Security Analysis and Improvements for the IETF MLS Standard for Group Messaging

Joël Alwen[3⋆], Sandro Coretti[2⋆⋆], Yevgeniy Dodis[1⋆⋆⋆], and Yiannis Tselekounis[1†]

[1] New York University
{dodis,tselekounis}@cs.nyu.edu
[2] IOHK
sandro.coretti@iohk.io
[3] Wickr Inc.
jalwen@wickr.com

Abstract. Secure messaging (SM) protocols allow users to communicate securely over untrusted infrastructure. In contrast to most other secure communication protocols (such as TLS, SSH, or Wireguard), SM sessions may be long-lived (e.g., years) and highly asynchronous. In order to deal with likely state compromises of users during the lifetime of a session, SM protocols do not only protect authenticity and privacy, but they also guarantee *forward secrecy (FS)* and *post-compromise security (PCS)*. The former ensures that messages sent and received before a state compromise remain secure, while the latter ensures that users can recover from state compromise as a consequence of normal protocol usage.

SM has received considerable attention in the two-party case, where prior work has studied the well-known double-ratchet paradigm in particular and SM as a cryptographic primitive in general. Unfortunately, this paradigm does not scale well to the problem of secure *group* messaging (SGM). In order to address the lack of satisfactory SGM protocols, the IETF has launched the message-layer security (MLS) working group, which aims to standardize an eponymous SGM protocol. In this work we analyze the *TreeKEM* protocol, which is at the core of the SGM protocol proposed by the MLS working group.

On a positive note, we show that TreeKEM achieves PCS in isolation (and slightly more). However, we observe that the current version of TreeKEM does not provide an adequate form of FS. More precisely, our work proceeds by formally capturing the exact security of TreeKEM as a so-called *continuous group key agreement (CGKA)* protocol, which we believe to be a primitive of independent interest. To address the insecurity of TreeKEM, we propose a simple modification to TreeKEM inspired by recent work of Jost *et al.* (EUROCRYPT '19) and an idea due to Kohbrok (MLS Mailing List). We then show that the modified version of TreeKEM comes with almost no efficiency degradation but achieves *optimal* (according to MLS specification) CGKA security, including FS and PCS. Our work also lays out how a CGKA protocol can be used to design a full SGM protocol.

Finally, we propose and motivate an extensive list of potential future research directions for the area.

1 Introduction

Secure messaging. End-to-end Secure Messaging (SM) allows people to exchange messages without compromising their authenticity and privacy. In contrast to common secure communication protocols such as TLS and SSH, SM protocols are designed for settings in which messages are exchanged *asynchronously* and in which sessions exist for long periods of time. Consequently, participants can be offline at times and their state is more likely to be exposed at some point during the lifetime of a session. SM protocols are therefore expected to satisfy so-called *forward secrecy (FS)* and *post-compromise security (PCS)* (a.k.a. backward secrecy). The former means that even when a participant's key material is compromised, past messages (delivered before the compromise) remain secure. Conversely, PCS means that once the compromise ends, the participants will eventually recover full security as a side effect of continued normal protocol usage.

⋆ Partially supported by the European Research Council under ERC Consolidator Grant (682815 - TOCNeT)
⋆⋆ Work partially done at NYU, where author was supported by NSF grants 1314568 and 1319051.
⋆⋆⋆ Partially supported by gifts from VMware Labs, Facebook and Google, and NSF grants 1314568, 1619158, 1815546.
† Supported by NSF grants 1314568 and 1319051.

The rigorous design and analysis of *two-party* SM protocols has received considerable attention in recent years. This is in no small part due to advent of the *double ratchet paradigm*, introduced by Marlinspike and Perrin [17]. Forming the cryptographic core of a slew of popular messaging applications (e.g., Signal, who first introduced it, as well as WhatsApp, Facebook Messenger, Skype, Google Allo, Wire, and more), double ratchet protocols are now regularly used by over a billion people worldwide.

However, double ratchet protocols are inherently designed for the case where only two users communicate with each other. In order to employ them for groups with more than two users, there is thus little or no alternative to running double ratchets between all pairs of users (at least to distribute and update key material). Unfortunately, that means the double ratchet paradigm does not scale well in settings with a large number of users. In particular, the communication complexity to update key material (an operation crucial to providing PCS) grows *linearly* in the group size. In fact, this poor performance holds for *all*, currently deployed, SM protocols enjoying some form of FS and PCS (i.e., including non-double ratchet based ones.)

This begs the natural question of how to build secure *group* messaging protocols (SGM) that enjoy similar security properties to the two-party ones but whose efficiency scales (say) logarithmically in the group size.

Message layer security and TreeKEM. In order to address the lack of satisfactory SGM protocols, the IETF has launched the message-layer security (MLS) working group, which aims to standardize an eponymous SGM protocol [3,19]. Following in the footsteps of the double ratchet, the MLS protocol promises to be widely deployed and heavily used. Indeed, the working group already includes various messaging companies (Cisco, Facebook, Google, Wickr, Wire, Twitter, etc.) whose combined messaging user base includes everything from government agencies, political organizations, and NGOs to companies both large and small—not to mention a major chunk of the world's consumer population.

The heart of the MLS standard is the so-called TreeKEM protocol. TreeKEM continuously generates fresh, shared, and secret randomness used by the participating parties to evolve the group key material. Each new group key is used to initiate a fresh symmetric hash ratchet that defines a stream of nonce/key pairs used to symmetrically encrypt/decrypt higher-level application messages (such as texts in a chat) using an AEAD. A stream is used until the next evolution of the group key at which point a new stream is initiated.

So not only is TreeKEM the most novel and intricate part of the MLS draft, but understanding it is also central to understanding the security and efficiency properties of full MLS protocol itself. In particular, TreeKEM is crucially involved in achieving PCS and FS.

1.1 Contributions

Continuous group key agreement. This paper makes progress in the formal study of secure group-messaging protocols (SGMs) by studying the security of the latest version of the TreeKEM protocol. First, our work defines the notion of *continuous group key agreement (CGKA)* and casts TreeKEM as a CGKA protocol. CGKA protocols provide methods for adding as well as removing group members and, most crucially, for performing *updates*. Each update operation is initiated by an arbitrary user and results in a new so-called *update secret*. Update secrets are high-entropy random values that the parties use to refresh their group key material in the higher-level protocols (e.g., in the SGM). In an update operation, the initiator also suitably encrypts information about the update secret for other group members.

Our security definition for CGKA protocols requires that (i) users obtain the same update secrets (correctness), (ii) update secrets look random to an attacker observing the protocol messages, (iii) past update secrets remain random even if the state of a party is compromised by the attacker (FS), and (iv) parties can recover from state compromise (PCS). All of these properties are captured by a single, fairly intuitive security game.

We argue that the formal security properties of CGKA are phrased in such a way that it is a suitable building block for full SGM protocols. In particular, CGKA is inspired by the modularization of Alwen *et al.* [1], who constructed a secure two-party messaging protocol (based on the double-ratchet paradigm) by combining three primitives: continuous key agreement (CKA), forward secure authenticated encryption with associated data (FS-AEAD), and a so-called PRF-PRNG. CGKA is therefore to be seen as the multi-user analogue of CKA and is tailored to be used in conjunction with a PRF-PRNG and the multi-user version of FS-AEAD. Specifically, the update secret is run through the PRF-PRNG in order to obtain new keys for the multi-user FS-AEAD. Due to the already quite high complexity of CGKA itself, this work focuses exclusively on CGKA and sketches how it can be used in a higher-level protocol to obtain a full SGM protocol.

TreeKEM has poor forward secrecy. Having defined the notion of CGKA, we analyze the latest version of TreeKEM w.r.t. the new definition. By doing so, we observe that there are serious issues with TreeKEM's forward secrecy, stemming from the fact that its users do not erase old keys sufficiently fast. Specifically, note that in order to efficiently perform updates (with packet sizes logarithmic in the number of users), TreeKEM arranges all group members at the leaves of a binary tree and uses public-key encryption (PKE) to encrypt information about update secrets I to specific subsets of members (determined by their position in the tree). After processing the update, however, parties do not erase or modify the PKE secret keys used to decrypt the update information, since they might need them to process future updates. Hence, corrupting any party other than the update initiator will completely reveal I to an attacker, thereby violating FS. In fact, in order for I to remain secret upon state compromise of an arbitrary user, logarithmically many additional updates are required before the compromise in the best case, and linearly many in the worst case (depending on the order of updates). Even worse, unless the sibling (in the tree) of I's initiator performs an update, I is never forward secret.

Our work formally captures the exact type of FS achieved by TreeKEM by providing an appropriate weakening of the CGKA security definition and proving that TreeKEM satisfies it. On a positive note, even the weakened definition provides PCS, i.e., TreeKEM's update secretes are at least backward secure.

Fixing TreeKEM. In order to remedy TreeKEM's issues with FS, we devise a new type of public-key encryption (PKE) (based on work by Jost *et al.* [15] and suggestions by Konrad Kohbrok on the MLS mailing list [16]) and show that using it in lieu of the (standard) PKE within TreeKEM results in a protocol with *optimal* FS. Specifically, with the new flavor of PKE, *public and secret keys suitably change with every encryption and decryption*, respectively. This kind of key evolution ensures that after decryption, the (evolved) secret key leaks no information about the original message, thereby thwarting the above attack. We also provide a very efficient instantiation of the new PKE notion, thereby ending up with a practical fix and *going from very loose to optimal security at negligible cost*, albeit under the following assumption about the order in which messages are delivered to all participants.

Global ordering of messages. All CGKA security definitions in this work encode the assumption that the delivery server (which caches protocol messages until users come online again) delivers CGKA messages in the same order to all users in a session. Having said that, the delivery server (which we modeled formally as the adversary) may still drop or delay messages at will, as well as decide on the delivery order between users arbitrarily (as long as each user eventually gets the same order of protocol messages). The assumption is taken directly from the MLS design specification where it is stated explicitly. (It is also worth noting that the assumption could also be practically realized in the public bulletin board model, e.g., using a block-chain protocol.)

Of course, an alternative approach would have been to remove the assumption from our security definitions. However, this would require one of two things: either *significantly* weakening the adversary to allow proving security for any TreeKEM-like protocol (as we demonstrate in the section on future directions), or making bigger changes to the TreeKEM construction. In particular, we suspect that any such changes would result in a protocol that is quite impractical for real world use. Indeed, this exact dynamic has already played out in the two-party case where the more secure protocols end up making use of, e.g., HIBEs with arbitrary depth. Thus, motivated by analyzing the IETF proposal and contributing to that design process, we opted to stick with the MLS delivery order assumption while considering only truly practical constructions—at least for this initial work.

Adaptive security. The security of both TreeKEM and the improved version mentioned above is proved w.r.t. a *non-adaptive* attacker, i.e., an attacker that is required to announce all corruptions at the beginning (as opposed to being able to corrupt on-the-fly depending on values and messages produced by the protocol). The reason for this restriction are commitment issues (with PKE ciphertexts) that are quite common in the cryptographic literature and quite difficult to deal with—unless one accepts an exponential loss in security. Given the importance of considering adaptive attackers in practice, we provide a first step towards proving TreeKEM adaptively secure by sketching how to apply a very clever reduction technique of Jafargholi *et al.* [14] to the TreeKEM setting. While the resulting bounds are substantially better than the trivial exponential security loss mentioned above, they are still quite far from usable in practice. We leave a formal adaptive proof and finding techniques leading to reasonable security bounds as an interesting and important open problem.

Future directions. SGM protocols have enormous real world applicability. Collectively, they already have well over a billion users today. Yet at the same time, the subject remains a surprisingly nascent area within cryptography with *many* very well motivated open problems still to be addressed. So, to help the area forward, in Section 8 and Appendix 8.2 we lay out some ideas for what we consider to be important open problems for future research in the area. In particular, we believe that the IETF's effort to standardize an SGM protocol could stand to benefit directly from answers to many of the problems we describe here. We also discuss some more theoretical open problems which we see as central to this area but which are not answered yet.

1.2 Related Work

The double ratchet paradigm was introduced by Marlinspike and Perrin [17], drawing on ideas from the OTR (off-the-record) protocol [5]. An early analysis of the double-ratchet algorithm was performed by Cohn-Gordon *et al.* [8]. An important line of work [4,21,13,15,11] studied the problem of two-party secure messaging as a cryptographic primitive. In particular, the work by Jost *et al.* [15] introduced the notion of updatable PKE most closely related to the one used in this paper. However, for our purposes a simpler definition is sufficient, although we use the same efficient construction as [15] to realize our notion. A paper by Alwen *et al.* [1] provided a modular design for double-ratchet algorithms and formal definition of secure messaging in the two-party setting.

In the group setting, Cremers *et al.* [9] note TreeKEM's disadvantages w.r.t. PCS in the setting with multiple groups, and Weider [24] suggests Causal TreeKEM, a variant that requires less ordering of protocol messages than plain TreeKEM. The messages on the MLS mailing list that suggested TreeKEM were [22,2]. The main precursor of TreeKEM, the so-called asynchronous ratchet tree (ART) protocol, was introduced by Cohn-Gordon *et al.* [7], and was very influential for the creating of the MLS group and the development of TreeKEM.

The TreeKEM protocol is related to schemes for (symmetric-key) broadcast encryption (BE) introduced by Fiat and Naor [12]. Unlike SGM, in BE schemes the trusted group manager distributed all the secret keys, as well as message content. Moreover, users do not need to update their secret keys (meaning the schemes are *stateless*), as the trusted manager can add or revoke users at will. Public-key BE schemes introduced by Dodis and Fazio [10] allow group members (and, in fact, anybody) to deliver content, but the group management is still done by the trusted authority. Multicast encryption (ME)[18,25,6] schemes improve the efficiency of BE schemes, by explicitly maintaining a single, "current" group of users who share a common key, at the cost of group manager needing to send explicit "membership update" messages to add or remove users; meaning that the schemes are no longer stateless. In this sense ME schemes are closer to SGM schemes studied here, but the group management is still done by the trusted authority, and no PCS is considered (in particular, there are no "key update" messages).

2 Preliminaries

This section introduces some general notation and basic concepts around binary trees. Definitions of PRGs and CPA-secure public-key encryption can be found in Section A.

Notation. For a positive integer a, $[a]$ denotes the set $\{1, 2, \ldots, a\}$. For an integer n, $\mathsf{mp2}(n)$ is the maximum power of 2 dividing n. Security games in this work involve *dictionaries*. The value stored with key x in a dictionary D is denoted by $D[x]$. The statement $D[\cdot] \leftarrow y$ (for any type of y) initializes a dictionary D in which the default initial value for each key is y.

Binary trees. This work considers rooted binary trees, in which all nodes have between 0 and 2 unique children. The *height* of τ is the length of the longest path from the root to any leaf.[4] A node with no children is called a *leaf*; all other nodes are called *internal*. A tree τ is *full* if it has height h and 2^h leaves. For an integer $h \geq 0$, denote by FT_h the full binary tree of hight h. For two leaf nodes ℓ and ℓ' in some tree, let $\mathsf{LCA}(\ell, \ell')$ be the least common ancestor of ℓ and ℓ', i.e., the node where the paths from these leaves to the root meet.

[4] In particular, the tree of height 0 consists of a single node, the root.

3 Continuous Group Key Agreement

The purpose of *continuous group key-agreement (CGKA)* is to continuously provide members of a messaging group with fresh secret random values, which they use to refresh their key material (in a higher-level protocol). This section formally defines the syntax of CGKA schemes and presents a security notion that simultaneously captures correctness, key indistinguishability, forward secrecy, as well as post-compromise security.

3.1 CGKA Syntax

A CGKA scheme provides algorithms to create a group, add as well as remove users, perform updates, and process protocol messages.

