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
eamonn.postlethwaite.2016@live.rhul.ac.uk

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 [14, 19, 52] 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.

CCS CONCEPTS

• **Security and privacy** → *Key management*; *Public key encryption.*

KEYWORDS

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

ACM Reference Format:
Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. 2021. A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3460120.3484817

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 [21]. 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 $N \leq 50000$ [52, §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 [35] and Signal [55] 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

*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.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484817

regular pace, thus limiting the scope of a compromise. This practice, called *ratcheting*, provides post-compromise security (PCS) and forward secrecy (FS) [7, 25, 28]. It also forms the basis for more sophisticated techniques [8, 9, 19] providing various levels of a stronger notion called post-compromise forward security (PCFS) [8–10].

Continuous Group Key Agreement. The notions of continuous (group) key agreement (CKA and CGKA) were put forward [6–10] 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 [19]. It underlies the IETF draft standard for secure messaging, MLS [14, 52]. TreeKEM derives its name from its use of *ratchet trees* (bottom left of Fig. 1, p. 3), and a significant amount of research and engineering effort has been undertaken to study the efficiency and security implications of this signature feature [6, 8–10, 20, 58].

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 [57]. 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 [57]), 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.

1.1.1 The Chained CmPKE Protocol. At a very high level, our protocol is inspired by the Chained mKEM protocol [18, 20]. 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,

[1] A documented property [6] 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 [24] 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*" [24]. This illustrates that mobile data can be a limited and expensive resource.

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

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 [46]. 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

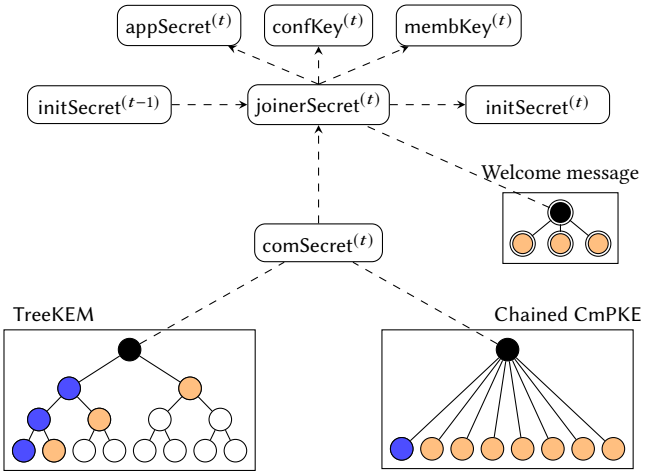


Figure 1: Initialization of a new epoch t, here with a group of $N = 8$ members. A dashed arrow from X to Y means that Y is computed by passing X (and possibly other values) to a HKDF, a dashed line means that $X = Y$.
Here, the leftmost user in the TreeKEM (resp. Chained CmPKE) box initiates a new epoch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

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) [43] 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 $(\mathsf{ct}_0, \widehat{\mathsf{ct}}_i)$. 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 token-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 [37], 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 [43] and a *key-committing* SKE [2, 32, 33, 37], 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 [15, 45] 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 [43]. 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 [43] 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 [43] by revisiting the designs of the NIST submissions [16, 51, 53] 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 [51] called Bilbo640, a variant of Kyber512 [53] called Ilum512, and a variant of LPRime653 [16] 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 [43] 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 [4, 11] and BKW [22] 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 the full version of this paper [40], 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. [10] that was used to analyze the TreeKEM version 10 in MLS, which is itself an extension of [9]. 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 [10] 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 [19] originates from *Asynchronous Ratcheting Trees* (ART) introduced by Cohn-Gordon et al. [27]. 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. [8] and Bhargavan et al. [20] 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. [10] 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 [6] enjoys efficiency advantages for large groups maintained by a small number of 'administrators'. Re-randomized TreeKEM [8] and TreeKEM with active security [9] improve the PCFS property against passive and active adversaries, respectively, but require relatively heavy cryptographic primitives. Finally, *Causal* TreeKEM [58] 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 [25, 26]. The notion of *Continuous Key Agreement* (CKA) (that is, CGKA in the two-party setting) is studied in [7]. This generalizes the public-key ratchet of Signal's Double Ratchet protocol [49]. The X3DH protocol [50] of the Signal protocol, used to establish the initial secret key required for CKA, is studied in [23, 39]. 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 [54] posits that post-quantum signatures are generally less efficient than post-quantum KEMs. Similarly, McTiny [17] and Post-Quantum WireGuard [41] 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 the full version of this paper [40]. 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 $|\Pr[(b, \mathsf{k}) \leftarrow\!\$ \{0,1\} \times \mathcal{K}, (\mathsf{M}_0, \mathsf{M}_1) \leftarrow \mathcal{A}(1^\kappa), \mathsf{ct}^* \leftarrow \mathsf{Enc}_\mathsf{s}(\mathsf{k}, \mathsf{M}_b), b' \leftarrow \mathcal{A}^{C(\cdot)}(\mathsf{ct}^*) : b =$

$b'] - 1/2| \leq \mathsf{negl}(\kappa)$, where $C(\mathsf{ct})$ returns $\mathsf{Dec_s}(\mathsf{k}, \mathsf{ct})$ if $\mathsf{ct} \neq \mathsf{ct}^*$ and $\perp$ otherwise.

