S}_{2\text{-}5}$ invoked party id is also identical to the above case. Therefore, the statement holds in case Node$[c_0]$ is set during a process query as well.

This concludes the proof. □

The following provides some useful equivalence relationships between the protocol states and nodes maintained by the history graph. Most of the relations are a simple consequence of the correctness of the real protocol and we provide them mainly for reference. Note that some relationships are not included in the following since we either do not require them or because we need to prove them. Specifically, *Case C* is missing many desirable consistency relation checks such as Node$[c_0]$.orig = id$_c$ and Node$[c_0]$.mem = G.memberIDsvks(). These relations are not simple consequence of the real protocol and will be handled separately below.

Fact 1 (Existence of id in history graph). *Let $G \neq \perp$ and G^{prev} (possibly $\perp$) be the current and previous protocol states*[16] *of party* id *that is internally simulated by $\mathcal{S}_{2\text{-}5}$, respectively. Then, if* Ptr[id] = c_0*, then one of the following three cases hold:*

Case A: [c_0 is the main root root$_0$*]*

- id = id$_{\text{creator}}$;
- $G^{\text{prev}} = \perp$;
- Node$[\text{root}_0]$.orig = id$_{\text{creator}}$;
- Node$[\text{root}_0]$.par = $\perp$;
- Node$[\text{root}_0]$.pro = $\perp$;
- Node$[\text{root}_0]$.mem = G.memberIDsvks()
- G.epoch = 0;
- G.confTransHash-w.o-'id$_c$' = $\perp$;
- G.confTransHash = $\perp$;
- G.groupCont() = (G.groupid, G.epoch, G.memberHash, G.confTransHash);
- G.confKey = H(G.joinerSecret, G.groupCont(), 'conf');
- G.membKey = H(G.joinerSecret, G.groupCont(), 'memb');
- G.appSecret = H(G.joinerSecret, G.groupCont(), 'app');
- G.interimTransHash = H(G.confTransHash, confTag).

Case B: [c_0 is a detached root (i.e., c_0 = root$_{rt}$ for some $rt \in \mathbb{N}$)] There exists a w_0 of the form ((groupid, epoch, memberPublicInfo, memberHash, confTransHash-w.o-'id$_c$', confTransHash, interimTransHash, confTag, id$_c$), T, sig) *such that*

- Wel$[w_0]$ = root$_{rt}$;
- $G^{\text{prev}} = \perp$;
- Node$[\text{root}_{rt}]$.orig = id$_c$;
- Node$[\text{root}_{rt}]$.par = $\perp$;
- Node$[\text{root}_{rt}]$.pro = $\perp$;
- Node$[\text{root}_{rt}]$.mem = G.memberIDsvks();
- G.groupid = groupid;
- G.epoch = epoch;
- G.memberPublicInfo() = memberPublicInfo;
- G.memberHash = *derive-member-hash(G);
- G.memberHash = memberHash;
- G.confTransHash-w.o-'id$_c$' = confTransHash-w.o-'id$_c$';
- G.confTransHash = confTransHash;
- G.confTransHash = H(G.confTransHash-w.o-'id$_c$', id$_c$);
- G.interimTransHash = interimTransHash;
- G.groupCont() = (G.groupid, G.epoch, G.memberHash, G.confTransHash);
- G.confKey = H(G.joinerSecret, G.groupCont(), 'conf');
- G.membKey = H(G.joinerSecret, G.groupCont(), 'memb');
- G.appSecret = H(G.joinerSecret, G.groupCont(), 'app');
- confTag = MAC.TagGen(G.confKey, G.confTransHash);
- G.confTag = confTag;
- G.interimTransHash = H(G.confTransHash, confTag).

Moreover, all such w_0 agrees on every entry expect for (T, sig)*. In particular, all such w_0 includes the same* confTag.

Case C: [c_0 is a non-root (i.e., c_0 = (groupid, epoch, id$_c$, 'commit', C_0 = (propIDs, kp, T), sig, confTag))]

- *for all $p \in \vec{p}$* = Node$[c_0]$.pro, *p is of the form* (groupid, epoch, id$_s$, 'proposal', P, sig′, membTag) *and satisfies*
  - G.groupid = groupid;
  - G^{prev}.epoch = epoch;
  - membTag = MAC.TagGen(G^{prev}.membKey, (G^{prev}.groupCont(), id$_s$, 'proposal', P, sig′));
  - G.membTags = $(\text{membTag})_{\text{membTag } \textit{included in } p \in \vec{p}}$;
  - propIDs = $(\text{H}(p))_{p \in \vec{p}}$.
- G.groupid = G^{prev}.groupid;
- G.epoch = G^{prev}.epoch + 1 = epoch + 1;
- G.memberHash = *derive-member-hash(G);
- G.confTransHash-w.o-'id$_c$' = H(G^{prev}.interimTransHash, (groupid, epoch, 'commit', C_0, sig));
- G.confTransHash = H(G.confTransHash-w.o-'id$_c$', id$_c$);
- G.joinerSecret = H(G^{prev}.initSecret, G.comSecret);
- G.groupCont() = (G.groupid, G.epoch, G.memberHash, G.confTransHash);
- G.confKey = H(G.joinerSecret, G.groupCont(), 'conf');
- G.membKey = H(G.joinerSecret, G.groupCont(), 'memb');
- G.appSecret = H(G.joinerSecret, G.groupCont(), 'app');
- confTag = MAC.TagGen(G.confKey, G.confTransHash);
- G.confTag = confTag;
- G.interimTransHash = H(G.confTransHash, confTag).

Proof. All the relations in *Case A* and *Case C* are a consequence of the correctness of the real protocol. For the latter case, observe that Node$[c_0]$.pro is only set during a commit or a process query. All the relations in *Case B* that do not concern Node$[*]$ is also a consequence of the correctness of the real protocol.

We check the remaining conditions for *Case B*. First, observe that the only place Node$[\text{root}_{rt}]$ for some $rt \in \mathbb{N}$ is set is during a join query when $\mathcal{S}_{2\text{-}5}$ returns (ack := true, $\perp$, orig′ := id$_c$, mem′ := mem) to $\mathcal{F}_{\text{CGKA,2-5}}$. Here, id$_c$ is those included in w_0. Then by

[16] We assume the state is incremented (i.e., move from G^{prev} to G) when processing either a commitment or a welcome message. Therefore, even though the state is updated after a commit in a strict sense, we view them as the same "current" state for simplicity.

the *create-root function in the ideal join protocol, we have $\mathsf{Node}[\mathsf{root}_{rt}] = \mathsf{id}_c$ as desired. Moreover, observing that every entry expect for $(\mathsf{T}, \mathsf{sig})$ in the welcome message w_0 is used to derive interimTransHash, the uniqueness of the remaining entries are guaranteed due to the modification we made in Hybrid 2-1 [*No collision in RO*]. □

The following is the main proposition of *Part 2-1*. It shows that two parties are assigned to the same node in the history graph if and only if they agree on the same group secrets. This allows us to relate the consistency of history graph and protocol states.

Proposition E.8 (Consistency of protocol secrets and history graph). *Let* id *and* id′ *be two parties such that* $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ *and* $\mathsf{Ptr}[\mathsf{id}'] \neq \bot$*, and let* G_{id} *and* $G_{\mathsf{id}'}$ *be their protocol states. Here,* id *and* id′ *may be the same party from different epochs. Then, we have* $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$ *if and only if either one of the following conditions hold:*

- $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$*;*
- $G_{\mathsf{id}}.\mathsf{membKey} = G_{\mathsf{id}'}.\mathsf{membKey}$*;*
- $G_{\mathsf{id}}.\mathsf{appSecret} = G_{\mathsf{id}'}.\mathsf{appSecret}$.

Proof. Let us first show the "if" direction of the statement. We only show the case $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$ as the other cases can be proven identically. The proof heavily relies on the equality relations provided in Fact 1. First, since we can assume there is no collision in the hash function H due to the modification we made in Hybrid 2-1 [*No collision in RO*], we have $G_{\mathsf{id}}.\mathsf{confTransHash} = G_{\mathsf{id}'}.\mathsf{confTransHash}$ (which are included in groupCont()). Then, this implies that $G_{\mathsf{id}}.\mathsf{confTag} = G_{\mathsf{id}'}.\mathsf{confTag}$. In case $\mathsf{Ptr}[\mathsf{id}]$ and $\mathsf{Ptr}[\mathsf{id}']$ are both non-roots, then this implies that $\mathsf{Ptr}[\mathsf{id}]$ and $\mathsf{Ptr}[\mathsf{id}']$ both include the same confTag. Hence, by the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com} *and* L_{wel}], we have $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$.

Now, let us consider the case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$. Then, since $\mathsf{Ptr}[\mathsf{id}]$ is assigned to a detached root only during a join query, there must exist w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$, where w_0 includes $G_{\mathsf{id}}.\mathsf{confTag}$ by the correctness of the protocol. Due to Proposition E.7, there does not exist a non-root c_0 such that $\mathsf{Node}[c_0] \neq \bot$ but c_0 includes $G_{\mathsf{id}}.\mathsf{confTag} = G_{\mathsf{id}'}.\mathsf{confTag}$. This implies that we must have $\mathsf{Ptr}[\mathsf{id}'] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$. Then, by Proposition E.6, we have $rt = rt'$ since there cannot exist two w_0 and w_0' such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt}$, $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt'}$, and $rt \neq rt'$ that include the same confTag. Therefore, we also have $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$ when they are assigned to detached roots.

It remains to show the "only if" direction of the statement. Again, we only show the case $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$ as the other cases can be proven identically. Assume $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$. In case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_0$, then $\mathsf{id} = \mathsf{id}' = \mathsf{id}_{\mathsf{creator}}$. Therefore, the statement holds trivially. The case $\mathsf{Ptr}[\mathsf{id}] = c_0$ for some non-root c_0 holds as a direct consequence of Fact 1. Finally, in case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$, then by Fact 1, any w_0 that satisfy the relations provide in *Case B* produce the same confTag. Then due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{wel}], we must have $G_{\mathsf{id}}.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$. The case $\mathsf{Ptr}[\mathsf{id}] = c_0$ for some non-root c_0 can be checked similarly to the case $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$.

This completes the proof. □

Remark 3 (Implication of $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$). *Proposition E.8 only focuses on the group secrets since we wanted an "if and only if" statement. However, if we only cared about the "only if" direction, there is much more we can deduce from* $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$*. Namely, following the "only if" direction of the proof of Proposition E.8 and the proof of Lem. E.3 to move from Hybrid 2-1 to 2-2, we can conclude that if two parties* id *and* id′ *satisfy* $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$*, then they agree on the same view of the group such as* $G_{\mathsf{id}}.\mathsf{groupid} = G_{\mathsf{id}'}.\mathsf{groupid}$ *and* $G_{\mathsf{id}}.\mathsf{memberIDsvks}() = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$ *as expected. We use this implication in Proposition E.11.*

Part 2-2. Consistency of proposal messages. The following proposition establishes that if a party outputs or receives a proposal p that already exists in the history graph (i.e., $\mathsf{Prop}[p] \neq \bot$), then it satisfies all the intuitive consistency checks.

Proposition E.9 (Consistency of existing proposal node). *Assume party* id *such that* $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ *outputs* p *on input* $(\mathsf{Propose}, \mathsf{act})$ *from* $\mathcal{S}_{2\text{-}5}$*, where* p *is of the form* $(\mathsf{groupid}, \mathsf{epoch}, \mathsf{id}, \text{'proposal'}, P, \mathsf{sig}, \mathsf{membTag})$*. Then, if* $\mathsf{Prop}[p] \neq \bot$*, we have the following (after* $\mathcal{S}_{2\text{-}5}$ *receives an output from* id *but before it provides input to* $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$*):*

- $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$*;*
- $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id}$*;*
- $\mathsf{Prop}[p].\mathsf{act} =$ 'upd'-svk *if* $P = (\text{'upd'}, \mathsf{kp})$*;* $\mathsf{Prop}[p].\mathsf{act} =$ 'add'-id_t-svk_t *if* $P = (\text{'add'}, \mathsf{kp}_t)$*; or* $\mathsf{Prop}[p].\mathsf{act} =$ 'rem'-id_t *if* $P = (\text{'rem'}, \mathsf{id}_t)$*, where* svk *and* svk_t *are included in* kp *and* kp_t*, respectively.*

Additionally, consider the following two cases:

- id *outputs* $(c_0, \vec{c}, w_0, \vec{w})$ *on input* $(\mathtt{Commit}, \vec{p}, \mathsf{svk})$ *from* $\mathcal{S}_{2\text{-}5}$*; or*
- id *outputs* $(\mathsf{id}_c, upd \| rem \| add)$ *on input* $(\mathtt{Process}, c_0, \vec{c}, \vec{p})$ *from* $\mathcal{S}_{2\text{-}5}$.

For these two cases, we have the following (after $\mathcal{S}_{2\text{-}5}$ *receives an output from* id *but before it provides input to* $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$*):*

- *for all* $p \in \vec{p}$*,* p *is of the form* $(\mathsf{groupid}, \mathsf{epoch}, \mathsf{id}_s, \text{'proposal'}, P, \mathsf{sig}, \mathsf{membTag})$ *and we have the following if* $\mathsf{Prop}[p] \neq \bot$*:*
  - $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$*;*
  - $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id}_s$*;*
  - $\mathsf{Prop}[p].\mathsf{act} =$ 'upd'-svk *if* $P = (\text{'upd'}, \mathsf{kp})$*;* $\mathsf{Prop}[p].\mathsf{act} =$ 'add'-id_t-svk_t *if* $P = (\text{'add'}, \mathsf{kp}_t)$*; or* $\mathsf{Prop}[p].\mathsf{act} =$ 'rem'-id_t *if* $P = (\text{'rem'}, \mathsf{id}_t)$*, where* svk *and* svk_t *are included in* kp *and* kp_t*, respectively.*

Proof. We first consider the former case where id executes a propose protocol. Let G_{id} be the protocol state of id. There are two places where $\mathsf{Prop}[p]$ can be set. One is during a propose query and the other is during *fill-prop which is invoked during a commit or process query. Assume $\mathsf{Prop}[p]$ is set during a propose query. Then, there exists some party id′ that was invoked by $\mathcal{S}_{2\text{-}5}$ for a propose query that output p. Since p includes id, we have id′ = id and the condition regarding $\mathsf{Prop}[p].\mathsf{act}$ holds by the correctness of the real propose protocol. Moreover, due to modification we made in Hybrid 2-2 [*Unique* membTag *in* L_{prop}], the protocol state G'_{id} of id when it outputs p the first time must satisfy $G_{\mathsf{id}}.\mathsf{membKey} = G'_{\mathsf{id}}.\mathsf{membKey}$. Then due to Proposition E.8, we have $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$ as desired. On the

other hand, assume Prop[p] is set during *fill-prop which is invoked during a commit or process query. Let the party being invoked be id′. By how $\mathcal{S}_{2\text{-}5}$ responds to *fill-prop it is clear that Prop[p].orig = id and the condition regarding Prop[p].act hold. Moreover, due to modification we made in Hybrid 2-2 [*Unique* membTag *in* L_{prop}], the protocol state $G_{\text{id}'}$ of party id′ that accepts p must satisfy G_{id}.membKey = $G_{\text{id}'}$.membKey. Then due to Proposition E.8, we have Ptr[id] = Ptr[id′]. Therefore, by the definition of *create-prop run within *fill-prop, we have Prop[p].par = Ptr[id] as desired.

The latter cases where id executes either a commit of process query consist of the exact same argument as above. Therefore, this completes the proof. □

Part 2-3. Consistency of commit messages. This is the final and most important part of the basic consistency checks. Unlike proposal messages, consistency of commit messages is proven by induction. Informally, this is because to conclude the current commit node is consistent, we must rely on the fact that the previous commit node is also consistent. We first show that if the current commit node id is assigned to agree with the group member as those included in the protocol state (i.e., G.memberIDsvks() = Node[c_0].mem), then the next commit node and the updated protocol state agrees on the group member of the next epoch.

Proposition E.10 (Consistency of commit and process protocol). *Assume party* id *and* c_0' *such that* Ptr[id] = c_0' *and* Node[c_0'] ≠ ⊥. *Let G be* id*'s protocol state and assume we have G*.memberIDsvks() = Node[c_0'].mem. *Consider the following two cases:*

- id *outputs* $(c_0, \vec{c}, w_0, \vec{w})$ *on input* (Commit, $\vec{p}$, svk) *from* $\mathcal{S}_{2\text{-}5}$*; or*
- id *outputs* $(\text{id}_c, upd \| rem \| add)$ *on input* (Process, $c_0, \widehat{c}, \vec{p}$) *from* $\mathcal{S}_{2\text{-}5}$.

After receiving the output from id, $\mathcal{S}_{2\text{-}5}$ *continues the simulation by providing input to* $\mathcal{F}_{\text{CGKA},2\text{-}5}$. *If the ideal commit or process protocols terminate without halting or outputting* ⊥*, then we have* G'.memberIDsvks() = Node[c_0].mem *in both cases, where* G' *is the new group state included in* G.pendCom[c_0] *in case of a commit query or the updated group state in case of a process query.*

Proof. Let us first consider the case id is invoked on a commit query. Condition on the ideal functionality not halting or outputting ⊥, we are guaranteed that the function *members on line 6 of the ideal commit protocol terminates as expected. In particular, since *members runs syntactically the same procedure as *apply-props on line 3 of the real commit protocol, we have G'.memberIDsvks() = mem if G.memberIDsvks() = Node[c_0'].mem, where mem is the outputs of *members. Now, if Node[c_0] is created via *create-child (i.e., Node[c_0] = ⊥ ∧ rt = ⊥), then we have Node[c_0].mem = mem as desired. Otherwise, if *consistent-com and *attach do not halt nor output ⊥, we have Node[c_0].mem = mem as desired in case Node[c_0] ≠ ⊥ or Node[c_0] = ⊥ ∧ rt ≠ ⊥. This completes the proof in case of a commit query.

Let us now consider the case id is invoked on a process query. Following the same argument as above, in case Node[c_0] is created via *create-child (i.e., Node[c_0] = ⊥ ∧ rt = ⊥), we have Node[c_0].mem = mem as desired conditioned on the ideal functionality not halting or outputting ⊥. Otherwise, if *valid-successor and *attach do not halt nor output ⊥, we have Node[c_0].mem = mem as desired in case Node[c_0] ≠ ⊥ or Node[c_0] = ⊥ ∧ rt ≠ ⊥. This completes the proof in case of a join query. □

We next show that if a party id is assigned to some commit node in the history graph, then the group member stored on that commit node should be consistent with the members stored in the protocol state. The proof is by induction where the base case is guaranteed by Fact 1 and we use the previous Proposition E.10 to move up the epoch. Specifically, any party is first assigned to a root in the beginning (*Case A* or *B* in Fact 1), and in this case, the commit node and protocol state are guaranteed to store the same group members.

Proposition E.11 (Consistency of current commit node). *Assume party* id *and a non-root* c_0 *of the form* (groupid, epoch, id_c, 'commit', C_0 = (propIDs, kp, T), sig, confTag) *such that* Ptr[id] = c_0 *and* Node[c_0] ≠ ⊥. *Let G be the protocol state of* id. *Then we have the following:*

- Node[c_0].orig = id_c*;*
- Node[c_0].mem = G.memberIDsvks().

Proof. We first prove the relation Node[c_0].orig = id_c. Observe Node[c_0] is set either during a commit or a process query. Below, we only consider the case Node[c_0] is set during a commit query since the case for a process query is almost identical. There are further two cases to consider: Node[c_0] was initially ⊥ and $\mathcal{S}_{2\text{-}5}$ outputs rt = ⊥ or Node[c_0] was initially ⊥ and $\mathcal{S}_{2\text{-}5}$ outputs $rt \in \mathbb{N}$. In the former case, due to the *create-child function in the ideal commit protocol, we have Node[c_0].orig = id_c as desired. In the latter case, there exists a detached root root_{rt} and a welcome message w_0 that includes the same confTag as c_0 such that Wel[w_0] = root_{rt}. First, by how $\mathcal{S}_{2\text{-}5}$ simulates the join query, we have Node[root_{rt}].orig = id_c', where id_c' is included in groupInfo of the welcome message w_0. Next, due to the *attach function in the ideal commit protocol, we have Node[c_0].orig = Node[root_{rt}].orig = id_c'. Finally, due to Hybrid 2-2 [*Unique* confTag *in* L_{com} *and* L_{wel}], we have id_c' = id_c. Therefore, if Node[c_0] is set during a commit query, then we have Node[c_0].orig = id_c as desired.

So far we established the first part of the statement: Node[c_0].orig = id_c. It remains to prove the second part of the statement: Node[c_0].mem = G.memberIDsvks(). Below, we prove by contradiction. Assume we have Ptr[id] = c_0 and Node[c_0] ≠ ⊥ for a non-root c_0 but Node[c_0].mem ≠ G.memberIDsvks(). Observe that Ptr[id] is assigned to a new value only during a process or a join query; Ptr[id] remains the same during a process or a commit query. Let us consider the latter case where Ptr[id] = c_0 is assigned during join query, that is, id is invoked by $\mathcal{S}_{2\text{-}5}$ on input $(w_0, \widehat{w})$. We show that this case boils down to checking the former case. Since c_0 is a non-detached root by the assumption in the statement and by how $\mathcal{S}_{2\text{-}5}$ simulates the join query, Node[c_0] ≠ ⊥ and c_0 includes the same confTag as w_0. Since Node[c_0] is set only during a commit or a process query, this implies that there is some party id′ that outputs (resp. inputs) c_0 during a commit (resp. process) query. In case id′ is invoked on a process query, then Ptr[id′] = c_0 due to the ideal process function. Then, due to Rem. 3, if G.memberIDsvks() = G'.memberIDsvks(), where G and G' are the protocol states of id and id′, respectively. Specifically, it suffices

to check that $\mathsf{Node}[c_0].\mathsf{mem} \neq G'.\mathsf{memberIDsvks}()$ cannot happen during a process query. The case id' is invoked on a commit query is handled in the same way.

It remains to consider the former case where $\mathsf{Ptr}[\mathsf{id}] = c_0$ is assigned during process query. Then, taking the contrapositive of Proposition E.10, since $\mathsf{Node}[c_0].\mathsf{mem} \neq G.\mathsf{memberIDsvks}()$, we must have $\mathsf{Node}[c_0'].\mathsf{mem} \neq G'.\mathsf{memberIDsvks}()$. We can iteratively apply this argument till we reach a point that $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$ for some $rt \in \{0\} \cup \mathbb{N}$. This is because any party is initially assigned to either a root or non-root via the join query (or to root_0 by default if $\mathsf{id} = \mathsf{id}_{\mathsf{creator}}$), and in case we arrive at a non-root, then by the above argument, we can focus on the commit or process query that generated the non-root and repeat the same argument till we reach a root. Finally, if G'' is the protocol state of id when $\mathsf{Ptr}[\mathsf{id}] = \mathsf{root}_{rt}$, then we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} \neq G''.\mathsf{memberIDsvks}()$. However, by Fact 1, we must have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G''.\mathsf{memberIDsvks}()$. Therefore, this is a contradiction. This establishes the second part of the statement.

This completes the proof. □

Finally, the following proposition is an analog of Proposition E.9 regarding the consistency check of existing proposal nodes. Specifically, the following establishes that if a party outputs or receives a commit c_0 that already exists in the history graph (i.e., $\mathsf{Node}[c_0] \neq \bot$), then it satisfies intuitive consistency checks.

Proposition E.12 (Consistency of existing commit node). *Let party* id *satisfy* $\mathsf{Ptr}[\mathsf{id}] \neq \bot$ *and consider the following two cases:*

- id *outputs* $(c_0, \vec{c}, w_0, \vec{w})$ *on input* $(\mathtt{Commit}, \vec{p}, \mathsf{svk})$ *from* $\mathcal{S}_{2\text{-}5}$*; or*
- id *outputs* $(\mathsf{id}_c, upd \| rem \| add)$ *on input* $(\mathtt{Process}, c_0, \widehat{c}, \vec{p})$ *from* $\mathcal{S}_{2\text{-}5}$.

Let G' *be either the protocol state included in* $G.\mathsf{pendCom}[c_0]$ *of* id *after executing a commit protocol or the updated protocol state of* id *after executing the process protocol Then, we have the following (after* $\mathcal{S}_{2\text{-}5}$ *receives an output from* id *but before it provides input to* $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$*):*

- *if* $\mathsf{Node}[c_0] \neq \bot$*, then*
  - $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}$ *(resp.* id_c*) if* id *executed a commit (resp. process) protocol;*
  - $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$*;*
  - $\mathsf{Node}[c_0].\mathsf{pro} = \vec{p}$*;*
  - $\mathsf{Node}[c_0].\mathsf{mem} = G'.\mathsf{memberIDsvks}()$.
- *if* c_0 *is attached to a detached root* root_{rt} *(i.e., either* $\mathsf{Node}[c_0] = \bot$ *and* $\mathcal{S}_{2\text{-}5}$ *outputs* $(ack := \mathtt{true}, rt \neq \bot, c_0, \vec{c}, w_0, \vec{w})$ *if* id *executed a commit protocol; or* $\mathsf{Node}[c_0] = \bot$ *and* $\mathcal{S}_{2\text{-}5}$ *outputs* $(ack := \mathtt{true}, rt \neq \bot, \bot, \bot)$ *if* id *executed a process protocol), then*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}$ *(resp.* id_c*) if* id *executed a commit (resp. process) protocol;*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} = \bot$*;*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{pro} = \bot$*;*
  - $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G'.\mathsf{memberIDsvks}()$.

Proof. We first prove the simpler second case where $\mathsf{Node}[c_0] = \bot$ and c_0 is assigned to a detached root. Notice that $\mathsf{Node}[\mathsf{root}_{rt}]$ is only created during a join query. Let w_0' be the associating welcome message that is used to create $\mathsf{Node}[\mathsf{root}_{rt}]$ and assume party id' was invoked by this join query. Let $G_{\mathsf{id}'}$ be the protocol state after id' processes the welcome message. Then, by *Case (B)* of Fact 1, we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}_c'$ and $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$, where id_c' is those included in w_0'. By how $\mathcal{S}_{2\text{-}5}$ simulates the commit and process queries, if id was invoked on a commit or a process query, then c_0 includes the same confTag as w_0'. Then, due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com} *and* L_{wel}], we have $G'.\mathsf{confKey} = G_{\mathsf{id}'}.\mathsf{confKey}$ and $G'.\mathsf{confTransHash} = G_{\mathsf{id}'}.\mathsf{confTransHash}$. Then, by Fact 1 and due to the modification we made in Hybrid 2-1 [*No collision in RO*], we also have $G'.\mathsf{memberHash} = G_{\mathsf{id}'}.\mathsf{memberHash}$ and $\mathsf{id} = \mathsf{id}_c'$ (resp. $\mathsf{id}_c = \mathsf{id}_c'$) if id is invoked on a commit (resp. process) query. Finally, by the definition of `*derive-member-hash` (see Fig. 25), we have $G'.\mathsf{memberIDsvks}() = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$. Since we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} = \mathsf{Node}[\mathsf{root}_{rt}].\mathsf{pro} = \bot$ by definition, this concludes the proof for the second case where c_0 is assigned to a detached root.