Definition 1. *A* continuous group key-agreement (CGKA) *scheme* $\mathsf{CGKA} = (\mathsf{init}, \mathsf{create}, \mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{proc})$ *consists of the following algorithms:*

- Initialization: init *takes an ID* ID *and outputs an initial state* γ*.*
- Group creation: create *takes a state* γ *and a list of IDs* $\mathsf{G} = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$*, and outputs a new state* γ' *and a control message* W*.*
- Add: add *takes a state* γ *and an ID* ID'*, and outputs a new state* γ *as well as control messages* W *and* T*.*
- Remove: rem *takes a state* γ *and an ID* ID' *and outputs a new state* γ' *and a control message* T*.*
- Update: upd *takes a state* γ *and outputs a new state* γ' *and a control message* T*.*
- Process: proc *takes a state* γ *and a control message* T *and outputs a new state* γ' *and an update secret* I*.*

The basic usage of a CGKA scheme is as follows: Generally, once a group is established using create, any group member, referred to as the *sender*, may call any of the algorithms to add or remove members or to perform updates. Each time, such a call results in a new so-called *epoch*. It is implicitly the task of a server connecting the parties to then relay the resulting *control* messages to all current group members (including the sender). Observe that there are two types of control messages: *welcome* messages W, which are sent to parties *joining* a group, and normal control messages T, which are intended for parties already in the group. Whenever the server delivers a control message to a group member, they process it using proc. Algorithm proc also outputs an *update secret* I, where the intention is that $I \neq \bot$ if and only if the control message corresponds to an update.

3.2 CGKA Security

Informally, the basic properties that any CGKA scheme must satisfy are the following:

- *Correctness:* All group members output the same update secret I in update epochs.
- *Privacy:* The update secrets look random given the transcript of control messages.
- *Forward secrecy (FS):* If the state of any group member is leaked at some point, all previous update secrets remain hidden from the attacker.
- *Post-compromise security (PCS):* After every group member whose state was leaked performs an update (that is processed by the group) update secrets become secret again.

These properties are captured by the security game presented in this section (cf. Figure 1). In the game the attacker is given access to various oracles to drive the execution of a CGKA protocol. It is important to note that the capabilities of the attacker and restrictions on the order in which the attacker may call the oracles is motivated by how a CGKA protocol would be used in a higher-level protocol. Most importantly, the attacker will not be allowed to modify or inject any control messages. The corresponding design choices are justified in Section 3.3.

Oracles of Security Game for CGKA

```
init
    b ←$ {0,1}
    ∀ID : γ[ID] ← init(ID)
    lead[·], I[·], G[·] ← ϵ
    ep[·], ctr[·] ← 0
    D[·] ← true
    chall[·] ← false
    pub M[·] ← ϵ

create-group (ID_0, ID_1, ..., ID_n)
    t ← ep[ID]
    req t = 0
    c ← ++ ctr[ID_0]
    (γ[ID_0], W) ← create(γ[ID_0], ID_1, ..., ID_n)
    for i = 0, ..., n
        M[t+1, ID_0, ID_i, c] ← W
    G[t+1, ID, c] ← {ID_0, ID_1, ..., ID_n}

reveal (t)
    req I[t] ∉ {ϵ, ⊥} ∧ ¬chall[t]
    chall[t] ← true
    return I[t]

chall (t)
    req I[t] ∉ {ϵ, ⊥} ∧ ¬chall[t]
    I_0 ← I[t]
    I_1 ← K
    chall[t] ← true
    return I_b

add-user (ID, ID')
    t ← ep[ID]
    req t > 0 ∧ ID' ∉ G[t]
    c ← ++ ctr[ID]
    (γ[ID], W, T) ← add(γ[ID], ID')
    M[t+1, ID, ID', c] ← (W, T)
    for ĨD ∈ G[t]
        M[t+1, ID, ĨD, c] ← T
    G[t+1, ID, c] ← G[t] ∪ {ID'}

remove-user (ID, ID')
    t ← ep[ID]
    req t > 0 ∧ ID' ∉ G[t] > 0
    c ← ++ ctr[ID]
    (γ[ID], T) ← rem(γ[ID], ID')
    for ĨD ∈ G[t]
        M[t+1, ID, ĨD, c] ← T
    G[t+1, ID, c] ← G[t] \ {ID'}

send-update (ID)
    t ← ep[ID]
    req t > 0
    c ← ++ ctr[ID]
    (γ[ID], T) ← upd(γ[ID])
    for ĨD ∈ G[t]
        M[t+1, ID, ĨD, c] ← T
    G[t+1, ID, c] ← G[t]

deliver (t, ID, ID', c)
    req lead[t] ∈ {ϵ, (ID, c)} ∧
          (t = ep[ID'] + 1 ∨ added(t, ID, ID', c))
    T ← M[t, ID, ID', c]
    (γ[ID'], I) ← proc(γ[ID'], T)
    if lead[t] = ϵ
        lead[t] ← (ID, c)
        I[t] ← I
        G[t] ← G[t, ID, c]
    else if I ≠ I[t]
        win
    if removed(t, ID')
        ep[ID'] ← −1
    else
        ep[ID'] ++
    ctr[ID'] ← 0

corr (ID)
    return γ[ID]

no-del (ID)
    D[ID] ← false
```

Fig. 1. Oracles for the CGKA security game for a scheme $\mathsf{CGKA} = (\mathsf{init}, \mathsf{create}, \mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{proc})$. The functions added and $\mathsf{removed}$ are defined in the text.

Epochs. The main oracles to drive the execution are the oracles to create groups, add users, remove users, and to deliver control messages, i.e., **create-group**, **add-user**, **remove-user**, **send-update**, and **deliver**. The first four oracles allow the adversary to instruct parties to initiate new epochs, whereas the deliver oracle makes parties actually proceed to the next epoch. The server connecting the parties is trusted to provide parties with a consistent view of which operation takes place in each epoch. That is, while multiple parties may initiate a new epoch, the attacker is forced to pick a single operation that defines the new epoch; the corresponding sender is referred to as the *leader* of the epoch. Observe that the parties may advance at various speeds and therefore be in epochs arbitrarily far apart.

The game forces the attacker to initially, i.e., in epoch 1, create a group. Thereafter, any group member may add new parties, remove current group members, or perform an update. The attacker may also corrupt any party at any point (thereby learning that party's secret state) and challenge the update secret in any epoch where the leader performed an update operation. Furthermore, the adversary can instruct parties to stop deleting old secrets. There will be restrictions checked at the end of the execution of the game to ensure that the attacker's challenge/corruption/no-deletion behavior does not lead to trivial attacks.

Initialization. The **init** oracle sets up the game and all the variables needed to keep track of the execution. The random bit b is used for real-or-random challenges, and the dictionary γ keeps track of all the users' states. For every epoch, the dictionaries lead, $\mathbf{I}$, and $\mathbf{G}$ record the leader, the update secret, and the group members, respectively, and ep records which epoch each user is currently in. The array ctr counts all new operations initiated by a user in their current epoch. Moreover, D keeps track of which parties delete their old values and which do not. Dictionary chall is used to ensure that the adversary can issue at most a single challenge per (update) epoch. Finally, M records all control messages produced by parties; the adversary has read access to M (as indicated by the keyword **pub**).

Initiating operations and choosing epoch leaders. As mentioned above, the attacker must choose a leader in every epoch, i.e., a sender whose control message is ultimately processed by all group members. More precisely, for each user ID currently in some epoch t, $\mathsf{ctr}[\mathsf{ID}]$ can be thought of as a (local) "version number" that counts the various operations initiated by ID in epoch t. The counter is incremented each time ID initiates a new operation. The resulting control messages for users ID_i are stored in M with key $(t+1, \mathsf{ID}, \mathsf{ID}_i, \mathsf{ctr}[\mathsf{ID}])$,

representing the number of the next epoch, the sender, the recipient, and the (local) version number of the operation. Similarly, dictionary $\mathbf{G}$ stores the new group that would result from the operation with key $(t+1, \mathsf{ID}, \mathsf{ctr}[\mathsf{ID}])$.

For every epoch t, the first control message $M[t, \mathsf{ID}, \mathsf{ID}', c]$ delivered via **deliver**, for some users ID and ID' and version number c, determines that ID is the leader and c the version that was chosen by the server. Correspondingly, the game records $\mathsf{lead}[t] \leftarrow (\mathsf{ID}, c)$ and sets the group membership to $\mathbf{G}[t] \leftarrow \mathbf{G}[t, \mathsf{ID}, c]$.

In general, whenever a party ID' processes any control message, the counter $\mathsf{ctr}[\mathsf{ID}']$ is reset to 0 as all operations initiated by ID' in its current epoch are now obsolete (either processed by ID or rejected by the server in favor of some other operation). Note that the sender of an operation also sends a control message addressed to themselves to the server. The server confirms an operation by returning that message back to the sender.

Group creation. The oracle **create-group** causes ID_0 to create a group with members $\{\mathsf{ID}_0, \ldots, \mathsf{ID}_n\}$. This is only allowed if ID_0 is currently in epoch 0, which is enforced by the **req** statement. Thereafter, ID_0 calls the group creation algorithm and sends the resulting welcome messages to all users involved (including itself).

Adding and removing users and performing updates. For all three oracles **add-user**, **remove-user** and **send-update**, the **req** statement checks that the call makes sense (e.g., checking that a party added to the group is not currently a group member). Subsequently, the oracles call the corresponding CGKA algorithms (add, rem, and upd, respectively) and store the resulting control messages in M.

Delivering control messages. The oracle **deliver** is called with the same four arguments $(t, \mathsf{ID}, \mathsf{ID}', c)$ that are used as keys for the M array. The **req** statement at the beginning checks that (1) either there is no leader for epoch t yet or version c of ID is the leader already and (2) the recipient ID' is currently either in epoch $t-1$ or a newly added group member, which is checked by predicate added defined by

$$\mathsf{added}(t, \mathsf{ID}, \mathsf{ID}', c) \;:=\; \mathsf{ID}' \notin \mathbf{G}[t-1] \wedge \mathsf{ID}' \in \mathbf{G}[t, \mathsf{ID}, c] \ .$$

If the checks are passed, the appropriate control message is retrieved from M and run through proc on the state of ID'. If there is no leader for epoch t yet, the game sets the leader as explained above and also records the update secret $\mathbf{I}[t]$ output by proc. In all future calls to **deliver**, the values I output by process will be checked against $\mathbf{I}[t]$, and, in case of a mismatch, the instruction **win** reveals the secret bit b to the attacker; this ensures correctness. Finally, the epoch counter for ID' is incremented—or set to -1 if the operation just processed removes ID' from the group. This involves a check via predicate $\mathsf{removed}$ defined by

$$\mathsf{removed}(t, \mathsf{ID}') \;:=\; \mathsf{ID}' \in \mathbf{G}[t-1] \wedge \mathsf{ID}' \notin \mathbf{G}[t] \ .$$

Challenges, corruptions, and deletions. In order to capture that update secrets must look random, the attacker is allowed to issue a challenge for any epoch corresponding to an update operation. When calling **chall**(t) for some t, the oracle first checks that t indeed corresponds to an update epoch and that a leader already exists. Similarly, using **reveal**, the attacker can simply learn the update secret of an epoch. It is also ensured that for each epoch, the attacker can make at most one call to either **chall** or **reveal**.

To formally model forward secrecy and PCS, the attacker is also allowed to learn the current state of any party by calling the oracle **corrupt**. Finally, the attacker can instruct a party ID to stop deleting old values by calling **no-del**(ID). Subsequently, the game will *implicitly* store all old states of ID (instead of overriding them) and leak it to the attacker when he calls **corrupt**.[5] The game also sets the corresponding flag.

Avoiding trivial attacks. In order to ensure that the attacker may not win the CGKA security game with trivial attacks (such as, e.g., challenging an epoch t's update secret and leaking some party's state in epoch t), at the end of the game, the predicate **safe** is run on the queries $\mathbf{q}_1, \ldots, \mathbf{q}_q$ in order to determine whether the execution was devoid of such attacks. The predicate tests whether the attacker can trivially compute the update secret in a challenge epoch t^* using the state of a party ID in some epoch t and the control messages observed on the network. This is the case if either (1) ID has not performed an update or been

[5] Modeling no-deletions explicitly would clutter Figure 1 quite a bit.

Safety Predicate

safe $(\mathbf{q}_1, \ldots, \mathbf{q}_q)$
  for (i, j) *s.t.* $\mathbf{q}_i = \mathbf{corrupt}(\mathsf{ID})$ *for some* ID *and* $\mathbf{q}_j = \mathbf{chall}(t^*)$ *for some* t^*
    if $\mathsf{q2e}(\mathbf{q}_i) \leq t^*$ *and* $\nexists k$ *s.t.* $0 < \mathsf{q2e}(\mathbf{q}_i) < \mathsf{q2e}(\mathbf{q}_k) \leq t^*$ *and* $\mathbf{q}_k \in \{\mathbf{send\text{-}update}(\mathsf{ID}), \mathbf{remove\text{-}user}(*, \mathsf{ID})\}$
      return 0
    if $\mathsf{q2e}(\mathbf{q}_i) > t^*$ *and* $\exists k$ *s.t.* $\mathsf{q2e}(\mathbf{q}_k) \leq t^*$ *and* $\mathbf{q}_k = \mathbf{no\text{-}del}(\mathsf{ID})$
      return 0
  return 1

Fig. 2. The safety predicate determines whether a sequence of oracle calls $(\mathbf{q}_1, \ldots, \mathbf{q}_q)$ allows the attacker to trivially win the CGKA security game.

removed before epoch t^* or (2) ID stopped deleting values at some point up to epoch t^* and was corrupted thereafter. The predicate is depicted in Figure 3.2. The figure uses the function $\mathsf{q2e}(\mathbf{q})$, which returns the epoch corresponding to query $\mathbf{q}$. Specifically, for $\mathbf{q} \in \{\mathbf{corrupt}(\mathsf{ID}), \mathbf{no\text{-}del}(\mathsf{ID})\}$, if ID is member of the group when $\mathbf{q}$ is made, $\mathsf{q2e}(\mathbf{q})$ is the value of $\mathsf{ep}[\mathsf{ID}]$ (when the query is made), otherwise, $\mathsf{q2e}(\mathbf{q})$ returns $\perp$. For $\mathbf{q} \in \{\mathbf{send\text{-}update}(\mathsf{ID}), \mathbf{remove\text{-}user}(\mathsf{ID}, \mathsf{ID}')\}$, $\mathsf{q2e}(\mathbf{q})$, is the epoch for which ID initiates the operation. If $\mathbf{q}$ is not processed by any user we set $\mathsf{q2e}(\mathbf{q}) = \perp$.[6]

Observe that the predicate **safe** can in general be replaced by any other predicate P, potentially weakening the resulting security notion.

Advantage. In the following, a (t, c, n)-*attacker* is an attacker $\mathcal{A}$ that runs in time at most t, makes at most c challenge queries, and never produces a group with more than n members. The attacker wins the CGKA security game if he correctly guesses the random bit b in the end *and* the safety predicate P evaluates to true on the queries made by the attacker. The advantage of $\mathcal{A}$ with safety predicate P against a CGKA scheme CGKA is defined by

$$\mathsf{Adv}^{\mathsf{CGKA},\mathsf{P}}_{\text{cgka-na}}(\mathcal{A}) \;:=\; \left|\mathsf{Pr}[\mathcal{A} \text{ wins}] - \frac{1}{2}\right| .$$

Definition 2 (Non-adaptive CGKA security). *A continuous group key-agreement protocol* CGKA *is* non-adaptively $(t, c, n, \mathsf{P}, \varepsilon)$-secure *if for all* (t, c, n)*-attackers,*

$$\mathsf{Adv}^{\mathsf{CGKA},\mathsf{P}}_{\text{cgka-na}}(\mathcal{A}) \;\leq\; \varepsilon \; .$$

3.3 CGKA in Higher-Level Protocols

Syntax and security of CGKA protocols are defined in such a way that they can be used by a higher-level protocol (e.g., the full MLS protocol) in a modular fashion. This design is inspired by the modularization provided by Alwen *et al.* [1] for the two-party Signal protocol. In that work, the two-party analogue of CGKA, *continuous key agreement (CKA)*, is combined with so-called forward-secure AEAD and a hash function with particular properties in order to obtain a protocol for *full* two-party messaging. One of the crucial features of this approach lies in the fact that the CKA component can be analyzed in a setting with authenticated channels. This is due to the fact that the higher-level protocol suitably authenticates all messages sent by the CKA scheme.