We also define key commitment for a SKE [33] 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, 32, 33, 37], we define this notion by providing the (non-uniform) adversary oracle access to $\mathsf{Enc_s}$ and $\mathsf{Dec_s}$, 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 $\Pr[(\mathsf{k}_0, \mathsf{k}_1, \mathsf{ct}) \leftarrow \mathcal{A}^{\mathsf{Enc_s}, \mathsf{Dec_s}}(1^\kappa), (\mathsf{M}_b \leftarrow \mathsf{Dec_s}(\mathsf{k}_b, \mathsf{ct}))_{b\in\{0,1\}} : \mathsf{M}_0 \neq \perp \wedge \mathsf{M}_1 \neq \perp] \leq \mathsf{negl}(\kappa)$.

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

2.2 Decomposable Multi-Recipient PKE

Decomposable multi-recipient PKE (mPKE) was introduced in [43]. Similarly to a standard mPKE [12, 45, 56], 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 [43] 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 [43] 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:

- $\mathsf{mSetup}(1^\kappa) \rightarrow \mathsf{pp}$: On input the security parameter 1^κ, it outputs a public parameter pp.
- $\mathsf{mGen}(\mathsf{pp}) \rightarrow (\mathsf{ek}, \mathsf{dk})$: On input a public parameter pp, it outputs a pair of encryption key and decryption key $(\mathsf{ek}, \mathsf{dk})$.
- $\mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M}; \mathsf{r}_0, (\mathsf{r}_i)_{i\in[N]}) \rightarrow \vec{\mathsf{ct}} = (\mathsf{ct}_0, (\widehat{\mathsf{ct}}_i)_{i\in[N]})$: The (decomposable) encryption algorithm running with randomness $(\mathsf{r}_0, \mathsf{r}_1, \cdots, \mathsf{r}_N)$, splits into a pair of algorithms $(\mathsf{mEnc^i}, \mathsf{mEnc^d})$:
  - $\mathsf{mEnc^i}(\mathsf{pp}; \mathsf{r}_0) \rightarrow \mathsf{ct}_0$: On input a public parameter pp and randomness r_0, it outputs an (encryption key *independent*) ciphertext ct_0.
  - $\mathsf{mEnc^d}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{M}; \mathsf{r}_0, \mathsf{r}_i) \rightarrow \widehat{\mathsf{ct}}_i$: 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 $\widehat{\mathsf{ct}}_i$.
- $\mathsf{mDec}(\mathsf{dk}, \mathsf{ct}_i) \rightarrow \mathsf{M}$ 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 $\mathsf{mEnc^i}$ is the null-function and $\mathsf{mEnc^d}$ 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 [36], where the latter informally states that the ciphertext has high min-entropy. Due to space constraints, definitions are given in the full version of this paper [40]. 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 $|\Pr[\mathcal{A}\ \textit{wins}] - 1/2| \leq \mathsf{negl}(\kappa)$. If $\mathcal{A}$ is not given access to the corruption oracle C, this game corresponds to standard IND-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, 32, 33, 37], 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 [32], (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]

[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.

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 (ek, 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 $\mathsf{pp} \leftarrow \mathsf{CmSetup}(1^\kappa)$, $((\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{CmGen}(\mathsf{pp}))_{i\in[N]}$, 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 token 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 [9]), 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 $\perp$).

[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.

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 the full version of this paper [40].

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 [44]. Due to space constraints, the detailed construction and its associated proofs of correctness and security are given in the full version of this paper [40].

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 $[C \cap S = \emptyset]$ **then**
  return b
return $[b = b']$

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

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 $[C \cap S = \emptyset]$ **then**
  return b
return $[b = b']$

Corruption Oracle $\mathcal{C}(i)$

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

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 $[\mathsf{M}_0 \neq \mathsf{M}_1] \wedge [\mathsf{M}_0 \neq \bot] \wedge [\mathsf{M}_1 \neq \bot]$

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**.**

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 [20]. 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 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 [9, 10]. 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. A.

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 [10, 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