It remains to prove the first case where $\mathsf{Node}[c_0] \neq \bot$. There are four cases where $\mathsf{Node}[c_0]$ can be created when c_0 is a non-root.

(1) Some party id' output $(c_0, \vec{c}_0', w_0', \vec{w}_0')$ on input $(\mathtt{Commit}, \vec{p}', \mathsf{svk}')$ from $\mathcal{S}_{2\text{-}5}$ and there does not exist any w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for any $rt' \in \mathbb{N}$ that includes the same confTag as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$);
(2) Some party id' output $(c_0, \vec{c}_0', w_0', \vec{w}_0')$ on input $(\mathtt{Commit}, \vec{p}', \mathsf{svk}')$ from $\mathcal{S}_{2\text{-}5}$ and there exists a w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ that includes the same confTag as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$);
(3) Some party id' output $(\mathsf{id}_c', upd' \| rem' \| add')$ on input $(\mathtt{Process}, c_0, \widehat{c}', \vec{p}')$ from $\mathcal{S}_{2\text{-}5}$ and there does not exist any w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for any $rt' \in \mathbb{N}$ that includes the same confTag as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$);
(4) Some party id' output $(\mathsf{id}_c', upd' \| rem' \| add')$ on input $(\mathtt{Process}, c_0, \widehat{c}', \vec{p}')$ from $\mathcal{S}_{2\text{-}5}$ and there exists a w_0 such that $\mathsf{Wel}[w_0] = \mathsf{root}_{rt'}$ for some $rt' \in \mathbb{N}$ that includes the same confTag as c_0 (before $\mathcal{S}_{2\text{-}5}$ provides input to $\mathcal{F}_{\mathsf{CGKA},2\text{-}5}$).

Since the proof for the latter two cases are almost identical to the former two cases we only prove Cases (1) and (2).

For both cases, let $G'_{\mathsf{id}'}$ be the pending protocol state included in $G_{\mathsf{id}'}.\mathsf{pendCom}[c_0]$ of the protocol state $G_{\mathsf{id}'}$ of id' after executing the commit query. Then, by the correctness of the ideal commit protocol, we have $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}'$, $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}']$, $\mathsf{Node}[c_0].\mathsf{pro} = \vec{p}'$, and $\mathsf{Node}[c_0].\mathsf{mem} = G_{\mathsf{id}'}.\mathsf{memberIDsvks}()$, where the last condition holds due to Proposition E.10. Note that we have $\mathsf{Node}[c_0].\mathsf{par}$ and $\mathsf{Node}[c_0].\mathsf{pro}$ in Case (2) due to the `*consistent-com` and `*attach` functions in the ideal commit protocol.

Now, since id' and id output the same c_0, and c_0 includes id, we have $\mathsf{id} = \mathsf{id}'$. Hence, $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}$. Moreover, by the modification in Hybrid 2-2 [*Unique* confTag *in* L_{com}], we have $(G'_{\mathsf{id}'}.\mathsf{confKey}, G'_{\mathsf{id}'}.\mathsf{confTransHash}) = (G'.\mathsf{confKey}, G'.\mathsf{confTransHash})$. Then, by Proposition E.8, since both confKey are the same we have $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}']$. Hence, $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$. Also, due to the modification we made in Hybrid 2-1 [*No collision in RO*] and by the definition of `*derive-member-hash` (see Fig. 25), we also have

$\vec{p}' = \vec{p}$ and G'.memberIDsvks() = $G'_{\mathsf{id}'}$.memberIDsvks(). Therefore, Node[c_0].pro = $\vec{p}$ and Node[c_0].mem = G_{id}.memberIDsvks(). This completes the proof for Cases (1) and (2).

This concludes the proof. □

Combining the propositions in *Part 2*, we obtain the following corollary.

Corollary E.13 (Invariant `cons-invariant`). `cons-invariant` *in Fig. 19 always outputs* `true` *(condition on* $\mathcal{S}_{2\text{-}5}$ *not aborting).*

Proof. The first two conditions (a) and (b) hold due to Fact 1 and Propositions E.9, E.11 and E.12. Moreover, Condition (c) holds due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}] and the fact that attaching detached roots to an existing non-root can not cause a cycle in the history graph. □

Part 3. Analysis of the simulation. We are finally ready to analyze that $\mathcal{S}_{2\text{-}5}$ provides the same view to $\mathcal{Z}$ as in the previous Hybrid 2-4. Below, we only focus on the case $\mathcal{S}_{2\text{-}5}$ receives a non-$\perp$ from the simulated parties. Otherwise, $\mathcal{S}_{2\text{-}5}$ can perfectly simulate the previous hybrid by simply setting *ack* = `false`.

(1) Analysis of Create. It is clear that $\mathcal{F}_{\mathsf{CGKA,2\text{-}5}}$ outputs $\perp$ to $\mathcal{Z}$ if and only if $\mathcal{S}_{2\text{-}4}$ returned $\perp$ to $\mathcal{Z}$ (or to $\mathcal{F}_{\mathsf{CGKA,2\text{-}5}}$ to be more precise) in Hybrid 2-4. Therefore, the view of $\mathcal{Z}$ remains identical in both hybrids.

(2) Analysis of Proposal. If party id returns p to $\mathcal{S}_{2\text{-}5}$, we need to check that $\mathcal{F}_{\mathsf{CGKA,2\text{-}5}}$ also returns p to $\mathcal{Z}$ as in the previous hybrid. We only focus on act = 'upd'-svk since the other cases are just a simplification of this check. We first check that the `*valid-svk` check made by $\mathcal{F}_{\mathsf{CGKA,2\text{-}5}}$ on line 4 of (`Process`, act) in Fig. 15 succeeds. For id to have output p, we need `*fetch-ssk-if-nec`(G, svk) = ssk $\neq \perp$ in the real protocol (see Fig. 22). Within `*fetch-ssk-if-nec` (G, svk), if G.member[G.id].svk $\neq$ svk, then we must have SSK[id, svk] $\neq \perp$ due to the check made by $\mathcal{F}_{\mathsf{AS}}$. This implies that $\mathcal{F}_{\mathsf{AS}}$ run within `*valid-svk` in $\mathcal{F}_{\mathsf{CGKA,2\text{-}5}}$ outputs `true` on input (`has-ssk`, id, svk), and hence, `*valid-svk`(id, svk) also outputs `true`. On the other hand, since we have Node[c_0].mem = G.memberIDsvks() due to Proposition E.11, if G.member[G.id].svk = svk, then we have Node[Ptr[id]].mem[id] = svk. Therefore, `*valid-svk`(id, svk) also outputs `true` in this case as well. Therefore, `*valid-svk`(id, svk) outputs `true` if id did not return $\perp$ in the real protocol.

Finally, due to Proposition E.9, the **assert** condition checked within `*consistent-prop` on line 9 of (`Propose`, act) is not triggered when Prop[p] $\neq \perp$. Therefore, if party id returns p to $\mathcal{S}_{2\text{-}5}$, then $\mathcal{F}_{\mathsf{CGKA,2\text{-}5}}$ also returns p to $\mathcal{Z}$ as specified.

(3) Analysis of Commit. Assume Ptr[id] = c'_0 and let $G \neq \perp$ be the protocol state of the simulated party id before it executes the commit. The check made by `*valid-svk` is the same check we covered in the analysis of proposal (see above *(2)*). We therefore first check the **assert** condition on line 7 is never triggered. To do so, we first establish that the set mem output by `*members` is identical to G'.memberIDs (but possibly ordered differently), where G' is the protocol state generated on line 3 in the real commit protocol. This consists of three checks. First, by Proposition E.11 we have Node[c'_0].mem = G.memberIDsvks(). Therefore, if commit succeeds in the real protocol, then we have (id, $*$) $\in$ Node[c'_0].mem. Next, due to how $\mathcal{S}_{2\text{-}5}$ answers to `*fill-prop` and by Proposition E.9, we have Prop[p] $\neq \perp$ and Prop[p].par = c'_0 for all $p \in \vec{p}$ and the contents of Prop[p] (i.e., Prop[p].orig and Prop[p].act) are consistent with p. Combining the three checks, we are guaranteed that `*members` outputs mem is identical to those created in `*apply-props` in the real protocol. Therefore, this establishes that the **assert** condition mem $\neq \perp$ and (id, svk) $\in$ mem are satisfied.

To finish the remaining analysis, we consider three cases: Node[c_0] = $\perp \wedge$ $rt = \perp$, Node[c_0] = $\perp \wedge$ $rt \in \mathbb{N}$, and Node[c_0] $\neq \perp$. Here, recall that the **assert** condition `cons-invariant` $\wedge$ `auth-invariant` in line 21 of commit is never triggered as they are always set to `true`. This follows from the fact that **sig-inj-allowed** and **mac-inj-allowed** are always set to `true` in this hybrid and `cons-invariant` is always set to `true` due to Corollary E.13.

[*Case 1:* Node[c_0] = $\perp$ and $rt = \perp$] It suffices to show that Wel[w_0] = $\perp$ when $w_0 \neq \perp$ (see line 11 in ideal commit protocol). Let us prove by contradiction and assume Wel[w_0] $\neq \perp$. First, by how $\mathcal{S}_{2\text{-}5}$ simulates the commit query (see **(3)** of *Part 1*), there does not exist w'_0 such that Wel[w'_0] = root$_{rt'}$ for some $rt' \in \mathbb{N}$ and w'_0 includes the same confTag as w_0. This means, Wel[w_0] = c''_0 for some non-root $c''_0 \neq c_0$ such that Node[c''_0] $\neq \perp$. Since Wel[w_0] is assigned a non-root value only during a commit query, we must have that some id$'$ (possibly id) was invoked by $\mathcal{S}_{2\text{-}5}$ and output c''_0 and w_0. Due to correctness of the protocol, c''_0 and w_0 must include the same confTag. Similarly, c_0 and w_0 must also include the same confTag. However, due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}], the simulator $\mathcal{S}_{2\text{-}5}$ aborts the simulation as in the prior hybrid. Therefore, $\mathcal{S}_{2\text{-}5}$ provides the same view to $\mathcal{Z}$ as in the prior hybrid.

[*Case 2:* Node[c_0] = $\perp$ and $rt \neq \perp$] It suffices to verify that the checks run within `*consistent-com` are satisfied and Wel[w_0] $\in \{\perp, c_0\}$ when $w_0 \neq \perp$ (see line 11 in ideal commit protocol). Let us consider the former check. Due to Proposition E.12, the only check within `*consistent-com` that we need to verify is whether we have RandCor[id] = 'bad'. Here, note that the condition Node[c_0].mem = mem is satisfied since we established mem = G'.memberIDsvks() above. Now, due to the modification we made in Hybrid 2-4 [*Unique* c_0 *with good randomness*] , unless $\mathcal{S}_{2\text{-}5}$ runs party id on the same randomness, we must have Node[c_0] = $\perp$ since every c_0 output by the parties include a unique confTag. Hence, we must have RandCor[id] = 'bad' as desired and all the checks run within `*consistent-com` are satisfied. Finally, it is clear that `*attach` on line 17 of the commit procedure assigns c_0 to Wel[w_0]. Therefore, the latter check on Wel[w_0] = c_0 is also satisfied.

[*Case 3:* Node[c_0] $\neq \perp$] It suffices to verify that the checks run within `*consistent-com` are satisfied and Wel[w_0] $\in \{\perp, c_0\}$ when $w_0 \neq \perp$ (see line 11 in ideal commit protocol). Since the check regarding `*consistent-com` is identical to the above *Case 2*, we only consider the latter check. Assume for the sake of contradiction that Wel[w_0] = $c''_0 \neq c_0$ for $c''_0 \neq \perp$ and Node[c''_0] $\neq \perp$. Observe the only situation the value of Wel[w_0] is set is either during a commit query or a join query. We first consider the case Wel[w_0] is set during a commit query. If this case occurs, then this implies that some id$'$ output (c''_0, w_0) as otherwise the **assert** condition regarding Wel[w_0] in the commit procedure is triggered.

Due to the correctness of the real protocol, all c_0, c_0'', and w_0 include the same confTag. However, due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}], we must have $c_0'' = c_0$ or otherwise the simulator $\mathcal{S}_{2\text{-}5}$ aborts the simulation as in the previous hybrid. Hence, we have $\mathsf{Wel}[w_0] = c_0$ as desired.

Let us consider the other case where $\mathsf{Wel}[w_0]$ is set during a join query, which implies that some id′ output c_0''. We have two cases to consider: c_0'' is a non-root or a detached root. If c_0'' is a non-root, then due to how $\mathcal{S}_{2\text{-}5}$ simulates the join query (see **(5)** of *Part 1*), this implies that c_0'' includes the same confTag as the one included in w_0 and we have $\mathsf{Node}[c_0''] \neq \bot$ when answering the join query. Recall that when c_0'' is a non-root, $\mathsf{Node}[c_0'']$ is set only during a commit or process query. Then, due to the modification we made in Hybrid 2-2 [*Unique* confTag *in* L_{com}], since c_0'' and c_0 include the same confTag, we must have $c_0'' = c_0$ as desired or otherwise the simulator $\mathcal{S}_{2\text{-}5}$ aborts the simulation as in the previous hybrid. On the other hand, if c_0'' is a detached root, then `*attach` on line 17 of the commit procedure assigns c_0 to $\mathsf{Wel}[w_0]$.

Collecting all the checks, we have either $\mathsf{Wel}[w_0] = \bot$ or $\mathsf{Wel}[w_0] = c_0$ as desired. Therefore, $\mathcal{S}_{2\text{-}5}$ provides the same view to $\mathcal{Z}$ as in the prior hybrid.

(4) Analysis of Process. Let $G \neq \bot$ be the protocol state of the simulated party id after it executes the process protocol. Moreover, assume id outputs $(\mathsf{id}_c, \mathit{upd} \| \mathit{rem} \| \mathit{add})$ on input $(\mathtt{Process}, c_0, \widehat{c}, \vec{p})$. There are three cases that can occur while $\mathcal{S}_{2\text{-}5}$ answers the process query (see **(4)** of *Part 1*): *Case 1:* $\mathsf{Node}[c_0] = \bot$ and $rt = \bot$; *Case 2:* $\mathsf{Node}[c_0] = \bot$ and $rt \neq \bot$; and *Case 3:* $\mathsf{Node}[c_0] \neq \bot$. We analyze each cases separately.

[*Case 1:* $\mathsf{Node}[c_0] = \bot$ and $rt = \bot$] Following the same argument we made for analyzing the commit query (see *(3)* above), `*members` on line 6 of the ideal process protocol outputs (mem, propSem), where mem is identical to those created in `*apply-props` in the real protocol and propSem is identical to $\mathit{upd} \| \mathit{rem} \| \mathit{add}$ output by id. This implies that the **assert** conditions on line 7 and line 20 of the ideal process protocol are never triggered. Finally, in *Case 1*, $\mathcal{S}_{2\text{-}5}$ sets $\mathsf{orig}' = \mathsf{id}_c$, where id_c is those output by id, so we conclude that `*output-proc`(c_0) outputs $(\mathsf{id}_c, \mathsf{propSem})$ as in the previous hybrid.

[*Case 2:* $\mathsf{Node}[c_0] = \bot$ and $rt \neq \bot$] Identically to *Case 1*, `*members` on line 6 of the ideal process protocol outputs (mem, propSem), where mem is identical to those created in `*apply-props` in the real protocol and propSem is identical to $\mathit{upd} \| \mathit{rem} \| \mathit{add}$ output by id. Moreover, by Proposition E.12, we have $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig} = \mathsf{id}_c$, $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} = \bot$, $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{pro} = \bot$, and $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem} = G.\mathsf{memberIDsvks}()$. Therefore, the check within the `*valid-successor` and `*attach` functions on line 15 and line 16, respectively, all passes. Hence, `*output-proc`(c_0) outputs $(\mathsf{id}_c, \mathsf{propSem})$ as in the previous hybrid.

[*Case 3:* $\mathsf{Node}[c_0] \neq \bot$] This is almost identical to *Case 2*. The only difference is that by Proposition E.12, we have $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c$, $\mathsf{Node}[c_0].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$, $\mathsf{Node}[c_0].\mathsf{pro} = \vec{p}$, and $\mathsf{Node}[c_0].\mathsf{mem} = G.\mathsf{memberIDsvks}()$. Observe the check within the `*valid-successor` function on line 15 all passes. Hence, `*output-proc`(c_0) outputs $(\mathsf{id}_c, \mathsf{propSem})$ as in the previous hybrid.

(5) Analysis of Join. Let $G \neq \bot$ be the protocol state of the simulated party id after it executes the join protocol. Assume id outputs $(\mathsf{id}_c, G.\mathsf{memberIDsvks}())$ on input $(\mathtt{Join}, w_0, \widehat{w})$. There are four cases that can occur while $\mathcal{S}_{2\text{-}5}$ answers the join query (see **(5)** of *Part 1*): *Case 1:* $\mathsf{Wel}[w_0] \neq \bot$; *Case 2:* $\mathsf{Wel}[w_0] = \bot$ but there exists a unique c_0 including the same confTag as w_0 satisfying $\mathsf{Node}[c_0] \neq \bot$; *Case 3:* $\mathsf{Wel}[w_0] = \bot$ and no such c_0 exists but there exists a (possibly non-unique) w_0' including the same confTag as w_0 satisfying $\mathsf{Wel}[w_0] \neq \bot$; and *Case 4:* $\mathsf{Wel}[w_0] = \bot$ and no such c_0 or w_0' exist. We analyze each cases separately.

[*Case 1:*] In case $\mathsf{Wel}[w_0] = c_0$ already exists, it suffices to consider the case it was initially set. Namely, it suffices to check that $\mathcal{S}_{2\text{-}5}$ simulates the previous hybrid in the below *Cases 2, 3,* and *4.*

[*Case 2:*] $\mathsf{Node}[c_0]$ can be set only during a commit or process query. Due to Propositions E.11 and E.12 and correctness of the real protocol, we have $(\mathsf{id}, *) \in \mathsf{Node}[c_0].\mathsf{mem}$. Therefore, the **assert** condition on line 12 if the ideal join protocol is never triggered. Moreover, due to the same reason, the output of the ideal join protocol $(\mathsf{Node}[c_0].\mathsf{orig}, \mathsf{Node}[c_0].\mathsf{mem})$ is identical to those from the previous hybrid (i.e., those output by id).

[*Case 3:*] Since w_0' includes the same confTag as w_0, we have $\mathsf{Wel}[w_0]$ is assigned to $\mathsf{Wel}[w_0']$ due to Proposition E.6. Note that there may exist many w_0' but all $\mathsf{Wel}[w_0']$ are identical, so this is well-defined. Moreover, since there is no c_0 that includes the same confTag as w_0 and w_0', we must have $\mathsf{Wel}[w_0'] = \mathsf{root}_{rt}$ for some $rt \in \mathbb{N}$. Hence, it suffices to check that $\mathcal{S}_{2\text{-}5}$ simulates the previous hybrid in case $\mathsf{Wel}[w_0']$ was initially set to root_{rt}, which we provide in the final *Cases 4.*

[*Case 4:*] Since $\mathcal{S}_{2\text{-}5}$ sets $\mathsf{orig}' := \mathsf{id}_c$ and $\mathsf{mem}' := G.\mathsf{memberIDsvks}()$, it is clear that the **assert** condition on line 12 of the ideal join protocol is not triggered. Moreover, since the ideal join protocol outputs $(\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{orig}', \mathsf{Node}[c_0].\mathsf{mem} = \mathsf{mem}')$, $\mathcal{S}_{2\text{-}5}$ simulates the previous hybrid perfectly.

(6) Analysis of Key Query. Due to Proposition E.8, every id and id′ such that $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Ptr}[\mathsf{id}]$ contain the same appSecret. Therefore, $\mathcal{S}_{2\text{-}5}$ provides an identical view to $\mathcal{Z}$ of the previous hybrid. □

E.3.6 From Hybrid 2-5 to 2-6: Proof of Lem. E.14.

Lemma E.14. *Hybrid 2-5 and Hybrid 2-6 are indistinguishable assuming* CmPKE *and* SIG *are correct with overwhelming probability.*

Proof. The only different between the previous hybrid occurs when id outputs $\bot$ when invoked on a commit, process, or join query by $\mathcal{S}_{2\text{-}5}$ but `*succeed-com`, `*succeed-proc`, or `*succeed-wel` output `true`, respectively. Notice the ideal `succeed-*` functionalities only care for commit and proposal messages that are not adversarially generated (i.e., $\mathsf{Node}[c_0].\mathsf{stat} \neq$ 'adv' and $\mathsf{Prop}[p].\mathsf{stat} \neq$ 'adv'). Therefore, as long as CmPKE and SIG are do not produce ciphertexts or signatures that do not correctly decrypt or verify, then the statement holds. Here, note that this must hold even the ciphertexts and signatures are created with maliciously generated randomness since the ideal functionality allows for $\mathsf{RandCor}[\mathsf{id}] =$ 'bad'. However, since we use the global random oracle to expand the randomness, no adversary can find an input that maps to a bad randomness assuming CmPKE and SIG are correct with overwhelming

probability (on honestly generated randomness). This completes the proof. □

E.4 From Hybrid 3 to 4: Lem. E.15

Hybrid 4 concerns the authenticity of the signature scheme. The functionality $\mathcal{F}_{\text{CGKA},4}$ halts if the **sig-inj-allowed** predicate returns false, i.e., $\mathcal{Z}$ succeeds to forge a signature without knowing signing key. To prove Lem. E.15 (i.e., the probability $\mathcal{F}_{\text{CGKA},4}$ halts is negligible), we show that, if $\mathcal{Z}$ can inject a message for which **sig-inj-allowed** predicate returns false, it can be used to break the sEUF-CMA security of SIG. In other words, if SIG is sEUF-CMA secure, the simulator receives only messages which the **sig-inj-allowed** predicate returns true. Thus, $\mathcal{F}_{\text{CGKA},4}$ never halts, and we conclude that Hybrid 3 and Hybrid 4 are indistinguishable. We below provide a formal proof of the above overview.

Lemma E.15. *Hybrid 3 and Hybrid 4 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. To show Lem. E.15, we consider the following sub-hybrids between Hybrid 3 and Hybrid 4.

Hybrid 3-0 := Hybrid 3. This is identical to Hybrid 3. We use the functionality $\mathcal{F}_{\text{CGKA},3}$ and the simulator $\mathcal{S}_{3\text{-}0} := \mathcal{S}_3$. In this hybrid, the **sig-inj-allowed** predicate always returns true.

Hybrid 3-1. This concerns injection of commit messages. The simulator $\mathcal{S}_{3\text{-}1}$ is defined exactly as $\mathcal{S}_{3\text{-}0}$ except that it aborts if the following condition holds (the simulator checks the condition whenever it updates the history graph).

- **Condition (A):** There exists a non-root node c_0 such that $\mathsf{Node}[c_0].\mathsf{stat} = \text{`adv'}$ and $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{Exposed}$, where $c_p := \mathsf{Node}[c_0].\mathsf{par}$ is the parent of c_0 and $\mathsf{id}_c := \mathsf{Node}[c_0].\mathsf{orig}$ is the committer of c_0.

Condition (A) relates to Condition (a) of `auth-invariant`: If a non-root node c_0 satisfying Condition (A) exists, Condition (a) of `auth-invariant` returns false. We show in Lem. E.16 that Hybrid 3-0 and Hybrid 3-1 are indistinguishable.

Hybrid 3-2. This concerns injection of proposal messages. The simulator $\mathcal{S}_{3\text{-}2}$ is defined exactly as $\mathcal{S}_{3\text{-}1}$ except that it aborts if the following condition holds (the simulator checks the condition whenever it updates the history graph).

- **Condition (B):** There exists a proposal node p such that $\mathsf{Prop}[p].\mathsf{stat} = \text{`adv'}$ and $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_s] \notin \mathsf{Exposed}$, where $c_p := \mathsf{Prop}[p].\mathsf{par}$ is the parent commit node of p and $\mathsf{id}_s := \mathsf{Prop}[p].\mathsf{orig}$ is the sender of p.

Condition (B) relates to Condition (b) of `auth-invariant`: If a node p satisfying Condition (B) exists, Condition (b) of `auth-invariant` returns false. We show in Lem. E.17 that Hybrid 3-1 and Hybrid 3-2 are indistinguishable.

Hybrid 3-3. This concerns injection of welcome messages. The simulator $\mathcal{S}_{3\text{-}3}$ is defined exactly as $\mathcal{S}_{3\text{-}2}$ except that it aborts if the following condition holds (the simulator checks the condition whnever it updates the history graph).

- **Condition (C):** There exists a detached root root_{rt} such that $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{Exposed}$, where $\mathsf{id}_c := \mathsf{Node}[\mathsf{root}_{rt}].\mathsf{orig}$ is the committer of the corresponding welcome message.

Condition (C) relates to Condition (c) of `auth-invariant`: If a root node root_{rt} satisfying Condition (C) exists, Condition (c) of `auth-invariant` returns false. We show in Lem. E.18 that Hybrid 3-2 and Hybrid 3-3 are indistinguishable.

Hybrid 3-4 := Hybrid 4. This is identical to Hybrid 4. We replace the functionality $\mathcal{F}_{\text{CGKA},3}$ with $\mathcal{F}_{\text{CGKA},4}$, that is, we use the original **sig-inj-allowed** predicate. The simulator $\mathcal{S}_{3\text{-}4}$ is defined exactly as $\mathcal{S}_{3\text{-}3}$ except that it no longer abort. Since $\mathcal{S}_{3\text{-}3}$ aborts if and only if $\mathcal{F}_{\text{CGKA},4}$ halts, Hybrid 3-3 and Hybrid 3-4 are identical.