The approach just outlined can also be taken in the group setting. In fact, the design of the full MLS protocol follows the above paradigm [3]. In particular, TreeKEM's protocol messages are authenticated by the group members, which justifies the fact that in the CGKA game, the attacker cannot change control messages sent by parties.

4 TreeKEM

4.1 Overview

The TreeKEM CGKA protocol is based on so-called (binary) *ratchet trees (RTs)*. In a TreeKEM RT, group members are arranged at the leaves, and all nodes have an associated public-key encryption (PKE) key pair,

[6] Any boolean expression containing $\perp$ is false.

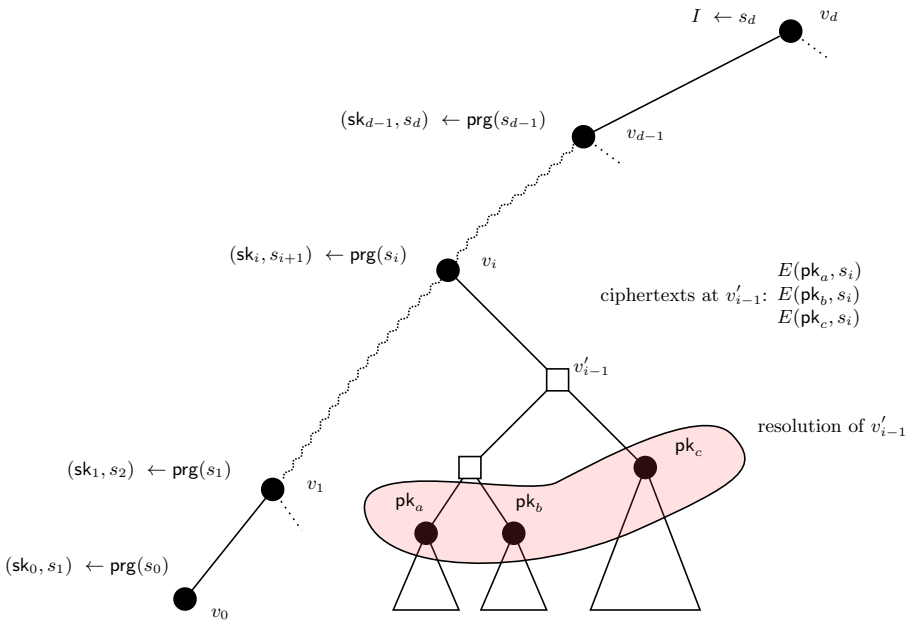

Fig. 3. An update operation initiated by the party at leaf v_0: First, a random "seed value" s_0 is chosen. Thereafter, a PRG is applied iteratively at every level i of v_0's direct path in order to derive (i) a PKE secret key sk_i for that level (from which a public key can be computed using the key generation algorithm) and (ii) a seed s_{i+1} for the next level. Every seed s_i is encrypted using the public key of the corresponding co-path node v'_{i-1}. Sometimes, such a node can be blank, in which case s_i must be encrypted using the public keys of each node in the *resolution*, which is the smallest set of nodes covering all leaves in the subtree of v'_{i-1}. This ensures that all these nodes are able to compute the keys from v_i upward. The update secret I produced by such an update is the seed value s_d at the root.

except for the root. The tree invariant is that each user knows all secret keys on their *direct path*, i.e., on the path from their leaf node to the root. In order to perform an update—the most crucial operation of a CGKA—and produce a new update secret I, a party first generates fresh key pairs on every node of their direct path. Then, for every node v' on its *co-path*—the sequence of siblings of nodes on the direct path—it encrypts specific information under the public key of v' that allows each party in the subtree of v' to learn all new secret keys from v's parent up to the root (cf. Figure 3 and Section 4.4).

Before presenting the formal description of TreeKEM in Section 4.4, basic concepts around ratchet trees are explored in Section 4.3. Moreover, Section 4.2 quickly discusses the simple PKI model used in this work.

4.2 PKI

The TreeKEM protocol requires a public-key infrastructure (PKI) where parties can register ephemeral keys. The MLS documents [19,3] lay out explicitly how users are to generate, authenticate, distribute, and verify each others initialization keys. For simplicity and in order not to detract from the essential components of TreeKEM, this work models the PKI by providing protocol algorithms and attackers with access to the following PKI functionality:

- Any user ID may request a *fresh* (encryption) public key pertaining to some other user ID'. That is, when ID calls $\mathsf{get\text{-}pk}(\mathsf{ID}')$, the PKI functionality generates a fresh key pair $(\mathsf{pk}, \mathsf{sk})$ and returns pk to ID. The PKI also records the triple $(\mathsf{pk}, \mathsf{sk}, \mathsf{ID}')$ and passes the information $(\mathsf{pk}, \mathsf{ID}')$ to the attacker.
- Any user ID' may request secret keys corresponding to public keys associated with them. Specifically, when ID' calls $\mathsf{get\text{-}sk}(\mathsf{pk})$, if a triple $(\mathsf{pk}, \mathsf{sk}, \mathsf{ID}')$ is recorded, the PKI functionality returns sk to ID'.

Note in particular that the PKI ensures that every public key is only used once. Of course, in practice such a PKI functionality would actually be implemented by having users generate key pairs themselves and registering them with the PKI. However, the above formalization simplifies the description of the protocols.

4.3 Ratchet Trees

Basics. The following are some basic concepts around TreeKEMs ratchet trees.

LBBTs. An RT in TreeKEM is a so-called *left-balanced binary tree (LBBT)*. In a nutshell, an LBBT on n nodes (is defined recursively and) has a maximal full binary tree as its left child and an LBBT on the remaining nodes as its right child:

Definition 3 (Left-Balanced Binary Tree). *For $n \in \mathbb{N}$ the* left-balanced binary tree (LBBT) on n nodes LBBT_n *is the binary tree constructed as follows:*

- *The tree LBBT_1 is a single node.*
- *Let $x = \mathsf{mp2}(n)$.*[7] *Then, the root of LBBT_n has the full subtree FT_x as the left subtree and LBBT_{n-x} as the right subtree.*

Observe that LBBT_n has exactly n leaves and that every internal node has both children. In an RT, nodes are *labeled* as follows:

- *Root:* The root is labeled by an *update secret* I.
- *Internal nodes:* Internal nodes are labeled by a *key pair* $(\mathsf{pk}, \mathsf{sk})$ for the PKE scheme Π.
- *Leaf nodes:* Leaf nodes are labeled like internal nodes, except that they additionally have an *owner* ID.

Labels are referred to using dot-notation (e.g., $v.\mathsf{pk}$ is v's public key). As a shorthand, $\tau.\mathsf{ID}$ is the leaf node with label ID. Any subset of a node's labels may be undefined, which is indicated by the special symbol $\perp$. Furthermore, a node v may be *blank*. A blank node has all of its labels set to $\perp$. As explained below, all internal nodes in a freshly initialized RT are blank, and, moreover, blanks can result from adding and removing users to and from a group, respectively.

Paths and blanking. As hinted at the beginning of this section, it will be useful to consider the following types of paths:

- the *direct path* $\mathsf{dPath}(\tau, \mathsf{ID})$, which is the path from the leaf node labeled by ID to the root;
- the *co-path* $\mathsf{coPath}(\tau, \mathsf{ID})$, which is the sequence of siblings of nodes on the direct path $\mathsf{dPath}(\tau, \mathsf{ID})$;

Furthermore, given an ID ID and an RT τ, the function $\tau' \leftarrow \textsc{Blank}(\tau, \mathsf{ID})$ blanks all nodes on $\mathsf{dPath}(\tau, \mathsf{ID})$.

Resolutions and representatives. A crucial notion in TreeKEM is that of a resolution. Intuitively, the resolution of a node v is the smallest set of non-blank nodes that covers all leaves in v's subtree.

Definition 4 (Resolution). *Let τ be a tree with node set V. The* resolution $\mathsf{Res}(v) \subseteq V$ *of a node $v \in V$ is defined recursively as follows:*

- *If v is* not blank*, then $\mathsf{Res}(v) = \{v\}$.*
- *If v is a* blank leaf*, then $\mathsf{Res}(v) = \emptyset$.*
- *Otherwise, $\mathsf{Res}(v) := \cup_{v' \in C(v)} \mathsf{Res}(v')$, where $C(v)$ are the children of v.*

Each leaf ℓ' in the subtree τ' of some node v' has a representative in τ':

Definition 5 (Representative). *Consider a tree τ and two leaf nodes ℓ and ℓ'.*

1. *Assume ℓ' is non-blank and in the subtree rooted at v'. The* representative $\mathsf{Rep}(v', \ell')$ of ℓ' in the subtree of v' *is the first filled node on the path from v' (down) to ℓ.*
2. *Consider the least common ancestor $w = \mathsf{LCA}(\ell, \ell')$ of ℓ and ℓ'. Let v be the child of w on the direct path of ℓ, and v' that on the direct path of ℓ'. The* representative $\mathsf{Rep}(\ell, \ell')$ of ℓ' w.r.t. ℓ *is defined to be the representative $\mathsf{Rep}(v', \ell')$ of ℓ' in the subtree of v'.*

It is easily seen that $\mathsf{Rep}(v', \ell') \in \mathsf{Res}(v')$.

[7] Recall that $\mathsf{mp2}(n)$ is the maximum power of two dividing n.

Simple RT operations. The following paragraphs describe how RTs are initialized as well as how they grow and shrink. The proofs of Lemmas 1 and 2 below can be found in Appendix C.

RT Initialization. Given lists of users $G = (\mathsf{ID}_0, \mathsf{ID}_1, \ldots, \mathsf{ID}_n)$ and public keys $\mathbf{pk} = (\mathsf{pk}_0, \mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ as well as an integer j and a secret key sk_j, a new RT is initialized as the left-balanced binary tree LBBT_{n+1} where

- all the internal nodes as well as the root are blanked,
- the label of every leaf i is set to $(\mathsf{ID}_i, \mathsf{pk}_i, \bot)$, and
- the secret key at leaf j is additionally set to sk_j.

In the following, the above operation is denoted by $\textsc{Init}(G, \mathbf{pk}, j, \mathsf{sk}_j)$.

Adding IDs to the RT. Given an RT τ, the procedure $\tau' \leftarrow \textsc{AddID}(\tau, \mathsf{ID}, \mathsf{pk})$, sets the labels of the first blank leaf of τ to $(\mathsf{ID}, \mathsf{pk}, \bot)$, and outputs the resulting tree, τ'. If there is no blank leaf in the tree $\tau = \mathsf{LBBT}_n$, method $\textsc{AddLeaf}(\tau)$ is called, which adds a leaf z to it, resulting in a new tree $\tau' = \textsc{AddLeaf}(\tau)$:

1. If n is a power of 2, create a new node r' for τ'. Attach the root of τ as its left child and z as its right child.
2. Otherwise, let r be the root of τ, and let τ_L and τ_R be r's left and right subtrees, respectively. Recursively insert z into τ_R to obtain a new tree τ'_R, and let τ' be the tree with r as a root, τ_L as its left subtree and τ'_R as its right subtree.

Lemma 1. *If $\tau = \mathsf{LBBT}_n$, then $\tau' = \mathsf{LBBT}_{n+1}$.*

Removing an ID. The procedure $\tau' \leftarrow \textsc{RemID}(\tau, \mathsf{ID})$ blanks the leaf labeled with ID and truncates the tree such that the rightmost non-blank leaf is the last node of the tree. Specifically, the following recursive procedure $\textsc{Trunc}(v)$ is called on the rightmost leaf v of τ, resulting in a new tree $\tau' \leftarrow \textsc{Trunc}(\tau)$:[8]

1. If v is blank and not the root, remove v as well as its parent and place its sibling v' where the parent was. Then, execute $\textsc{Trunc}(v')$.
2. If v is non-blank and the root, execute $\textsc{Trunc}(v'')$ on the rightmost leaf node in the tree.
3. Otherwise, do nothing.

Lemma 2. *If $\tau = \mathsf{LBBT}_n$, then $\tau' = \mathsf{LBBT}_y$ for some $0 < y \leq n$. Furthermore, unless $y = 1$, the rightmost leaf of τ' is non-blank.*

Public copy of an RT. Given an RT τ, $\tau' \leftarrow \textsc{Pub}(\tau)$ creates a public copy, τ', of the RT by setting all secret-key labels to $\bot$.

4.4 TreeKEM Protocol

This section now explains the TreeKEM protocol in detail by describing all the algorithms involved in the scheme, which is depicted in Figure 4. For simplicity, the state γ is not made explicit; it consists of the variables initialized by init. TreeKEM makes (black-box) use of the following cryptographic primitives:

- a pseudo random generator prg, and
- a CPA-secure public-key encryption scheme $\Pi = (\mathsf{PKEG}, \mathsf{Enc}, \mathsf{Dec})$.

TreeKEM as described here is slightly different from TreeKEM as described in the current MLS draft [3]. These differences are elaborated on in Appendix D. Essentially, they are small efficiency improvements that do not affect security.

[8] Overloading function $\textsc{Trunc}$ for convenience here.

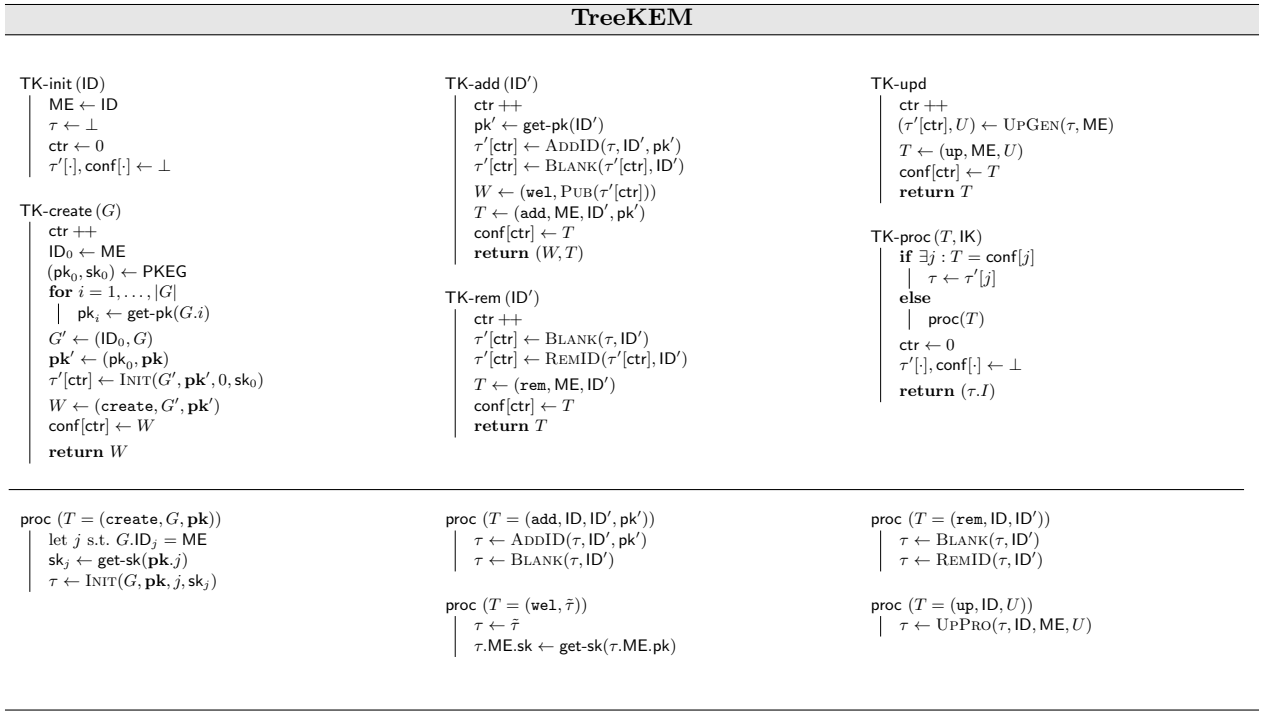

Fig. 4. The TreeKEM protocol operations. The functions AddID, RemID, and Blank are defined in Section 4.3, while UpGen and UpPro are defined in Section 4.4.