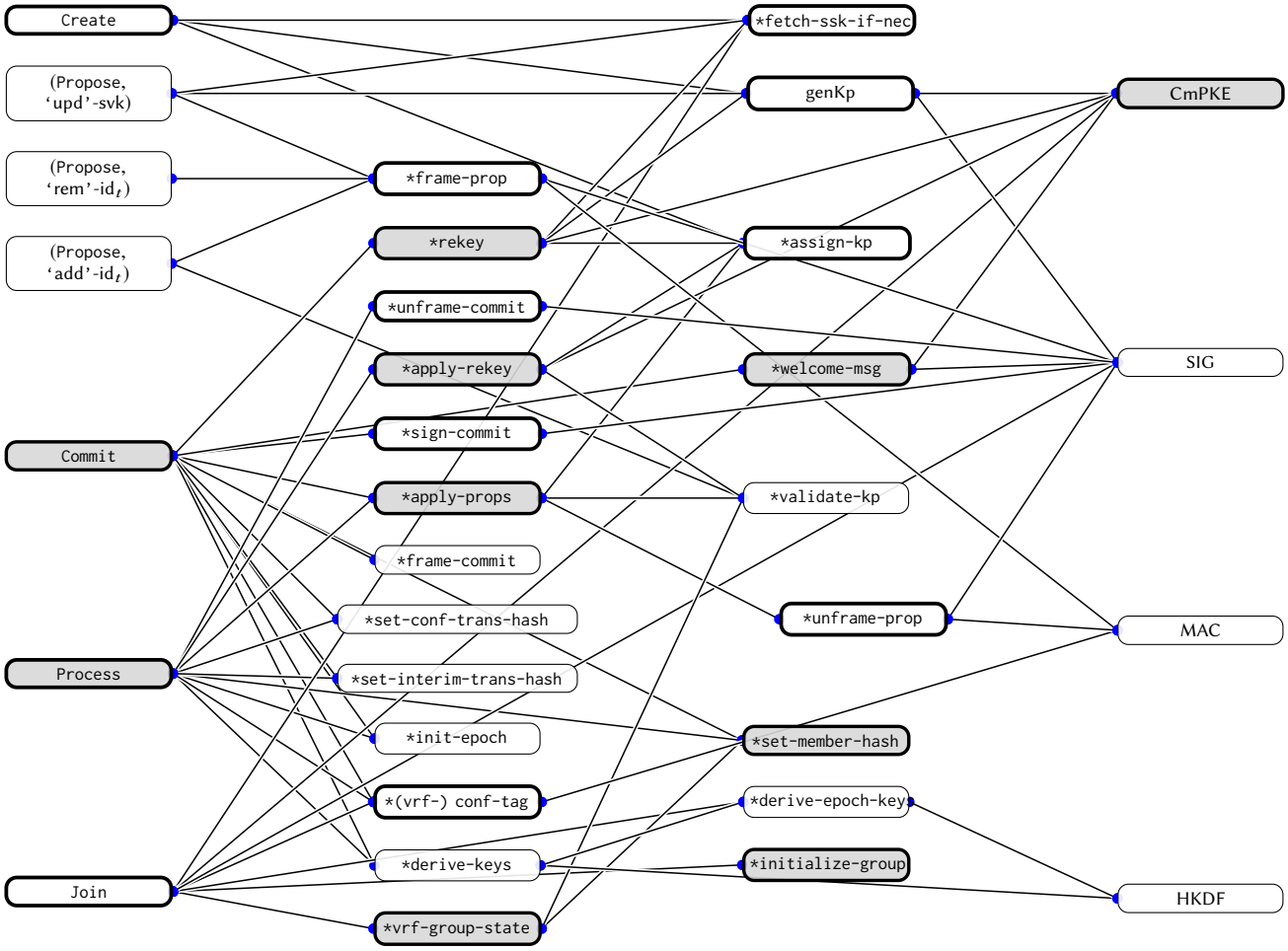


Figure 4: Call graph of Chained CmPKE. We use the notations `function`, `function` and `function` to denote functions that undergo respectively minimal, moderate and strong changes compared to [9, 10].

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.

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.

[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 [10]). 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.

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 [13], 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 [43], 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 [53], NTRU LPRime [16] and FrodoKEM [51]. 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.

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 [10] 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)			
	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$	$\lvert\mathsf{ek}\rvert$	$\lvert\mathsf{ct}_0\rvert$	$\lvert\widehat{\mathsf{ct}}_i\rvert$	$\lvert\mathsf{sig}\rvert$
[10]	$\lceil \log N \rceil$	$\lceil \log N \rceil$	$\lceil \log N \rceil$	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	$2(N-1)$	$2(N-1)$	$2N$

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. [10]. The security model presented by [10] is an extension of [9] 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 [10] 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, thus our model is a strict generalization of prior models. The security of Chained CmPKE is established by Thm. 4.1. Our detailed security model and the proof of Thm. 4.1 are provided in the full version of this paper [40].

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** that decides whether the epoch key is secure, 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 [43], reproduced in Fig. 5, which adapts the Lindner–Peikert framework [47] to the mPKE setting. As observed by [43], Fig. 5 can be readily applied to the (possibly alternative) finalists FrodoKEM [51], Kyber [53], NTRU LPRime [16] and Saber [30]. We take this one step further and propose new parametrizations of [16, 51, 53] 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.

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	[53]	768 (+32)	640	128
Ilum512	Sec. 5	768	704	48
LPRime653	[16]	865 (+32)	865 (+32)	128
LPRime757	Sec. 5	1076	1076	32
Frodo640	[51]	9600 (+16)	9600	120
Bilbo640	Sec. 5	10240	10240	24
SIKEp434	[42]	330	330	16

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 [16, 51, 53]. Finally, the full version of this paper [40] contains an additional section that 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 [16] and [53] respectively. Concretely, for modulus rounding we will focus on the Compress and Decompress functions of [53]. 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 [51], 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.