From Lems. E.16 to E.18 provided below, Hybrid 3-0 and Hybrid 3-3 are indistinguishable. Moreover, Hybrid 3-3 and Hybrid 3-4 are identical. Therefore, we conclude that Hybrid 3 and Hybrid 4 are indistinguishable. □

E.4.1 From Hybrid 3-0 to 3-1: Proof of Lem. E.16.

Lemma E.16. *Hybrid 3-0 and Hybrid 3-1 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{3\text{-}0}$ and $\mathcal{S}_{3\text{-}1}$ is that $\mathcal{S}_{3\text{-}1}$ aborts if Condition (A) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature; and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}_{3\text{-}1}$ except for signature generation. Observer that honest signing keys are generated through `register-svk` queries to $\mathcal{F}_{\text{AS}}^{\text{IW}}$. Let svk^* be the challenge signing key received from the sEUF-CMA game. At the beginning of the game, $\mathcal{B}$ chooses an index $i \in [Q]$ at random, where Q is the largest total number of `register-svk` queries from $\mathcal{Z}$. $\mathcal{B}$ embeds the challenge key svk^* in the i-th `register-svk` query (if the i-th signature key is generated with bad randomness, $\mathcal{B}$ aborts). For the other `register-svk` queries, $\mathcal{B}$ generates signature keys as in the previous hybrid. We assume svk^* is embedded in id^*'s signature key. Whenever id^* creates key packages, proposals, or commit/welcome messages using svk^*, $\mathcal{B}$ uses the signing oracle to generate signatures. If id^* is corrupted or it generates a signature using bad randomness while it holds svk^*, $\mathcal{B}$ aborts.

$\mathcal{B}$ extracts a forgery as follows: Whenever $\mathcal{B}$ creates a node c_0, $\mathcal{B}$ checks whether c_0 satisfies Condition (A). If a node c_0 satisfies the condition, $\mathcal{B}$ retrieves the signature sig and the signed message $m :=$ comCont from c_0. If the sender of c_0 is id^*, and the corresponding signature key is svk^*, $\mathcal{B}$ submits (m, sig) to the challenger. Note that (m, sig) is a valid message-signature pair because the node is created only if (m, sig) is valid.

We argue (m, sig) is a valid forgery. Since c_0 satisfies Condition (A) and c_0 is sent from id^* using svk^*, the following holds:

Fact (1): $\mathsf{Node}[c_0].\mathsf{stat} = \text{`adv'}$;
Fact (2): $\mathsf{Node}[c_0].\mathsf{orig} = \mathsf{id}_c = \mathsf{id}^*$;
Fact (3): $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] = \mathsf{svk}^*$, where $c_p := \mathsf{Node}[c_0].\mathsf{par}$; and
Fact (4): $\mathsf{Node}[c_p].\mathsf{mem}[\mathsf{id}_c] \notin \mathsf{Exposed}$.

Fact (1) implies (m, sig) has not been generated by $\mathcal{B}$. Therefore, the sign oracle has not output (m, sig). Facts (2)-(4) imply svk^* has

not been exposed. Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the sEUF-CMA game.

We finally evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. Therefore, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and we conclude that Hybrid 3-0 and Hybrid 3-1 are indistinguishable to $\mathcal{Z}$. □

E.4.2 From Hybrid 3-1 to 3-2: Proof of Lem. E.17.

Lemma E.17. *Hybrid 3-1 and Hybrid 3-2 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{3\text{-}1}$ and $\mathcal{S}_{3\text{-}2}$ is $\mathcal{S}_{3\text{-}2}$ aborts if Condition (B) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature; and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as shown in Lem. E.16, and extracts the forgery as follows: Whenever $\mathcal{B}$ creates a node p, $\mathcal{B}$ checks whether p satisfies Condition (B). If some node p satisfies the condition, $\mathcal{B}$ retrieves the signature sig and the signed message $m :=$ propCont from p. If the sender of p is id^* and the corresponding signature key is svk^*, $\mathcal{B}$ submits (m, sig) as the forgery.

We argue (m, sig) is a valid forgery. Since p satisfies Condition (B) and p is sent from id^* with svk^*, the following holds:

Fact (1): $\text{Prop}[p].\text{stat} = \text{'adv'}$;
Fact (2): $\text{Prop}[p].\text{orig} = \text{id}_c = \text{id}^*$;
Fact (3): $\text{Node}[c_p].\text{mem}[\text{id}_c] = \text{svk}^*$, where $c_p := \text{Node}[c_0].\text{par}$; and
Fact (4): $\text{Node}[c_p].\text{mem}[\text{id}_c] \notin \text{Exposed}$.

Note that $c_p := \text{Prop}[p].\text{par}$ is the parent of p. Fact (1) implies (m, sig) has not been generated by $\mathcal{B}$. Therefore, the sign oracle has not output (m, sig). Facts (2)-(4) implies svk^* has not been exposed. Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the sEUF-CMA game.

We evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. Therefore, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and we conclude that Hybrid 3-1 and Hybrid 3-2 are indistinguishable to $\mathcal{Z}$. □

E.4.3 From Hybrid 3-2 to 3-3: Proof of Lem. E.18.

Lemma E.18. *Hybrid 3-2 and Hybrid 3-3 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{3\text{-}2}$ and $\mathcal{S}_{3\text{-}3}$) is $\mathcal{S}_{3\text{-}3}$ aborts if Condition (C) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature; and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as shown in Lem. E.16, and extracts the forgery as follows: Whenever $\mathcal{B}$ creates a node root_{rt}, $\mathcal{B}$ checks whether root_{rt} satisfies Condition (C). If some node root_{rt} satisfies the condition, $\mathcal{B}$ retrieves the signature sig and the signed message $m := (\text{groupInfo}, \mathsf{T})$ from w_0. If the sender of w_0 is id^* and the corresponding signature key is svk^*, $\mathcal{B}$ submits (m, sig) as the forgery.

We argue (m, sig) is a valid forgery. Since root_{rt} satisfies Condition (C) and w_0 is valid on $(\text{id}^*, \text{svk}^*)$, the following facts hold:

Fact (1): $\text{Node}[\text{root}_{rt}].\text{stat} = \text{'adv'}$;
Fact (2): $\text{Node}[\text{root}_{rt}].\text{orig} = \text{id}_c = \text{id}^*$;
Fact (3): $\text{Node}[\text{root}_{rt}].\text{mem}[\text{id}_c] = \text{svk}^*$; and
Fact (4): $\text{Node}[\text{root}_{rt}].\text{mem}[\text{id}_c] \notin \text{Exposed}$.

Fact (1) implies (m, sig) has not been generated by $\mathcal{B}$. Therefore, the sign oracle has not output (m, sig). Facts (2)-(4) implies svk^* has not been exposed. Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the sEUF-CMA game.

We evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. Therefore, if $\mathcal{Z}$ distinguishes the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability at least ϵ/Q, which is also non-negligible. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and we conclude that Hybrid 3-2 and Hybrid 3-3 are indistinguishable to $\mathcal{Z}$. □

E.5 From Hybrid 4 to 5: Lem. E.19

Hybrid 5 concerns the authenticity of MAC. The functionality $\mathcal{F}_{\text{CGKA},5}$ halts if the **mac-inj-allowed** predicate returns false, i.e., $\mathcal{Z}$ succeeds to forge a MAC tag without knowing the MAC key. To show Lem. E.19 (i.e., the probability $\mathcal{F}_{\text{CGKA},5}$ halts is negligible), we show that, if $\mathcal{Z}$ can distinguish the two hybrids, we can break the Chained CmPKE conforming GSD security of CmPKE. To this end, we consider that the simulator creates the GSD graph based on epoch secrets and MAC tags. The GSD graph represents the relationship of epoch secrets and MAC tags, and indicates which MAC key is exposed (Note that to discuss which MAC key is exposed, we will uses the sEUF-CMA security of SIG.). We show that, if $\mathcal{Z}$ injects a message for which the **mac-inj-allowed** predicate returns false, it can be used to break the Chained CmPKE conforming GSD security. In other words, if CmPKE is Chained CmPKE conforming GSD secure, the simulator receives only messages for which the **mac-inj-allowed** predicate returns true. Thus, $\mathcal{F}_{\text{CGKA},5}$ never halts, and we conclude that Hybrid 4 and Hybrid 5 are indistinguishable. We below provide a formal proof of the above overview.

Lemma E.19. *Hybrid 4 and Hybrid 5 are indistinguishable assuming* SIG *is* sEUF-CMA *secure and* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. To prove the lemma, we consider the following sub-hybrids between Hybrids 4 and 5:

Hybrid 4-0 := Hybrid 4. This is identical to Hybrid 4. We use the functionality $\mathcal{F}_{\text{CGKA},4}$ and the simulator $\mathcal{S}_{4\text{-}0} := \mathcal{S}_4$. In

this hybrid, the **mac-inj-allowed** predicate always returns true.[17]

Hybrid 4-1. This concerns injection of key packages. The simulator $\mathcal{S}_{4\text{-}1}$ is defined exactly as $\mathcal{S}_{4\text{-}0}$ except that it aborts if the following condition holds.

Condition (KP): There exists a proposal node p such that $\mathsf{Prop}[p].\mathsf{act} = \text{`add'-id}_t\text{-svk}_t, \mathsf{svk}_t \notin \mathsf{Exposed}$ and $\mathsf{DK}[\mathsf{id}_t, \mathsf{kp}_t] = \perp$, where kp_t is the key package in p.

We show in Lem. E.20 that Hybrid 4-0 and Hybrid 4-1 are indistinguishable assuming SIG is sEUF-CMA secure. In the following hybrids, if a valid key package is injected, the corresponding signing key is always exposed. We will use this fact to discuss which secret is exposed in the GSD graph.

Hybrid 4-2. We modify the simulator $\mathcal{S}_{4\text{-}1}$ so that it creates the GSD graph based on CmPKE keys, epoch secrets and MAC tags. Roughly speaking, the GSD graph represents the relationship of CmPKE keys, epoch secrets and MAC tags, and it indicates which secret is exposed. The simulator $\mathcal{S}_{4\text{-}2}$ internally runs two simulators $\mathcal{S}_{\mathsf{GSD}}$ and $\mathcal{S}'_{4\text{-}2}$: $\mathcal{S}_{\mathsf{GSD}}$ simulates the GSD oracles and creates the GSD graph as shown in Fig. 29; $\mathcal{S}'_{4\text{-}2}$ simulates the interaction for the environment $\mathcal{Z}$, the functionality, and the adversary $\mathcal{A}$ by using the GSD oracles provided by $\mathcal{S}_{\mathsf{GSD}}$. In addition, $\mathcal{S}'_{4\text{-}2}$ always rejects injected commit/proposal messages if the corresponding MAC key is not exposed, and aborts if a commit node is attached to a detached root although the corresponding initial secret is not exposed.[18] So as not to interrupt the proof, we formally explain how $\mathcal{S}_{\mathsf{GSD}}$ and $\mathcal{S}'_{4\text{-}2}$ are defined. We show in Lem. E.21 that Hybrid 4-1 and Hybrid 4-2 are indistinguishable assuming CmPKE is Chained CmPKE conforming GSD secure.

Hybrid 4-3. We undo the changes made between Hybrid 4-0 and Hybrid 4-2. That is, the simulator $\mathcal{S}_{4\text{-}3}$ no longer aborts the simulation. Using the same arguments to move through Hybrid 4-0 to Hybrid 4-2, Hybrid 4-2 and Hybrid 4-3 remain indistinguishable.

Hybrid 4-4 := Hybrid 5. This is identical to Hybrid 5. We replace the functionality $\mathcal{F}_{\mathsf{CGKA},4}$ with $\mathcal{F}_{\mathsf{CGKA},5}$, that is, we use the original **mac-inj-allowed** predicate. The simulator $\mathcal{S}_{4\text{-}4}$ is defined exactly as $\mathcal{S}_{4\text{-}3}$. We show in Lem. E.28 that Hybrid 4-3 and Hybrid 4-4 are identical.

From Lems. E.20, E.21 and E.28 provided below, Hybrid 4-0 and Hybrid 4-4 are indistinguishable. Therefore, we conclude that Hybrid 4 and Hybrid 5 are indistinguishable.

□

E.5.1 From Hybrid 4-0 to 4-1: Proof of Lem. E.20.

Lemma E.20. *Hybrid 4-0 and Hybrid 4-1 are indistinguishable assuming* SIG *is* sEUF-CMA *secure.*

Proof. The only difference between $\mathcal{S}_{4\text{-}0}$ and $\mathcal{S}_{4\text{-}1}$ is $\mathcal{S}_{4\text{-}1}$ aborts Condition (KP) holds. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that breaks the sEUF-CMA security of SIG. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ extracts a valid signature forgery using $\mathcal{Z}$; we then show the validity of the forged signature; and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\mathsf{AS}}^{\mathsf{IW}}$ and $\mathcal{F}_{\mathsf{KS}}^{\mathsf{IW}}$, and the protocol execution of each party as in $\mathcal{S}_{4\text{-}1}$ except for the signature generation. Let svk^* be the challenge signature key provided by the sEUF-CMA game. Observer that honest signing keys are generated on `register-svk` queries to $\mathcal{F}_{\mathsf{AS}}^{\mathsf{IW}}$. We assume $\mathcal{Z}$ issues at most Q `register-svk` queries. At the beginning of the game, $\mathcal{B}$ chooses an index $i \in [Q]$ at random, and embeds the challenge key svk^* in the i-th `register-svk` query (if the i-th signature key is generated with bad randomness, $\mathcal{B}$ aborts). For other `register-svk` queries, $\mathcal{B}$ generates signature keys following the description of $\mathcal{F}_{\mathsf{AS}}^{\mathsf{IW}}$. Assume svk^* is embedded in id^*'s signing key. Whenever id^* creates key packages, proposals, or commit/welcome messages using svk^*, $\mathcal{B}$ uses the signing oracle to generate signatures. If id^* is corrupted or it generates a signature using bad randomness while it holds svk^*, $\mathcal{B}$ aborts.

$\mathcal{B}$ extracts the forgery as follows: Whenever $\mathcal{B}$ creates a node p, $\mathcal{B}$ checks whether p satisfies Condition (KP). If a node p satisfies the condition, $\mathcal{B}$ retrieves the signature sig and the signed message $m := (\mathsf{id}, \mathsf{ek}, \mathsf{svk})$ from p. If $(\mathsf{id}, \mathsf{svk}) = (\mathsf{id}^*, \mathsf{svk}^*)$, $\mathcal{B}$ submits (m, sig) as the forgery. (Note that the proposal node is created only if (m, sig) is valid.)

We argue (m, sig) is a valid forgery. Since p satisfies Condition (KP) and p contains the key package on $(\mathsf{id}^*, \mathsf{svk}^*)$, the following facts hold:

Fact (1): $\mathsf{Prop}[p].\mathsf{act} = \text{`add'-id}^*\text{-svk}^*$;
Fact (2): $\mathsf{DK}[\mathsf{id}^*, \mathsf{kp}^*] = \perp$, where kp^* is the key package in p; and
Fact (3): $\mathsf{svk}^* \notin \mathsf{Exposed}$.

Fact (1) implies the adversary outputs the valid signature on svk^* because history graph nodes are created when messages are valid. Fact (2) implies (m, sig) has not been generated by Key Service via `register-kp` query; Specifically the sign oracle has not outputs (m, sig). Fact (3) implies svk^* has not been exposed when $\mathcal{B}$ obtains (m, sig). Therefore, (m, sig) is a valid forgery on svk^*, and $\mathcal{B}$ wins the game.

We evaluate the success probability of $\mathcal{B}$. The probability that $\mathcal{B}$ correctly guesses the signature key used to forge is $1/Q$. If $\mathcal{Z}$ can distinguish the hybrids with probability ϵ, $\mathcal{B}$ wins the game within probability ϵ/Q. If ϵ is non-negligible, $\mathcal{B}$ wins sEUF-CMA game with non-negligible probability. This contradicts the assumption that SIG is sEUF-CMA secure. Therefore, ϵ must be negligible, and Hybrid 4-0 and Hybrid 4-1 are indistinguishable to $\mathcal{Z}$. □

E.5.2 Proof of Lem. E.21: From Hybrid 4-1 to 4-2.

The proof consists of two parts: We first explain how simulator $\mathcal{S}_{4\text{-}2}$ simulates Hybrid 4-2 while creating the GSD graph based on secrets and MAC tags. (see *Part 1*); we then show $\mathcal{S}_{4\text{-}2}$ provides an indistinguishable view to $\mathcal{Z}$ (see *Part 2*).

Part 1: Description of the Simulator $\mathcal{S}_{4\text{-}2}$. We consider that $\mathcal{S}_{4\text{-}2}$ internally runs two simulators $\mathcal{S}_{\mathsf{GSD}}$ and $\mathcal{S}'_{4\text{-}2}$: $\mathcal{S}_{\mathsf{GSD}}$ simulates the GSD oracles and creates the GSD graph following the procedures shown in Fig. 29, and $\mathcal{S}'_{4\text{-}2}$ simulates the interaction for the environment $\mathcal{Z}$, the functionality, and the adversary $\mathcal{A}$ by using the GSD oracles provided by $\mathcal{S}_{\mathsf{GSD}}$. Looking ahead, we use the GSD game

[17] Since the **sig-inj-allowed** predicate always returns true (cf. Lem. E.15), the truth value of `auth-invariant` and the truth value of **mac-inj-allowed** are the same.
[18] We define $\mathcal{S}'_{4\text{-}2}$ so that it never creates history graph nodes for which **mac-inj-allowed** returns false.

to show the several hybrids remain indistinguishable. We allow the reduction to simulate $\mathcal{S}_{4\text{-}2}$ by running $\mathcal{S}'_{4\text{-}2}$ on its own while using the challenger provided by the GSD game as a replacement of $\mathcal{S}_{\mathrm{GSD}}$.

$\mathcal{S}'_{4\text{-}2}$ creates the GSD graph based on secrets (CmPKE decryption keys and epoch secrets) and MAC tags created by simulated parties. $\mathcal{S}'_{4\text{-}2}$ keeps a counter ctr (it is initialized with 1), denoting the smallest unused GSD node. Whenever deriving a new encryption key, epoch secret, or MAC tag, $\mathcal{S}'_{4\text{-}2}$ assigns a GSD node to the secret/tag. For example, when $\mathcal{S}'_{4\text{-}2}$ generates a random secret, it sends an unused GSD node to the GSD oracle and the oracle chooses the value. When $\mathcal{S}'_{4\text{-}2}$ uses a specific value as the secret, it assigns the value to an unused GSD node by using `Set-Secret` oracle. When $\mathcal{S}'_{4\text{-}2}$ computes an epoch secret or MAC tag using random oracle, it calls `Join-Hash`/`Hash` oracle to assign the derived secret/tag to an unused GSD node. Throughout the proof, we denote the secret (decryption keys and epoch secrets) by (s, u), the pair of the secret s and the assigned GSD node u. When the node is assigned to the secret, if $\mathcal{S}'_{4\text{-}2}$ knows the secret $s \neq \bot$, the secret is set to (s, u), otherwise set to $(\bot, u)$. The value of each secret will be updated adaptively during the simulation: As soon as $\mathcal{S}'_{4\text{-}2}$ corrupts some GSD node v (i.e., query $\mathtt{Corr}(u)$), for each node u such that $\mathbf{gsd\text{-}exp}(u)$ becomes true, $\mathcal{S}'_{4\text{-}2}$ computes the secret s_u from previously obtained ciphertexts and corrupted secrets, and replaces all occurrence of $(\bot, u)$ with (s_u, u). Hence, if $\mathbf{gsd\text{-}exp}(u) = \mathtt{true}$, then $\mathcal{S}'_{4\text{-}2}$ knows the secret s assigned to the node u. The special case is the encryption key is generated by $\mathcal{A}$ (this case occurs when an injected add/update proposal or commit message is received). In this case, since $\mathcal{S}'_{4\text{-}2}$ does not know the corresponding decryption key, it sets the decryption key to $(\bot, \bot)$.

$\mathcal{S}'_{4\text{-}2}$ maintains the following lists for the simulation:

- $L_{\text{`memb'}}$: It contains tuple $(u_{\text{`memb'}}, m, \mathsf{membTag})$ where $u_{\text{`memb'}}$ is a GSD node assigned to the membership key, $m$ is a MACed message under the key $u_{\text{`memb'}}$ and membTag is a corresponding a membership tag generated by $\mathcal{S}_{\mathrm{GSD}}$.
- $L_{\text{`epoch'}}$: It contains tuple $(u_{\mathsf{par\text{-}init}}, \mathsf{comSecret}, \mathsf{joinerSecret}, \mathsf{confTransHash}, \mathsf{confTag})$ where $u_{\mathsf{par\text{-}init}}$ is a GSD node assigned to the parent initial secret, comSecret is a commit secret, joinerSecret is a joiner secret, confTransHash is a confirmation hash and confTag is a confirmation tag (i.e., epoch secrets and MAC tag of each epoch).
- $L_{\text{`enc'}}$: It contains tuple $((s, u), u_{\mathsf{id}}, (\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}))$ where (s, u) is a encrypted secret (comSecret or joinerSecret), u_{id} is a GSD node assigned to the encryption key of party id and $(\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}})$ is a corresponding CmPKE ciphertext.

The first two lists are used when $\mathcal{S}'_{4\text{-}2}$ needs to recompute the same epoch secrets or MAC tags. In particular, due to the modification we made in Hybrid 2-2, confTag is unique for each joinerSecret and confTransHash (and joinerSecret is unique for each initSecret and comSecret (cf. Hybrid 2-1)). The third $L_{\text{`enc'}}$ is used to check whether the received ciphertext can be sent to `CmDec` oracle.

Now we explain how $\mathcal{S}'_{4\text{-}2}$ simulates the protocol using the GSD oracles. Other procedures not described are identical to the previous hybrid. Note that to make the proof of Proposition E.27 easier to read, oracle calls corresponding to the corruption of a GSD node are highlighted with red underline.

Key package creation for id**.** When $\mathcal{S}'_{4\text{-}2}$ creates a key package, it generates a CmPKE's encryption key for the key package with the help of the GSD oracle.

- If $\mathsf{RandCor}[\mathsf{id}] =$ 'good', $\mathcal{S}'_{4\text{-}2}$ generates a CmPKE key pair as $(\mathsf{ek}, \mathsf{dk}) := (\mathtt{*get\text{-}ek}(u_{\mathsf{ctr}}), (\bot, u_{\mathsf{ctr}}))$ (and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$). Here, $\mathsf{ek} \leftarrow \mathtt{*get\text{-}ek}(u)$ denotes that $\mathcal{S}'_{4\text{-}2}$ obtains the encryption key ek on a node u by calling the oracle $\mathtt{CmEnc}(\{u\}, 0)$ (the special node 0 is only used here). $\mathcal{S}'_{4\text{-}2}$ creates a key package based on the encryption key.
- Else, key packages are generated as in the previous hybrid. After generating the key package, $\mathcal{S}'_{4\text{-}2}$ assigns the seed s of the CmPKE key to an unused GSD node $u_{\mathsf{id}} := u_{\mathsf{ctr}}$ (ctr is incremented) by querying $\mathtt{Set\text{-}Full\text{-}Secret}(s, u_{\mathsf{id}})$. It sets $\mathsf{dk} := (\mathsf{dk}_{\mathsf{id}}, u_{\mathsf{id}})$.

Simulation of $\mathsf{id}_{\mathsf{creator}}$ **on input** (**`Create`**, svk)**.** $\mathcal{S}'_{4\text{-}2}$ creates a new group following the protocol, except for the initialization of epoch secrets and the confirmation tag. (The initial key package is created following the procedure in *Key package creation for* id above.)

$\mathcal{S}'_{4\text{-}2}$ generates the initial epoch secrets as follows: $\mathcal{S}'_{4\text{-}2}$ first prepares new GSD nodes $u_{\text{`joi'}}$, $u_{\text{`conf'}}$, $u_{\text{`app'}}$, $u_{\text{`memb'}}$, and $u_{\text{`init'}}$ (their values are set to $u_{\mathsf{ctr}}, u_{\mathsf{ctr}+1}, \ldots, u_{\mathsf{ctr}+4}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 5$).

- If $\mathsf{RandCor}[\mathsf{id}_{\mathsf{creator}}] =$ 'good', $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Hash}(u_{\text{`joi'}}, u_{\mathsf{lbl}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{$'conf', 'app', 'memb', 'init'$\}$.[19] It sets $G.\mathsf{joinerSecret} := (\bot, u_{\text{`joi'}})$, $G.\mathsf{confKey} := (\bot, u_{\text{`conf'}})$, $G.\mathsf{appSecret} := (\bot, u_{\text{`app'}})$, $G.\mathsf{membKey} := (\bot, u_{\text{`memb'}})$, $G.\mathsf{initSecret} := (\bot, u_{\text{`init'}})$. Note that the value assigned to $u_{\text{`joi'}}$ is set to random during the first call of `Hash`.
- Else, $\mathcal{S}'_{4\text{-}2}$ generates the joiner secret $s_{\text{`joi'}}$ using the randomness provided from $\mathcal{A}$, and computes $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{`joi'}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{$'conf', 'app', 'memb', 'init'$\}$. Then, $\mathcal{S}'_{4\text{-}2}$ assigns the epoch secrets to GSD oracles: It queries $\mathtt{Set\text{-}Secret}(u_{\text{`joi'}}, s_{\text{`joi'}})$ and $\mathtt{Hash}(u_{\text{`joi'}}, u_{\mathsf{lbl}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{$'conf', 'app', 'memb', 'init'$\}$. $\mathcal{S}'_{4\text{-}2}$ sets $G.\mathsf{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G.\mathsf{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G.\mathsf{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G.\mathsf{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$, $G.\mathsf{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$.

$\mathcal{S}'_{4\text{-}2}$ generates the confirmation tag as follows:

- If $\mathsf{RandCor}[\mathsf{id}_{\mathsf{creator}}] =$ 'good', $\mathcal{S}'_{4\text{-}2}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ (and sets $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$). Then, it queries $\mathtt{Hash}(u_{\text{`conf'}}, u_{\mathsf{ctag}}, G.\mathsf{confTransHash})$ and $G.\mathsf{confTag} := \mathtt{Corr}(u_{\mathsf{ctag}})$. (See Rem. 1.)
- Else, $\mathcal{S}'_{4\text{-}2}$ computes $G.\mathsf{confTag} := \mathsf{RO}(s_{\text{`conf'}}, G.\mathsf{confTransHash})$ as in the previous hybrid.

Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`epoch'}} \xleftarrow{+} (\bot, (\bot, \bot), G.\mathsf{joinerSecret}, G.\mathsf{confTransHash}, G.\mathsf{confTag})$.

Simulation of id **on input** (**`Propose`**, act)**.** $\mathcal{S}'_{4\text{-}2}$ generates a proposal message p as in the previous hybrid, except for the generation of key packages and membership tags.

When $\mathcal{S}'_{4\text{-}2}$ generates an update proposal, it creates a key package following the procedure in *Key package creation for* id above. Then,

[19] $G.\mathsf{groupCont}()$ returns the group information defined in Tab. 8. It is determined before the party computes the epoch secret.

$\mathcal{S}'_{4\text{-}2}$ keeps the generated decryption key dk for p (dk will be used when $\mathcal{S}'_{4\text{-}2}$ receives p in the commit or process protocol).

$\mathcal{S}'_{4\text{-}2}$ generates the membership tag as follows, by using the membership key G.membKey = ($s_{\text{`memb'}}$, $u_{\text{`memb'}}$) at epoch Ptr[id].

- If **gsd-exp**($u_{\text{`memb'}}$) = true (i.e., $s_{\text{`memb'}} \neq \bot$)[20], $\mathcal{S}'_{4\text{-}2}$ computes membTag := RO($s_{\text{`memb'}}$, (propCont, sig)) as in the previous hybrid.
- Else if ($u_{\text{`memb'}}$, (propCont, sig), membTag) $\in L_{\text{`memb'}}$ exists for some membTag, it is used as the membership tag. Note that this case occurs when the same p is generated multiple times.
- Else, $\mathcal{S}'_{4\text{-}2}$ prepares a new GSD node $u_{\text{mtag}} := u_{\text{ctr}}$ (and sets ctr ← ctr+1) and queries Hash($u_{\text{`memb'}}$, $u_{\text{mtag}}$, (propCont, sig)) and membTag := Corr($u_{\text{mtag}}$). $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`memb'}} \xleftarrow{+}$ ($u_{\text{`memb'}}$, (propCont, sig), membTag).

Simulation of id **on input** (**Commit**, $\vec{p}$, svk)**.** $\mathcal{S}'_{4\text{-}2}$ generates a commit message as in the previous hybrid, except for the differences shown below:

In *unframe-prop, $\mathcal{S}'_{4\text{-}2}$ verifies the membership tag membTag in p as follows, by using the membership key G.membKey = ($s_{\text{`memb'}}$, $u_{\text{`memb'}}$) at epoch Ptr[id]:

- If **gsd-exp**($u_{\text{`memb'}}$) = true (i.e., $s_{\text{`memb'}} \neq \bot$), $\mathcal{S}'_{4\text{-}2}$ computes membTag′ = RO($s_{\text{`memb'}}$, (propCont, sig)) and checks whether membTag = membTag′ holds.
- Else if ($u_{\text{`memb'}}$, (propCont, sig), membTag′) $\in L_{\text{`memb'}}$ exists for some membTag′, $\mathcal{S}'_{4\text{-}2}$ checks whether membTag = membTag′ holds.
- Else, $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\text{inj-p}}$ and in Proposition E.22 we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\text{inj-p}}$ occurs. Thus, it does not alter the view of $\mathcal{Z}$

In *apply-props, $\mathcal{S}'_{4\text{-}2}$ updates the membership list as follows:

- If p is an update proposal sent from $\mathsf{id}_s \neq \mathsf{id}$, $\mathcal{S}'_{4\text{-}2}$ stores the decryption key corresponding to p to G'.member[id_s].dk. Namely, if p has been generated by $\mathcal{S}'_{4\text{-}2}$, it knows the decryption key. Otherwise (i.e., $\mathcal{S}'_{4\text{-}2}$ does not know the decryption key), $\mathcal{S}'_{4\text{-}2}$ sets G'.member[id_s].dk := ($\bot$, $\bot$).
- If p is an add proposal that contains id_t and kp_t, $\mathcal{S}'_{4\text{-}2}$ copies the decryption key stored in DK[id_t, kp_t] to G'.member[id_t].dk. If kp_t is not registered to Key Service $\mathcal{F}_{\text{KS}}^{\text{IW}}$ (i.e.,DK[id_t, kp_t] = $\bot$), $\mathcal{S}'_{4\text{-}2}$ sets G'.member[id_t].dk := ($\bot$, $\bot$).

After the execution of *apply-props, $\mathcal{S}'_{4\text{-}2}$ obtains the new membership list and the decryption key of each member. In other words, for all members id, G.member[id].dk is of the form ($*$, u_{id}) or ($\bot$, $\bot$), where $*$ is either $\bot$ or $s_{\mathsf{id}} \neq \bot$.

$\mathcal{S}'_{4\text{-}2}$ runs *rekey as follows: It first creates a key package following the procedure in *Key package creation for* id above. Then $\mathcal{S}'_{4\text{-}2}$ generates a new commit secret:

- If RandCor[id] = 'good', $\mathcal{S}'_{4\text{-}2}$ first prepares a GSD node $u_{\text{com}} := u_{\text{ctr}}$ (and sets ctr ← ctr + 1) and generates a random commit secret as follows. Let G'.member be the new membership list and receivers be the set of the identity of the existing parties.
  - If G'.member[id].dk ≠ ($\bot$, $\bot$) for all id ∈ receivers, each G'.member[id].dk is of the form ($*$, u_{id}). $\mathcal{S}'_{4\text{-}2}$ composes the set $S_{\text{receivers}} := \{ u_{\mathsf{id}} \}_{\mathsf{id} \in \text{receivers}}$ and queries CmEnc($S_{\text{receivers}}$, u_{com}) to compute the ciphertext ($\mathsf{T}, \vec{\mathsf{ct}} = (\mathsf{ct}_u)_{u \in S_{\text{receivers}}}$). Note that the commit secret is chosen at random by the CmEnc oracle.
    - If **gsd-exp**(u_{id}) = true for some $u_{\mathsf{id}} \in S_{\text{receivers}}$, $\mathcal{S}'_{4\text{-}2}$ decrypts ($\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}$) using the secret s_{id} of u_{id}, and obtains s_{com}. Then, $\mathcal{S}'_{4\text{-}2}$ sets G'.comSecret := (s_{com}, u_{com}) and stores $L_{\text{`enc'}} \xleftarrow{+}$ ((s_{com}, u_{com}), u_{id}, ($\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}$)) for each $u_{\mathsf{id}} \in S_{\text{receivers}}$.
    - Else, $\mathcal{S}'_{4\text{-}2}$ sets G'.comSecret := ($\bot$, u_{com}) and stores $L_{\text{`enc'}} \xleftarrow{+}$ (($\bot$, u_{com}), u_{id}, ($\mathsf{T}, \mathsf{ct}_{u_{\mathsf{id}}}$)) for each $u_{\mathsf{id}} \in S_{\text{receivers}}$.
  - Else (i.e., G'.member[id].dk = ($\bot$, $\bot$) for some id ∈ receivers), $\mathcal{S}'_{4\text{-}2}$ generates and encrypts the commit secret s_{com} as in the previous hybrid. Then, $\mathcal{S}'_{4\text{-}2}$ queries Set-Secret(u_{com}, s_{com}) and sets G'.comSecret := (s_{com}, u_{com}).
- If RandCor[id] = 'bad', $\mathcal{S}'_{4\text{-}2}$ generates and encrypts the commit secret s_{com} as in the previous hybrid. In this case, $\mathcal{S}'_{4\text{-}2}$ may have generated or received the same commit message. Thus, $\mathcal{S}'_{4\text{-}2}$ checks if the same commit message has been generated or received earlier, and if not, it assigns a GSD node to the commit secret s_{com}. This check is done when $\mathcal{S}'_{4\text{-}2}$ derives the epoch secret in *derive-keys function (see below).

To compute the epoch secrets and confirmation tag, $\mathcal{S}'_{4\text{-}2}$ runs *derive-keys and *gen-conf-tag as follows: Let ($s_{\text{par-init}}$, $u_{\text{par-init}}$) be the initial secret at epoch ($s_{\text{par-init}}$ can be $\bot$). Note that G'.groupCont() (including the confirmation hash G'.confTransHash) for the new epoch is computed before running *derive-keys function.

- If RandCor[id] = 'good', $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes $u_{\text{`joi'}}$, $u_{\text{`conf'}}$, $u_{\text{`app'}}$, $u_{\text{`memb'}}$, and $u_{\text{`init'}}$ (their values are set to $u_{\text{ctr}}$, $u_{\text{ctr+1}}, \ldots, u_{\text{ctr+4}}$ and ctr is incremented as ctr ← ctr+5) and queries Join-Hash($u_{\text{par-init}}$, $u_{\text{com}}$, $u_{\text{`joi'}}$, 'joi') and Hash($u_{\text{`joi'}}$, u_{lbl}, (G'.groupCont(), lbl)) for each lbl ∈ {'conf', 'app', 'memb', 'init'}.
  - If **gsd-exp**($u_{\text{par-init}}$) = false ∨ **gsd-exp**(u_{com}) = false, $\mathcal{S}'_{4\text{-}2}$ sets G'.joinerSecret := ($\bot$, $u_{\text{`joi'}}$), $G'$.confKey := ($\bot$, $u_{\text{`conf'}}$), G'.appSecret := ($\bot$, $u_{\text{`app'}}$), $G'$.membKey := ($\bot$, $u_{\text{`memb'}}$) and G'.initSecret := ($\bot$, $u_{\text{`init'}}$). Then, $\mathcal{S}'_{4\text{-}2}$ prepares a new GSD node $u_{\text{ctag}} := u_{\text{ctr}}$ (and sets ctr ← ctr+1), and queries Hash($u_{\text{`conf'}}$, u_{ctag}, G'.confTransHash) and G'.confTag := Corr(u_{ctag}).
  - Else, (i.e., **gsd-exp**($u_{\text{par-init}}$) = true ∧ **gsd-exp**(u_{com}) = true), $\mathcal{S}'_{4\text{-}2}$ knows both $s_{\text{par-init}}$ and s_{com}. It computes $s_{\text{`joi'}}$ := RO($s_{\text{par-init}}$, $s_{\text{com}}$, 'joi') and $s_{\text{lbl}}$ := RO($s_{\text{`joi'}}$, (G'.groupCont(), lbl)) for each lbl ∈ {'conf', 'app', 'memb', 'init'}, and sets G'.joinerSecret := ($s_{\text{`joi'}}$, $u_{\text{`joi'}}$), G'.confKey := ($s_{\text{`conf'}}$, $u_{\text{`conf'}}$), G'.appSecret := ($s_{\text{`app'}}$, $u_{\text{`app'}}$), G'.membKey := ($s_{\text{`memb'}}$, $u_{\text{`memb'}}$) and G'.initSecret := ($s_{\text{`init'}}$, $u_{\text{`init'}}$). $\mathcal{S}'_{4\text{-}2}$ generates the confirmation tag as G'.confTag := RO($s_{\text{`conf'}}$, G'.confTransHash).

[20] Recall that if **gsd-exp**(u) = true for the node u, then $\mathcal{S}'_{4\text{-}2}$ knows the secret $s \neq \bot$ assigned to u. This is because if **gsd-exp**(u) = true , $\mathcal{S}'_{4\text{-}2}$ can compute the secret s from the known information (e.g., previously generated ciphertexts).

Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`epoch'}} \xleftarrow{+} (u_{\text{par-init}}, G'.\text{comSecret}, G'.\text{joinerSecret}, G'.\text{confTransHash}, G'.\text{confTag})$.

- If RandCor[id] = 'bad', $\mathcal{S}'_{4\text{-}2}$ knows $s_{\text{com}} \neq \bot$. $\mathcal{S}'_{4\text{-}2}$ does as follows.
  - If $(u_{\text{par-init}}, (s_{\text{com}}, *), *, G'.\text{confTransHash}, \text{confTag}') \in L_{\text{`epoch'}}$ exists for some unique confTag′,[21] $\mathcal{S}'_{4\text{-}2}$ uses confTag′ as the confirmation tag. Note that this case occurs if some party has derived the same epoch secrets in the commit or process protocol.
  - Else, $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes u_{com}, $u_{\text{`joi'}}$, $u_{\text{`conf'}}$, $u_{\text{`app'}}$, $u_{\text{`memb'}}$, and $u_{\text{`init'}}$ (their values are set to $u_{\text{ctr}}, u_{\text{ctr+1}}, \ldots, u_{\text{ctr+5}}$ and ctr is incremented as ctr ← ctr + 6) and queries $\texttt{Set-Secret}(u_{\text{com}}, s_{\text{com}})$, $\texttt{Join-Hash}(u_{\text{par-init}}, u_{\text{com}}, u_{\text{`joi'}}, \text{`joi'})$ and $\texttt{Hash}(u_{\text{`joi'}}, u_{\text{lbl}}, (G'.\text{groupCont}(), \text{lbl}))$ for each lbl ∈ {'conf', 'app', 'memb', 'init'}.
    - If $\textbf{gsd-exp}(u_{\text{par-init}}) = \texttt{false}$, $\mathcal{S}'_{4\text{-}2}$ prepares a new node $u_{\text{ctag}} := u_{\text{ctr}}$ (and sets ctr ← ctr + 1) and computes the confirmation tag by querying $\texttt{Hash}(u_{\text{`conf'}}, u_{\text{ctag}}, G'.\text{confTransHash})$ and $G'.\text{confTag} := \texttt{Corr}(u_{\text{ctag}})$. Then it checks the following.
      - If $(\bot, (\bot, \bot), *, G'.\text{confTransHash}, \text{confTag}') \in L_{\text{`epoch'}}$ exists for some confTag′ such that $G'.\text{confTag} = \text{confTag}'$, $\mathcal{S}'_{4\text{-}2}$ aborts. We call this event $\text{Abort}_{\text{attach}}$, and in Proposition E.26 we will prove that the probability the simulator aborts due to this event is negligible.[22]
      - Else, the simulator $\mathcal{S}'_{4\text{-}2}$ sets $G'.\text{comSecret} := (s_{\text{com}}, u_{\text{com}})$, $G'.\text{joinerSecret} := (\bot, u_{\text{`joi'}})$, $G'.\text{confKey} := (\bot, u_{\text{`conf'}})$, $G'.\text{appSecret} := (\bot, u_{\text{`app'}})$, $G'.\text{membKey} := (\bot, u_{\text{`memb'}})$ and $G'.\text{initSecret} := (\bot, u_{\text{`init'}})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`epoch'}} \xleftarrow{+} (u_{\text{par-init}}, G'.\text{comSecret}, G'.\text{joinerSecret}, G'.\text{confTransHash}, G'.\text{confTag})$.
    - If $\textbf{gsd-exp}(u_{\text{par-init}}) = \texttt{true}$ (i.e., $\mathcal{S}'_{4\text{-}2}$ knows $s_{\text{par-init}} \neq \bot$), then $\mathcal{S}'_{4\text{-}2}$ computes the joiner secret $s_{\text{`joi'}} := \text{RO}(s_{\text{par-init}}, s_{\text{com}}, \text{`joi'})$ and the epoch secrets $s_{\text{lbl}} := \text{RO}(s_{\text{`joi'}}, (G'.\text{groupCont}(), \text{lbl}))$ for each lbl ∈ {'conf', 'app', 'memb', 'init'}. $\mathcal{S}'_{4\text{-}2}$ sets $G'.\text{comSecret} := (s_{\text{com}}, u_{\text{com}})$, $G'.\text{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G'.\text{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G'.\text{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G'.\text{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$ and $G'.\text{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$. It also computes the confirmation tag $G'.\text{confTag} := \text{RO}(s_{\text{`conf'}}, G'.\text{confTransHash})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`epoch'}} \xleftarrow{+} (u_{\text{par-init}}, G'.\text{comSecret}, G'.\text{joinerSecret}, G'.\text{confTransHash}, G'.\text{confTag})$.

If a new member exists, $\mathcal{S}'_{4\text{-}2}$ runs `*welcome-msg` following the protocol description except that $\mathcal{S}'_{4\text{-}2}$ encrypts the joiner secret $G'.\text{joinerSecret}$ as follows:

- If RandCor[id] = 'good', $\mathcal{S}'_{4\text{-}2}$ does as follows. Let $G'.\text{member}$ be the new membership list and addedMem be the set of new party's identity.
  - If $G'.\text{member}[\text{id}].\text{dk} \neq (\bot, \bot)$ for all id ∈ addedMem, each $G'.\text{member}[\text{id}].\text{dk}$ is of the form $(*, u_{\text{id}})$. $\mathcal{S}'_{4\text{-}2}$ composes the set $S_{\text{addedMem}} := \{ u_{\text{id}} \}_{\text{id} \in \text{addedMem}}$, and queries $\texttt{CmEnc}(S_{\text{addedMem}}, u_{\text{`joi'}})$ to compute the ciphertext $(\text{T}, \vec{\text{ct}} = (\text{ct}_u)_{u \in S_{\text{addedMem}}})$.
    - If $\textbf{gsd-exp}(u_{\text{id}}) = \texttt{true}$ for some $u_{\text{id}} \in S_{\text{addedMem}}$, it decrypts $(\text{T}, \text{ct}_{u_{\text{id}}})$ using the secret s_{id} of u_{id}, and obtains $s_{\text{`joi'}}$. $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`enc'}} \xleftarrow{+} ((s_{\text{`joi'}}, u_{\text{`joi'}}), u_{\text{id}}, (\text{T}, \text{ct}_{u_{\text{id}}}))$ for each $u_{\text{id}} \in S_{\text{addedMem}}$. Then, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\text{lbl}} := \text{RO}(s_{\text{`joi'}}, (G'.\text{groupCont}(), \text{lbl}))$ for each lbl ∈ {'conf', 'app', 'memb', 'init'} and updates the epoch secrets as $G'.\text{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G'.\text{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G'.\text{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G'.\text{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$ and $G'.\text{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$. ($L_{\text{`epoch'}}$ is also updated accordingly.)
    - Else, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`enc'}} \xleftarrow{+} (G'.\text{joinerSecret}, u_{\text{id}}, (\text{T}, \text{ct}_{u_{\text{id}}}))$ for each $u_{\text{id}} \in S_{\text{addedMem}}$.
  - Else (i.e., $G'.\text{member}[\text{id}].\text{dk} = (\bot, \bot)$ for some id ∈ addedMem), $\mathcal{S}'_{4\text{-}2}$ queries $s_{\text{`joi'}} := \texttt{Corr}(u_{\text{`joi'}})$ and encrypts $s_{\text{`joi'}}$ as in the previous hybrid. Then, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\text{lbl}} := \text{RO}(s_{\text{`joi'}}, (G'.\text{groupCont}(), \text{lbl}))$ for each lbl ∈ {'conf', 'app', 'memb', 'init'} and updates the epoch secrets as $G'.\text{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G'.\text{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G'.\text{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G'.\text{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$ and $G'.\text{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$. ($L_{\text{`epoch'}}$ is also updated accordingly.)
- If RandCor[id] = 'bad', $\mathcal{S}'_{4\text{-}2}$ queries $s_{\text{`joi'}} := \texttt{Corr}(u_{\text{`joi'}})$ and encrypts $s_{\text{`joi'}}$ as in the previous hybrid. Then, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\text{lbl}} := \text{RO}(s_{\text{`joi'}}, (G'.\text{groupCont}(), \text{lbl}))$ for each lbl ∈ {'conf', 'app', 'memb', 'init'} and updates the epoch secrets as $G'.\text{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G'.\text{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G'.\text{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G'.\text{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$ and $G'.\text{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$. ($L_{\text{`epoch'}}$ is also updated accordingly.)

Simulation of id **on input** (**Process**, $c_0, \widehat{c}, \vec{p}$)**.** $\mathcal{S}'_{4\text{-}2}$ simulates the process protocol as in the previous hybrid, except for the differences shown below:

Firstly, $\mathcal{S}'_{4\text{-}2}$ simulates `*unframe-prop` and `*apply-props` identically to when simulating the `Commit` protocol. After the execution of `*apply-props`, $\mathcal{S}'_{4\text{-}2}$ obtains the new membership list and the decryption key of each party. In other words, for all members id, $G.\text{member}[\text{id}].\text{dk}$ is of the form $(*, u_{\text{id}})$ or $(\bot, \bot)$.

During execution of `*apply-rekey`, $\mathcal{S}'_{4\text{-}2}$ decrypts the ciphertext $\text{ct} = (\text{T}, \widehat{\text{ct}})$ in $(c_0, \widehat{c})$ as follows, by using id's current decryption key $(s_{\text{id}}, u_{\text{id}}) = G.\text{member}[\text{id}].\text{dk}$[23]. Let $(*, u_{\text{par-init}})$ be the initial secret at epoch Ptr[id] and let confTag be the confirmation tag in c_0. Note that the confirmation hash $G'.\text{confTransHash}$ for the new epoch is computed before running `*apply-rekey` function.

Case (1): If $s_{\text{id}} \neq \bot$, $\mathcal{S}'_{4\text{-}2}$ simply decrypts ct by itself to obtain the commit secret s_{com}.

[21] Due to the argument we made in Hybrid 2-2, confTag is unique for each initSecret, comSecret and confTransHash.

[22] $\text{Abort}_{\text{attach}}$ occurs if the created commit node by the commit protocol is attached to an existing detached root even if the corresponding initial secret is not leaked. If such node is created, the **mac-inj-allowed** predicate returns false. Namely, we prove in Proposition E.26 that the probability of such node is created is negligible.

[23] u_{id} is always non-⊥ because id uses the decryption key generated by itself: When it joins the group, it fetches the CmPKE key registered by itself (cf. `get-dks` queries to $\mathcal{F}_{\text{KS}}$ (line 12 in the join protocol)); When it updates its state, it replaces the old CmPKE key with a new CmPKE key generated by itself which stored in pendUpd (cf. line 11 in `*apply-props` function) or pendCom array (cf. line 4 in the process protocol).

Case (2): Else, if $(*, u_{\mathsf{id}}, \mathsf{ct}) \notin L_{\text{`enc'}}$ (i.e., ct can be sent to CmDec oracle), $\mathcal{S}'_{4\text{-}2}$ sends $(u_{\mathsf{id}}, \mathsf{ct})$ to CmDec oracle to obtain the commit secret s_{com}.

Case (3): Else, if $((s', u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ for some $(s' \neq \bot, u')$, s' is used as the commit secret s_{com} (i.e., $s_{\mathsf{com}} := s'$). $\mathcal{S}'_{4\text{-}2}$ retrieves such s_{com}.

Case (4)[24]**:** Else, if $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ and $(u_{\mathsf{par\text{-}init}}, (\bot, u'), *, G'.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{`epoch'}}$ exist for some u' and $\mathsf{confTag}'$, then $\mathcal{S}'_{4\text{-}2}$ skips running *derive-keys, and verifies the confirmation tag confTag in c_0 by checking $\mathsf{confTag} = \mathsf{confTag}'$.

Case (5)[25]**:** Else, if $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ for some u', $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$. In Proposition E.23, we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$ occurs. Thus, $\mathcal{Z}$'s view is indistinguishable.

It remains to explain what $\mathcal{S}'_{4\text{-}2}$ does with the commit secret s_{com} for Cases (1)-(3). The simulator finishes by funning *derive-keys and *vrf-conf-tag as follows.

- If $(u_{\mathsf{par\text{-}init}}, (s_{\mathsf{com}}, *), *, G'.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{`epoch'}}$ exists for some $\mathsf{confTag}'$, $\mathcal{S}'_{4\text{-}2}$ verifies the confirmation tag by checking $\mathsf{confTag} = \mathsf{confTag}'$ and skips *derive-keys and *vrf-conf-tag. Note that this case occurs if $\mathcal{S}'_{4\text{-}2}$ derived the epoch secret corresponding to the received commit message c_0.
- Else if $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathsf{false}$, $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$. In Proposition E.24 below, we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$ occurs, i.e., $\mathcal{Z}$'s view is not changed.
- Else (i.e., $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathsf{true}$), $\mathcal{S}'_{4\text{-}2}$ knows both $s_{\mathsf{par\text{-}init}}$ and s_{com}. It computes the joiner secret $s_{\text{`joi'}} := \mathsf{RO}(s_{\mathsf{par\text{-}init}}, s_{\mathsf{com}}, \text{`joi'})$ and the epoch secrets $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{`joi'}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{`conf'}, \text{`app'}, \text{`memb'}, \text{`init'}\}$. Then, $\mathcal{S}'_{4\text{-}2}$ recomputes the confirmation tag $\mathsf{confTag}' := \mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ and checks whether $\mathsf{confTag} = \mathsf{confTag}'$ holds. If true, $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes u_{com}, $u_{\text{`joi'}}$, $u_{\text{`conf'}}$, $u_{\text{`app'}}$, $u_{\text{`memb'}}$, and $u_{\text{`init'}}$ (their values are set to $u_{\mathsf{ctr}}, u_{\mathsf{ctr+1}}, \ldots, u_{\mathsf{ctr+5}}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 6$) and queries Set-Secret$(u_{\mathsf{com}}, s_{\mathsf{com}})$, Join-Hash$(u_{\mathsf{par\text{-}init}}, u_{\mathsf{com}}, u_{\text{`joi'}}, \text{`joi'})$ and $\mathsf{Hash}(u_{\text{`joi'}}, u_{\mathsf{lbl}}, (G'.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{`conf'}, \text{`app'}, \text{`memb'}, \text{`init'}\}$. $\mathcal{S}'_{4\text{-}2}$ sets $G'.\mathsf{comSecret} := (s_{\mathsf{com}}, u_{\mathsf{com}})$, $G'.\mathsf{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G'.\mathsf{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G'.\mathsf{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G'.\mathsf{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$ and $G'.\mathsf{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`epoch'}} \xleftarrow{+} (u_{\mathsf{par\text{-}init}}, G'.\mathsf{comSecret}, G'.\mathsf{joinerSecret}, G'.\mathsf{confTransHash}, \mathsf{confTag})$.

Simulation of id **on input** $(\mathtt{Join}, w_0, \widehat{w})$**.** $\mathcal{S}'_{4\text{-}2}$ simulates the join protocol as in the previous hybrid, except for the differences shown below:

When initializing the group membership list, for each encryption key in $G.\mathsf{member}$, if $\mathcal{S}'_{4\text{-}2}$ knows the corresponding decryption key, $\mathcal{S}'_{4\text{-}2}$ copies it to $G.\mathsf{member}$. Otherwise, the decryption key is set to $(\bot, \bot)$.