Initialization. The initialization procedure TK-init expects as input an ID ID and initializes several state variables: Variable ME remembers the ID of the party running the scheme and τ will keep track of the RT used. The other variables are used to keep track of all the operations (creates, adds, removes, and updates) initiated by ME but not confirmed yet by the server. Specifically, each time a party performs a new operation, it increases ctr and stores the potential next state in $\tau'[\mathsf{ctr}]$. Moreover, conf[ctr] will store the control message the party expects from the server as confirmation that the operation was accepted. These variables are reset each time proc processes a control message (which can either be one of the messages in conf or a message sent by another party).

Group creation. Given lists of users $G = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$, TK-create initializes a new ratchet tree by first creating a new PKE key pair $(\mathsf{pk}_0, \mathsf{sk}_0)$, fetching public keys $\mathbf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ corresponding to the IDs in G from the PKI, and then calling Init with[9] $G' = (\mathsf{ID}_0, G)$ and $\mathbf{pk}' = (\mathsf{pk}_0, \mathbf{pk})$ as well as 0 and sk_0. The welcome message simply consists of G' and $\mathbf{pk}'$.

Adding a group member. To add new group member ID′, add first obtains a corresponding public key pk′ from the PKI and then updates the RT by calling AddID (described above) followed by Blank, which removes all keys from the new party's leaf up to the root. This ensures that the new user does not know any secret keys used by the other group members before he joined. The welcome message for the new user simply consists of a public copy of the current RT (specifically, Pub sets the sender's secret-key label to $\perp$), and the control message for the remaining group members of the IDs of the sender and the new user as well as the latter's public key.

Removing a group member. A group member ID′ is removed by first blanking all the keys from the leaf node of ID′ to the root. This prevents parties from using keys known to ID′ in the future. User ID′ is

[9] Here we slightly abuse vector notation.

subsequently removed from the tree by calling RemID. The control message for the remaining group members consists of the IDs of the sender and the removed user.

Performing an update. A user ME performs an update by choosing new key pairs on their direct path as follows:

- *Compute path secrets*: Let $v_0 = v, v_1, \ldots, v_d$, be the nodes on the direct path of ME's leaf node v. First, ME chooses a uniformly random s_0. Then, it computes
$$\mathsf{sk}_i \| s_{i+1} \leftarrow \mathsf{prg}(s_i), \text{ for } i = 0, \ldots, d-1.$$
- *Update RT labels*: For $i = 0, \ldots, d-1$, ME computes $\mathsf{pk}_i \leftarrow \mathsf{PKEG}(\mathsf{sk}_i)$ and updates the PKE label of v_i to $(\mathsf{pk}_i, \mathsf{sk}_i)$.
- *Root node*: For the root node, ME sets $I := s_d$.

The above operation is denoted by $\tau' \leftarrow \textsc{PropUp}(\tau, v, s_0)$. Having computed the new keys on its direct path, ME proceeds as follows:

- *Encrypt path secrets*: Let $v'_0, \ldots, v'_{d-1}$ be the nodes on the co-path of v (i.e., v'_i is the sibling of v_i). For every value s_i and every node $v_j \in \mathsf{Res}(v'_{i-1})$, ME computes $c_{ij} \leftarrow \mathsf{Enc}(v_j.\mathsf{pk}, s_i)$.
- *Output*: All ciphertexts c_{ij} are concatenated to an overall ciphertext $\mathbf{c}$ (in some canonical order[10]). Let $U \leftarrow (\mathsf{PK}, \mathbf{c})$, where $\mathsf{PK} := (\mathsf{pk}_0, \ldots, \mathsf{pk}_{d-1})$ be the update information for the remaining group members.

The entire update process described above is denoted by $(\tau', U) \leftarrow \textsc{UpGen}(\tau, \mathsf{ID})$. The control message for this operation simply consists of ME's ID and U.

Notation. Later on, it will also be convenient to refer to the set of secret keys
$$\mathsf{RecKeys}(s_i) \;:=\; \{\mathsf{sk}_i, \ldots, \mathsf{sk}_{d-1}, s_d\}$$
that can be recovered from path secret s_i. Moreover, let
$$\mathsf{PKeys}(s_i) \;:=\; \{\mathsf{sk} \mid s_i \text{ is encrypted under the public key corresponding to } \mathsf{sk}\}$$
be the set of secret keys such that s_i is encrypted under the corresponding public keys.

Processing control messages. When processing a control message T, a user first checks whether T corresponds to an operation they initiated. If so, they simply adopt the corresponding RT in $\tau'[\cdot]$.

Whenever T was sent from another user, depending on the type of the control message, proc operates as follows:

- $T = (\texttt{create}, G, \mathbf{pk})$*:* In this case, simply determine the position j of ME in the G list, retrieve the appropriate secret key sk_j from the PKI, and initialize the RT via $\tau \leftarrow \textsc{Init}(G, \mathbf{pk}, j, \mathsf{sk}_j)$.
- $W = (\texttt{wel}, \tilde{\tau})$*:* Simply adopt $\tilde{\tau}$ as the current RT τ and set the secret key at ME's node to the key $\mathsf{get\text{-}sk}(\tau.\mathsf{ME.pk})$ retrieved from the PKI.
- $T = (\mathsf{add}, \mathsf{ID}, \mathsf{ID}', \mathsf{pk}')$*:* Add the new user ID' to the RT and blank all nodes in the direct path of the new user.
- $T = (\texttt{rem}, \mathsf{ID}, \mathsf{ID}')$*:* Blank all nodes on the direct path of user ID' and remove ID' from the RT.
- $T = (\texttt{up}, \mathsf{ID}, U)$*:* A user ID' at some leaf ℓ' receiving $U = (\mathsf{PK}, \mathbf{c})$, issued by the user with id ID at leaf v, recovers the update information as follows: Let $w := \mathsf{Rep}(v, \ell')$. The user with ID', uses $w.\mathsf{sk}$ to decrypt c_{ij} (for the appropriate j) and obtain s_i. Finally, update the ratchet tree by overriding the public-key labels on the v-root-path by the keys in PK, and by then producing a new tree $\tau' \leftarrow \textsc{PropUp}(\tau, \mathsf{LCA}(v, \ell'), s_i)$. The entire process just described is denoted by
$$\tau' \;\leftarrow\; \textsc{UpPro}(\tau, \mathsf{ID}, \mathsf{ID}', U).$$

Irrespective of whether T was created by ME or another user, after processing it, TK-proc resets the variables pertaining to keeping track of ME's unconfirmed operations.

[10] For the sake of concreteness, consider the order obtained by first sorting the c_{ij} by i and then by j, using the natural ordering for resolutions obtained by first considering the left child and then the right child (cf. Definition 4).

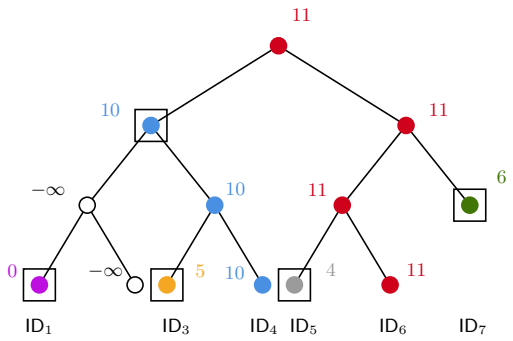

Fig. 5. A ratchet tree showing only the epoch numbers of secret keys; empty nodes are blank. This tree was created by (say) the following sequence of 11 operations: initialization with eight parties $\mathsf{ID}_1, \ldots, \mathsf{ID}_8$; updates by ID_5, ID_2, ID_5, ID_3, and ID_7; removal of ID_8; update by ID_2; removal of ID_2; updates by ID_4 and ID_6. The boxed nodes contain keys from which the attacker can compute the update secret of epoch 11.

5 Security of TreeKEM

Ideally, a CGKA scheme satisfies Definition 2 w.r.t. the safety predicate **safe**. However, this is not the case for TreeKEM. Specifically, while TreeKEM achieves post-compromise security (PCS), it only provides a very weak notion of forward secrecy. We first illustrate this with a simple example in Section 5.1 and then proceed to characterize the exact security of the TreeKEM protocol in Section 5.2, using a predicate **tkm**. While precise (cf. Section 4.4), predicate **tkm** is quite unintuitive and cumbersome. To that end, we show that a scheme secure w.r.t. **tkm** is also secure w.r.t. to the slightly weaker but more intuitive predicates **fsu** and **pcs**; the former captures a notion of forward security while the latter captures PCS without guaranteeing forward secrecy.

5.1 TreeKEM is Not Forward Secret

On an intuitive level, the reason the TreeKEM protocol fails to be forward secret is that after processing the messages generated by an update operation, parties must keep the secret keys used to decrypt the update information since they might be needed for processing future updates. Therefore, corrupting any party other than the update initiator completely reveals the update secret of the previous epoch, and *potentially keys of older epochs* as well, violating forward secrecy.

In order to better understand this issue, imagine that for every secret key that appears in the ratchet tree, the epoch number in which it was created is recorded; for keys retrieved from the PKI, epoch 0 is assigned. At any point during the game, the annotated ratchet tree will be of the following type: each node is either blank or has a secret key whose epoch number equals the maximum of the epoch numbers of its children (where, for simplicity, the epoch number is $-\infty$ for blank nodes). An example of a ratchet tree annotated with these epoch numbers is given in Figure 5.

Consider the security of the update secret I produced in epoch 11 against future corruptions. As per TreeKEMs definition, information about I is encrypted under the public keys of all nodes on the co-path of ID_6. The nodes on said co-path are the epoch-4 key at ID_5's leaf, the epoch-6 key at ID_7's leaf, and highest epoch-10 key in the tree. The latter key, however, can also be recovered from the initial key of ID_1 or the epoch-5 key at ID_3's leaf (since those keys are on the co-path of ID_4, who performed the update in epoch 10). These "dangerous" keys are highlighted by boxes in Figure 5.

Observe now that if the attacker corrupts any party ID *after* epoch 11 but before a boxed key known to ID is overridden by an update, he can compute I. In particular, each of the parties in $\{\mathsf{ID}_1, \mathsf{ID}_3, \mathsf{ID}_5, \mathsf{ID}_7\}$ must execute an update before they are corrupted in order for I to remain secure (as these parties are the only ones that can override the corresponding boxed leaf keys).

Predicate tkm

```
tkm(q_1, ..., q_q)
  (V, E) ← KG(q_1, ..., q_q)
  for (i, j) s.t. q_i = corr(ID) for some ID and t = q2e(q_i), q_j = chall(t*) for some t*
    if I[t*] ∈ K^t_ID
      return 0
  return 1
```

Fig. 6. The predicate **tkm** for the TreeKEM protocol.

5.2 Capturing TreeKEM's Security

In order to capture the security level achieved by the TreeKEM protocol exactly, this section defines a safety predicate **tkm** based on the notion of a *key graph*. The key graph records the relationships among secret keys in an execution of the protocol. That is, it keeps track of which keys can be computed given which other keys (learned via state compromise). Specifically, given a sequence of oracle queries $Q = (\textbf{create-group}, \mathbf{q}_1, \ldots, \mathbf{q}_t)$ the key graph $(\mathcal{V}_t, \mathcal{E}_t) \leftarrow \mathsf{KG}(Q)$ is defined as follows:

- **create-group**$(\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$: The **create-group** operation defines $(\mathcal{V}_0, \mathcal{E}_0)$ as follows:
  - $\mathcal{V}_0 \leftarrow \{\mathsf{sk}_{\mathsf{ID}_1}, \ldots, \mathsf{sk}_{\mathsf{ID}_n}\}$, i.e., $\mathcal{V}_0$ consists of the secret keys of all users in the initial group.
  - $\mathcal{E}_0 \leftarrow \emptyset$.
- $\mathbf{q}_i = \textbf{send-update}(\mathsf{ID})$: Let $\mathsf{sk}_0, \ldots, \mathsf{sk}_{d-1}$ and $s_0, \ldots, s_d$ be the secret keys and path secrets generated by the update operation. Compute[11]
  - $\mathcal{V}_i \leftarrow \mathcal{V}_{i-1} \cup \{\mathsf{sk}_0, \ldots, \mathsf{sk}_{d-1}, s_d\}$.
  - For $j = 1, \ldots, d$, $K_j \leftarrow \{(\mathsf{sk}, \mathsf{sk}') \mid \mathsf{sk} \in \mathsf{PKeys}(s_j), \mathsf{sk}' \in \mathsf{RecKeys}(s_j)\}$.
  - Set $\mathcal{E}_i \leftarrow \mathcal{E}_{i-1} \cup \left(\bigcup_{j \in [d]} K_j\right)$.
- $\mathbf{q}_i = \textbf{add-user}(\mathsf{ID}, \mathsf{ID}')$: Set $\mathcal{V}_i \leftarrow \mathcal{V}_{i-1} \cup \{\mathsf{sk}_{\mathsf{ID}'}\}$.

The queries **remove-user**, **deliver**, do not make any modifications to the key graph, but they indirectly affect the way it evolves.

Let $(\mathcal{V}, \mathcal{E})$ be the key graph defined by executing a sequence of operations of the TreeKEM protocol. For a user with ID ID and an epoch t, $\mathcal{K}^t_{\mathsf{ID}}$ consists of the following elements:

1. The private keys in the state of ID in epoch t.
2. The private keys in $\mathcal{V}$ that are are reachable from the above keys in the key graph $(\mathcal{V}, \mathcal{E})$.

Having defined TreeKEM's key graph, admissible adversaries are now captured via the predicate **tkm** in Figure 6. The predicate essentially makes sure that the attacker does not learn any keys from which a challenged update secret is reachable.

More intuitive predicates. Since predicate **tkm** is very specific to TreeKEM, the security level achieved by TreeKEM is perhaps understood more easily by considering the following two predicates:

- The *PCS predicate*, denoted **pcs**, captures PCS only, i.e., without any kind of forward secrecy. This is achieved by excluding corruptions after any challenge (on top of the normal safety predicate).
- The notion of limited forward secrecy (FS) captured here is *FS with updates (FSU)*. Specifically, when the state fo a party ID is leaked, then all keys *before* the most recent update by ID remain secret.

In the following lemma, we establish relations between the these predicates and **tkm**. The proof of the lemma can be found in Appendix E.

Lemma 3. *For any sequence of queries Q, if* $\mathbf{pcs}(Q) = 1$ *or* $\mathbf{fsu}(Q) = 1$*, then* $\mathbf{tkm}(Q) = 1$.

In Section 5.4, we also show that the formalization introduced above is necessary for evaluating and proving security for the TreeKEM protocol.

[11] See Section 4.4, page 13, for a definition of the sets PKeys and $\mathsf{RecKeys}$.