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

1: $\mathbf{A} \leftarrow\!\!\$\ R_q^{n\times n}$
2: **return** $\mathsf{pp} := \mathbf{A}$

Algorithm 2 $\mathsf{mGen}(\mathsf{pp})$

1: $\mathbf{S} \leftarrow\!\!\$\ D_s^{n\times\bar{n}}$
2: $\mathbf{E} \leftarrow\!\!\$\ D_e^{n\times\bar{n}}$
3: $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
4: **return** $\mathsf{ek} := \mathbf{B}, \mathsf{dk} := \mathbf{S}$

Algorithm 3 $\mathsf{mEnc}(\mathsf{pp}, (\mathsf{ek}_i)_{i\in[N]}, \mathsf{M})$

1: $\mathsf{r}_0 := (\mathbf{R}, \mathbf{E}') \leftarrow\!\!\$\ D_s^{\bar{m}\times n} \times D_{e'}^{\bar{m}\times n}$
2: $\mathsf{ct}_0 := \mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0)$
3: **foreach** $i \in [N]$ **do**
4: $\quad \mathsf{r}_i := \mathbf{E}_i'' \leftarrow\!\!\$\ D_{e''}^{\bar{m}\times\bar{n}}$
5: $\quad \widehat{\mathsf{ct}}_i := \mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{M}; \mathsf{r}_0, \mathsf{r}_i)$
6: **return** $\vec{\mathsf{ct}} := (\mathsf{ct}_0, \widehat{\mathsf{ct}}_1, \ldots, \widehat{\mathsf{ct}}_N)$

Algorithm 4 $\mathsf{mEnc}^{\mathsf{i}}(\mathsf{pp}; \mathsf{r}_0)$

1: $\mathbf{U} \leftarrow \mathbf{RA} + \mathbf{E}'$
2: **return** $\mathsf{ct}_0 := \mathbf{U}$

Algorithm 5 $\mathsf{mEnc}^{\mathsf{d}}(\mathsf{pp}, \mathsf{ek}_i, \mathsf{M}; \mathsf{r}_0, \mathsf{r}_i)$

1: $\mathbf{V}_i \leftarrow \mathbf{RB}_i + \mathbf{E}_i'' + \mathsf{Encode}(\mathsf{M})$
2: **return** $\widehat{\mathsf{ct}}_i := \mathbf{V}_i$

Algorithm 6 $\mathsf{mDec}(\mathsf{sk}, \mathsf{ct})$

1: **return** $\mathsf{M} := \mathsf{Decode}(\mathbf{V} - \mathbf{US})$

Figure 5: Lattice-based mPKE construction of [43]. R is the base ring, $D_s, D_e, D_{e'}, D_{e''}$ are distributions over R.

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.

Coefficient Dropping. When trying to decode a message M from $(\mathbf{U}, \mathbf{V}_i)$ using $\mathbf{S}$, 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.

Modulus Rounding. Rounding away the least significant bits of $\mathbf{B}, \mathbf{U}$, and $\mathbf{V}_i$ provides more compact $\mathsf{ek}, \mathsf{ct}_0$ 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.

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 [29] or analyzed [31, 38]. 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. The cryptanalytic model, known attacks and concrete security estimates of our schemes against them are provided in the full version of this paper [40].

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.

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. 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 [5, 59]. 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 [53], we have $\{R : \mathbb{Z}[x]/(x^{256}+1), n : k, q : q, \bar{n} = \bar{m} : 1, D_s = D_e : B_{\eta_1}, D_{e'} = D_{e''} : B_{\eta_2}\}$.

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

Table 3: Parameter sets of Kyber512 and Ilum512, using the notation of [53], 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

requires 32 bits of randomness per sample, see Tab. 4. We have $\{R : \mathbb{Z}, n : n, q : 2^{16}, \bar{n} : \bar{n}, \bar{m} : \bar{m}, D_s = D_e = D_{e'} = D_{e''} = \chi_{\text{Bilbo640}}\}$.

Table 4: Parameter sets of Frodo640 and Bilbo640, using the notation of [51], plus b/s to denote the random bits needed to sample an integer coefficient, and $\{\mathsf{D_B}, \mathsf{D_U}, \mathsf{D_V}\}$ to denote the bits/coefficient in $\{\mathsf{B}, \mathsf{U}, \mathsf{V}\}$ (instead of a common D in [51]).