$\mathcal{S}'_{4\text{-}2}$ decrypts the ciphertext $\mathsf{ct} = (\mathsf{T}, \widehat{\mathsf{ct}})$ in $(w_0, \widehat{w})$ using id's decryption key $(s_{\mathsf{id}}, u_{\mathsf{id}}) = G.\mathsf{member}[\mathsf{id}].\mathsf{dk}$[26] as follows. Let $G.\mathsf{confTransHash}$ be the confirmation hash and confTag be the confirmation tag included in w_0.

Case (1): If $s_{\mathsf{id}} \neq \bot$, $\mathcal{S}'_{4\text{-}2}$ simply decrypts ct by itself to obtain the joiner secret $s_{\text{`joi'}}$.

Case (2): Else, if $(*, u_{\mathsf{id}}, \mathsf{ct}) \notin L_{\text{`enc'}}$ (i.e., ct can be sent to CmDec oracle), $\mathcal{S}'_{4\text{-}2}$ sends $(u_{\mathsf{id}}, \mathsf{ct})$ to obtain the joiner secret $s_{\text{`joi'}}$.

Case (3): Else, if $((s', u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ for some $(s' \neq \bot, u')$, $s'$ is used as the joiner secret $s_{\text{`joi'}}$ (i.e., $s_{\text{`joi'}} := s'$). $\mathcal{S}'_{4\text{-}2}$ retrieves such $s_{\text{`joi'}}$.

Case (4)[27]**:** Else, if $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ and $(*, *, (\bot, u'), G.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{`epoch'}}$ exist for some u' and $\mathsf{confTag}'$, then $\mathcal{S}'_{4\text{-}2}$ skips the derivation of the epoch secrets, and verifies the confirmation tag confTag in w_0 by checking $\mathsf{confTag} = \mathsf{confTag}'$.

Case (5)[28]**:** If $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$ for some u', $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$. We call this event $\mathsf{E}_{\mathsf{inj\text{-}w}}$. In Proposition E.25, we will prove that the simulator in the previous hybrid also outputs $\bot$ when $\mathsf{E}_{\mathsf{inj\text{-}w}}$ occurs, i.e., $\mathcal{Z}$'s view is indistinguishable.

It remains to explain what $\mathcal{S}'_{4\text{-}2}$ does with the commit secret $s_{\text{`joi'}}$ for Cases (1)-(3). The simulator verifies confTag in w_0 as follows.

- If $(*, *, (s_{\text{`joi'}}, *), G.\mathsf{confTransHash}, \mathsf{confTag}') \in L_{\text{`epoch'}}$ exists for some $\mathsf{confTag}'$, $\mathcal{S}'_{4\text{-}2}$ verifies the confirmation tag by checking $\mathsf{confTag} = \mathsf{confTag}'$.
- Else, $\mathcal{S}'_{4\text{-}2}$ computes $s_{\mathsf{lbl}} := \mathsf{RO}(s_{\text{`joi'}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{`conf'}, \text{`app'}, \text{`memb'}, \text{`init'}\}$, and verifies the confirmation tag by checking $\mathsf{confTag} = \mathsf{RO}(s_{\text{`conf'}}, G.\mathsf{confTransHash})$. If the tag is valid, $\mathcal{S}'_{4\text{-}2}$ prepares new GSD nodes $u_{\text{`joi'}}$, $u_{\text{`conf'}}$, $u_{\text{`app'}}$, $u_{\text{`memb'}}$, and $u_{\text{`init'}}$ (their values are set to $u_{\mathsf{ctr}}, u_{\mathsf{ctr+1}}, \ldots, u_{\mathsf{ctr+4}}$ and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 5$) and queries Set-Secret$(u_{\text{`joi'}}, s_{\text{`joi'}})$ and $\mathsf{Hash}(u_{\text{`joi'}}, u_{\mathsf{lbl}}, (G.\mathsf{groupCont}(), \mathsf{lbl}))$ for each $\mathsf{lbl} \in \{\text{`conf'}, \text{`app'}, \text{`memb'}, \text{`init'}\}$. Then, $\mathcal{S}'_{4\text{-}2}$ sets $G.\mathsf{joinerSecret} := (s_{\text{`joi'}}, u_{\text{`joi'}})$, $G.\mathsf{confKey} := (s_{\text{`conf'}}, u_{\text{`conf'}})$, $G.\mathsf{appSecret} := (s_{\text{`app'}}, u_{\text{`app'}})$, $G.\mathsf{membKey} := (s_{\text{`memb'}}, u_{\text{`memb'}})$ and $G.\mathsf{initSecret} := (s_{\text{`init'}}, u_{\text{`init'}})$. Finally, $\mathcal{S}'_{4\text{-}2}$ stores $L_{\text{`epoch'}} \xleftarrow{+} (\bot, (\bot, \bot), G.\mathsf{joinerSecret}, G.\mathsf{confTransHash}, \mathsf{confTag})$.

Simulation of id **on input** Key**.** Let $G.\mathsf{appSecret}$ be the application secret at epoch Ptr[id]. If it is of the from $(s_{\text{`app'}} \neq \bot, u_{\text{`app'}})$, $\mathcal{S}'_{4\text{-}2}$ returns $s_{\text{`app'}}$. Else, $\mathcal{S}'_{4\text{-}2}$ queries $s_{\text{`app'}} := \mathsf{Corr}(u_{\text{`app'}})$ and returns $s_{\text{`app'}}$; Then $\mathcal{S}'_{4\text{-}2}$ replaces $(\bot, u_{\text{`app'}})$ with $(s_{\text{`app'}}, u_{\text{`app'}})$

Corruption query for id**.** id's state at epoch Ptr[id] contains the following secrets:

- The decryption key stored in $G.\mathsf{member}[\mathsf{id}].\mathsf{dk}$ and $G.\mathsf{pendUpd}$ array;

[24] This case occurs if the received commit message is generated by the commit protocol with good randomness, and the encrypted commit secret has not been leaked.

[25] This case occurs when the ciphertext was generated by CmEnc oracle, but the simulator did not derive the corresponding epoch secret. Put differently, this occurs when $\mathcal{Z}$ generates a malicious welcome message using an honestly generated ciphertext.

[26] u_{id} is always non-$\bot$ because when a party id joins the group, id uses the decryption key generated by itself (cf. get-dks queries to $\mathcal{F}_{\mathsf{KS}}$ (line 12 in the join protocol)).

[27] This case occurs if the received welcome message is generated by the commit protocol with good randomness, and the encrypted joiner secret has not been leaked.

[28] This case occurs when the ciphertext was generated by CmEnc oracle, but the simulator did not derive the corresponding epoch secret. Put differently, it occurs when $\mathcal{Z}$ generates a malicious commit message using an honestly generated ciphertext.

- The decryption keys of the key packages registered to Key Service. They are stored in DK[id, ∗]; and
- The current epoch secrets confKey, membKey, appSecret, and initSecret. Note that id holds appSecret only if Key query has not been queried to Ptr[id].
- The epoch secrets stored in G.pendCom array;

For each of the above secrets, if the secret is of the form $(\perp, u)$, $\mathcal{S}'_{4\text{-}2}$ queries $s := \texttt{Corr}(u)$ and replaces $(\perp, u)$ with (s, u). Then, for each node v such that **gsd-exp**(v) becomes true due to the corruption, $\mathcal{S}'_{4\text{-}2}$ computes the secret s_v from previously obtained ciphertexts and corrupted secrets, and replaces all occurrence of $(\perp, v)$ with (s_v, v). Finally, $\mathcal{S}'_{4\text{-}2}$ returns id's secrets listed above to the adversary $\mathcal{A}$.

Part 2: Proof of Indistinguishability of the Two Hybrid. We next prove indistinguishability between Hybrid 4-1 and Hybrid 4-2.

Lemma E.21. *Hybrid 4-1 and Hybrid 4-2 are indistinguishable assuming* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. The difference from the previous hybrid is that the simulator $\mathcal{S}'_{4\text{-}2}$ always outputs $\perp$ when events $\mathsf{E}_{\text{inj-p}}$, $\mathsf{E}_{\text{inj-c-1}}$, $\mathsf{E}_{\text{inj-c-2}}$, and $\mathsf{E}_{\text{inj-w}}$ occur, and it aborts when event $\mathsf{Abort}_{\text{attach}}$ occurs. In the following, we show that, if $\mathcal{Z}$ can distinguish the two hybrids (i.e., $\mathcal{S}'_{4\text{-}2}$ provides a different view to $\mathcal{Z}$ when the events occur), then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game.

We first show that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\text{inj-p}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a proposal message without knowing the membership key.

Proposition E.22. *$\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-p}}$ occurs is indistinguishable between the two hybrids if* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\perp$ when event $\mathsf{E}_{\text{inj-p}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\text{GSD}}$. We assume $\mathcal{B}$ receives at most N distinct proposal messages as input to the commit and process protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [N]$ at random, and hopes that the i-th proposal message triggers $\mathsf{E}_{\text{inj-p}}$, and the simulator in the previous hybrid outputs non-$\perp$ when it receives the message. ($\mathcal{B}$ succeeds to guess with probability $1/N$.) When $\mathcal{B}$ receives the i-th proposal message p, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume p is processed by id without loss of generality, and let $u_{\text{`memb'}}$ be the GSD node assigned to the membership key at epoch Ptr[id]. Also, let membTag be the membership tag in p.

(1) $\mathcal{B}$ prepares a new GSD node $u_{\text{mtag}} := u_{\text{ctr}}$ and queries $\texttt{Hash}(u_{\text{`memb'}}, u_{\text{mtag}}, (\text{propCont}, \text{sig}))$, where (propCont, sig) is taken from p.
(2) $\mathcal{B}$ queries $\text{membTag}' := \texttt{Chall}(u_{\text{mtag}})$
(3) If membTag = membTag′, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can send (propCont, sig) to Hash oracle (that is, $\mathcal{B}$ has not queries (propCont, sig) to Hash oracle before) because $(u_{\text{`memb'}}, (\text{propCont}, \text{sig}), *) \notin L_{\text{`memb'}}$ when $\mathsf{E}_{\text{inj-p}}$ occurs. If $\mathcal{B}$ succeeds the guess, membTag in p is valid, i.e., it satisfies membTag = $\text{RO}(s_{\text{`memb'}}, (\text{propCont}, \text{sig}))$ for the membership key $s_{\text{`memb'}}$ assigned to $u_{\text{`memb'}}$. Moreover, $\mathcal{B}$ assigns $\text{RO}(s_{\text{`memb'}}, (\text{propCont}, \text{sig}))$ to u_{mtag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), membTag = membTag′ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We then check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{mtag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}\textbf{gsd-exp}(u_{\text{mtag}}) &= \big(u_{\text{mtag}} \in \text{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{`memb'}}) \\ &= \texttt{false}.\end{aligned}$$

This is because u_{mtag} is not sent to Corr or Set-Secret oracle, and $\textbf{gsd-exp}(u_{\text{`memb'}}) = \texttt{false}$ when $\mathsf{E}_{\text{inj-p}}$ occurs. Hence, u_{mtag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ wins the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds to guess the proposal message. If $\mathcal{Z}$ distinguish the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-p}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We then prove that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\text{inj-c-1}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a commit message without knowing the encrypted commit secret.

Proposition E.23. *$\mathcal{Z}$'s view when $E_{inj\text{-}c\text{-}1}$ occurs is indistinguishable between the two hybrids if* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\perp$ when event $\mathsf{E}_{\text{inj-c-1}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\text{GSD}}$. We assume $\mathcal{B}$ receives at most Q distinct commit messages as input to the process protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random, and hopes that the i-th commit message triggers $\mathsf{E}_{\text{inj-c-1}}$, and the simulator in the previous hybrid outputs non-$\perp$ when it receives the message. ($\mathcal{B}$ succeeds to guess with probability $1/Q$.) When $\mathcal{B}$ receives the i-th commit message c_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume c_0 is processed by id without loss of generality and let $u_{\text{par-init}}$ be the GSD node assigned to the initial secret at epoch Ptr[id]. Let confTag be the confirmation tag in c_0. In addition, when $\mathsf{E}_{\text{inj-c-1}}$ occurs, there

exists a node u' such that $((\bot, u'), u_{\mathsf{id}}, \mathsf{ct}) \in L_{\text{`enc'}}$, where u_{id} is the GSD node corresponding to id's current CmPKE key and ct is the ciphertext in c_0

(1) $\mathcal{B}$ prepares new GSD nodes $u_{\text{`joi'}}$ and $u_{\text{`conf'}}$ (their values are set to u_{ctr} and $u_{\mathsf{ctr}+1}$, and ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr} + 2$) and queries $\mathtt{Join\text{-}Hash}(u_{\mathsf{par\text{-}init}}, u', u_{\text{`joi'}}, \text{`joi'})$ and $\mathtt{Hash}(u_{\text{`joi'}}, u_{\text{`conf'}}, (G'.\mathsf{groupCont}(), \text{`conf'}))$.
(2) $\mathcal{B}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ and queries $\mathtt{Hash}(u_{\text{`conf'}}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$.
(3) $\mathcal{B}$ queries $\mathsf{confTag}' := \mathtt{Chall}(u_{\mathsf{ctag}})$.
(4) If $\mathsf{confTag} = \mathsf{confTag}'$, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can issue the above queries to oracles Hash and Join-Hash because the inputs are new GSD nodes. Note also that the confirmation key derived from the current initial secret and c_0 is correctly computed to $u_{\text{`conf'}}$ because the commit secret encrypted in ct is assigned to $u'$. If $\mathcal{B}$ succeeds the guess, confTag in $c_0$ is valid, i.e., it satisfies $\mathsf{confTag} = \mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ for the confirmation key $s_{\text{`conf'}}$ assigned to $u_{\text{`conf'}}$. Moreover, $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ to u_{ctag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\mathsf{confTag} = \mathsf{confTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{ctag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}
\mathbf{gsd\text{-}exp}(u_{\mathsf{ctag}}) &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{`conf'}}) \\
&= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{`joi'}}) \\
&= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee \\
&\qquad \big(\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) \wedge \mathbf{gsd\text{-}exp}(u')\big) \\
&= \mathtt{false}.
\end{aligned}$$

This is because u_{ctag}, $u_{\text{`conf'}}$, and $u_{\text{`joi'}}$ are not sent to Corr or Set-Secret oracle, and $\mathbf{gsd\text{-}exp}(u') = \mathtt{false}$ when $\mathsf{E}_{\text{inj-c-1}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds to guess the message. If $\mathcal{Z}$ distinguish the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-c-1}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We also prove that that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\text{inj-c-2}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a commit message without knowing the initial secret of the parent node.

Proposition E.24. *$\mathcal{Z}$'s view when $E_{inj\text{-}c\text{-}2}$ occurs is indistinguishable between the two hybrids if* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\bot$ when event $\mathsf{E}_{\text{inj-c-2}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}$ and $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\mathrm{GSD}}$. We assume $\mathcal{B}$ receives at most Q distinct commit messages as input to the process protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random, and hopes that the i-th commit message triggers $\mathsf{E}_{\text{inj-c-2}}$, and the simulator in the previous hybrid outputs non-$\bot$ when it receives the message. ($\mathcal{B}$ succeeds to guess with probability $1/Q$.) When $\mathcal{B}$ receives the i-th commit message c_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume c_0 is processed by id without loss of generality and let $u_{\mathsf{par\text{-}init}}$ be the GSD node assigned to the initial secret at epoch $\mathsf{Ptr}[\mathsf{id}]$. Let confTag be the confirmation tag in c_0. Note that when $\mathsf{E}_{\text{inj-c-2}}$ occurs, $\mathcal{B}$ succeeds to decrypt the commit secret s_{com}.

(1) $\mathcal{B}$ prepares new GSD nodes u_{com}, $u_{\text{`joi'}}$ and $u_{\text{`conf'}}$ (their values are set to u_{ctr}, $u_{\mathsf{ctr}+1}$ and $u_{\mathsf{ctr}+2}$. ctr is incremented as $\mathsf{ctr} \leftarrow \mathsf{ctr}+3$) and queries $\mathtt{Set\text{-}Secret}(u_{\mathsf{com}}, s_{\mathsf{com}})$, $\mathtt{Join\text{-}Hash}(u_{\mathsf{par\text{-}init}}, u_{\mathsf{com}}, u_{\text{`joi'}}, \text{`joi'})$ and $\mathtt{Hash}(u_{\text{`joi'}}, u_{\text{`conf'}}, (G'.\mathsf{groupCont}(), \text{`conf'}))$.
(2) $\mathcal{B}$ prepares a new GSD node $u_{\mathsf{ctag}} := u_{\mathsf{ctr}}$ and queries $\mathtt{Hash}(u_{\text{`conf'}}, u_{\mathsf{ctag}}, G'.\mathsf{confTransHash})$.
(3) $\mathcal{B}$ queries $\mathsf{confTag}' := \mathtt{Chall}(u_{\mathsf{ctag}})$.
(4) If $\mathsf{confTag} = \mathsf{confTag}'$, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can issue above queries because $\mathcal{B}$ prepares new GSD nodes. Note also that the confirmation key derived from the current initial secret and c_0 is correctly computed to $u_{\text{`conf'}}$. If $\mathcal{B}$ succeeds to guess, confTag in $c_0$ is valid, i.e., it satisfies $\mathsf{confTag} = \mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ for the confirmation key $s_{\text{`conf'}}$ assigned to $u_{\text{`conf'}}$. Moreover, $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ to u_{ctag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\mathsf{confTag} = \mathsf{confTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{ctag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}
\mathbf{gsd\text{-}exp}(u_{\mathsf{ctag}}) &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{`conf'}}) \\
&= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \mathbf{gsd\text{-}exp}(u_{\text{`joi'}}) \\
&= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee \\
&\qquad \big(\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) \wedge \mathbf{gsd\text{-}exp}(u_{\mathsf{com}})\big) \\
&= \mathtt{false}.
\end{aligned}$$

This is because u_{ctag}, $u_{\text{`conf'}}$, $u_{\text{`joi'}}$ are not sent to Corr or Set-Secret oracle, and $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{false}$ when $\mathsf{E}_{\text{inj-c}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds to guess the message. If $\mathcal{Z}$ distinguish the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE

conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-c-2}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We then prove that $\mathcal{Z}$'s view is not changed when $\mathsf{E}_{\text{inj-w}}$ occurs. In other words, $\mathcal{Z}$ cannot inject a welcome message without knowing the encrypted joiner secret.

Proposition E.25. *$\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-w}}$ occurs is indistinguishable between the two hybrids if* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. The difference from the previous hybrid is that $\mathcal{S}'_{4\text{-}2}$ outputs $\perp$ when event $\mathsf{E}_{\text{inj-w}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\text{GSD}}$. We assume $\mathcal{B}$ receives at most N distinct welcome messages as input to the join protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [N]$ at random, and hopes that the i-th welcome message triggers $\mathsf{E}_{\text{inj-w}}$, and the simulator in the previous hybrid outputs non-$\perp$ when it receives the message (i.e., the message is valid). ($\mathcal{B}$ succeeds to guess with probability $1/N$.) When $\mathcal{B}$ receives the i-th welcome message w_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume w_0 is processed by id without loss of generality and let confTag be the confirmation tag in w_0. In addition, when $\mathsf{E}_{\text{inj-w}}$ occurs, there exists a node u' such that $((\perp, u'), u_{\text{id}}, \text{ct}) \in L_{\text{`enc'}}$, where u_{id} is the GSD node corresponding to id's current CmPKE key and ct is the ciphertext in w_0

(1) $\mathcal{B}$ prepares new GSD nodes $u_{\text{`conf'}} := u_{\text{ctr}}$ (ctr is incremented) and queries $\mathsf{Hash}(u', u_{\text{`conf'}}, (G.\mathsf{groupCont}(), \text{`conf'}))$.
(2) $\mathcal{B}$ prepares a new GSD node $u_{\text{ctag}} := u_{\text{ctr}}$ and queries $\mathsf{Hash}(u_{\text{`conf'}}, u_{\text{ctag}}, G'.\mathsf{confTransHash})$.
(3) $\mathcal{B}$ queries $\text{confTag}' := \mathtt{Chall}(u_{\text{ctag}})$.
(4) If confTag = confTag′, $\mathcal{B}$ submits 0; otherwise submits 1.

Note that $\mathcal{B}$ can issue above queries because $\mathcal{B}$ prepares new GSD nodes. Note also that the confirmation key derived from w_0 is correctly computed to $u_{\text{`conf'}}$ because the joiner secret encrypted in ct is assigned to $u'$. If $\mathcal{B}$ succeeds the guess, confTag in $w_0$ is valid, i.e., it satisfies $\text{confTag} = \mathsf{RO}(s_{\text{`conf'}}, G.\mathsf{confTransHash})$ for the confirmation key $s_{\text{`conf'}}$ assigned to $u_{\text{`conf'}}$. Moreover, $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{`conf'}}, G.\mathsf{confTransHash})$ to u_{ctag}. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\text{confTag} = \text{confTag}'$ holds with probability 1; otherwise with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that the GSD graph is acyclic and u_{ctag} is a sink node. In addition, we have

$$\begin{aligned}\textbf{gsd-exp}(u_{\text{ctag}}) &= (u_{\text{ctag}} \in \text{Corr}) \vee \textbf{gsd-exp}(u_{\text{`conf'}}) \\ &= (u_{\text{ctag}} \in \text{Corr}) \vee (u_{\text{`conf'}} \in \text{Corr}) \vee \textbf{gsd-exp}(u') \\ &= \mathtt{false}.\end{aligned}$$

This is because u_{ctag} and $u_{\text{`conf'}}$ are not sent to `Corr` or `Set-Secret` oracle, and $\textbf{gsd-exp}(u') = \mathtt{false}$ when $\mathsf{E}_{\text{inj-w}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds the guess. If $\mathcal{Z}$ distinguish the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that $\mathcal{Z}$'s view when $\mathsf{E}_{\text{inj-w}}$ occurs is indistinguishable between Hybrid 4-1 and Hybrid 4-2. □

We finally prove that the probability the simulator aborts due to $\text{Abort}_{\text{attach}}$ is negligible.

Proposition E.26. *The probability $\text{Abort}_{\text{attach}}$ occurs is negligible if* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. The difference from the previous hybrid is $\mathcal{S}'_{4\text{-}2}$ aborts when event $\text{Abort}_{\text{attach}}$ occurs. We show that, if $\mathcal{Z}$ can distinguish the two hybrids, then there exists an adversary $\mathcal{B}$ that can win the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$ and how $\mathcal{B}$ embeds the GSD challenge. We then show the validity of the GSD challenge, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\text{GSD}}$. We assume $\mathcal{Z}$ generates at most Q commit message by invoking the commit protocol. At the beginning of the game, $\mathcal{B}$ chooses $i \in [Q]$ at random, and hopes that $\text{Abort}_{\text{attach}}$ occurs while $\mathcal{B}$ generates the i-th commit message. ($\mathcal{B}$ succeeds to guess such message with probability $1/Q$.) When $\mathcal{B}$ generates the i-th commit message c_0, $\mathcal{B}$ embeds the GSD challenge and determines the challenge bit as follows. We assume c_0 is generated by id without loss of generality. Let $u_{\text{par-init}}$ be the GSD node assigned to the initial secret at epoch `Ptr[id]` and let u_{com} be the GSD node assigned to the commit secret. Note that the confirmation hash $G'.\mathsf{confTransHash}$ of new epoch is computed before deriving epoch secret.

(1) $\mathcal{B}$ prepares new GSD nodes $u_{\text{`joi'}}$ and $u_{\text{`conf'}}$ (their values are set to u_{ctr} and $u_{\text{ctr+1}}$. ctr is incremented.) and queries $\mathtt{Join\text{-}Hash}(u_{\text{par-init}}, u_{\text{com}}, u_{\text{`joi'}}, \text{`joi'})$ and $\mathsf{Hash}(u_{\text{`joi'}}, u_{\text{`conf'}}, (G'.\mathsf{groupCont}(), \text{`conf'}))$.
(2) $\mathcal{B}$ prepares a new GSD node $u_{\text{ctag}} := u_{\text{ctr}}$ and queries $\mathsf{Hash}(u_{\text{`conf'}}, u_{\text{ctag}}, G'.\mathsf{confTransHash})$.
(3) $\mathcal{B}$ queries $\text{confTag} := \mathtt{Chall}(u_{\text{ctag}})$ instead of $\mathtt{Corr}(u_{\text{ctag}})$.
(4) If $(\perp, (\perp, \perp), *, G'.\mathsf{confTransHash}, \text{confTag}') \in L_{\text{`epoch'}}$ exists for some confTag′ such that $\text{confTag} = \text{confTag}'$, $\mathcal{B}$ submits 0 to the GSD challenger; otherwise submits 1.

Note that $\mathcal{B}$ can issue the above queries to oracles `Hash` and `Join-Hash` because the inputs are new GSD nodes. Note also that the confirmation key derived from the current initial secret and u_{com} is correctly computed to $u_{\text{`conf'}}$. $\mathcal{B}$ assigns $\mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$ to u_{ctag}. If $\text{Abort}_{\text{attach}}$ occurs, $\mathcal{B}$ has obtained confTag′ (stored in $L_{\text{`epoch'}}$) that satisfies $\text{confTag}' = \mathsf{RO}(s_{\text{`conf'}}, G'.\mathsf{confTransHash})$. Thus, if the challenge oracle returns the real value (i.e., the challenge bit is 0), $\text{confTag} = \text{confTag}'$ holds with probability 1; otherwise

with negligible probability. Therefore, $\mathcal{B}$ can output the correct challenge bit with overwhelming probability.

We check the validity of the GSD challenge. Observe that GSD graph is acyclic and u_{ctag} is a sink node. In addition, by the structure of the GSD graph, we have

$$\begin{aligned}\textbf{gsd-exp}(u_{\mathsf{ctag}}) &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{`conf'}}) \\ &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{`joi'}}) \\ &= \big(u_{\mathsf{ctag}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`conf'}} \in \mathsf{Corr}\big) \vee \big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee \\ &\quad (\textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}) \wedge \textbf{gsd-exp}(u_{\mathsf{com}})) \\ &= \mathtt{false}.\end{aligned}$$

This is because u_{ctag}, $u_{\text{`conf'}}$, $u_{\text{`joi'}}$ are not sent to `Corr` or `Set-Secret` oracle, and $\textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{false}$ when $\mathsf{Abort}_{\mathsf{attach}}$ occurs. Hence, u_{ctag} is a valid challenge node, and $\mathcal{B}$ wins the GSD game.