PCS and FSU Predicates

pcs $(\mathbf{q}_1, \ldots, \mathbf{q}_q)$
  if $\exists (i,j)$, *s.t.* $\mathbf{q}_i = \mathbf{corr}(\mathsf{ID})$ *for some* ID, $\mathbf{q}_j = \mathbf{chall}(t^*)$ *for some* t^*, *and* $\mathsf{q2e}(\mathbf{q}_i) > t^*$
    return 0
  return safe$(\mathbf{q}_1, \ldots, \mathbf{q}_q)$

fsu $(\mathbf{q}_1, \ldots, \mathbf{q}_q)$
  for (i,j) *s.t.* $\mathbf{q}_i = \mathbf{corr}(\mathsf{ID})$ *for some* ID, $\mathbf{q}_j = \mathbf{chall}(t^*)$ *for some* t^*
    if $t^* < \mathsf{q2e}(\mathbf{q}_i)$ *and* $\nexists\, k$ *s.t.* $\mathbf{q}_k = \mathbf{send\text{-}update}(\mathsf{ID})$ *s.t.* $t^* < \mathsf{q2e}(\mathbf{q}_k) \leq \mathsf{q2e}(\mathbf{q}_i)$
      return 0
  return safe$(\mathbf{q}_1, \ldots, \mathbf{q}_q)$

Fig. 7. The PCS predicate **pcs** and the FS-with-updates predicate **fsu**.

5.3 Proof of Security of TreeKEM

This section presents the following security result for the TreeKEM protocol and provides high-level intuition for the security proof. The details of the proof can be found in Appendix F.1.

Theorem 1 (Non-adaptive security of TreeKEM). *Assume that*

- prg *is a* $(t_{\mathsf{prg}}, \varepsilon_{\mathsf{prg}})$*-secure pseudo-random generator,*
- Π *is a* $(t_{\mathsf{cpa}}, \varepsilon_{\mathsf{cpa}})$*-CPA-secure public-key encryption scheme,*

Then, TreeKEM is a $(t, c, n, \mathsf{P}, \varepsilon)$*-secure CGKA protocol, for* $\mathsf{P} \in \{\mathbf{tkm}, \mathbf{pcs}, \mathbf{fsu}\}$, $\varepsilon = 2cn(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}})$, *and* $t \approx t_{\mathsf{prg}} \approx t_{\mathsf{cpa}}$.

Proof intuition. Consider an execution of the (single-challenge) CGKA game with the TreeKEM scheme. Recall that an update operation by a node at depth d produces, for a uniformly random s_0, the values

$$s_0 \xrightarrow{\mathsf{prg}} (\mathsf{sk}_0, s_1) \xrightarrow{\mathsf{prg}} (\mathsf{sk}_1, s_2) \xrightarrow{\mathsf{prg}} \ldots \xrightarrow{\mathsf{prg}} (\mathsf{sk}_{d-1}, s_d)$$

where $I = s_d$ is the update secret. In the example tree in Figure 5, assume that the update secret $I = s_3$ created in epoch 11 is challenged. Observe that the last update (by ID_6) encrypts information about I under the keys at the nodes on ID_6's co-path. These keys stem from epochs 4, 6, and 10, respectively. To use the CPA security of said keys to argue that no information about I is obtained by the attacker, one has to recursively check under which other keys information about them has been encrypted. For example, in epoch 10, information was encrypted using a key from epoch 5 and the initial key of ID_1 (who has never performed an update).

Therefore, the proof proceeds in a series of hybrids that fake ciphertexts and replace PRG outputs by random values in a bottom-up fashion, i.e., beginning with the nodes at the greatest depths. In the example of Figure 5, the hybrids would be the following (highlighting the differences in each step):

- H_d^{c}: Is identical to the original CGKA experiment.
- H_d^{p}: When the updates in epochs 4, 5, 10, and 11 are computed, the output of the first application of the PRG is replaced by a uniformly random value, i.e., instead of computing $(\mathsf{sk}_0, s_1) \leftarrow \mathsf{prg}(s_0)$, sk_0 and s_1 are simply chosen randomly. The rest of the update is computed normally. The security of this step follows from that of the PRG.
- H_{d-1}^{c}: When the updates in epochs 10 and 11 are computed, instead of encrypting s_1 under the corresponding key on the co-path, the all-zero string is encrypted. This step is safe by the CPA security of the PKE in use and the fact that the secret keys at depth d produced by the updates in epochs 4, 5, 10, and 11 are chosen randomly.
- H_{d-1}^{p}: When the updates in epochs 6, 10, and 11 are computed, all PRG computations at depth $d-1$ are replaced by choosing uniformly random values. That is, instead of applying the PRG, the values (sk_0, s_1) (in the case of epoch 6) and (sk_1, s_2) (in the case of epoch 10 and 11) are chosen randomly. The security of this step follows from that of the PRG and by observing that encryptions of s_1 have been replaced by dummy encryptions in the previous hybrid.

– H^{c}_{d-1}: When the updates in epochs 10 and 11 are computed, instead of encrypting s_2 under the corresponding keys on the resolution of the co-path nodes, the all-zero string is encrypted. This step is safe by the CPA security of the PKE in use and the fact that the secret key at depth $d-1$ produced by the update in epoch 6 and the initial key of ID_1 are chosen randomly.
– H^{p}_{d-2}: Similarly to H^{p}_{d-1}, values (sk_2, s_3) are chosen randomly when computing updates in epochs 10 and 11.
– H^{c}_{d-3}: In epoch 11, the encryption of s_3 is replaced by a dummy encryption.

Observe that in H^{c}_{d-3}, the adversary is now not provided with any information about $s_3 = I$ in update 11. Hence, its advantage in the final hybrid is 0.

Adaptive security for TreeKEM. Due to space limitations, adaptive security is discussed in Section F.2, where we derive that TreeKEM is adaptively secure with security loss factor of $O(n^{\log n})$.

5.4 Is the TreeKEM Key Graph Necessary?

In the current section we argue about the necessity of key graphs for the TreeKEM protocol. First we prove that if **tkm** is not satisfied, then there exists an attack against TreeKEM.

Lemma 4. *Let Q be a sequence of queries to the oracles to the CGKA game (cf. Section 3). Then, if $\mathbf{tkm}(Q) = 0$, security of the TreeKEM protocol breaks with probability 1.*

Proof. Since $\mathbf{tkm}(Q) = 0$, then by the definition of **tkm**, either the challenge key I^* is in the private state of a corrupted user or there is a key sk in the private state of the corrupted user such that there exists a path from sk to I^* in the key graph with respect to Q. Clearly, in the former case security trivially breaks as the adversary learns I^*. In the latter case assume a path $\mathsf{sk} = sk_1, \ldots, \mathsf{sk}_l = I^*$ in TreeKEM's key graph. Then, by the definition of the key graph, there exist path secrets $s_2, \ldots, s_l$ such that s_i is encrypted under pk_{i-1} (which is the public key that corresponds to sk_{i-1}) and sk_i is computable from s_i via a sequence of zero or more prg and Dec operations. Thus, the adversary by learning $\mathsf{sk} = \mathsf{sk}_1$ it can compute all the keys in the path $\mathsf{sk} = sk_1, \ldots, \mathsf{sk}_l = I^*$. □

It would be tempting to argue that if the adversary compromises keys that have been overwritten by the protocol update operation, this would not affect the security of the protocol, thus a version of the key graph that omits overwritten keys, would suffice for describing the TreeKEM predicate and proving security for the TreeKEM protocol. However, this is not the case as we formally argue in the following lemma.

Lemma 5. *There exists a sequence of TreeKEM protocol operations, under which the recovery of overwritten keys breaks security of the protocol with probability 1.*

Proof. Assume the adversary is allowed to recover overwritten keys and consider the execution depicted in Figure 8. Observe that in the given sequence of update operations $\mathsf{sk}^1_{c,d}$ is overwritten by $\mathsf{sk}^3_{c,d}$ when id c performs the last update operation. Now, if id d is compromised in epoch 3, the adversary recovers sk^0_d. Let $\mathsf{c} \leftarrow \mathsf{Enc}_{\mathsf{pk}^0_d}(s_1)$, be the ciphertext generated by c during the update. Given sk^0_d, the adversary can compute $s_1 \leftarrow \mathsf{Dec}_{\mathsf{sk}^0_d}(\mathsf{c})$ and $(\cdot, I_2) \leftarrow \mathsf{prg}(s_1)$, recovering the update secret of epoch 2. □

6 Optimal Forward Secrecy

The level of security satisfied by the TreeKEM protocol is limited, as shown in Section 5. In order to achieve better security, this section presents a modified version of TreeKEM that is secure even w.r.t. to predicate **safe**. The new version of the protocol is based on a suggestion by Kohbrok [16] on the MLS mailing list and uses so-called *updatable public-key encryption (UPKE)* (cf. Jost *et al.* [15]). In this work we use a variant of UPKE, which we formally present in Section 6.2; a construction of a slightly stronger variant can be found in Section 7.

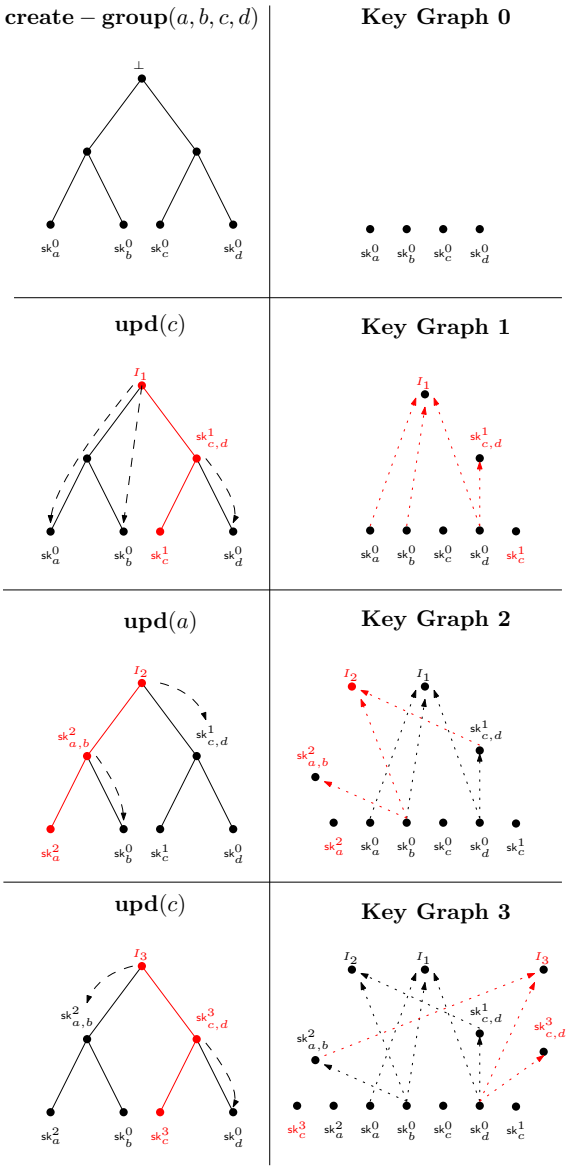

Fig. 8. The figure for the proof of Lemma 5. It depicts an example of the ratchet tree (left column) and the corresponding key graph (right column) for an execution with 4 users/ids, a, b, c, d, with the following sequence of update operations: first c updates, then a, and then c again. The edges in the ratchet tree along the direct path of the user who is issuing an update operation are marked red; a dashed black edge (u, v) denotes the fact that the secret of u is encrypted under the pk_v. Also by $\mathsf{sk}^i_{x,y}$, we denote a secret key introduced in epoch i and is known to the ids x, y. The corresponding key graph (cf. Section 5.2) is depicted in the left column. Newly introduced (dotted) edges are marked by red color.

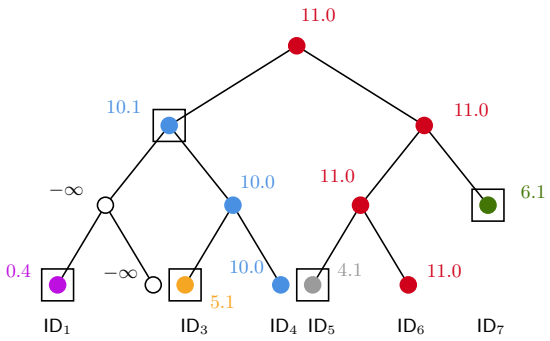

Fig. 9. A ratchet tree showing only the epoch and version numbers of secret keys; empty nodes are blank. This tree was created by (say) the following sequence of 11 operations: initialization with eight parties $\mathsf{ID}_1, \ldots, \mathsf{ID}_8$; updates by ID_5, ID_2, ID_5, ID_3, and ID_7; removal of ID_8; update by ID_2; removal of ID_2; updates by ID_4 and ID_6. The boxed nodes contain keys under whose earlier versions information leading to update secret of epoch 11 was encrypted.

6.1 Fixing TreeKEM

In a nutshell, the new TreeKEM protocol uses UPKE instead of normal PKE. On an intuitive level, the encryption algorithm of a UPKE scheme outputs a new public key (to be used for future encryptions) along with the ciphertext. Similarly, the decryption algorithm outputs a corresponding new secret key. This is done in such a fashion that even given the new version of the secret key, no information about the plaintexts encrypted under older versions is revealed.

In order to better understand how this solves the issue with TreeKEM's subpar forward secrecy (FS), consider the example execution of the TreeKEM protocol depicted in Figure 9 (which was already used in Section 5). Once more, imagine that for every secret key that appears in the ratchet tree, the epoch number in which it was created is recorded, where keys retrieved from the PKI and used when the group is created, are assigned epoch 0. Imagine further that, in addition to the epoch number, the *version* number of each key is recorded. More precisely, a UPKE key generated by an update operation has version 0, and with every plaintext encrypted under it, the version number is incremented by 1. For example, in Figure 9, the initial key of ID_1 has been used by four different update operations.

It is now easy to see how UPKE solves the FS issue: While in the plain version, the boxed keys could be used to recover the update secret I of epoch 11, in the new TreeKEM, the information that would allow recovery of I was encrypted under the second most recent version of the boxed keys. However, UPKE now guarantees that the most recent version of any boxed key obtained upon corruption of a corresponding user after epoch 11 reveals no information about I. For example, information about epoch 11's update secret was encrypted under version 0 of the highest epoch-10 key. In turn, information about the latter key was encrypted under version 3 of the key at ID_1's leaf. As a result of these encryptions the two keys are now at versions 1 and 4, respectively. Thus, even if, say, ID_1 is compromised after epoch 11, the new key versions reveal no useful information about epoch 11's update secret.

6.2 Updatable Public-Key Encryption

Below we define a variant of UPKE in which the public (resp. private) key update functionality is implemented by the encryption (resp. decryption) operation. This is how our UPKE notion deviaties from that of [15], in which the key update operations are implemented by independent functionalities.

Definition 6. *An* updatable public-key encryption *(UPKE) scheme is a triple of algorithms* $\mathsf{UPKE} = (\mathsf{PKEG}, \mathsf{Enc}, \mathsf{Dec})$ *with the following syntax:*

- Key generation: PKEG *receives a uniformly random key* sk_0 *and outputs a fresh initial public key* $\mathsf{pk}_0 \leftarrow \mathsf{PKEG}(\mathsf{sk}_0)$.
- Encryption: Enc *receives a public key* pk *and a message* m *and produces a ciphertext* c *and a new public key* pk'.

– Decryption: Dec *receives a secret key* sk *and a ciphertext* c *and outputs a message* m *and a new secret key* sk'.

Correctness. A UPKE scheme must satisfy the following correctness property. For any sequence of randomnesss and message pairs $\{r_i, m_i\}_{i=1}^{q}$,

$$\mathsf{P}\left[\begin{array}{c}\mathsf{sk}_0 \leftarrow \mathcal{SK}; \mathsf{pk}_0 \leftarrow \mathsf{PKEG}(\mathsf{sk}_0);\ \text{For } i \in [q], (c_i, \mathsf{pk}_i) \leftarrow \mathsf{Enc}(\mathsf{pk}_{i-1}, m_i; r_i);\\ (m_i', \mathsf{sk}_i) \leftarrow \mathsf{Dec}(\mathsf{sk}_{i-1}, c_i) : m_i = m_i'\end{array}\right] \ = \ 1.$$

The notion of CPA security that we define below is along the lines of CPA-secure PKE, with the only difference being that for honestly generated ciphertexts the adversary receives access to the randomness that produced them. In this way we capture protocol executions in which prior to the challenge epoch, the adversary receives access to the ciphertexts generated by users (by observing the network), as well as to the randomness used by corrupted users to encrypt path secrets prior to the challenge epoch.