Scheme	n	$\mathsf{D_B}$	$\mathsf{D_U}$	$\mathsf{D_V}$	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

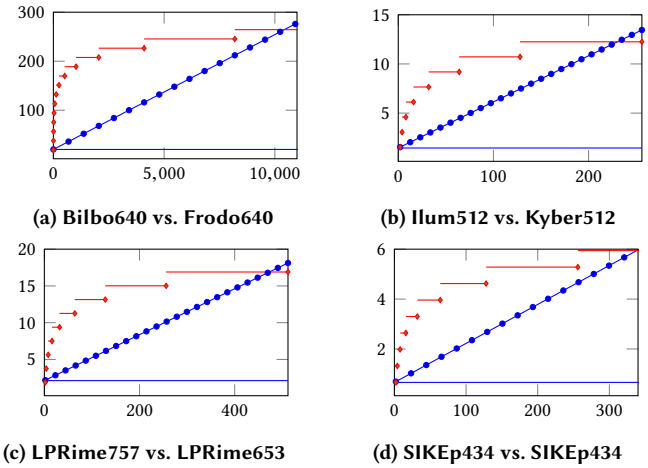
NTRU LPRime. We introduce a new parameter set, LPRime757. We reduce the number of bits per entry of $\mathbf{V}$ from 4 to 2, and must increase the modulus, and decrease the weight, to account for this, see Tab. 5. The authors of NTRU LPRime [16] place a great emphasis on having $(x^p - x - 1)$ irreducible in $\mathbb{Z}_q$ and a DFR equal to zero. This is also the case for LPRime757.

We slightly alter the rounding function Top to Top′ which maintains perfect correctness while allowing us a larger weight than otherwise. 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 [16], 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 $\text{Right}(\text{Top}'(C)) - C$ over all $C \in \mathbb{Z}_q$. We have $\{R : \mathbb{Z}[x]/(x^p - x - 1), n = \bar{n} = \bar{m} : 1, q : q, D_s : \text{Short}, D_e = D_{e'} : X, D_{e''} : Y\}$.

Table 5: Parameter sets of LPRime653 and LPRime757, using the notation of [16]. We drop $p - I$ coefficients from $\mathbf{V}$.

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 [43]. Bandwidth-wise, it seems asymptotically optimal, as $\widehat{\text{ct}}_i$ is κ bits. Security-wise, [43] provides a security reduction to the SSDDH problem [34], 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.



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

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

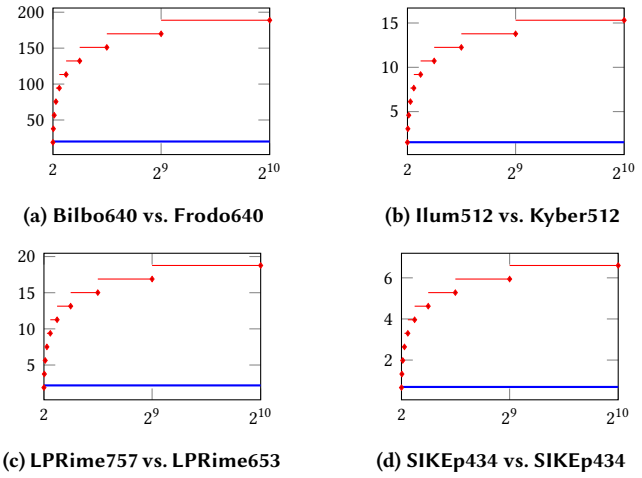
Figure 6: The graphs "*X vs Y*" give the bandwidth overhead (in term of encryption keys and ciphertexts) of commit messages when using Chained CmPKE with the CmPKE X (—● when uploaded, —— when downloaded), compared to TreeKEM with the KEM Y (—♦— *both* when uploaded and downloaded). The x-axis is the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

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 [48].
- mPKE. If we choose to rely on isogeny-based assumptions, we may use the SIKE mPKE from [43]. If we rely on lattice-based assumptions, we may use one of our three lattice-based mPKEs from Sec. 5: Bilbo640, Ilum512, LPRime757.

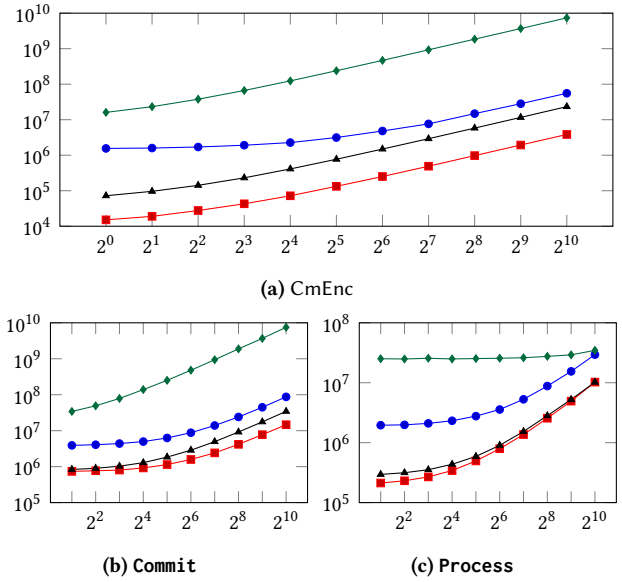
The mKEMs which are at the core of the mPKEs are implemented in C, starting from the optimized public platform-independent implementations of [16, 42, 51, 53]. For Ilum512 and SIKEp434, the changes are straightforward. The modifications for Bilbo640 are only slightly more involved due to the new distribution 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.



(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).

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]

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

[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).

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, and Ludovic Perret. 2014. Algebraic Algorithms for LWE. Cryptology ePrint Archive, Report 2014/1018. https://eprint.iacr.org/2014/1018.

[5] 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.

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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.