We finally evaluate the advantage of $\mathcal{B}$. $\mathcal{B}$ can win the GSD game if $\mathcal{Z}$ distinguishes the hybrids and $\mathcal{B}$ succeeds the guess. If $\mathcal{Z}$ distinguish the two hybrids with non-negligible probability ϵ, $\mathcal{B}$ wins the game with probability ϵ/Q, which is also non-negligible. However, This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, ϵ must be negligible, and we conclude that the probability $\mathsf{Abort}_{\mathsf{attach}}$ occurs is negligible, i.e., the two hybrids are indistinguishable for $\mathcal{Z}$. □

From the above propositions, we conclude that Hybrid 4-1 and Hybrid 4-2 are indistinguishable for $\mathcal{Z}$. □

E.5.3 From Hybrid 4-3 to 4-4: Proof of Lem. E.28. Before proving Lem. E.28, we provide the key proposition that establishes the relationship between the safety predicate (**safe** and **know** shown in Fig. 28) and the **gsd-exp** predicate. It will be used in the proof of Lems. E.28 and E.30.

Proposition E.27. *Let $u_{\text{`app'}}, u_{\text{`memb'}}, u_{\text{`conf'}}, u_{\text{`init'}}$ be the GSD node assigned to each epoch secret at epoch c_0. The following statements hold.*

- *If* $\textbf{\textit{know}}(c_0, \text{`epoch'}) = \mathtt{false}$, *then* $\textbf{\textit{gsd-exp}}(u) = \mathtt{false}$ *for* $u \in \{ u_{\text{`memb'}}, u_{\text{`conf'}}, u_{\text{`init'}} \}$.
- *If* $\textbf{\textit{safe}}(c_0) = \mathtt{true}$, *then* $\textbf{\textit{gsd-exp}}(u_{\text{`app'}}) = \mathtt{false}$.

Proof. We show the contraposition of the statements. That is, we prove

- If $\textbf{gsd-exp}(u) = \mathtt{true}$ for $u \in \{ u_{\text{`memb'}}, u_{\text{`conf'}}, u_{\text{`init'}} \}$, then $\textbf{know}(c_0, \text{`epoch'}) = \mathtt{true}$, and
- If $\textbf{gsd-exp}(u_{\text{`app'}}) = \mathtt{true}$, then $\textbf{safe}(c_0) = \mathtt{false}$.

Recalling the definition of **gsd-exp** and the GSD graph created by $\mathcal{S}'_{4\text{-}2}$, for each $\hat{u} \in \{ u_{\text{`memb'}}, u_{\text{`conf'}}, u_{\text{`init'}}, u_{\text{`app'}} \}$, we have

$$\begin{aligned}&\textbf{gsd-exp}(\hat{u}) \\ &\iff \big(\hat{u} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\text{`joi'}}) \\ &\iff \big(\hat{u} \in \mathsf{Corr}\big) \vee \big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee \textbf{gsd-exp}(u_{\mathsf{id},\text{`joi'}}) \\ &\quad \vee \big(\textbf{gsd-exp}(u_{\mathsf{com}}) \wedge \textbf{gsd-exp}(u_{\mathsf{par\text{-}init}})\big) \\ &\iff \big(\hat{u} \in \mathsf{Corr}\big) \vee \big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee \big(u_{\mathsf{id},\text{`joi'}} \in \mathsf{Corr}\big) \\ &\quad \vee \big(\big((u_{\mathsf{com}} \in \mathsf{Corr}) \vee \textbf{gsd-exp}(u_{\mathsf{id},\text{`com'}})\big) \wedge \textbf{gsd-exp}(u_{\mathsf{par\text{-}init}})\big) \\ &\iff \hat{u} \in \mathsf{Corr} \cdots \text{ Case (A)} \\ &\quad \vee \big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee \big(u_{\mathsf{id},\text{`joi'}} \in \mathsf{Corr}\big) \cdots \text{ Case (B)} \\ &\quad \vee \big((u_{\mathsf{com}} \in \mathsf{Corr}) \vee (u_{\mathsf{id},\text{`com'}} \in \mathsf{Corr})\big) \wedge \textbf{gsd-exp}(u_{\mathsf{par\text{-}init}}), \cdots \text{ Case (C)}\end{aligned}$$

where $u_{\text{`joi'}}$ (resp. $u_{\mathsf{com}}$) is the GSD node assigned to the joiner (resp. commit) secret at epoch $c_0$, $u_{\mathsf{id},\text{`joi'}}$ (resp. $u_{\mathsf{id},\text{`com'}}$) is the GSD node assigned to an encryption key used to encrypt $u_{\text{`joi'}}$ (resp. u_{com}) at c_0, and $u_{\mathsf{par\text{-}init}}$ is the GSD node assigned to the initial secret at the parent node of c_0. Note that Case (C) is evaluated when c_0 is a non-root node because otherwise the GSD graph starts from $u_{\text{`joi'}}$. In the following, we analyze Case (A), Case (B), and Case (C) in order.

[Case (A): $\big(\hat{u} \in \mathsf{Corr}\big) = \mathtt{true}$]. $\hat{u} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Corr}(\hat{u})$. (Note that $\mathcal{S}'_{4\text{-}2}$ never queries $\mathtt{Set\text{-}Secret}(\hat{u}, *)$.) By the description of $\mathcal{S}'_{4\text{-}2}$, it queries $\mathtt{Corr}(\hat{u})$ if (1) a party at c_0 is corrupted via Corruption query; or (2) the committer of c_0 is corrupted via Corruption query while it is at the parent node of c_0 (this corresponds to the fact that the committer stores pending commits in pendCom array).

Case (A-1): It immediately implies $\textbf{know}(c_0, \text{`epoch'}) = \mathtt{true}$ because $\mathsf{Node}[c_0].\mathsf{exp} \neq \emptyset$ becomes true when a party at $c_0$ is corrupted. In particular, if the party is corrupted before `Key` query is issued to the epoch c_0, $(*, \mathtt{true}) \in \mathsf{Node}[c_0].\mathsf{exp}$ becomes true. This implies $\textbf{safe}(c_0) = \mathtt{false}$.

Case (A-2): When the committer is corrupted while it is at the parent node of c_0, the status of the node c_0 is set to 'bad' (cf. *update-stat-after-exp). This implies $\textbf{know}(c_0, \text{`epoch'}) = \mathtt{true}$ and $\textbf{safe}(c_0) = \mathtt{false}$ due to Condition (a) of ***secrets-injected**.

[Case (B): $\big(u_{\text{`joi'}} \in \mathsf{Corr}\big) \vee \big(u_{\mathsf{id},\text{`joi'}} \in \mathsf{Corr}\big) = \mathtt{true}$]. First, $u_{\text{`joi'}} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Corr}(u_{\text{`joi'}})$ or $\mathtt{Set\text{-}Secret}(u_{\text{`joi'}}, *)$.

Recalling the description of $\mathcal{S}'_{4\text{-}2}$, it issues $\mathtt{Corr}(u_{\text{`joi'}})$ if (1) in the commit protocol, a new member is added with a malicious key package generated by $\mathcal{A}$; or (2) in the commit protocol, the committer encrypts the joiner secret at c_0 using bad randomness.

Case (B-1): When the adversary succeeds to inject a malicious key package, the signing key used to generate the key package must be exposed due to the modification we made in Hybrid 4-1. Thus, due to Condition (a) of ***can-traverse**, $^*\textbf{can-traverse}(c_0) = \mathtt{true}$.

Case (B-2): If the committer encrypts the joiner secret, $\mathsf{Node}[c_0].\mathsf{pro}$ contains at least one add proposal. In addition, the status of c_0 is set to 'bad' because the committer uses bad randomness. Thus, due to Condition (c) of ***can-traverse**, $^*\textbf{can-traverse}(c_0) = \mathtt{true}$.

It issues $\mathtt{Set\text{-}Secret}(u_{\text{`joi'}}, *)$ if (3) in the create protocol, the group creator initializes the group using bad randomness; or (4) in the join protocol, a party joins a group that $\mathcal{S}'_{4\text{-}2}$ has not created (i.e., the party assigns to a detached root).

Case (B-3): When the group creator initialize c_0 (= root_0) using bad randomness, the status of c_0 is set to 'bad'. Thus, due to Condition (a) of ***secrets-injected**, $\textbf{know}(c_0, \mathsf{id}_{\mathsf{creator}}) = \mathtt{true}$. Since c_0 is a root node, due to Condition (d) of ***can-traverse**, we have $^*\textbf{can-traverse}(c_0) = \mathtt{true}$.

Case (B-4): When a party joins a group that the adversary creates, the party is assigned to a detached root, and its status is 'adv'. Thus, due to Condition (a) of ***secrets-injected**,

$\mathbf{know}(c_0, \mathsf{id}_c) = \mathtt{true}$, where id_c is the sender of the welcome message. Moreover, c_0 is a root node. Thus, due to Condition (d) of ***can-traverse**, ***can-traverse**$(c_0) = \mathtt{true}$.

Next, $u_{\mathsf{id},\text{`joi'}} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Corr}(u_{\mathsf{id},\text{`joi'}})$ or $\mathtt{Set\text{-}Full\text{-}Secret}(u_{\mathsf{id},\text{`joi'}}, *)$. Recalling the description of $\mathcal{S}'_{4\text{-}2}$, it issues $\mathtt{Corr}(u_{\mathsf{id},\text{`joi'}})$ if (5) the adversary corrupts the new member id_t before it joins a group; or (6) the encryption key of a new member id_t used to encrypt the joiner secret at c_0 is corrupted after epoch c_0.

Case (B-5): When the adversary corrupts a member id_t before id_t joins a group, the signing key used to generate the key package is marked as exposed (cf. $(\mathtt{exposed}, \mathsf{id}_t)$ query to $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$). Thus, due to Condition (a) of ***can-traverse**, ***can-traverse**$(c_0) = \mathtt{true}$.

Case (B-6): This case happens if a party id_t holds the same encryption key at both c_0 and c'_0, where c'_0 is a descendant node of c_0, and id_t is corrupted at c'_0. Since id_t is corrupted at c'_0, $(\mathsf{id}_t, *) \in \mathsf{Node}[c'_0].\mathsf{exp}$ becomes true, and we have $\mathbf{know}(c'_0, \mathsf{id}_t) = \mathtt{true}$. In addition, since id_t holds the same encryption key at both c_0 and c'_0, id_t did not perform any actions that replace id_t's encryption key between c_0 and c'_0. Thus, $\neg^*$**secrets-replaced**$(c''_0, \mathsf{id}_t) = \mathtt{true}$ for each c''_0 on c_0–c'_0. Therefore, due to Condition (b) of ***can-traverse**, ***can-traverse**$(c_0) = \mathtt{true}$ (cf. ***reused-welcome-key-leaks**).

It issues $\mathtt{Set\text{-}Full\text{-}Secret}(u_{\mathsf{id},\text{`joi'}}, *)$ if (7) the key package in the add proposal is generated using bad randomness.

Case (B-7): When the key package for add proposals is generated using bad randomness, $\mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}$ marks that the signing key is exposed (cf. `register-kp` query with bad randomness). Thus, due to Condition (a) of ***can-traverse**, ***can-traverse**$(c_0) = \mathtt{true}$.

In all cases, if Case (B) is true, then ***can-traverse**$(c_0) = \mathtt{true}$; It implies $\mathbf{know}(c_0, \text{`epoch'}) = \mathtt{true}$ and $\mathbf{safe}(c_0) = \mathtt{false}$.

**[Case (C): $\big((u_{\mathbf{com}} \in \mathsf{Corr}) \vee (u_{\mathsf{id},\text{`}\mathbf{com}\text{'}} \in \mathsf{Corr})\big) \wedge \mathbf{gsd\text{-}exp}(u_{\mathbf{par\text{-}init}}) = \mathtt{true}$].** We below show that $u_{\mathsf{com}} \in \mathsf{Corr} \vee u_{\mathsf{id},\text{`com'}} \in \mathsf{Corr}$ (i.e., $\mathbf{gsd\text{-}exp}(u_{\mathsf{com}}) = \mathtt{true}$) implies $\mathbf{know}(c_0, \mathsf{id}) = \mathtt{true}$ for some $\mathsf{id} \in \mathsf{Node}[c_0].\mathsf{mem}$. By applying Proposition E.27 to the parent node of c_0 (i.e., $\mathsf{Node}[c_0].\mathsf{par}$), we can show that $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{true}$ implies $\mathbf{know}(\mathsf{Node}[c_0].\mathsf{par}, \text{`epoch'}) = \mathtt{true}$. As a result, due to Condition (d) of $^*$**can-traverse**, if Case (C) is true, then $^*$**can-traverse**$(c_0) = \mathtt{true}$, which implies $\mathbf{know}(c_0, \text{`epoch'}) = \mathtt{true}$ and $\mathbf{safe}(c_0) = \mathtt{false}$.

First, $u_{\mathsf{com}} \in \mathsf{Corr}$ becomes true when $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Set\text{-}Secret}(u_{\mathsf{com}}, *)$[29] Recalling the description of $\mathcal{S}'_{4\text{-}2}$, it issues $\mathtt{Set\text{-}Secret}(u_{\mathsf{com}}, *)$ if (1) in the commit protocol, a malicious encryption key generated by $\mathcal{A}$ is used to encrypt the commit secret at c_0; (2) in the commit protocol, the committer generates c_0 using bad randomness; or (3) in the process protocol, a party processes an injected commit message.

Case (C-1): This case occurs if the committer of c_0 has applied (I) an injected update proposal that contains a malicious encryption key or (II) an add proposal that contains a malicious key package generated by $\mathcal{A}$ at a previous epoch, and the committer uses the same encryption key to generate c_0.

Case (C-1-I): Assume the committer has applied the injected update proposal sent from id at the node c'_0, which is an ancestor of c_0 (including the case $c'_0 = c_0$). When the adversary succeeds to inject an update proposal that contains the encryption key generated by the adversary, the corresponding proposal node has the states 'adv'. Hence, due to Condition (b) of ***secrets-injected**, we have $\mathbf{know}(c'_0, \mathsf{id}) = \mathtt{true}$. In addition, if id uses the same encryption key at both c'_0 and c_0, id did not perform any actions that replace id's encryption key between c'_0 and c_0. Thus, $\neg^*$**secrets-replaced**(c''_0, id) is true for each c''_0 on c'_0–c_0 path. Therefore, Condition (c) of $\mathbf{know}(c_0, \mathsf{id})$ returns true.

Case (C-1-II): Assume the committer has applied the add proposal from id at the node c'_0, which is an ancestor of c_0. When the adversary succeeds to inject a key package that contains malicious encryption key, the signing key used to generate the key package must be exposed due to the modification we made in Hybrid 4-1 (or Hybrid 5-1). Hence, due to Condition (c) of ***secrets-injected**, we have $\mathbf{know}(c'_0, \mathsf{id}) = \mathtt{true}$. In addition, since id uses the same encryption key at both c'_0 and c_0, id did not perform any actions that replace id's encryption key between c'_0 and c_0. Thus, $\neg^*$**secrets-replaced**(c''_0, id) is true for each c''_0 on c'_0–c_0 path. Therefore, Condition (c) of $\mathbf{know}(c_0, \mathsf{id})$ returns $\mathtt{true}$.

Case (C-2): If the committer generates c_0 using bad randomness, the status of c_0 is set to 'bad'. Thus, due to Condition (a) of ***secrets-injected**, $\mathbf{know}(c_0, \mathsf{id}_c) = \mathtt{true}$.

Case (C-3): Due to the modification we made in Hybrid 4-2 (cf. $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}2}}$), if the node c_0 is created by the process protocol, both $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \mathtt{true}$ and $\mathbf{gsd\text{-}exp}(u_{\mathsf{com}}) = \mathtt{true}$ hold. Moreover, the status of c_0 is set to 'adv'. Thus, due to Condition (a) of ***secrets-injected** and Condition (d) of ***can-traverse**, $\mathbf{know}(c_0, \text{`epoch'}) = \mathtt{true}$ and $\mathbf{safe}(c_0) = \mathtt{false}$ always hols in this case.

Next, $u_{\mathsf{id},\text{`com'}} \in \mathsf{Corr}$ becomes true if $\mathcal{S}'_{4\text{-}2}$ queries $\mathtt{Corr}(u_{\mathsf{id},\text{`com'}})$ or $\mathtt{Set\text{-}Full\text{-}Secret}(u_{\mathsf{id},\text{`com'}}, *)$. $\mathcal{S}'_{4\text{-}2}$ issues $\mathtt{Corr}(u_{\mathsf{id},\text{`com'}})$ if (4) a party id is corrupted via Corruption query while it holes the encryption key $u_{\mathsf{id},\text{`com'}}$; or (5) id issues an update proposal at the parent node of c_0 and id is corrupted before processing the commit message c_0 (this corresponds to the fact that id stores the pending update proposals including encryption keys in pendUpd array).

Case (C-4): Assume id is corrupted at epoch c'_0. That is, $(\mathsf{id}, *) \in \mathsf{Node}[c'_0].\mathsf{exp}$ is true. In addition, since party id uses the same encryption key $u_{\mathsf{id},\text{`com'}}$ at c_0, id does not perform any actions that replace id's encryption key between c_0 and c'_0. Thus, $\neg^*$**secrets-replaced**(c''_0, id) is true for each c''_0 on c'_0–c_0 or c_0–c'_0 path. Due to Condition (c) or (d) of $\mathbf{know}(c_0, \mathsf{id})$, we have $\mathbf{know}(c_0, \mathsf{id}) = \mathtt{true}$.

Case (C-5): If id is corrupted at the parent node of c_0, the status of the corresponding proposal nodes stored in pendUpd array are set to 'bad' (cf. `*update-stat-after-exp`). Thus, due to Condition (b) of ***secrets-injected**, we have $\mathbf{know}(c_0, \mathsf{id}) = \mathtt{true}$.

[29] By definition, $\mathcal{S}'_{4\text{-}2}$ never queries $\mathtt{Corr}(u_{\mathsf{com}})$.

$\texttt{Set-Full-Secret}(u_{\mathsf{id},\text{`com'}}, *)$ was queried to $u_{\mathsf{id},\text{`com'}}$ if (6) the committer of c_0 applied an update proposal from id generated with bad randomness and uses the same encryption key at c_0 or (7) the committer applied an add proposal from id generated with bad randomness and uses the same encryption key at c_0.

Case (C-6): Assume id issues an update proposal using bad randomness and the committer applies it at epoch c_0'. At the epoch, $\mathcal{S}'_{4\text{-}2}$ issues $\texttt{Set-Full-Secret}(u_{\mathsf{id},\text{`com'}}, *)$ during key package generation, and due to Condition (b) of $^*$**secrets-injected**, $\mathbf{know}(c_0', \mathsf{id})$ is true. In addition, since party id uses the encryption key $u_{\mathsf{id},\text{`com'}}$ at c_0, id does not perform any actions that replace id's encryption key between c_0 and c_0'. Thus, $\neg^*\mathbf{secrets\text{-}replaced}(c_0'', \mathsf{id})$ is true for each c_0'' on c_0'–c_0 path. Due to Condition (d) of $\mathbf{know}(c_0, \mathsf{id})$, we have $\mathbf{know}(c_0, \mathsf{id}) = \texttt{true}$.

Case (C-7): Assume id added a group at epoch c_0' using an add proposal generated with bad randomness. When the key package was generated, $\mathcal{S}'_{4\text{-}2}$ issues $\texttt{Set-Full-Secret}(u_{\mathsf{id},\text{`com'}}, *)$, and due to Condition (c) of $^*$**secrets-injected**, $\mathbf{know}(c_0', \mathsf{id})$ is true. In addition, since party id uses the encryption key $u_{\mathsf{id},\text{`com'}}$ at c_0, id does not perform any actions that replace id's encryption key between c_0 and c_0'. Thus, $\neg^*\mathbf{secrets\text{-}replaced}(c_0'', \mathsf{id})$ is true for each c_0'' on c_0'–c_0 path. Due to Condition (d) of $\mathbf{know}(c_0, \mathsf{id})$, we have $\mathbf{know}(c_0, \mathsf{id}) = \texttt{true}$.

In all cases, $\mathbf{know}(c_0, \mathsf{id}) = \texttt{true}$ for some $\mathsf{id} \in \mathsf{Node}[c_0].\mathsf{mem}$. Therefore, if $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \texttt{true}$ holds additionally, Case (C) implies $\mathbf{know}(c_0, \text{`epoch'}) = \texttt{true}$ and $\mathbf{safe}(c_0) = \texttt{false}$.

From the above discussion, we obtain the following statements.

- If $\mathbf{know}(c_0, \text{`epoch'}) = \texttt{false}$, then $\mathbf{gsd\text{-}exp}(u) = \texttt{false}$ for $u \in \{ u_{\text{`memb'}}, u_{\text{`conf'}}, u_{\text{`init'}} \}$.
- If $\mathbf{safe}(c_0) = \texttt{true}$, then $\mathbf{gsd\text{-}exp}(u_{\text{`app'}}) = \texttt{false}$.

□

Now we ready to prove Lem. E.28.

Lemma E.28. *Hybrid 4-3 and Hybrid 4-4 are identical.*

Proof. The difference between Hybrid 4-3 and Hybrid 4-4 is that in Hybrid 4-4 we use the original `auth-invariant` predicate and the functionally $\mathcal{F}_{\mathsf{CGKA},4}$ halts if `auth-invariant` returns false. We show that the simulator $\mathcal{S}_{4\text{-}4}$ never creates history graph nodes such that `auth-invariant` returns false, that is, $\mathcal{F}_{\mathsf{CGKA},4}$ never halts. We consider Condition (a) and Condition (b) of `auth-invariant` in order.

Condition (a) of `auth-invariant`. We first show that, for all non-root node c_0 in the history graph created by $\mathcal{S}_{4\text{-}4}$, if $\mathsf{Node}[c_0].\mathsf{stat}$ = 'adv', then $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{true}$, where $c_p := \mathsf{Node}[c_0].\mathsf{par}$ (non-root implies $c_p \neq \perp$). By the definition of the functionality, a non-root commit node c_0 with status 'adv' is created when (1) an existing detached root is attached to a commit node generated by the commit protocol using bad randomness; or (2) the commit node is created by the process protocol. On the other hand, (1) $\mathcal{S}_{4\text{-}4}$ aborts when an existing detached root will be attached to a commit node generated using bad randomness if $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \texttt{false}$ (cf. $\mathsf{Abort}_{\mathsf{attach}}$); and (2) $\mathcal{S}_{4\text{-}4}$ always rejects the injected commit message c_0 in the process protocol if $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \texttt{false}$ (cf. $\mathsf{E}_{\mathsf{inj\text{-}c\text{-}1}}$). Here, $u_{\mathsf{par\text{-}init}}$ is the GSD node assigned to the initial secret at the parent epoch of c_0. Thus, commit nodes with status 'adv' are created only if $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \texttt{true}$. Moreover, due to Proposition E.27, if $\mathbf{gsd\text{-}exp}(u_{\mathsf{par\text{-}init}}) = \texttt{true}$, then $\mathbf{know}(c_p, \text{`epoch'}) = \texttt{true}$, i.e., $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{true}$. Therefore, commit nodes with status 'adv' are created only if $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{true}$. In other words, there exists no node c_0 such that $\mathsf{Node}[c_0].\mathsf{stat}$ = 'adv' and $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{false}$ in the history graph created by $\mathcal{S}_{4\text{-}4}$.

Condition (b) of `auth-invariant`. We next show that, for all proposal node p in the history graph, if $\mathsf{Prop}[p].\mathsf{stat}$ = 'adv', then $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{true}$, where $c_p := \mathsf{Prop}[p].\mathsf{par}$ (by definition, c_p always non-$\perp$). By the definition of the functionality, a proposal node with status 'adv' is created if a proposal message is not generated by the propose protocol. On the other hand, if a received proposal is not generated by the propose protocol and $\mathbf{gsd\text{-}exp}(u_{\text{`memb'}}) = \texttt{false}$, the commit and process protocol always outputs $\perp$ (i.e., rejects the message), where $u_{\text{`memb'}}$ is the GSD node assigned to the corresponding membership key at c_p. In other words, proposal nodes with status 'adv' is created only if $\mathbf{gsd\text{-}exp}(u_{\text{`memb'}}) = \texttt{true}$. Moreover, due to Proposition E.27, if $\mathbf{gsd\text{-}exp}(u_{\text{`memb'}}) = \texttt{true}$, then $\mathbf{know}(c_p, \text{`epoch'}) = \texttt{true}$, i.e., $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{true}$. Therefore, proposal nodes with status 'adv' are created only if $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{true}$. In other words, there exists no proposal node p such that $\mathsf{Prop}[p].\mathsf{stat}$ = 'adv' and $\mathbf{mac\text{-}inj\text{-}allowed}(c_p) = \texttt{true}$ in the history graph created by $\mathcal{S}_{4\text{-}4}$.

From the above discussion, $\mathcal{S}_{4\text{-}4}$ never creates history graph nodes such that `auth-invariant` returns false, i.e., the functionality never halts. Thus, Hybrid 4-3 and Hybrid 4-4 are identical. □

E.6 From Hybrid 5 to 6: Lem. E.29

In Hybrid 6, receiving Key query, the functionality $\mathcal{F}_{\mathsf{CGKA}}$ returns a random application secret if **safe** is true for the queried epoch. To prove the indistinguishability of the two hybrids, we gradually replace each application secret with a random value instead of the real value if **safe** is true. We show that, if $\mathcal{Z}$ can distinguish whether the application secret is real or random, it can be used to break the Chained CmPKE conforming GSD security of CmPKE. In other words, if CmPKE is Chained CmPKE conforming GSD secure, Hybrid 5 and Hybrid 6 are indistinguishable. We below provide a formal proof of the above overview.

Lemma E.29. *Hybrid 5 and Hybrid 6 are indistinguishable assuming* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. We assume $\mathcal{Z}$ creates at most Q epochs (i.e., commit nodes). To show Lem. E.29, we consider the following sub-hybrids between Hybrid 5 and Hybrid 6.

Hybrid 5-0. This is identical to Hybrid 5. We use the functionality $\mathcal{F}_{\mathsf{CGKA},5}$, and all application secrets are set to the real value by the simulator $\mathcal{S}_{5\text{-}0} := \mathcal{S}_5$.