IND-CPA security for UPKE. For any adversary $\mathcal{A}$ with running time t we consider the IND-CPA security game:

– Sample $\mathsf{sk}_0 \leftarrow \mathcal{SK}$, $\mathsf{pk}_0 \leftarrow \mathsf{PKEG}(\mathsf{sk}_0)$, $b \leftarrow \{0,1\}$.
– $\mathcal{A}$ receives pk_0 and for $i = 1, \ldots, q$, $\mathcal{A}$ outputs m_i and receives $(c_i, \mathsf{pk}_i, r_i)$ such that $(c_i, \mathsf{pk}_i) \leftarrow \mathsf{Enc}(\mathsf{pk}_{i-1}, m_i; r_i)$, for uniformly random r_i
– $\mathcal{A}$ outputs (m_0^*, m_1^*)
– For $i = 1, \ldots, q$, compute $(m_i, \mathsf{sk}_i) \leftarrow \mathsf{Dec}(\mathsf{sk}_{i-1}, c_i)$
– Compute $(c^*, \mathsf{pk}^*) \leftarrow \mathsf{Enc}(\mathsf{pk}_q, m_b^*)$, $(\cdot, \mathsf{sk}^*) \leftarrow \mathsf{Dec}(\mathsf{sk}_q, c^*)$
– $b' \leftarrow \mathcal{A}(\mathsf{pk}^*, \mathsf{sk}^*, c^*)$

$\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ in winning the above game is denoted by $\mathsf{Adv}_{\mathrm{cpa}}^{\mathsf{UPKE}}(\mathcal{A})$.

Definition 7. *An updatable public-key encryption scheme* UPKE *is* (t, ε)-CPA-secure *if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}_{\mathrm{cpa}}^{\mathsf{UPKE}}(\mathcal{A}) \ \leq \ \varepsilon.$$

6.3 An Optimally Secure Protocol

The new TreeKEM protocol presented in the this section uses UPKE CPA-secure encryption in place of standard CPA-secure encryption. Using UPKE when a user issues an update operation not only updates the PKE keys in its direct path but also the PKE keys of all nodes in the resolution of the co-path nodes. The new TreeKEM protocol is presented by highlighting the differences to TreeKEM.

The initialization, group creation, user addition/removal operations of the protocol are identical to those of TreeKEM. The only difference is the use of UPKE. The *update* and *process* operations work as shown next.

Performing an update. A user performs an update as follows:

– *Compute path secrets*: Let $v_0 = v, v_1, \ldots, v_d$, be the nodes along the direct path of the node v who issues an update. For uniformly random s_0 compute

$$\mathsf{sk}_i \| s_{i+1} \leftarrow \mathsf{prg}(s_i), \ \text{for } i = 0, \ldots, d-1.$$

– *Update the RT labels along the direct path*: For $i = 0, \ldots, d-1$, compute $\mathsf{pk}_i \leftarrow \mathsf{PKEG}(\mathsf{sk}_i)$ and the PKE label of v_i is updated to $(\mathsf{pk}_i, \mathsf{sk}_i)$.
– *Root node*: For the root, set $I := s_d$.

Up to now, the computation is identical to the one in the TreeKEM protocol.

– *Encrypt path secrets and update public keys*: Let $v_0', \ldots, v_{d-1}'$, be the nodes on the co-path of v (i.e., v_i' is the sibling of v_i). For every value s_i and every node $v_j \in \mathsf{Res}(v_{i-1}')$, compute $(c_{ij}, \mathsf{pk}_{ij}) \leftarrow \mathsf{Enc}(v_j.\mathsf{pk}, s_i)$ and set the public key of v_j to pk_{ij}.
– *Output*: All ciphertexts c_{ij} are concatenated to an overall ciphertext $\mathbf{c}$ and all keys pk_{ij} are stored in $\bar{\mathsf{PK}}$. Return $U \leftarrow (\mathsf{PK}, \bar{\mathsf{PK}}, \mathbf{c})$, where $\mathsf{PK} := (\mathsf{pk}_0, \ldots, \mathsf{pk}_{d-1})$.

This extended update process is denoted by $(\tau', U) \leftarrow \textsc{ExtUpGen}(\tau, \mathsf{ID})$.

Processing control messages. Processing control messages is similar to the TreeKEM protocol. The main difference is in the way the users process the output of the public key encryption scheme. In particular, for any node of the ratchet tree, v, when processing the output of the encryption operation under the public key of v, $(c, \mathsf{pk}'_v) \leftarrow \mathsf{Enc}(\mathsf{pk}_v, s)$, users compute $(s, \mathsf{sk}'_v) \leftarrow \mathsf{Dec}(v.\mathsf{sk}, c)$, process the path secret s as in TreeKEM, but in addition they set the public and secret key of v to pk'_v and sk'_v, respectively.

6.4 Security of the New TreeKEM

The modified version of the TreeKEM protocol satisfies optimal security, i.e., security w.r.t. the predicate **safe**. The proof of Theorem 2 can be found in Appendix F.3.

Theorem 2 (Non-adaptive security of Modified TreeKEM). *Assume that*

- prg *is a* $(t_{\mathsf{prg}}, \varepsilon_{\mathsf{prg}})$*-secure pseudo-random generator,*
- Π *is a* $(t_{\mathsf{cpa}}, \varepsilon_{\mathsf{cpa}})$*-CPA-secure updatable public-key encryption scheme.*

Then, the protocol of Section 6.3 is a $(t, c, n, \mathbf{safe}, \varepsilon)$*-secure CGKA protocol, for* $\varepsilon = 2cn(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}})$ *and* $t \approx t_{\mathsf{prg}} \approx t_{\mathsf{cpa}}$*.*

Adaptive security for Modified TreeKEM. Due to space limitations, adaptive security is discussed in Section F.4, where we argue that Modified TreeKEM is adaptively secure with security loss factor of $O(n^{\log n})$.

7 Constructions of Updatable Encryption

In this section we construct UPKE satisfying a stronger notion of CPA security than the one we presented in Section 6. The new notion considers adversaries that control the randomness used by the encryption oracle, and can be potentially useful for constructing CGKA protocols that will be secure against adversaries that control the randomness used by honest users for the update operation.

Below we formally define security for the stronger notion which we denote by CPA*.

IND-CPA* security for UPKE. For any adversary $\mathcal{A}$ with running time t we consider the IND-CPA* security game:

- Sample $\mathsf{sk}_0 \leftarrow \mathcal{SK}$, $\mathsf{pk}_0 \leftarrow \mathsf{PKEG}(\mathsf{sk}_0)$
- $\mathcal{A}$ on input pk_0 outputs $(m_0^*, m_1^*), \{r_i, m_i\}_{i=1}^q$
- For $i = 1, \ldots, q$, compute $(c_i, \mathsf{pk}_i) \leftarrow \mathsf{Enc}(\mathsf{pk}_{i-1}, m_i; r_i); (m_i, \mathsf{sk}_i) \leftarrow \mathsf{Dec}(\mathsf{sk}_{i-1}, c_i)$
- Compute $b \leftarrow \{0, 1\}$, $(c^*, \mathsf{pk}^*) \leftarrow \mathsf{Enc}(\mathsf{pk}_q, m_b^*)$, $(\cdot, \mathsf{sk}^*) \leftarrow \mathsf{Dec}(\mathsf{sk}_q, c^*)$
- $b' \leftarrow \mathcal{A}(\mathsf{pk}^*, \mathsf{sk}^*, c^*)$

$\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ in winning the above game is denoted by $\mathsf{Adv}^{\mathsf{UPKE}}_{\mathrm{cpa}^*}(\mathcal{A})$.

Definition 8. *An updatable public-key encryption scheme* UPKE *is* (t, ε)-CPA*-secure *if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{UPKE}}_{\mathrm{cpa}^*}(\mathcal{A}) \ \leq \ \varepsilon \ .$$

When referring to security against polynomial-time adversaries having negligible (in the security parameter) advantage ϵ, we will simply use the term CPA*-secure, i.e., we omit (t, ε).

It is not hard to see that the above notion implies UPKE CPA security as defined in Section 6.

7.1 A UPKE scheme based on CDH and RO

We realize our notion using the construction of [15]. The scheme in formally defined in Figure 10.

Theorem 3. *Assuming the hardness of CDH over the group* G*, the UPKE scheme of Figure 10 is CPA*-secure if* H *is modeled as a random oracle.*

Proof. The proof is along the lines of the proof given by Jost *et al.* [15] for the construction of secretly key-updatable public-key encryption. Given an adversary $\mathcal{A}$ that breaks the CPA security of the UPKE scheme of Figure 10, we define an adversary $\mathcal{A}^*$ that breaks the CDH assumption (given a challenge $(A, B) = (g^a, g^b)$) assuming H is modeled as a random oracle. $\mathcal{A}^*$ is defined below.

Construction of Updatable Public Key Encryption

$\mathsf{PKEG}(\mathsf{sk})$
  return g^{sk}

$\mathsf{Enc}(\mathsf{pk}, m)$
  $(r, \delta) \leftarrow\$ \mathbb{Z}_q \times \mathbb{Z}_q$
  $c \leftarrow (g^r, H(\mathsf{pk}^r) \oplus (m \| \delta))$
  return $(c, \mathsf{pk} \cdot g^{\delta})$

$\mathsf{Dec}(\mathsf{sk}, (c_1, c_2))$
  $m \| \delta \leftarrow H(c_1^{\mathsf{sk}}) \oplus c_2$
  $\mathsf{sk}' \leftarrow (\mathsf{sk} + \delta) \mod q$
  return (m, sk')

Fig. 10. A UPKE scheme assuming Random Oracles and the hardness of Computational Diffie Hellman. g is the generator of a group G of prime order q and sk is in $\mathbb{Z}_q$.

$\mathcal{A}^*(A, B)$:

1. Sample $b \leftarrow \{0,1\}$, $\delta \leftarrow \mathbb{Z}_q$ and set $\mathsf{pk}_0 \leftarrow A \cdot g^{\delta}$
2. Execute $\mathcal{A}$ on input pk_0 and receive $(m_0^*, m_1^*), \{r_i, m_i\}_{i=1}^q$
3. For $i = 1, \ldots, q$, compute $(c_i, \mathsf{pk}_i) \leftarrow \mathsf{Enc}(\mathsf{pk}_{i-1}, m_i; r_i)$
4. Let $\mathsf{pk}_q = g^{a + \sum_{i=1}^q \delta_i + \delta}$ and $\Delta := \sum_{i=1}^q \delta_i + \delta$, where each δ_i is defined with respect to the randomness r_i chosen by $\mathcal{A}$.
5. Set $\mathsf{pk}^* \leftarrow \mathsf{pk}_q \cdot A^{-1}$ and $\mathsf{sk}^* \leftarrow \Delta$
6. Compute $R \leftarrow \{0,1\}^{|m|+|\delta|}$ and set $c^* \leftarrow (B, R)$
7. Let Q be the list of all queries made to the random oracle by $\mathcal{A}$. Multiply all elements by $B^{-\Delta}$ and output the output of the Diffie-Hellman self-corrector of [23] with respect to Q to obtain a solution.

Observe that, given the computation defined in the steps 1-5, the adversary should receive in step 7 the triple,

$$\left(\mathsf{pk}^* = g^{\Delta}, \mathsf{sk}^* = \Delta, c^* = \left(B, H\left(g^{(a+\Delta)b}\right) \oplus m_b^* || (-a)\right)\right).$$

However, in the above execution it receives

$$\left(\mathsf{pk}^* = g^{\Delta}, \mathsf{sk}^* = \Delta, c^* = (B, R)\right),$$

for uniformly random R. The only way for the adversary to distinguish between the above, is by querying the random oracle with $g^{(a+\Delta)b}$, thus by multiplying $g^{(a+\Delta)b} B^{-\Delta}$ it computes g^{ab}. □

8 Conclusions and Future Directions

8.1 Overview

In our paper, we formally have defined the notion of continuous group key agreement (CGKA). We have shown that TreeKEM, the current proposal for CGKA within the MLS messaging, has limited forward secrecy, and we have formally established the exact type of forward secrecy achieved by TreeKEM. Furthermore, we have provided a simple fix to the protocol and shown that the fixed version indeed has optimal forward secrecy.

Below we touch on several directions for future research surrounding secure group messaging (SGM)—and CGKA in particular. A more detailed discussion can be found in Section 8.2.

Provably secure group messaging. Probably one of the most immediate open problems is to provide a security definition and proof for a complete SGM protocol (i.e., supporting adds, removes, sends, etc.) To that end, the CGKA primitive defined in this work will (as explained in Section 3.3) play a key role in a (provably secure) SGM construction—just as its 2-party analog CKA did for two-party messaging in [1]. Indeed, the MLS protocol in its current form can be viewed as the result of such a composition (with TreeKEM as the CGKA). Thus, investigating the (provable security) properties for such a construction remains an important direction for future work.

Bridging the theory/practice divide. Given the very wide use of secure (group) messaging protocols in practice, it is especially interesting to try to further bridge the gaps between formal results and practice. One important step in this direction is to more accurately model the PKI upon which these protocols rely. Another step involves tighter concrete security bounds for adaptive security. A third one is to capture and prove security in a model where no assumptions are made about the delivery service (i.e., for an arbitrary adversarially controlled network).

Stronger security notions. We believe there is a lot of room for meaningful stronger security notions beyond the ones investigated in this work. These notions include security against *active* adversaries instead of just passive ones, protection of meta-data (especially from corrupt delivery servers), and deniability. Indeed, each of these has been explicitly mentioned as "great to have" security features during discussions in the MLS working group. Moreover, the MLS protocol already implements some techniques for achieving the latter two. So far, there has been no formal analysis of what those techniques achieve, let alone a discussion of how best to go about achieving such security goals.

Better constructions. There is much one could still hope to improve on in terms of SGM constructions. This holds both for practical constructions and for ones of only theoretical interest. As explained above, the gap between what we know how to build (never mind prove secure) and what could—at least in principle—still be possible remains wide open. Thus, even practically unusable but theoretically sound protocols that raise the bar for what we can prove and how secure we can make things would already be of great interest. On a more practical note, given that forward secrecy is a key security goal, practical post-quantum constructions would also be of value. More generally, however, it would be especially helpful to find constructions with better (exact) communication complexity than the current MLS draft. This is motivated by the fact that, in certain circumstances MLS packets can degrade all the way to size $\Omega(n)$. Worse, this behaviour can, in principle, continue indefinitely and/or reoccur in any session once the right conditions are created. Ideally, one would like a protocol where packets remain of size $O(\log(n))$ under any circumstances. Practically speaking, however, anything that improves on MLS's complexity would already be very helpful. In fact, this is probably one of the most important open problems in the area given that reducing packet size from $\Omega(n)$ to $O(\log(n))$ was the central reason the IETF initiated the MLS project in the first place. Furthermore, the way trees grow and shrink in TreeKEM seems rather arbitrary, and, in fact, the current draft does not seem to explain how trees with a lot of empty leaves are to be pruned. A protocol using balanced trees could possibly be a remedy here.

8.2 More Details on Future Directions

We expand further on open problems and future directions on secure group messaging.