[11] 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

[12] Manuel Barbosa and Pooya Farshim. 2007. Randomness reuse: Extensions and improvements. In *IMA International Conference on Cryptography and Coding*. Springer, 257–276.

[13] Richard Barnes. 2018. [MLS] Efficiency and "Ampelmann trees". IETF Mail Archive. https://mailarchive.ietf.org/arch/msg/mls/INcV28Jth25m_l__NMmQIYp13Po/.

[14] 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.

[15] 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

[16] 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.

[17] 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.

[18] Benjamin Beurdouche. 2020. *Formal Verification for High Assurance Security Software in F**. Ph.D. Dissertation.

[19] 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

[20] 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

[21] WhatsApp Blog. 2020. Two Billion Users – Connecting the World Privately. WhatsApp Blog. https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately/.

[22] 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

[23] 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.

[24] Cable.co.uk. 2021. Worldwide Mobile Data Pricing 2021 | 1GB Cost in 230 Countries. https://www.cable.co.uk/mobiles/worldwide-data-pricing/.

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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.

[31] 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

[32] 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

[33] 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.

[34] 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

[35] Electronic Frontier Foundation. 2021. Lavabit. EFF. https://www.eff.org/fr/cases/lavabit.

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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. Cryptology ePrint Archive. Full version - https://eprint.iacr.org/2021.

[41] 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

[42] 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.

[43] 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

[44] 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

[45] 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

[46] 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.
[47] 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
[48] 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.
[49] Moxie Marlinspike and Trevor Perrin. 2016. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/ https://signal.org/docs/specifications/doubleratchet/.
[50] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH key agreement protocol. https://signal.org/docs/specifications/x3dh/ https://signal.org/docs/specifications/x3dh/.
[51] 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.
[52] 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.
[53] 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.
[54] 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
[55] Signal. 2021. Grand jury subpoena for Signal user data, Central District of California. Signal Blog. https://signal.org/bigbrother/central-california-grand-jury/.
[56] 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
[57] Speedtest. 2021. Speedtest Global Index — Internet Speed around the world. https://www.speedtest.net/global-index.
[58] Matthew Weidner. 2019. *Group messaging for secure asynchronous collaboration*. MPhil dissertation. University of Cambridge, Cambridge, UK.
[59] 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 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 [10].

A.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 [10], 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.

A.2 Protocol Algorithms

The main protocol is depicted in Figs. 9 and 10. The associated helper functions are depicted in Figs. 12 to 16. In these figures, the differences from TreeKEM version 10 in MLS considered by [10] 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 (Create, 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

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.

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 (`Commit`, $\vec{p}$, 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.

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 $(\mathsf{id}, \mathsf{kphash}, \widehat{\mathsf{ct}}_{\mathsf{id}})$ 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. 16). 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 $\mathcal{F}_{\text{KS}}$ 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 $(w_0, \widehat{w}_{\text{id}})$. 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)

req $G = \bot \wedge \mathsf{id} = \mathsf{id}_{\text{creator}}$
G.groupid $\leftarrow_\$ \{0,1\}^\kappa$; G.joinerSecret $\leftarrow_\$ \{0,1\}^\kappa$
G.epoch $\leftarrow 0$
G.member[$*$] $\leftarrow \bot$; G.memberHash $\leftarrow \bot$
▷ memberHash is equivalent to the treehash used in TreeKEM
G.confTransHash-w.o-'id_c' $\leftarrow \bot$
G.confTransHash $\leftarrow \bot$
G.certSvks[$*$] $\leftarrow \emptyset$
G.pendUpd[$*$] $\leftarrow \bot$; G.pendCom[$*$] $\leftarrow \bot$
G.id $\leftarrow$ id
try ssk $\leftarrow$ *fetch-ssk-if-nec(G, svk)
(kp, dk) $\leftarrow$ genKp(id, svk, ssk)
$G \leftarrow$ *assign-kp(G, id, kp)
G.ssk $\leftarrow$ ssk
G.member[id].dk $\leftarrow$ dk
G.memberHash $\leftarrow$ *derive-member-hash(G)
(G, confKey)
    $\leftarrow$ *derive-epoch-keys(G, G.joinerSecret)
confTag $\leftarrow$ *gen-conf-tag(G, confKey)
$G \leftarrow$ *set-interim-trans-hash(G, confTag)

Input (Propose, 'upd'-svk)

req $G \neq \bot$
try ssk $\leftarrow$ *fetch-ssk-if-nec(G, svk)
(kp, dk) $\leftarrow$ genKp(id, svk, ssk)
$P \leftarrow$ ('upd', kp)
$p \leftarrow$ *frame-prop(G, P)
G.pendUpd[p] $\leftarrow$ (ssk, dk)
return p

Input (Propose, 'add'-id_t)

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

Input (Propose, 'rem'-id_t)