Hybrid 5-*i*. i runs through $[Q]$. The simulator $\mathcal{S}_{5\text{-}i}$ is defined exactly as $\mathcal{S}_{5\text{-}(i-1)}$ except that it sets the i-th application secret to random if **safe** is true. Note that we count application secrets in the order in which Key query is issued. We show in Lem. E.30 that Hybrid 5-$(i-1)$ and Hybrid 5-i are indistinguishable.

Hybrid 6. We replace the functionality $\mathcal{F}_{\mathsf{CGKA},5}$ with the original functionality $\mathcal{F}_{\mathsf{CGKA}}$. In this hybrid, all application secrets such that **safe** is true are set to random by $\mathcal{F}_{\mathsf{CGKA}}$. From $\mathcal{Z}$'s point of view, Hybrid 5-Q and Hybrid 6 are identical because the only difference is who sets application secrets to random.

The indistinguishability between Hybrid 5-0 and Hybrid 5-Q is derived by applying Lem. E.30 for all $i \in [Q]$. Therefore, we conclude that Hybrid 5 and Hybrid 6 are indistinguishable. □

Lemma E.30. *Hybrid 5-$(i-1)$ and Hybrid 5-i are indistinguishable assuming* CmPKE *is Chained CmPKE conforming GSD secure.*

Proof. The difference between Hybrid 5-$(i-1)$ and Hybrid 5-i is whether the i-th application secret is real or random if **safe** is true. We show if $\mathcal{Z}$ can distinguish the two hybrids, there exists an adversary $\mathcal{B}$ against the Chained CmPKE conforming GSD game. We first explain the description of $\mathcal{B}$. We then show the validity of the GSD challenge, and finally evaluate $\mathcal{B}$'s advantage.

$\mathcal{B}$ simulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}$ and $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$, and the protocol execution of each party as in $\mathcal{S}'_{4\text{-}2}$, except that $\mathcal{B}$ interacts with its GSD challenger instead of $\mathcal{S}_{\mathrm{GSD}}$. $\mathcal{B}$ embeds the GSD challenge in the i-th application secret as follows: Assume $\mathcal{Z}$ issues Key query to id and $c_0 = \mathsf{Ptr}[\mathsf{id}]$ is the j-th to be queried Key.

- If $\mathbf{safe}(c_0) = \mathtt{true}$ and $j < i$, $\mathcal{B}$ returns the random value.
- Else if $\mathbf{safe}(c_0) = \mathtt{true}$ and $j = i$, $\mathcal{B}$ queries $s_{\text{`app'}} := \mathtt{Chall}(u_{\text{`app'}})$ and returns the challenge value $s_{\text{`app'}}$, where $u_{\text{`app'}}$ is the GSD node corresponding to the application secret of c_0.
- Else, $\mathcal{B}$ returns the real application secret (if necessary, $\mathcal{B}$ corrupts the corresponding GSD node $u_{\text{`app'}}$).

Observe that, if $\mathbf{safe}(c_0) = \mathtt{false}$ for the i-th application secret, the challenge is not embedded. In this case, the two hybrids proceeds exactly the same.

We consider the case $\mathcal{B}$ embedded the GSD challenge. We argue the GSD challenge is valid. Observe that the GSD graph is acyclic and the challenge node $u_{\text{`app'}}$ is a sink node. In addition, due to Proposition E.27, $\mathbf{safe}(c_0) = \mathtt{true}$ implies $\mathbf{gsd\text{-}exp}(u_{\text{`app'}}) = \mathtt{false}$. Thus, $u_{\text{`app'}}$ is a valid challenge node.

We finally analyze $\mathcal{B}$'s advantage. If the challenge oracle returns the real value, $\mathcal{Z}$'s view is identical to Hybrid 5-$(i-1)$; else, i.e., the challenge oracle returns a random value, $\mathcal{Z}$'s view is identical to Hybrid 5-i. Hence, if $\mathcal{Z}$ distinguishes Hybrid 5-$(i-1)$ and Hybrid 5-i with non-negligible probability, $\mathcal{B}$ wins the GSD game with non-negligible probability by using $\mathcal{Z}$'s output. This contradicts the assumption that CmPKE is Chained CmPKE conforming GSD secure. Therefore, Hybrid 5-$(i-1)$ and Hybrid 5-i are indistinguishable. □

F A VARIANT OF GSD SECURITY TAILORED TO CHAINED CMPKE

We introduce a variant of the generalized selective decryption (GSD) security notion for public-key encryption tailored to our Chained CmPKE security proof, which we coin as a Chained CmPKE conforming GSD security. We then show that such variant is secure in the random oracle model assuming the hardness of the CmPKE. The formalization of our Chained CmPKE conforming GSD security is inspired by [9, 12, 13] but differs in the following way: we consider a (committing) multi-recipient encryption oracle rather than a single-recipient oracle; we restrict the hash oracles to *not* take as input the secrets used to generate the keys for CmPKE; the proof is simplified.[30] The restriction on the hash oracle is new to this work and effectively, this simplifies the proof while still being sufficient for proving our Chained CmPKE protocol.

In more detail, our Chained CmPKE conforming GSD is defined in Fig. 29. The game maintains a graph with M-vertices, where each node u stores a seed s_u initially set to $\perp$. If u is a source node, then s_u is further used to generate a key pair $(\mathsf{ek}_u, \mathsf{dk}_u)$ for CmPKE (see `*gen-full-key-if-nec`). An edge corresponds to dependencies between seeds, where there are three types of edges. One edge is a (multi-recipient) encryption edge created by `CmEnc`: if there is an edge from a source node u leading into v, then s_v is encrypted using ek_u. The second edge is a hash edge created by `Hash`: if there is an edge from a non-source node u leading into v, then s_v is (informally) the output of a hash function H on input s_u. The final type of edge is a join hash edge created by `Join-Hash`: if there is an edge from non-source nodes u and u' leading into v, then s_v is (informally) the output of a hash function H on input s_u and $s_{u'}$. See Fig. 1 for what these edges mean in the context of the CGKA protocol. In Fig. 1, the solid edge corresponds to encryption edges and the dashed edges correspond to either hash or join hash edges. Observe that `Hash` does not take as input an encryption node (i.e., EncSource) and `CmEnc` does not take as input a (non-join) hashed input (i.e., $s_v = \perp$ or $v \in$ WelcomeNode). This restriction is sufficient to prove security of our Chained CmPKE protocol, while also having the benefit that the GSD security proof will be simplified.

The GSD adversary can adaptively create the edges of the graph and also adaptively corrupt the nodes to obtain the stored secret seed. The **gsd-exp** function determines if the secret seed in node v is trivially exposed to the adversary. Specifically, u is exposed if it is corrupted or can be traversed from any corrupted nodes. Then, the security of GSD states that as long as node u does not satisfy **gsd-exp** (i.e., does not trivially expose the secret), then the secret seed s_u remains hidden.

More formally, we define the security notion as follows.

Definition F.1 (Chained CmPKE Conforming GSD Security). The security notion is defined by a game illustrated in Fig. 29, where we say the adversary $\mathcal{A}$ *wins* if the game outputs 1. We say the CmPKE is *Chained CmPKE conforming GSD secure* if for all PPT adversary $\mathcal{A}$, we have $|\Pr[\mathcal{A}\ \textit{wins}] - 1/2| \leq \mathrm{negl}(\kappa)$. We say it is *selectively* Chained CmPKE conforming GSD secure if $\mathcal{A}$ is required to commit to all of its oracle queries at the outset of the game.

The following is the main theorem of this section.

Theorem F.2. *A* CmPKE *is Chained CmPKE conforming GSD secure if* CmPKE *is* IND-CCA *secure with adaptive corruption. Additionally, it is selectively secure if* CmPKE *is only* IND-CCA *secure.*

Proof. We only consider the adaptive setting since downgrading the proof to the selective setting is straightforward. Our proof is in the non-programmable random oracle model.

[30] We also noticed that we would require oracles `Set-Secret` and `Set-Full-Secret` in the security proof.

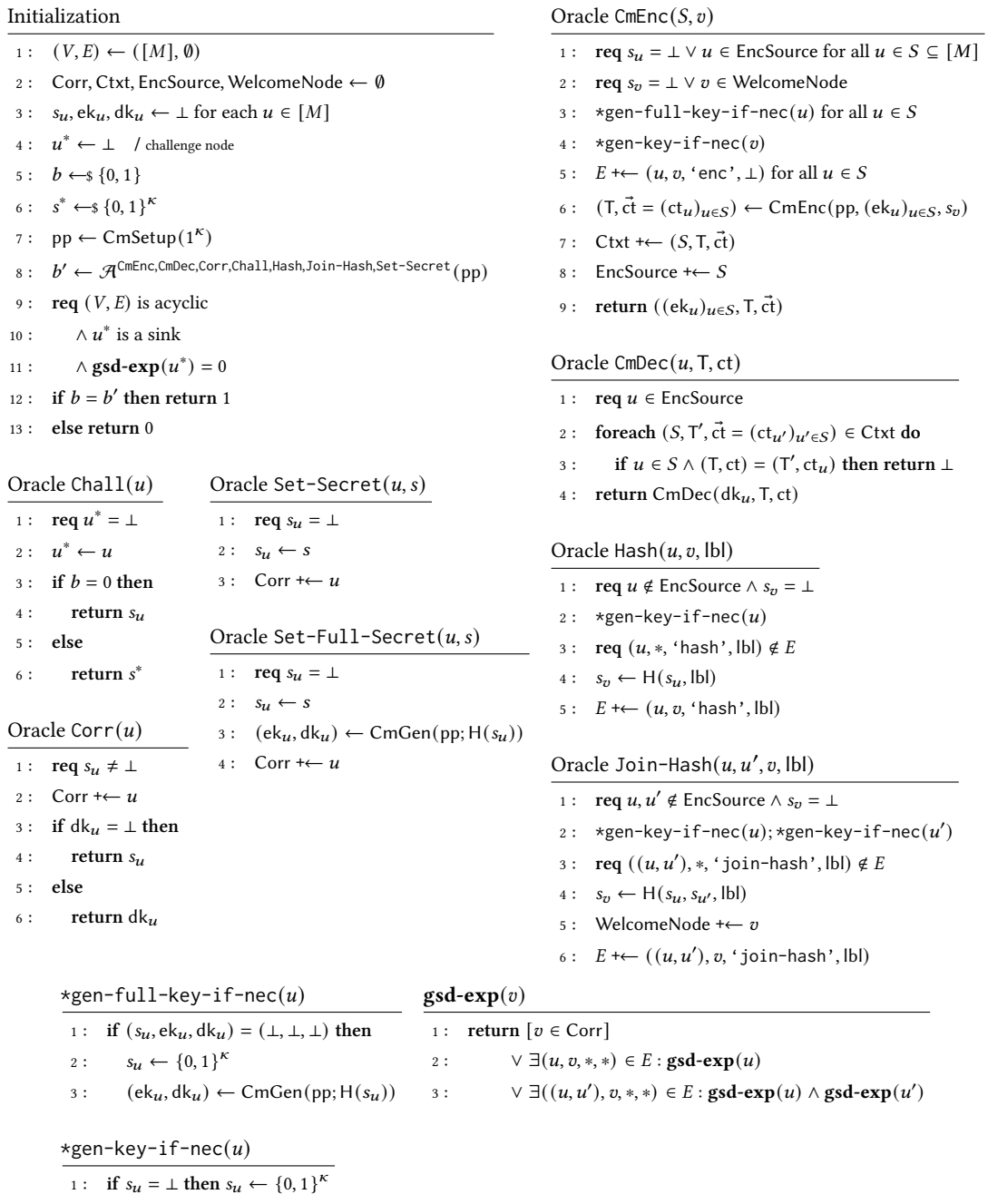


Figure 29: A Chained CmPKE conforming GSD game. The adversary $\mathcal{A}$ is also assumed to be given oracle access to the random oracle H.

To aid the proof, we define a helper function SecurePaths in Fig. 30 which takes as input a node $u \in [M]$ and a list L. At the end of the game, if an adversary $\mathcal{A}$ had chosen a valid challenge node u^*, then SecurePaths$(u^*, L := \emptyset)$ outputs all the "secure paths" that leads to u^*. To be more concrete, SecurePaths$(u^*, L := \emptyset)$ outputs a set of lists $D = \{L_k\}_{k \in [K]}$, where $K \leq M - 1$ and each L_k is either of the form $L_k = (u_1 := u^*, u_2, \cdots, u_{I_k})$ or $L_k = (u_1 := u^*, u_2, \cdots, u_{I_{k-1}}, S)$ for some integer I_k, nodes $u_i \in [M]$ such that **gsd-exp**$(u_i) = 0$, u_{I_k} is a source node, and a set of nodes $S \subset [M]$ such that $\forall u \in S$, $u \in$ EncSource and $u \notin$ Corr (which in particular implies **gsd-exp**$(u) = 0$). Intuitively, L_k is a path that consists of uncorrupt nodes (and possibly a set of nodes) that do not trivially leak the secret s^* associated to u^*. Note that it is not enough to simply check if a node is uncorrupted; even if v

SecurePaths(u, L)

1 : $L \twoheadleftarrow u$
2 : **if** $\exists v$ s.t. $(v, u, \text{`hash'}, *) \in E$ **then** / $u$ is connected only from a `hash` edge
3 : **if** **gsd-exp**$(v) = 0$ **then** / Such a v is unique if it exists
4 : SecurePaths(v, L)
5 : **elseif** u is not a source **then**
6 : $(v_1, \text{flag}_1, v_2, \text{flag}_2, S, \text{flag}_3) \leftarrow (\bot, \bot, \bot, \bot, \emptyset, \bot)$
7 : **if** $\exists (v, v')$ s.t. $((v, v'), u, \text{`join-hash'}, *) \in E$ **then**
8 : $(v_1, v_2) \leftarrow (v, v')$ / Such (v, v') is unique if it exists
9 : **if** **gsd-exp**$(v_1) = 0$ **then**
10 : $\text{flag}_1 \leftarrow \top$
11 : **if** **gsd-exp**$(v_2) = 0$ **then**
12 : $\text{flag}_2 \leftarrow \top$
13 : **if** $\exists v$ s.t. $(v, u, \text{`enc'}, *) \in E$ **then**
14 : $\text{flag}_3 \leftarrow \top$ / Remains $\top$ if all recipients are uncorrupted
15 : **foreach** v s.t. $(v, u, \text{`enc'}, *) \in E$ **do**
16 : $S \twoheadleftarrow v$
17 : **if** **gsd-exp**$(v) = 1$ **then**
18 : $\text{flag}_3 \leftarrow \bot$ / v is a corrupted (encryption) source
19 : **if** $(\text{flag}_1 = \top \vee \text{flag}_2 = \top) \wedge \text{flag}_3 = \top$ **then**
20 : **if** $\text{flag}_1 = \top$ **then** / u is connected from `join-hash` and `enc` edges
21 : SecurePaths(v_1, L)
22 : **if** $\text{flag}_2 = \top$ **then**
23 : SecurePaths(v_2, L)
24 : $L \twoheadleftarrow S$
25 : **return** L
26 : **elseif** $(\text{flag}_1 = \top \vee \text{flag}_2 = \top) \wedge S = \emptyset$
27 : **if** $\text{flag}_1 = \top$ **then** / u is connected only from a `join-hash` edge
28 : SecurePaths(v_1, L)
29 : **if** $\text{flag}_2 = \top$ **then**
30 : SecurePaths(v_2, L)
31 : **elseif** $v_1 = \bot \wedge \text{flag}_3 = \top$ **then** / u is connected only from `enc` edges
32 : $L \twoheadleftarrow S$
33 : **return** L
34 : **elseif** **gsd-exp**$(u) = 0$ **then** / u is a non-corrupted source
35 : **return** L

Figure 30: Helper function that outputs all the secure paths leading to the input node u, where $L = \emptyset$ by default.

and v' such that $((v, v'), u, \text{`join-hash'}, *) \in E$ are uncorrupted, we must also check that all $v''$ such that $(v'', u, \text{`enc'}, *) \in E$ are uncorrupted as well, since otherwise, u's secret s_u may trivially leak.

Stating the above more formally, we obtain the following lemma.

Lemma F.3. *The challenge sink u^* output by $\mathcal{A}$ is a valid challenge (i.e., **gsd-exp**$(u^*) = 0$) if and only if SecurePaths$(u^*) \neq \emptyset$.*

Proof. The proof simply consists of checking the conditions. The "only if" part of the proof is trivial since any path that satisfies **gsd-exp**$(u^*) = 0$ is also a path that will be output by SecurePaths. Therefore, let us focus on the "if" part of the proof. Let us assume SecurePaths$(u^*) \to D = \{ L_k \}_{k \in [K]}$. Consider any L_k of the form $L_k = (u_1 := u^*, \cdots, u_{I_k})$. (The case $L_k = (u_1 := u^*, \cdots, u_{I_{k-1}}, S)$ follows the same argument). By the definition of SecurePaths, it is clear that there is an edge between each adjacent nodes u_i and u_{i+1}. Moreover, we have **gsd-exp**$(u_i) = 0$ for every $i \in [I_k]$. This can be verified by checking the nodes in reverse order; The final output u_{I_k} is a source node that satisfies **gsd-exp**$(u_{I_k}) = 0$. For u_{I_k} to be output, SecurePaths$(u_{I_k}, L'_k = (u_i)_{i \in [I_k - 1]})$ must have been invoked within SecurePaths$(u_{I_k - 1}, L''_k = (u_i)_{i \in [I_k - 2]})$. If $(u_{I_k}, u_{I_k - 1})$ is connected by a `hash` edge, then **gsd-exp**$(u_{I_k - 1}) = 0$ and we have that u_{I_k} is the only node connected to $u_{I_k - 1}$. If $(u_{I_k}, u_{I_k - 1})$ is connected by an `enc` edge or by a `join-hash` edge (i.e., $\exists$ u such that $((u_{I_k}, u), u_{I_k - 1}, \text{`join-hash'}, *) \in E$), then all other `enc` edges leading to $u_{I_k - 1}$ come from uncorrupted nodes. Therefore, this establishes **gsd-exp**$(u_{I_k - 1}) = 0$. We can repeat this argument until we reach SecurePaths$(u_1 = u^*, L := \emptyset)$ to establish **gsd-exp**$(u_i) = 0$ for every $i \in [I_k]$. This completes the lemma. □

Let $D = \{ L_k \}_{k \in [K]} \leftarrow$ SecurePaths(u^*). There are two cases:

Case 1: $\exists L_k = (u_1 := u^*, \cdots, u_{I_k}) \in D$ such that for all $i \in [I_k]$, there does not exist v satisfying $(v, u_i, \text{`enc'}, *) \in E$.

Case 2: $\forall L_k \in D$, either $L_k = (u_1 := u^*, \cdots, u_{I_k})$ such that there exists $i \in [I_k]$ and v satisfying $(v, u_i, \text{`enc'}, *) \in E$ or $L_k = (u_1 := u^*, \cdots, u_{I_k - 1}, S)$.

Note that when $L_k = (u_1 := u^*, \cdots, u_{I_k - 1}, S)$, we have $S \subseteq$ EncSource in Fig. 29; that is, S is the set of nodes that were used to encrypt the secret $s_{u_{I_k - 1}}$ associated to node $u_{I_k - 1}$. The following Lems. F.4 and F.5, each corresponding to Case 1 and Case 2, respectively, completes the proof of the theorem. Note that although the lemma assumes either Case 1 or Case 2 always hold, this is without loss of generality since the reduction can always guess which case we end up in.

Lemma F.4. *If Case 1 occurs, then any adversary $\mathcal{A}$ making at most polynomially many oracle queries has negligible advantage against the Chained CmPKE conforming GSD security.*

Proof. When Case 1 occurs, there are no encryption edges coming into any of the nodes along the path $L_k = (u_1 := u^*, \cdots, u_{I_k})$. Therefore, since **gsd-exp**$(u_i) = 0$ for all $i \in [I_k]$, this means all the associated secrets $(s_{u_i})_{i \in [I_k]}$ are information theoretically hidden from the adversary $\mathcal{A}$ until they are queried to the random oracle. This can be argued more formally as follow by induction: Since u_{I_k} is a (non-encryption) source node, $s_{u_{I_k}}$ is information theoretically hidden. In case (u_i, u_{i+1}) is connected by a `hash` edge, then if s_{u_i} is hidden, so is $s_{u_{i+1}}$ in the random oracle model. On the other hand, in case (u_i, u_{i+1}) is connected by a `join-hash` edge, then even if the other node u such that $((u_i, u), u_{i+1}, \text{`join-hash'}, *) \in E$ is corrupted, $s_{u_{i+1}}$ is hidden as long as s_{u_i} is. Finally, since I_k and the number of random oracle queries $\mathcal{A}$ makes is polynomial, the probability of $\mathcal{A}$ obtaining any information on $(s_{u_i})_{i \in [I_k]}$ is negligible. This in particular implies that $s_{u_1} := s^*$ is uniform random in the view of $\mathcal{A}$. This concludes the lemma, where we note that the adaptivity of $\mathcal{A}$ is irrelevant. □

Lemma F.5. *If Case 2 occurs, then any PPT adversary $\mathcal{A}$ has negligible advantage against the Chained CmPKE conforming GSD security assuming the hardness of the* IND-CCA *security with adaptive corruption of* CmPKE.

Proof. Let $\mathcal{A}$ be an adversary against the Chained CmPKE conforming GSD security game that triggers Case 2. Consider the following three games where the first game corresponds to the real game depicted in Fig. 29 and the last game is where no (possibly inefficient) adversary has winning advantage. We denote E_i as the event that $\mathcal{A}$ wins in game Game i and show that each adjacent games are indistinguishable, thus establishing the hardness of the real game.

Game 0 : This is the real game depicted in Fig. 29.

Game 1 : The challenger guesses a random challenge sink $u^* \leftarrow\!\!\$\ [M]$ and a challenge source $v^* \leftarrow\!\!\$\ [M]$ conditioned on $u^* \neq v^*$. It then proceeds exactly as in the previous game except that it outputs a random bit on behalf of $\mathcal{A}$ if either u^* was not the node $\mathcal{A}$ queries to the challenge oracle or if there does not exist a list $L_k \in D \leftarrow \mathsf{SecurePaths}(u^*)$ such that either $L_k = (u_1 = u^*, \cdots, u_{I_k} = v^*)$ or $L_k = (u_1 = u^*, \cdots, u_{I_k-1}, S)$, where $v^* \in S$. Without loss of generality, we assume there is always an incoming edge to u^*. Then, by Lem. F.3, it is clear that $\Pr[E_1] \geq \Pr[E_0]/M^2$.

Game 2 : This is the same as the previous game except that the challenger answers to oracle CmEnc differently for those nodes connected to the challenge source node v^*. More concretely, when $\mathcal{A}$ queries (S, v) to oracle CmEnc, the challenger checks whether the following conditions (denoted as SetRand) hold:

- Is $s_v = \perp$ and $v^* \in S$?
- Is $v \in$ WelcomeNode and does there exist a set of edges in E that connects v^* to v?

If so, the challenger proceeds as in the previous game except that it samples a random message r_v and runs $\mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_u)_{u\in S}, r_v)$ instead of $\mathsf{CmEnc}(\mathsf{pp}, (\mathsf{ek}_u)_{u\in S}, s_v)$ on line 6. Otherwise, it is defined exactly as in the previous game. Intuitively, the challenger modifies all the incoming encryption edges to the secure path L_k to encrypt random values. Note that due to the way oracles Hash, Join-Hash, and CmEnc are defined, an input (S, v) that did not satisfy condition SetRand will remain unsatisfied since such a node v cannot be later connected to the secure path L_k.

Observe that Game 2 now boils down to the argument we made for Case 1 in Lem. F.4 since all the incoming encryption edges to the secure path L_k to encrypt random values. Specifically, there either exists a list $L_k = (u_1 = u^*, \cdots, u_{I_k} = v^*)$ or a list $L_k = (u_1 = u^*, \cdots, u_{I_k-1}, S)$, where $v^* \in S$. In the former case, since all the incoming encryption edges into the nodes in the list L_k are encrypting random values, we can simply ignore them. This is the same for the latter case, where we additionally observe that $\mathsf{CmEnc}(S, u_{I_k-1})$ provides an encryption of a random value. Therefore, following the same argument made in Case 1, $\Pr[E_2] = \mathsf{negl}(\kappa)$ even for a possible inefficient $\mathcal{A}$ that makes at most polynomially many queries.

To conclude the lemma, it remains to establish the bound between $\Pr[E_1]$ and $\Pr[E_2]$. We construct an adversary $\mathcal{B}$ against the IND-CCA security with adaptive corruption game of CmPKE that internally runs $\mathcal{A}$. Without loss of generality, we assume the "multi-challenge-ciphertext" variant where $\mathcal{B}$ can query polynomially-many challenge ciphertexts. By a basic hybrid argument, this variant is secure if the single-challenge version is secure. The description of $\mathcal{B}$ follows:

$\mathcal{B}^{C(\cdot)}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[M]})$: Whenever $\mathcal{B}$ needs to generate a new encryption and decryption key pair $(\mathsf{ek}_u, \mathsf{dk}_u)$, it simply uses an unused ek_u provided by its challenge. When $\mathcal{A}$ queries a corruption oracle on u, if dk_u is set, then $\mathcal{B}$ queries u to its corruption oracle and returns the received dk_u. Otherwise, s_u is generated on its own so it simply outputs s_u. When (S, v) queried to oracle CmEnc by $\mathcal{A}$ satisfies condition SetRand, then it samples a random message r_v and queries (s_v, r_v) as its challenge ciphertext. It then uses the provided challenge ciphertext to simulate CmEnc. Moreover, all oracle queries to CmDec can be answered by using its decryption oracle. Finally, when $\mathcal{A}$ makes a random oracle query, $\mathcal{B}$ simply relays this to its own random oracle.[31] $\mathcal{B}$ answers all other oracle queries, i.e., Set-Secret, Set-Full-Secret, Hash, Join-Hash on its own.

It can be checked that when $\mathcal{B}$ receives challenge ciphertexts for random messages, then the game it simulates is identical to Game 2. Otherwise, it is identical to Game 1. Therefore, assuming the hardness of the IND-CCA security with adaptive corruption of CmPKE, we have $|\Pr[E_1] - \Pr[E_2]| \leq \mathsf{negl}(\kappa)$.