Real-world PKI. A practical consideration which we believe merits further investigation is the accurate modeling of PKI. Concretely, MLS calls for each user to maintain a pool of, so called, InitKeys to be stored on an untrusted key server (KS). To invite Alice to a group, Bob must first obtain an InitKey for Alice from the KS and authenticate it using some external mechanism. In practice for example, this most often consists of Bob checking a digital signature of the retrieved InitKey using Alice's long term signing keys. So before any of this takes place Bob must first obtain Alice's long term verification key and authenticate it through some out-of-band mechanism (e.g., by visually comparing 2D-barcode fingerprints of the verification key he retrieved to the key on her device). Give that this mechanism forms the root of trust of all SGM sessions, formal security definitions and proofs for CGKA and SGM protocols should be extended to model it more accurately. In particular, this would allow better capturing the real world behaviour of such protocols.

Practical Adaptive Security. Another gap that could do with closing is to give tighter concrete bounds on a CGKA's (or SGM's) *adaptive* security. The current bounds we sketched above are derived analogously to the state-of-the-art for adaptive multi-cast in [14]. Unfortunately, they are so loose as to be somewhat meaningless for groups of any interesting size. (For example, to apply, they would require using block ciphers with keys of size $O(n^{\log(n)})$ which is, practically speaking, unrealistic.) As there is no reason to assume adversaries will not be adaptive, improving on these loose bounds remains an important open problem. Although improvements could come in the form of a new protocol (and security proof), we conjecture this is mainly a limitation of our proof techniques rather than a consequence of actual adaptive attacks.

8.3 Stronger Security Notions

Arbitrary networks. As mentioned already, one clear improvement would be to remove the assumption in this work that the delivery server (i.e., the underlying communication network) provides any kind of guarantees. In principle, this would be more aligned with the spirit of end-2-end cryptography. More concretely though, it is also an (at least implicit) design goal for the MLS protocol. That is, while MLS does use the delivery services to ensure functionality, it should *not* be that privacy or authenticity of chat sessions can be violated by a corrupt deliver service.

It is worth noting that, to the best of our knowledge, no known SGM (or even just CGKA) protocols exist achieving ideal security in an untrusted network model. By "ideal" security, we intuitively mean that the adversary should not be able to produce any messages (or group keys) using leaked states and network traffic, than what is implied by the desired functionality of the protocol. For example, if Alice should be able to process a given incoming protocol packet with her current state then so could the adversary. Ideal security means the adversary can do nothing beyond that.

To motivate the claim that we will need new constructions to achieve ideal security (even allowing for "only" state leakage corruptions) we briefly sketch an attack on TreeKEM in the arbitrary network model with state leakage. The attack generalizes to all current variants of TreeKEM. Thus, it implies that to achieve ideal security in the arbitrary network model (even only for state leakage) we will need new constructions.

Suppose, Alice, Bob and Charlie are in a (TreeKEM) group (along with other users). Moreover, assume they are in the same group state S_0 and Bob is Charlie's sibling in the ratchet tree in S_0. Alice and Bob now simultaneously each produce an update packet, defining group states S_A and S_B, respectively. Alice's update is delivered to Charlie who processes it leaving him in state S_A. Notice that Charlie is now no longer expected to be able to process Bob's update since he's in an incompatible state. However, suppose Charlie's state is now leaked to the adversary. As Bob and Charlie are siblings, Charlie would have needed his leaf key from S_0 to process Bob's update. Conversely, as Alice is *not* Charlie's sibling, processing updates from Alice results in no change to Charlie's leaf key. In other words his leaf secret in S_A is the same as in S_0. That means, the adversary can now actually process Bob's update to recover the group key for state S_B even though we did not expect Charlie to be able to do so.

We remark that using Kohbrok's improvement [16] here would not remove this vulnerability. (Albeit, it would fix the problem when Alice and Bob's direct paths intersect Charlie's at the same node.) Moreover, for Casual-TreeKEM [24], which explicitly permits computing updates, we could instead have Alice and Bob each send out an Add message followed by an update. Since Add messages do not commute with updates Charlie would again no longer be expected to be able to process Bob's messages after processing Alice's. Yet, the adversary might still be able to, e.g., if Bob is her sibling in the ratchet tree.

Active security. Another aspect one could hope to improve on is the somewhat weak corruption model. Currently we consider "only" local state leakage. However, a more robust notion would consider attacks by malicious insiders a.k.a. *active* security. It is relatively straightforward to come up with attacks in such security models against TreeKEM (and its variants) so we believe that advancements in this direction will also require new constructions (not just new definitions and proof techniques).

Meta-data protection. Another well motivated (but difficult to achieve) security property has seen essentially no formal work at the SGM protocol level, namely meta-data protection; in particular, from corrupt delivery servers. While almost all practical SGM protocols do indeed implement one form or another of defense against such attacks (e.g. MLS encrypts packet headers containing sender information) it would be helpful to more precisely analyze what these techniques achieve; not to mention investigate other better techniques to the same end.

Deniability. Finally, one security notion which has received a fair amount of attention already (but not for MLS) is deniability. Although so far the focus of the MLS work-group seems has been primarily on efficiency, privacy and authenticity, deniability has been identified as an additional desirable security property. Even in the 2-party case the double ratchet based protocols seem to aim for some form of deniability. However, to the best of our knowledge, no formal proof of deniability exist for any of these protocols.

8.4 Local Update Policies

At any given moment the security of a given group messaging session is intimately tied to the sequence of messages that lead up to this moment. Above all, who updated when determines for which keys and messages we can hope to expect security. What's more, at least for ratchet tree based protocols like TreeKEM/MLS and their variants, which users update maters a lot. Concretely, suppose Alice was the most recent person to update. Then it makes much more sense to have someone on the opposite half of the ratchet tree update next than her sibling since the update from the sibling will replace almost only exactly the keys that Alice just added to the group state herself while leaving all other, older, keys untouched. Already this simple example shows that for a fixed amount of bandwidth spent on updates the type of security actually achieved can vary quite widely from one execution to the next.

However, deciding on a local update policy (as every implementation will have to do) actually depends on a host of subtle issues. Roughly speaking these concern what the right model is, what plausible assumptions about clients can we leverage and what exactly are the goals of an update policy.

When it comes to the model there are several moving parts. How can we best model the online/offline behaviour of parties? (E.g. how often do they come online and for how long?) How can we best model bandwidth availability and cost? E.g. Is there a strict upper bound? Or does the cost of each additional MiB transmitted cost more than the last? A well designed update policy might also consider state specific information. E.g. for TreeKEM suppose some keys in the ratchet tree have been used a lot recently (e.g. to process update messages) while another key has not. It might be higher priority for the heavily used keys to be replaced than the unused ones.

In terms of plausible useful assumptions: normally in cryptography we tend to think of all parties as being essentially the same. However, in practice this can be far from true in ways that can be used to improve security. Can we leverage some of these differences between clients? E.g. some might have cheaper bandwidth than others, or be at a lower risk of compromise, or be online more often or for more time than others.

Finally, it is not clear what the precise goal of an update policy should be. E.g. is it to ensure that each successive message becomes forward secure as fast as possible in terms of number of epochs? Alternatively the goal might be to minimize the probabilities that messages are compromised given the risk profiles of different users in the group? What kind of trade-offs should/should not be made? Is it desirable to leave one message exposed for longer if that means that several others can be secured more quickly?

8.5 Decentralization

Currently MLS is defined to operate using untrusted but centralized server infrastructure. In particular, a key server (KS) and a delivery server (DS) are used heavily by the protocol e.g. to afford asynchronous communication. However, this focus seems to reflect the fact that the work-group is primarily driven by corporate / organizational interests. We believe there is also a practical need for SGM protocols that operate in a more decentralized setting, e.g., by piggy-backing on a blockchain. (Indeed, several such messaging projects are already deployed and in use.) Although, there are relatively immediate naïve ways to implement the KS and DS in a blockchain, it is unclear best to do this when we add goals such as minimizing meta-data leakage, latency and the cost of writing to the blockchain. Nor is it immediately clear what security is afforded by such a construction. With this in mind, we believe that developing fully decentralized SGMs to be a challenging and well motivated direction for future research.

References

1. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.
2. Richard Barnes. Subject: [MLS] Remove without double-join (in TreeKEM). MLS Mailing List. Mon, 06 August 2018 13:01 UTC, 2018. `https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik`.
3. Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-07, Internet Engineering Task Force, July 2019. Work in Progress.

4. Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650. Springer, Heidelberg, August 2017.
5. Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, October 28, 2004*, pages 77–84, 2004.
6. Ran Canetti, Juan A. Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM'99*, pages 708–716, New York, NY, USA, March 21–25, 1999.
7. Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018.
8. Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017*, pages 451–466, 2017.
9. Cas Cremers, Britta Hale, and Konrad Kohbrok. Revisiting post-compromise security guarantees in group messaging. Cryptology ePrint Archive, Report 2019/477, 2019. `https://eprint.iacr.org/2019/477`.
10. Yevgeniy Dodis and Nelly Fazio. Public key broadcast encryption for stateless receivers. In Joan Feigenbaum, editor, *Digital Rights Management*, pages 61–80, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
11. F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19*, volume 11689 of *LNCS*, pages 343–362. Springer, Heidelberg, August 2019.
12. Amos Fiat and Moni Naor. Broadcast encryption. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 480–491. Springer, Heidelberg, August 1994.
13. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.
14. Zahra Jafargholi, Chethan Kamath, Karen Klein, Ilan Komargodski, Krzysztof Pietrzak, and Daniel Wichs. Be adaptive, avoid overcommitting. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 133–163. Springer, Heidelberg, August 2017.
15. Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Heidelberg, May 2019.
16. Konrad Kohbrok. Subject: [MLS] Improve FS granularity at a cost. MLS Mailing List. Thu, 24 January 2019 09:51 UTC, 2019. `https://mailarchive.ietf.org/arch/msg/mls/WRdXVr8iUwibaQu0tH6sDnqU1no`.
17. M. Marlinspike and T. Perrin. The double ratchet algorithm, 11 2016. `https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf`.
18. Suvo Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of ACM SIGCOMM*, pages 277–288, Cannes, France, September 14–18, 1997.
19. Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-03, Internet Engineering Task Force, September 2019. Work in Progress.
20. Saurabh Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 21–40. Springer, Heidelberg, February 2007.
21. Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2018.
22. Eric Rescorla. Subject: [MLS] TreeKEM: An alternative to ART. MLS Mailing List. Thu, 03 May 2018 14:27 UTC, 2018. `https://mailarchive.ietf.org/arch/msg/mls/WRdXVr8iUwibaQu0tH6sDnqU1no`.
23. Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.
24. Matthew Weidner. Group messaging for secure asynchronous collaboration. MPhil Dissertation, 2019. Advisors: A. Beresford and M. Kleppmann, 2019. `https://mattweidner.com/acs-dissertation.pdf`.
25. Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, February 2000.

A Basic Cryptographic Primitives

A.1 Pseudorandom Generators

A *pseudorandom generator (PRG)* is a function $\mathsf{prg} : \mathcal{W} \to \mathcal{W} \times \mathcal{K}$ such that $\mathsf{prg}(U)$ is indistinguishable from U' for uniformly random $U \in \mathcal{W}$ and $U' \in \mathcal{W} \times \mathcal{K}$. The advantage of an attacker $\mathcal{A}$ at distinguishing between these two distributions is denoted by $\mathsf{Adv}^{\mathsf{prg}}_{\mathrm{prg}}(\mathcal{A})$; the attacker is parametrized by its running time t.

Definition 9. *A pseudorandom generator* prg *is* (t, ε)-secure *if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{prg}}_{\mathrm{prg}}(\mathcal{A}) \ \leq \ \varepsilon \ .$$

A.2 Public-Key Encryption

Definition 10. *A* public-key encryption *(PKE) scheme is a is a triple of algorithms* $\Pi = (\mathsf{PKEG}, \mathsf{Enc}, \mathsf{Dec})$ *with the following syntax:*

- Key generation: PKEG *receives (implicitly) a security parameter and outputs a fresh key pair* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKEG}$.
- Encryption: Enc *receives a public key* pk *and a message* m *and produces a ciphertext* c*.*
- Decryption: Dec *receives a secret key* sk *and a ciphertext* c *and outputs a message* m*.*

A PKE scheme must satisfy the following correctness property.

Correctness. For any message m,

$$\mathsf{Pr}[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKEG}; c \leftarrow \mathsf{Enc}(\mathsf{pk}, m); m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c) : m = m'] \ = \ 1.$$

IND-CPA security. For any adversary $\mathcal{A}$ with running time t we consider the IND-CPA security game:

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKEG}$
- $(m_0, m_1) \leftarrow \mathcal{A}(\mathsf{pk})$
- $b \leftarrow \{0, 1\}; c \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$
- $b' \leftarrow \mathcal{A}(c)$

$\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ in winning the above game is denoted by $\mathsf{Adv}^{\Pi}_{\mathrm{cpa}}(\mathcal{A})$.

Definition 11. *A public-key encryption scheme* Π *is* (t, ε)-CPA-secure *if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\Pi}_{\mathrm{cpa}}(\mathcal{A}) \ \leq \ \varepsilon.$$

B Single-Challenge Security to Multi-Challenge Security

For CGKA schemes, single-challenge (non-adaptive) security implies multi-challenge security, as shown by the following lemma:

Lemma 6 (Single-challenge to multi-challenge). *Let* $\mathsf{P} \in \{\mathbf{safe}, \mathbf{pcs}, \mathbf{fsu}\}$ *and assume that a CGKA protocol* CGKA *is* $(t, 1, n, \mathsf{P}, \varepsilon)$*-secure. Then,* CGKA *is also* $(t', c, n, \mathsf{P}, \varepsilon')$*-secure for* $t' \approx t$ *and* $\varepsilon' = c\varepsilon$*.*

Proof. To prove the lemma, we show that for any $\mathcal{A}$ playing the cgka-na game there exists $\mathcal{A}'$ with a polynomially related run-time to $\mathcal{A}$ and for which:

$$\mathsf{Adv}^{\mathsf{CGKA},\mathsf{P}}_{\text{cgka-na-1}}(\mathcal{A}') \geq \mathsf{Adv}^{\mathsf{CGKA},\mathsf{P}}_{\text{cgka-na}}(\mathcal{A})/t \ .$$

We use a standard hybrid argument. Let H_0 be identical to the cgka-na game with $b = 0$. For $i \in [1, t]$ let hybrid H_i be just as H_{i-1} except that on the i^{th} query to **chall** it responds as if $b = 1$. In particular, H_t is identical to cgka-na with $b = 1$. For all $i \in [0, t]$ let $h_i = \mathsf{Pr}[H_i(\mathcal{A}) \to 1]$ and let $\bar{h}_i = 1 - h_i$.

Suppose for a second that for any $i \in [0, t-1]$ we have that $|(h_{i+1} - h_i)/2| \leq \varepsilon$. This would prove the lemma because then we could write:

$$\mathsf{Adv}^{\mathsf{CGKA},\mathsf{P}}_{\text{cgka-na}}(\mathcal{A}) = \left|\frac{\bar{h}_0 + h_t}{2} - \frac{1}{2}\right| = \left|\frac{h_t - h_0}{2}\right| = \left|\sum_{i\in[0,t-1]} \frac{h_{i+1} - h_i}{2}\right|$$
$$\leq \sum_{i\in[0,t-1]} \left|\frac{h_{i+1} - h_i}{2}\right| \leq t\varepsilon \ .$$

It remains to prove the following claim.

Claim. For all $i \in [0, t-1]$ it holds that $\left|\frac{h_{i+1}-h_i}{2}\right| \leq \varepsilon$.

To prove the claim, we define reduction R_i for each i and show its advantage for game cgka-na-1 is at least $\left|\frac{h_{i+1}-h_i}{2}\right|$. By assumption CGKA is $(\mathsf{P}, t, \varepsilon)$-1-secure so this proves the claim (and lemma).