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

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

req $G \neq \bot$
$G' \leftarrow$ *init-epoch(G)
try (G', *upd*, *rem*, *add*) $\leftarrow$ *apply-props(G, G', $\vec{p}$)
req ($*$, 'rem'-id) $\notin$ *rem* $\wedge$ (id, $*$) $\notin$ *upd*
addedMem $\leftarrow \{\, \mathsf{id}_t \mid (*, \text{'add'-}\mathsf{id}_t\text{-}*) \in \mathit{add} \,\}$
▷ Recipients of the welcome message
receivers $\leftarrow G'$.memberIDs() $\setminus$ addedMem
▷ Recipients of the new commit secret (including the committer)
try (G', comSecret, kp, T, $\vec{\mathsf{ct}} = (\widehat{\mathsf{ct}}_{\mathsf{id}})_{\mathsf{id} \in \text{receivers}}$)
    $\leftarrow$ *rekey(G', receivers, id, svk)
$G' \leftarrow$ *set-member-hash(G')
propIDs $\leftarrow$ ()
foreach $p \in \vec{p}$ **do** propIDs $\mathbin{+\!\!\leftarrow}$ H(p)
$C_0 \leftarrow$ (propIDs, kp, T)
sig $\leftarrow$ *sign-commit(G, C_0)
$G' \leftarrow$ *set-conf-trans-hash(G, G', id, C_0, sig)
(G', confKey, joinerSecret)
    $\leftarrow$ *derive-keys(G, G', comSecret)
confTag $\leftarrow$ *gen-conf-tag(G', confKey)
$c_0 \leftarrow$ *frame-commit(G, C_0, sig, confTag)
$G' \leftarrow$ *set-interim-trans-hash(G', confTag)
$\vec{c} \leftarrow \emptyset$
foreach id $\in G$.memberIDs() **do**
    if id $\in$ receivers **then**
        $\vec{c} \mathbin{+\!\!\leftarrow} \widehat{c}_{\mathsf{id}} = (\mathsf{id}, \widehat{\mathsf{ct}}_{\mathsf{id}})$
    else
        $\vec{c} \mathbin{+\!\!\leftarrow} \widehat{c}_{\mathsf{id}} = (\mathsf{id}, \bot)$
if *add* $\neq$ () **then**
    (G', w_0, $\vec{w}$) $\leftarrow$ *welcome-msg(G', addedMem, joinerSecret, confTag)
else
    $w_0 \leftarrow \bot$; $\vec{w} \leftarrow \emptyset$
G.pendCom[c_0] $\leftarrow$ (G', $\vec{p}$, *upd*, *rem*, *add*)
return (c_0, $\vec{c}$, w_0, $\vec{w}$)

Input Key

req $G \neq \bot$
k $\leftarrow G$.appSecret
G.appSecret $\leftarrow \bot$
return k

Figure 9: Main protocol: Create, Propose, and Commit. The major changes from [10] are highlighted in yellow.

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

req $G \neq \bot$
$(\mathrm{id}_c, C_0, \mathrm{sig}, \mathrm{confTag}) \leftarrow$ *unframe-commit(G, c_0)
if then $\mathrm{id}_c = \mathrm{id}$
  parse($G', \vec{p}', upd, rem, add$) $\leftarrow G.\mathrm{pendCom}[c_0]$
  req $\vec{p} = \vec{p}'$
  return $(\mathrm{id}_c, upd \| rem \| add)$
parse($\mathrm{propIDs}, \mathrm{kp}_c, \mathsf{T}$) $\leftarrow C_0$
parse($\mathrm{id}', \widehat{\mathrm{ct}}_{\mathrm{id}'}$) $\leftarrow \widehat{c}$
req $\mathrm{id} = \mathrm{id}'$
for $i \in 1, \ldots, |\vec{p}|$ **do**
  req $\mathsf{H}(\vec{p}[i]) = \mathrm{propIDs}[i]$
$G' \leftarrow$ *init-epoch(G)
try $(G', upd, rem, add) \leftarrow$ *apply-props($G, G', \vec{p}$)
req $(*, \mathrm{id}_c) \notin rem \wedge (\mathrm{id}_c, *) \notin upd$
if $(*,$ 'rem'-id$) \in rem$ **then**
  $G' \leftarrow \bot$
else
  $G' \leftarrow$ *set-conf-trans-hash($G, G', \mathrm{id}_c, C_0, \mathrm{sig}$)
  $(G', \mathrm{comSecret}) \leftarrow$ *apply-rekey($G', \mathrm{id}_c, \mathrm{kp}_c, \mathsf{T}, \widehat{\mathrm{ct}}_{\mathrm{id}}$)
  $G' \leftarrow$ *set-member-hash(G')
  $(G', \mathrm{confKey}, \mathrm{joinerSecret})$
    $\leftarrow$ *derive-keys($G, G', \mathrm{comSecret}$)
  req *vrf-conf-tag($G', \mathrm{confKey}, \mathrm{confTag}$)
  $G' \leftarrow$ *set-interim-trans-hash($G', \mathrm{confTag}$)
return $(\mathrm{id}_c, upd \| rem \| add)$