Combining all the bounds, we have $\Pr[E_0] = \mathsf{negl}(\kappa)$ as desired. This completes the lemma. □

□

Remark 4 (Adaptive security from CmPKE with no adaptive corruption). *In the above proof, if we want to base adaptive security of the Chained CmPKE conforming GSD security from a* CmPKE *that is only* IND-CCA *secure (i.e., without adaptive corruption security), then we will incur an exponential reduction loss during the game transition of* Game 1 *to* Game 2. *This is because we need to guess correctly all the encryption keys that will not get corrupted from the set $[M]$ in order to simulate the corruption queries. In the worst case, we will lose a factor of $O(2^M)$.*

G CRYPTANALYSIS IN THE mPKE SETTING

In this appendix we describe our cryptanalytic model, how it differs from a more standard cryptanalytic model for PKEs, and how we incorporate parts of the cryptanalysis of the schemes for which we have provided alternative parametrizations in Sec. 5.2. At a high level, the main difference is the availability in the mPKE setting of many additional 'samples' (see below) to an adversary from the $N-1$ ciphertexts $\widehat{\mathsf{ct}}_i$.

In the PKE setting $N = 2$, whereas we consider up to $N = 2^{16}$ in the mPKE setting. The number of extra available samples becomes considerable when taking $N = 2^{16}$. For example, an adversary attacking Frodo640 has $640 + 8 = 648$ samples available for a row of $\mathbf{R}$ in the PKE case. This becomes $640+8\cdot(N-1) \approx 2^{19}$ in the $N = 2^{16}$ Bilbo640 case, though these extra samples have different properties

[31] To be precise, we assume the IND-CCA security with adaptive corruption game is defined in the random oracle model. This is without loss of generality.

to the original 640 in our new parametrization. In particular, they have much larger error, due to the modulus rounding.

Nonetheless, these extra samples require us to consider two attacks that are usually absent from concrete security analyses of lattice PKEs, namely the sample heavy Arora–Ge and BKW style attacks.

We first describe the number of samples available to an adversary in more detail, and the error these samples have. We then describe the three attacks we are considering, following broadly from the cryptanalysis present in several NIST final round lattice candidates, but adding Arora–Ge and BKW style attacks. After this, in App. G.2 we describe our cryptanalytic model, which targets NIST Security Level I. A scheme satisfying this definition of security should have security comparable to AES-128, both against classical and quantum adversaries. Finally, we discuss in more depth the following aspects of our new parametrizations; any subtle algorithmic changes beyond the reparametrizations, any ways in which our cryptanalysis significantly differs from what is present in their respective submission documents, and where we have opted to incorporate parts of their individual cryptanalyses. We reference Fig. 5 throughout, and in particular let $R = \mathbb{Z}[x]/(f)$ and $d = \deg(f)$, noting that we can recover Bilbo640 by setting $f(x) = x$.

G.1 Samples and Attacks

Samples. We talk of two distinct types of sample, a sample for $\mathbf{S}$ or a sample for $\mathbf{R}$. (See Fig. 5). The number of samples given for (a single column of) $\mathbf{S}$ is $d \cdot n$, and they come from $\mathbf{B}$. Only Bilbo640 has $\bar{n} > 1$, and these extra columns of $\mathbf{S}$ can be accounted for by a hybrid argument. We can similarly count the samples for $\mathbf{R}$. The number of samples given for (a single row of) $\mathbf{R}$ is $d \cdot n$ from $\mathbf{U}$ and $d \cdot \bar{n} - C$ from $\mathbf{V}$. Here C represents the number of coefficients dropped, so in particular $d \cdot \bar{n} - C$ is 128 for LPRime757 and Ilum512, and 8 for Bilbo640. Again it is only Bilbo640 for which $\bar{m} > 1$, and a similar hybrid argument can account for these extra rows of $\mathbf{R}$. The error for a sample is given by the appropriate choice from $D_e, D_{e'}$ and $D_{e''}$, for samples from $\mathbf{B}, \mathbf{U}$ and $\mathbf{V}$ respectively, plus any modulus rounding that may be applied to $\mathbf{U}$ or $\mathbf{V}$. Note that for NTRU LPRime all errors come from rounding, and so there is no 'extra' modulus rounding.

We consider up to $N = 2^{16}$ users and reevaluate the following attacks; primal lattice, Arora–Ge with Gröbner bases, and Coded BKW. This value of N comes from [76, §2.4], i.e. we have chosen the smallest power of 2 such that $N \geq 50000$. For all three of these attacks the standard deviations of the distributions from which errors and secrets are sampled play an important role, and we summarize them in Tab. 10. Our scripts to estimate the complexity of these attacks, along with various other tasks, are available at https://anonymous.4open.science/r/chained-cmpke-2C20/README.md. These scripts make use of the lwe-estimator,[32] an automated estimator based on [7], for the Arora–Ge and BKW style attacks. For the primal lattice attack we make use of the leaky-LWE-estimator,[33] see the Primal Attack paragraph below. We do not consider the dual lattice attack for the same reasons as argued in [77, §5.2.1], that is, the assumptions that make it competitive with the primal lattice attack in the core-SVP model are not compatible with recent advances in lattice sieving, i.e. the dimensions for free techniques [45], used in the 'Beyond core-SVP hardness' model. We use this latter model in our primal lattice attack estimation. We also do not, beyond their inclusion in the NTRU LPRime estimation script,[34] consider hybrid attacks. In particular, we do not consider them against either Bilbo640 or Ilum512, as [74, 77] do not consider them against either FrodoKEM or Kyber. A common theme throughout will be, though an adversary against mPKEs is granted a large number of extra samples, these extra samples are less useful than the majority of samples an adversary against an ordinary PKE would also receive, namely the $d \cdot n$ from either $\mathbf{B}$ or $\mathbf{U}$. Indeed, by a serendipitous turn of events our desire to minimize $|\widehat{\mathrm{ct}}_i|$ also minimizes the usefulness of these extra ciphertexts from a cryptanalytic perspective. For example, performing more rounding on these ciphertexts increases their error, and performing as much coefficient dropping as possible reduces their number; the hope being that the potential new avenues for cryptanalysis are nullified by these facts.

Primal Attack. The primal attack embeds [16, 65] a vector containing the error, and possibly also the secret, of an LWE or NTRU instance as a unique short vector in a lattice. It then applies lattice reduction to retrieve this vector. The primal attack requires a small number of samples and can therefore be used against either $\mathbf{S}$ or $\mathbf{R}$. In particular the optimal primal attack requires some linear multiple $c_p \cdot d \cdot n$ of samples, and typically $c_p \in [1, 2]$. We will use the methodology espoused in [74, 77] for the primal attack. This uses the NIST-round3 branch of the leaky-LWE-estimator, an implementation of [39] which studies the probabilistic behaviour of the primal attack. In the case of attacking $\mathbf{S}$ we have $d \cdot n$ samples from $\mathbf{B}$, i.e. $c_p = 1$. In the case of attacking $\mathbf{R}$ we have $d \cdot n$ samples from $\mathbf{U}$ and numerous samples from the $\mathbf{V}_i$. Due to our heavy rounding on the $\mathbf{V}_i$, the errors are far larger than those on $\mathbf{U}$ after rounding, see $\sigma_{\mathrm{r}(e')}$ and $\sigma_{\mathrm{r}(e'')}$ in Tab. 10 for their respective standard deviations. To be conservative while using the above primal attack methodology we use the smaller $\sigma_{\mathrm{r}(e')}$ for all ciphertext errors when attacking $\mathbf{R}$. We increase the number of samples available until the complexity estimate converges, which always occurs for $c_p < 2$, and take the 'Attack Estimation via simulation + probabilistic model' estimate. If we fix $c_p = 1$, that is, use only the samples from $\mathbf{U}$, then our estimates increase by less than a factor of two; in short the primal attack makes effectively no use of the extra samples afforded to it in our setting, even if we artificially assume they have much narrower errors. Our adaptation of the NIST-round3 branch outputs both classical and quantum gate counts using the estimated values for lattice sieves given in [6].

Coded BKW. The BKW style of attacks against LWE originate from the first subexponential time algorithm against the LPN problem [28]. They add samples together in such a way that the dimension of the instance is iteratively decreased, while keeping the error small enough to solve the final instance, for a practical explanation in the LWE case see [4]. The BKW style attacks are sample heavy, requiring superpolynomially many samples in $d \cdot n$. There are methods [30, §3.3] in the literature used to form new samples from already known samples, and some experimental evidence on

[32] https://bitbucket.org/malb/lwe-estimator/src/master/.
[33] https://github.com/lducas/leaky-LWE-Estimator/tree/NIST-round3.
[34] https://ntruprime.cr.yp.to/estimate-20200927.sage.

small dimensional instances suggesting the increase in the number of samples required is small when these 'sample amplification' techniques are used [56, VI]. We note that these results say that the *number* of samples required does not grow too much when sample amplification techniques are used, not that the complexity of the attack remains the same. This is discussed more below. In any case, it is standard not to consider BKW style attacks when attacking **S**. In the case of attacking **R** in an mPKE the picture becomes somewhat more mixed. We have $d \cdot n$ samples from **U** with error standard deviation $\sigma_{\mathrm{r}(e')}$ and $(N-1) \cdot (d \cdot \bar{n} - C)$ samples from the $\mathbf{V}_i$ with a larger error standard deviation $\sigma_{\mathrm{r}(e'')}$. Again, in the PKE case where $N = 2$, it is standard to assume that this is not enough samples to perform a BKW style attack. However for larger N this number of samples may be sufficient, and as such a BKW adversary may use some combination of these samples with different errors. We therefore report the estimates given by the lwe-estimator for the cost of the Coded-BKW [53] attack assuming first that the adversary has access to unlimited samples 'from **U**', and second that the adversary has access to unlimited samples 'from the $\mathbf{V}_i$', and assume that the cost for Coded-BKW lies somewhere between these estimates.

In general there is limited experimental data on the performance of the numerous BKW variants against LWE, especially on medium sized instances. Theoretical works focus on parametrizations that use standard deviations well above what is seen in practical schemes, and assume infinitely many fresh samples, although BKW does perform favorably to lattice attacks in asymptotic settings [52, 60]. We also note that there have been some improvements to BKW style attacks since Coded-BKW. In particular there has been Coded-BKW with sieving [51], which also allows quantum speedups to be incorporated during the sieving subroutine, and a number of other improvements [30]. The above, and the lack of publicly accessible estimation scripts for these new approaches, makes it difficult to precisely cost this attack against the parametrizations we suggest. We will appeal to the limited simulated and experimental results of the most recent practical study [30], namely Table 2 and Table 3, respectively. In Table 2 we see the primal attack remaining the most efficient for all simulated parameters in the low error rate setting, in particular for $\alpha = \sigma/q = 0.005$. If we restrict the primal attack to using only the samples from **U** then the highest error rate of our three parametrizations is 1.18/3329, more than an order of magnitude smaller. We recall that the primal attack makes effectively no use of samples from the $\mathbf{V}_i$. We also note that the BKW complexities estimated here assume access to an unlimited amount of samples. Looking at experimental data, the required number of samples from [30, Tab. 3] suggests that significant sample amplification would be required, e.g. for the $(n, \alpha) = (40, 0.005)$ case with 40 available samples from **U**, one is required to combine 6-tuples of samples as in [30, §3.3] to receive the required 45000000 samples. When assuming an unbounded number of fresh samples [30, Tab. 3] reports that attacking these parameters takes 12 minutes. The same work reports on solving the same instance, but limited to 1600 samples.[35] They therefore only need to take triples of samples to receive the required number, and report on some subtle difficulties encountered when creating enough triples of the correct form. This attack using triples for their sample amplification is reported to take over 3 hours. This increase in time complexity can be explained by an increase in the error standard deviation by a factor of $\sqrt{3}$ due to the sample amplification. We note the experiments of [56, VI] mentioned earlier lowered the error standard deviation by this factor before performing sample amplification to examine the effect on the required number of samples in isolation. In the more realistic setting of an adversary receiving $d \cdot n$ samples from **U**, and therefore having to perform more sample amplification, we assume the complexity increase will be greater still.

In conclusion, depending on the relative sizes of $\sigma_{\mathrm{r}(e')}$ and $\sigma_{\mathrm{r}(e'')}$ an adversary will choose to perform a certain amount of sample amplification on the samples from **U**, and potentially subsequently use samples from the $\mathbf{V}_i$. In either case, we expect the estimate we produce for an adversary given unlimited samples 'from **U**' will be an underestimate of the complexity of a BKW style attack. In general, more experimental work is needed to understand the performance of BKW variants in medium sized instances, using limited numbers of samples. We also note that the practical implications of sample amplification techniques in the ring setting [82], or whether the rounding we apply affects the algebraic structure they use, has not been investigated.

Arora–Ge with Gröbner Bases. The Arora–Ge attack [15] is a linearization attack that, by knowing the support of the error distribution, is able to create a linear system such that part of the solution encodes the secret. It then attempts to solve this linear system, in the original work by matrix inversion, and in the work that followed [5] by using Gröbner bases. The best known Arora–Ge style attacks require a superlinear number of samples [5] in $d \cdot n$, even in the bounded errors case, and therefore can only be used against **R**. The complexity of the linear system to be solved is very sensitive to the support size of the error distributions being considered, intuitively explaining why our heavily rounded extra samples do not give us a practical attack. We again use the lwe-estimator, and are able to take into account the differences between the errors of ct_0 and the $\widehat{\mathrm{ct}}_i$. If an adversary uses M of its available samples with error from some distribution D, we calculate the expected number e of distinct elements of $\mathsf{Supp}(D)$ that are sampled in these M samples. We assume the adversary can guess with probability one which e elements of $\mathsf{Supp}(D)$ have been sampled, and restrict the support of the error distribution to have size e for this estimate, making the attack cheaper. We always assume the adversary will use all the samples from **B** and **U**, and then increase the number of samples used from the $\mathbf{V}_i$, reporting the lowest complexity. For our parametrizations the most efficient Arora–Ge adversary uses very few of the samples available from the $\mathbf{V}_i$, in particular never more than those given in $N = 3$ users case. In the case of Frodo640, where there is no rounding on the $\mathbf{V}_i$, the most efficient Arora–Ge attack makes use of *all* the samples available from $N = 2^{16}$ users. While it is still secure against the attack (the estimated complexity is 2^{3193}), it shows the positive effect that rounding the modulus, and therefore increasing the size of the error support, has against the Arora–Ge attack. To make this effect more extreme we give an artificial 'Kyber like' parameter set which is Kyber512 except

[35] This value is n^2 and comes from https://www.latticechallenge.org/lwe_challenge/challenge.php. It does not represent a lattice scheme.

$D_s = D_e = B_2$, $D_{e'} = D_{e''} = B_1$ and somehow the implementer forgets to include *any* modulus rounding. We of course stress that these are not parameters suggested in [74], and even if they were, they would not have been suggested for the mPKE setting. Even so, these parameters are almost secure under our primal attack estimation methodology, at an estimated 2^{133} classical gates. It might be that hybrid attacks are relevant for these parameters, but assuming they are in a similar ballpark, then our Arora–Ge estimate, which suggests a complexity of 2^{62}, is far cheaper. Again, it uses all possible samples from the $\mathbf{V}_i$ as they have no modulus rounding, and shows in theory the necessity of a cryptanalytic model tailored to the mPKE setting.

G.2 The Cryptanalytic Model

Here we introduce the requirements we make for an mPKE scheme to be called secure. It is effectively the same model that NIST laid out in their call for proposals [75] but we take into account sample heavy attacks, and the impact of having many more samples than usual, as described above.

Cost of Attacks. We require an mPKE to be parametrized such that none of the attacks listed above give costs (whether in gate count, or in the 'ring operations' reported by the lwe-estimator) of less than 2^{143} classically, and 2^{117} quantumly (where appropriate).

These gate counts are from [75] and [64]. Indeed, for Security Level I, [75] requires 2^{143} classical gates, and, using the updated values of [64, Tab. 12] requires 2^{117} quantum gates. We note the strange phenomenon that the lower the MAXDEPTH allowed to a quantum computer, the harder the quantum gate count requirement becomes to satisfy. This follows from the poor parallelizability of quantum search, and therefore the more constrained the depth of a quantum computation, the more it must rely on parallelization, and the less efficient it becomes. In our case, this means that as the MAXDEPTH decreases, breaking AES-128 becomes harder, see [64] for detailed exposition. One could therefore argue that taking a smaller MAXDEPTH could render our parametrizations insecure with respect to quantum gate count, however we follow [75] in setting the minimum considered MAXDEPTH as 2^{40}. See the discussion [77, §5.3] for the potential impact of refinements to the primal attack on our gate count estimates.

Decryption Failure Rate. We require an mPKE to be such that the DFR remains below 2^{-120}, the largest of a final round lattice KEM [41, Tab. 1]. The largest DFR of any of our parametrizations is 2^{-125} for Ilum512. We note that for classical PKEs the DFR is often 0, that is, they exhibit perfect correctness. This is also the case for NTRU LPRime and our reparametrization thereof. The DFR is formally defined as the amount the expectation in Def. A.3 differs from 1. In the lattice PKEs with non zero DFR, a decryption failure can be used within *reaction attacks* [42, 54, 57] to learn information about the secret. Decryption failures also make future decryption failures easier to trigger [42]. Even successful decryptions can be used to inform the search for decryption failures [26]. Therefore, PKEs which are not parametrized to have perfect correctness instead aim to minimize their DFR. The concrete effect of the DFR in the mKEM setting is described by [66, Thm. 4.1]. We leave as an open research problem the concrete importance of the DFR in the CmPKE setting.

Table 10: Standard deviations for the various secret and error distributions, see Figure 5. The values $\sigma_{\mathbf{r}(e')}$ and $\sigma_{\mathbf{r}(e'')}$ denote the standard deviation of errors after rounding U and V respectively. As LPRime757 uses rounding for all errors, we report these errors as $\sigma_e, \sigma_{e'}$, and $\sigma_{e''}$.

Scheme	σ_s	σ_e	$\sigma_{e'}$	$\sigma_{e''}$	$\sigma_{\mathrm{r}(e')}$	$\sigma_{\mathrm{r}(e'')}$
Bilbo640	2.91	2.91	2.91	2.91	2.91	2364
Ilum512	$\sqrt{3/2}$	$\sqrt{3/2}$	1	1	1.18	120
LPRime757	$\sqrt{242/757}$	$\sqrt{2/3}$	$\sqrt{2/3}$	568	$\sigma_{e'}$	$\sigma_{e''}$

The results of our security estimation are given in Fig. 31. In all cases it is the primal lattice attack that remains the most efficient. We do not cost quantum variants of Coded-BKW or Arora–Ge with Gröbner bases, and leave this as future work. Below we discuss each of the new parametrizations in turn. In particular we discuss any subtle algorithmic changes and any differences (beyond the newly considered attacks) with their original cryptanalyses. We also mention any elements of their original cryptanalyses that we are able to incorporate, that are not required to satisfy our cryptanalytic model for mPKEs.

Ilum512. We make one substantive change in our cryptanalysis of Ilum512 compared to Kyber512. It comes from the different amounts of rounding on $\mathbf{U}$ and the $\mathbf{V}_i$, which firstly increases the DFR to 2^{-125}. More importantly though, we no longer satisfy the arguments of [77, §4.4] regarding estimating the primal attack on $\mathbf{R}$ with equal standard deviation for the secret and error distributions. That is, as we have reduced the amount of rounding on $\mathbf{U}$, we have $\sigma_{\mathrm{r}(e')} < \sigma_s \approx 1.22$, where the latter standard deviation is of the distribution used to generate $\mathbf{R}$. Using the reduction of [14] which allows one to sample the secret of an LWE instance from the same distribution as the error (and ignoring the samples this costs), we must therefore assume the elements of $\mathbf{R}$ are also drawn from the distribution with the smaller standard deviation $\sigma_{\mathrm{r}(e')}$. This must be taken into account for all three attacks we consider.

Bilbo640. Other than increasing the amount of bits of randomness required to sample from $D_s, D_e, D_{e'}$, and $D_{e''}$ (see below for an explanation why) and performing modulus rounding on the $\mathbf{V}_i$, we make no substantive changes to the algorithms of FrodoKEM to attain Bilbo640. Nor, other than including Arora–Ge and BKW style attacks, do we make any changes to our cryptanalysis. We may however reuse part of the security methodology of FrodoKEM. For example, we may wish to appeal to [74, Thm. 5.9], which relates the IND-CPA security of the PKE to the LWE problem, for up to our $N = 2^{16}$ users. As in [66, Lem. 5.1], modulo notational differences, we can adapt this to the mPKE setting as follows

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{mPKE},N}(\mathcal{A}) \leq N \cdot \bar{n} \cdot \mathsf{Adv}^{\mathsf{LWE}}_{n,n}(\mathcal{B}_1) + \bar{m} \cdot \mathsf{Adv}^{\mathsf{LWE}}_{n,n+N\bar{m}}(\mathcal{B}_2). \tag{2}$$

Intuitively we have a hybrid over $N \cdot \bar{n}$ columns of $\mathbf{S}$, each having n samples, and a hybrid over $\bar{m}$ rows of $\mathbf{R}$, each having $n + N \cdot \bar{n}$ samples. For both $\mathbf{S}$ and $\mathbf{R}$ the columns and rows, respectively, are secrets of length n. Conservatively therefore, in our setting we may take the larger of the two advantages and multiply by $N\bar{n} + \bar{m}$ to upper bound the advantage against the mPKE . From Fig. 31 we

Figure 31: All values are given as log base 2. The columns P-S-c, P-S-q, P-R-c, and P-R-q represent the classical primal attack against S, the quantum primal attack against S, the classical primal attack against R, and the quantum primal attack against R, respectively. The columns BKW-U and BKW-V represent the Coded-BKW attack assuming unlimited samples 'from U' and 'from the $\mathbf{V}_i$', respectively. The column AG represents the Arora–Ge with Gröbner bases attack.

Scheme	P-S-c	P-S-q	P-R-c	P-R-q	BKW-U	BKW-V	AG	DFR
Bilbo640	164	154	163	154	224	334	4601	-129
Ilum512	151	143	150	142	157	224	2227	-125
LPRime757	177	166	177	166	184	259	1493	-∞

assume a uniform t gate classical adversary has advantage no more than $2^{-163} \cdot t$ against either of the LWE problems, noting that we have gone from a Core-SVP estimate for this quantity in [74] to a gate count estimate here. Therefore a t gate IND-CPA $\mathcal{A}$ has an advantage of no more than $2^{-144} \cdot t$, and this mPKE is then a starting point for the constructions of Sec. 3.2. Another facet of FrodoKEM's security analysis we may wish to reuse is their Rényi divergence arugment. The main security theorem of FrodoKEM [74, Thm. 5.1] regarding the IND-CCA security of the KEM, while not applicable here, accounts for the Rényi divergence between the actual sampled distribution χ_{Frodo} and the rounded Gaussian Ψ_s, as well as the number of samples drawn from χ_{Frodo}. The number of samples drawn from χ_{Frodo} is $2n\bar{n} + 2\bar{m}n + \bar{m}\bar{n} = 20554$, which increases to $2\bar{m}n + N \cdot (2n\bar{n} + \bar{m}\bar{n}) \leq 675293184$ for χ_{Bilbo640} in the mPKE setting with $N \leq 2^{16}$. As Thm. 3.6, the respective theorem for CmPKEs, does not proceed via a search problem, i.e. the OW-PCA problem of FrodoKEM, similar Rényi divergence arguments are not made. However, we give here a distribution to show the plausibility of efficiently sampling sufficiently close distributions in the CmPKE setting. Using the methods of [61, §5.2] we produce the following distribution χ_{Bilbo640}, which has a Rényi divergence of 2.144×10^{-10} from $\Psi_{2.9\sqrt{2\pi}}$[36] at order 200. It is symmetric around 0 and described in the following figure as $\{\pm x : p(\pm x) \cdot 2^{32}\}$,

- 0 : 587928496
- ±1 : 554318271
- ±2 : 464582536
- ±3 : 346126223
- ±4 : 229230439
- ±5 : 134950272
- ±6 : 70621314
- ±7 : 32851452
- ±8 : 13583937
- ±9 : 4992798
- ±10 : 1631188
- ±11 : 473696
- ±12 : 122271
- ±13 : 28052
- ±14 : 5720
- ±15 : 1037
- ±16 : 167
- ±17 : 24
- ±18 : 3

This means that by using exactly twice as much randomness to sample an element of χ_{Bilbo640} we can keep the $\exp(s \cdot \mathrm{D}_\alpha(P\|Q))^{1-1/\alpha}$ term of [74, Thm. 5.1] below its value in the Frodo640 case, even in the presence of these extra samples.

LPRime757. We make a small algorithmic change in LPRime757 compared to NTRU LPRime to reduce the size of the $\mathbf{V}_i$ and allow slightly larger weights w than otherwise. To reduce the size of $\mathbf{V}$ in LPRime757 we must ensure the rounding procedure Top has codomain $\{0, \ldots, \tau - 1\}$ for $\tau < 16$. In particular we define Top′ which achieves this for $\tau = 4$ as follows

$$\mathsf{Top}'(C) = (\tau_1(C+\tau_0)+2^{15})/2^{16}, (\tau_0, \tau_1, \tau_2, \tau_3) = (3011, 33, 1995, 1978).$$

We note the that powers of 2 have each increased by one, compared to [21, §3.3]. This allows us a slightly larger weight w than otherwise. We do not alter Right. For our cryptanalysis we calculate a 'per coefficient' variance for secret polynomials of NTRU LPRime. The secret polynomials of NTRU LPRime are degree d and have exactly w non zero coefficients. These w positions are chosen uniformly and the value for each of them is independently and uniformly sampled from $\{-1, 1\}$, i.e. they are fixed weight, but not fixed sum. Given (w, d) and a fixed coefficient in an NTRU LPRime secret polynomial, its probability taken over all possible secret polynomials of being 0 is $1 - w/d$. Similarly, its probability of being either 1 or -1 is $w/2d$. We therefore calculate the variance using $p(\pm 1) = w/2d$ and $p(0) = 1 - w/d$. We reuse part of the security methodology of [21]. In particular we choose a weight w such that $1/4 \leq w/d \leq 1/2$, and parameters that satisfy both 'bulletproof' definitions for Level I security.[37] We also note that, by the necessary alterations to Eq. (2) in the LPRime757 mPKE case we can absorb the hybrid loss factor of $N + 1$.

[36] We have the relation $s = \sigma\sqrt{2\pi}$ for Ψ_s.

[37] https://ntruprime.cr.yp.to/estimate-20200927.sage using `run(757, 7879, 242, 'product')`.

Contents