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

req $G = \bot$
parse($\mathrm{groupInfo}, \mathsf{T}, \mathrm{sig}$) $\leftarrow w_0$
parse($\mathrm{id}', \mathrm{kphash}, \widehat{\mathrm{ct}}_{\mathrm{id}'}$) $\leftarrow \widehat{w}$
req $\mathrm{id} = \mathrm{id}'$
try $(G, \mathrm{confTag}, \mathrm{id}_c) \leftarrow$ *initialize-group($G, \mathrm{id}, \mathrm{groupInfo}$)
req $G.\mathrm{confTransHash} = \mathsf{H}(G.\mathrm{confTransHash\text{-}w.o\text{-}`id}_c\text{'}, \mathrm{id}_c)$
req $G.\mathrm{interimTransHash} = \mathsf{H}(G.\mathrm{confTransHash}, \mathrm{confTag})$
req SIG.Verify($G.\mathrm{member}[\mathrm{id}_c].\mathrm{svk}, \mathrm{sig}, (\mathrm{groupInfo}, \mathrm{ct}_0)$)
try $G \leftarrow$ *vrf-group-state(G)
svk $\leftarrow G.\mathrm{member}[\mathrm{id}].\mathrm{svk}$
try $G.\mathrm{ssk} \leftarrow$ *fetch-ssk-if-nec(G, svk)
Send get-dks to $\mathcal{F}_{\mathrm{KS}}$ and receive kbs
joinerSecret $\leftarrow \bot$
foreach $(\mathrm{kp}, \mathrm{dk}) \in \mathrm{kbs}$ **do**
  if $\mathsf{H}(\mathrm{kp}) = \mathrm{kphash}$ **then**
    req $G.\mathrm{member}[\mathrm{id}].\mathrm{kp}() = \mathrm{kp}$
    $G.\mathrm{member}[\mathrm{id}].\mathrm{dk} \leftarrow \mathrm{dk}$
    $\mathrm{joinerSecret} \leftarrow \mathrm{CmDec}(\mathrm{dk}, \mathsf{T}, \widehat{\mathrm{ct}}_{\mathrm{id}})$
req $\mathrm{joinerSecret} \neq \bot$
$(G, \mathrm{confKey}) \leftarrow$ *derive-epoch-keys($G, \mathrm{joinerSecret}$)
req *vrf-conf-tag($G, \mathrm{confKey}, \mathrm{confTag}$)
return $(\mathrm{id}_c, G.\mathrm{memberIDsvks}())$

Figure 10: Main protocol: Process and Join. The major changes from [10] are highlighted in yellow.

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

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

*validate-kp(G, kp, id)

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

*assign-kp(G, kp)

parse($\mathrm{id}, \mathrm{ek}, \mathrm{svk}, \mathrm{sig}$) $\leftarrow \mathrm{kp}$
$G.\mathrm{member}[\mathrm{id}].\mathrm{ek} \leftarrow \mathrm{ek}$
$G.\mathrm{member}[\mathrm{id}].\mathrm{svk} \leftarrow \mathrm{svk}$
$G.\mathrm{member}[\mathrm{id}].\mathrm{sig} \leftarrow \mathrm{sig}$
return G

Figure 11: Helper functions: key material related.

*init-epoch(G)

$G' \leftarrow G$.clone()
G'.epoch $\leftarrow G$.epoch + 1
G'.pendUpd[$*$] $\leftarrow \perp$; G'.pendCom[$*$] $\leftarrow \perp$
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 always is non-empty because it is sure to contain 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', comSecret, kp, 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', comSecret)

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

$\vec{\text{ek}} \leftarrow (G'.\text{member}[\text{id}_t].\text{ek})_{\text{id}_t \in \text{addedMem}}$
$(\mathsf{T}, \vec{\text{ct}} = (\widehat{\text{ct}}_{\text{id}_t})_{\text{id}_t \in \text{addedMem}})$
$\leftarrow \text{CmEnc}(\text{pp}_{\text{CmPKE}}, \vec{\text{ek}}, \text{joinerSecret})$
groupInfo $\leftarrow$ (G'.groupid, G'.epoch, G'.memberPublicInfo(), G'.memberHash, G'.confTransHash-w.o-'id_c', G'.confTransHash, G'.interimTransHash, confTag, G'.id)
sig $\leftarrow \text{SIG.Sign}(\text{pp}_{\text{SIG}}, G'.\text{ssk}, (\text{groupInfo}, \mathsf{T}))$
$w_0 \leftarrow$ (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
    $upd \mathrel{+}\!\!\leftarrow (\text{id}_s, \text{'upd'-svk})$
  else if 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] $\leftarrow \perp$
    $rem \mathrel{+}\!\!\leftarrow (\text{id}_s, \text{'rem'-id}_t)$
  else if 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)
    $add \mathrel{+}\!\!\leftarrow (\text{id}_s, \text{'add'-id}_t\text{-svk}_t)$
  else
    return $\perp$
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[$*$] $\leftarrow \perp$; G.pendCom[$*$] $\leftarrow \perp$
G.id $\leftarrow$ id
return (G, confTag, id_c)

Figure 12: Helper functions: Commit, Process and Join. The major changes from [10] are highlighted in yellow.

*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 13: 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 14: Helper function: Member hash and Transcript hash. The major changes from [10] are highlighted in yellow.

*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 15: 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 16: Helper function: Frame and unframe packets.