Reduction R_i expects to play the cgka-na-1 game while using (black-box) access to $\mathcal{A}$. To do this R_i forwards all oracle calls between $\mathcal{A}$ and cgka-na-1 and returns the response back to $\mathcal{A}$ except for **chall** queries. The i^{th} **chall** query is forwarded to the game faithfully and the response sent back to $\mathcal{A}$. However, for any other **chall** queries R_i simulates the answer to $\mathcal{A}$ as follows. First, it obtains the challenged epoch key $\mathbf{I}[t]$ by calling the **reveal** oracle. Next, it checks that the required condition in the **chall** oracle is satisfied. (The second clause is checked by keeping a local copy of the chall array and populating it as $\mathcal{A}$ makes **chall** and **reveal** queries.) If the condition's are not satisfied R_i ignores the query returning nothing to $\mathcal{A}$ (just as cgka-na would). Otherwise, if this is one of the first $i-1$ **chall** calls in the execution, then R_i returns $\mathbf{I}[t]$. Otherwise (when $\mathcal{A}$ has already made at least i previous **chall** calls) R_i samples and returns a random key from $\mathcal{K}$. Finally, it outputs the same guess for the bit b as made by $\mathcal{A}$.

It is also clear the run-time of R_i is polynomially related to that of $\mathcal{A}$. We also remark that if $\mathcal{A}$ is non-adaptive then so is R_i. Indeed, the sequence of oracle calls made by R_i depends only on the choice of calls made by $\mathcal{A}$. All R_i need do to decide on its calls is switch all but the i^{th} call to **chall** into a call to **reveal** (with the same argument).

Next, we observe that from $\mathcal{A}$'s point of view R_i interacting with cgka-na-1 with bit b behaves identically to hybrid H_{i+b}. Indeed, the only (non-syntactic) difference between an execution of $\mathcal{A}$ with H_{i+b} (for random b) and with R_i is that in the later case some calls to **chall** are replaced with **reveal** calls for the same epoch. In either case though, the same **req** condition is checked and the same change to chall is made. So the only difference in the view of $\mathcal{A}$ can arise at the end of the execution if the value of P is different. However, for each of $\mathsf{P} \in \{\mathbf{safe}, \mathbf{pcs}, \mathbf{fsu}\}$ its value can only change from false to true (if at all). This follows from the observation that the value of those predicates depends only on the sequence of oracle calls made and if we remove **chall** queries from any such sequence then the for loops check a subset of the original conditions. Moreover, those conditions are either unaffected or switch from true to false. Thus, overall the predicates will either remain unaffected or return 1 instead of 0. Finally, inserting **reveal** queries has no effect on any of the predicates.

In summery, we can now write:

$$\varepsilon \geq \mathsf{Adv}^{\mathsf{CGKA},\mathsf{P}}_{\text{cgka-na-1}}(\mathsf{R}_i) = \left|\mathsf{Pr}[\mathsf{R}_i \text{ wins}] - \frac{1}{2}\right| \geq \left|\frac{\bar{h}_i + h_{i+1}}{2} - \frac{1}{2}\right| = \left|\frac{h_{i+1} - h_i}{2}\right| \ .$$

□

C Growing and Truncating LBBTs

This section contains the deferred proofs for the lemmas dealing with LBBT growth and truncation.

Lemma 1. *If* $\tau = \mathsf{LBBT}_n$*, then* $\tau' = \mathsf{LBBT}_{n+1}$.

Proof. The statement is proved by strong induction on n. For the base case, $n = 1$, it is easy to verify that $\tau' = \mathsf{LBBT}_2$.

Assume now that $n > 2$ and the statement holds for all $k \leq n$. Consider the following two cases:

– If n is a power of 2, then τ' has FT_n as its left subtree and LBBT_1 as its right subtree, as required by the definition.

– For the case where n is not a power of 2, let $x = \mathsf{mp2}(n)$. Recall that by definition, τ has FT_x as the left subtree and a non-full LBBT_{n-x} as the right subtree. Procedure AddLeaf now recursively inserts the node into LBBT_{n-x}. By the inductive hypothesis, this results in LBBT_{n-x+1}, where $n - x + 1 \leq x$ since n is not a power of two. Hence, $\tau' = \mathsf{LBBT}_{n+1}$.

□

Lemma 2. *If $\tau = \mathsf{LBBT}_n$, then $\tau' = \mathsf{LBBT}_y$ for some $0 < y \leq n$. Furthermore, unless $y = 1$, the rightmost leaf of τ' is non-blank.*

Proof. The statement is proved by strong induction on n. If $n = 1$ then clearly $T' = \mathsf{LBBT}_1$, as a single node is never removed by Trunc.

Suppose now that $n > 2$ and that the lemma holds for all $k \leq n$, and consider $\tau = \mathsf{LBBT}_{n+1}$. Assume that the rightmost leaf node v of τ is blank as, otherwise, there is nothing to prove. Note that τ has FT_x as left subtree and LBBT_{n-x} as the right subtree, where $x = n/2$ when n is a power of 2 and $x = \mathsf{mp2}(n)$ otherwise. Irrespectively of which is the case, by the induction hypothesis, executing Trunc on v results in LBBT_y for some $0 < y < n - x$ just before Trunc is called on the (potentially new) root r_R of the right subtree. If r_R is non-blank, then the recursion ends here and the result is clearly an LBBT. If r_R is blank, the root r is removed, and the root of FT_x becomes the new root. Clearly, FT_x is an LBBT, and, hence, calling Trunc on FT_x's rightmost leaf results in an LBBT by the induction hypothesis. □

D Comparison To TreeKEM in MLSv7

The CGKA we analyze under the name TreeKEM in this work is mostly identical to the one described in Version 7 of the MLS protocol's RFC draft (i.e., currently the most recent version of the MLS specification) [3]. However, there are two differences which we describe and motivate here.

First, we augmented the protocol with the internal function Trunc used when removing parties from the tree. This helps prune blank leaves to a smaller (and potentially shallower) left-balanced binary tree. While not particularly interesting from a security perspective, this can result in meaningful efficiency improvements in sessions where many people have left a group of the course of its existence. Since the performance of TreeKEM is tightly correlated with the depth of the ratchet tree, pruning unused (i.e., blank) leaves potentially saves on computation and communication complexity.

The second difference is that add and remove operations now consist only of the tree manipulations (if needed) and blanking. In contrast, in MLSv7 the sender of add and remove packets also perform an update along their direct path. That means that in our version, after such an operation, the root is actually blank. Thus, we require that the higher level protocol (e.g., the SGM) ensures that before any new key material is needed (e.g., to encrypt/decrypt a new message), some group member first performs an update to repopulate the root node of the ratchet tree. These changes can make the CGKA protocol more efficient, and, yet, the proofs and attacks in our work carry over to the original.

In more detail, on the one hand, our variant more efficiently supports add/remove batches of users to an existing group. In the original version of TreeKEM both add and remove operations apply to a single user only. Moreover all such operations require every group member to download several ciphertexts and perform some expensive cryptographic operations. However, if the higher-level application makes no use of the new (root) key material before the next add/remove operation, then much of the just derived key material will immediately be overwritten without ever having been used. So, by deferring all such cryptographic operations to the next update, such needless communication and computation is avoided. On the other hand, it is also easy to recover the MLSv7 variant of TreeKEM from the one in this work by simply having the sender of add or remove operations also immediately perform an update. Thus,, the security statements proved in this work carry over to the original. Finally, note that the problems with forward secrecy discussed in this work apply just as much to MLSv7 (since these issues also occur without having adds and removes).

E Relationships among Predicates

This section contains the deferred proof for Lemma 3, which deals with the relationships among the **tkm**, **pcs**, and **fsu** predicates.

Lemma 3. *For any sequence of queries Q, if* $\mathbf{pcs}(Q) = 1$ *or* $\mathbf{fsu}(Q) = 1$*, then* $\mathbf{tkm}(Q) = 1$.

Proof. Assume $\mathbf{pcs}(Q) = 1$. It is not hard to see that the lemma holds since, clearly, if $\mathbf{pcs}(Q) = 1$, then there are no corruptions in epochs after t^*, and, thus, $\mathbf{pcs}(Q) = \mathbf{safe}(Q) = 1$.

For the FSU predicate, we prove the contrapositive, i.e., for any $Q = (\mathbf{q}_1, \ldots, \mathbf{q}_q)$, if $\mathbf{tkm}(Q) = 0$, then $\mathbf{fsu}(Q) = 0$. Since $\mathbf{tkm}(Q) = 0$ we have that for some i, $\mathbf{q}_i = \mathbf{corr}(\mathsf{ID})$, ID is corrupted when being in epoch $t = \mathsf{q2e}(\mathbf{q}_i)$ and the challenged key, $I^* := \mathbf{I}[t^*]$, belongs to K^t_{ID}. If a corruption happens after epoch t^* and no update is issued by the user on the interval $(t^*, t]$, then **fsu** outputs 0. Assume, there is no such corruption. Since $I^* \in K^t_{\mathsf{ID}}$, we consider two cases: (i) I^* is part of the private state of ID when ID is corrupted, and (ii) there exists a key sk and a path from sk to I^* in the key graph with respect to Q.

Case (i): First assume that $t > t^*$. Then we have that ID is corrupted after the challenge epoch, still I^* is in the private state of ID, which implies that there exists k such that $\mathbf{q}_k = \mathbf{no\text{-}del}(\mathsf{ID})$, $\mathsf{q2e}(\mathbf{q}_k) \leq t^*$, thus $\mathbf{fsu}(Q) = 0$. If $t = t^*$, then clearly no **send-update**(ID) or **remove-user**(ID) query is processed by the users in between the corruption and the challenge query and $\mathbf{fsu}(Q)$ outputs 0 (note that if $t < t^*$, I^* does not belong to the private state of ID when it is corrupted).

Case (ii)**,** $t \leq t^*$: For case (ii) and $t \leq t^*$, we have that there exists a key sk in the private state of ID at epoch t, such that I^* is reachable from sk in the key graph with respect to Q. Let $t \leq t^*$. Towards contradiction, assume there exists k and query $\mathbf{q}_k = \mathbf{send\text{-}update}(\mathsf{ID})$, such that $t < \mathsf{q2e}(\mathbf{q}_k) \leq t^*$. This operation generates a set of private keys, say K, which by the protocol specification, they are independent of sk and any other key in the private state of ID in epoch t; clearly, there can be no path from sk to any key in K in the key graph. In addition, all subsequent update operations by other users, when encrypting for user ID, they encrypt under some public key for which the secret key is in K. Thus, in order to for I^* to be reachable from sk, it requires some element in K to be reachable by sk, reaching a contradiction.

Similarly we prove that there can be no k, such that $\mathbf{q}_k = \mathbf{remove\text{-}user}(*, \mathsf{ID})$ and $t < \mathsf{q2e}(\mathbf{q}_k) \leq t^*$. Towards contradiction assume such k exists. Then, due to the blanking performed by the protocol when removing users, the remaining group members will erase all public and secret keys in the direct path of ID, thus all subsequent update operations will encrypt under public keys for which the secret keys are not known to ID. Thus, I^* cannot be reachable by sk if ID is removed from the group. The above imply that $\mathbf{fsu}(Q) = 0$ for case (ii) and $t \leq t^*$.

Case (ii)**,** $t > t^*$: This case is similar to the above. If a user is corrupted in epoch $t > t^*$ and there exists a key sk in the private state of ID at epoch t, such that I^* is reachable from sk, then the user has not issued an update before corruption: if there was an update before corruption then reachability between sk and I^* wouldn't hold (the argument is identical to the above).

□

F TreeKEM: Deferred Security Proofs

F.1 Proof of Non-Adaptive Security for TreeKEM

Theorem 1 (Non-adaptive security of TreeKEM). *Assume that*

- prg *is a* $(t_{\mathsf{prg}}, \varepsilon_{\mathsf{prg}})$*-secure pseudo-random generator,*
- Π *is a* $(t_{\mathsf{cpa}}, \varepsilon_{\mathsf{cpa}})$*-CPA-secure public-key encryption scheme,*

Then, TreeKEM is a $(t, c, n, \mathsf{P}, \varepsilon)$*-secure CGKA protocol, for* $\mathsf{P} \in \{\mathbf{tkm}, \mathbf{pcs}, \mathbf{fsu}\}$*,* $\varepsilon = 2cn(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}})$*, and* $t \approx t_{\mathsf{prg}} \approx t_{\mathsf{cpa}}$*.*

Proof. The theorem is proved w.r.t. **tkm** and by considering an attacker $\mathcal{A}$ that makes only a single challenge query. The final result is obtained by applying Lemmas 6 and 3.

Hybrid H_i^x, $i \in [d]$, $x \in \{\mathsf{c}, \mathsf{p}\}$:

1. *Initialization*: Execute **init**, **create-group**(G) and set $S[\cdot] \leftarrow \epsilon$. If $x = \mathsf{p}$, define $H := H_i^{\mathsf{c}}$, otherwise, $H := H_{i+1}^{\mathsf{p}}$.[a]
2. *Protocol execution*: Compute $(\mathbf{q}_1, \ldots, \mathbf{q}_q) \leftarrow \mathcal{A}$. For $r \in [q]$:
   (a) If $\mathbf{q}_r$ is a query for the oracles **corr**, **chall**, **add-user**, **remove-user**, **no-del**, process it as in H.
   (b) If $\mathbf{q}_r =$ **send-update** (ID), then, if the query is not processed by any user, process it as in H. Otherwise, let t be the epoch defined by such a query a. If for all $w \in [l]$, $t \neq t_w$, process the queries as in H. Otherwise, let d' be the depth of the leaf in τ^*, for which $t_w = t \notin \{\perp, 0\}$, for some $w \in [l]$. Compute:
      i. Let $X := \{x \mid x \geq i,\ t_{x,j} = t,\ j \in [2^x]\}$.
         – If $X = \emptyset$ then process everything as in H and go the next query.
         – Otherwise, let $i' \leftarrow \min_x X$ and let j' be such that $t_{i',j'} = t$ for $j' \in \left[2^{i'}\right]$.
            • If $x = \mathsf{p}$, $(i^*, j^*) \leftarrow (i', j')$.
            • Otherwise, if $i' > i$, $i^* \leftarrow i'$, otherwise $i^* \leftarrow i + 1$.

         Set $\mathsf{d} \leftarrow d - d' + 1$.
      ii. (**PRG to uniform**): Consider a modified version of the UpGen operation, UpGen$'$, that computes the path secrets as follows:
$$\$ \rightarrow (\mathsf{sk}_0, s_1), \$ \rightarrow \ldots, \$ \rightarrow (\mathsf{sk}_{\mathsf{d}-i^*}, s_{\mathsf{d}-i^*+1}) \xrightarrow{\mathsf{prg}} \ldots \xrightarrow{\mathsf{prg}} (\mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}})^b$$
      iii. (**Faking ciphertexts**):
         – Set $s_1^*, \ldots, s_{\mathsf{d}-i^*+1}^* \leftarrow \mathsf{0}$.
         – If $x = \mathsf{p}$, publish encryptions of $s_1^*, \ldots, s_{\mathsf{d}-i^*}^*, s_{\mathsf{d}-i^*+1}, \ldots, s_{\mathsf{d}}$.
         – If $x = \mathsf{c}$, publish encryptions of $s_1^*, \ldots, s_{\mathsf{d}-i^*+1}^*, s_{\mathsf{d}-i^*+2}, \ldots, s_{\mathsf{d}}$.
         – Set $S[\mathsf{ID}] \leftarrow (\mathsf{sk}_1, \ldots, \mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}})$.
   (c) If $\mathbf{q}_r =$ **deliver**$(t, \mathsf{ID}, \mathsf{ID}', c)$:
      – If for all $w \in [l]$, $t \neq t_w$, execute **deliver** normally.
      – Otherwise, $(\mathsf{sk}_1, \ldots, \mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}}) \leftarrow S[\mathsf{ID}]$. Let $v = \mathsf{LCA}(\mathsf{ID}, \mathsf{ID}')$ and let d'' be its depth. Set the keys from node v to the root to $\mathsf{sk}_{d'-d''}, \ldots, \mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}}$, for ID'.
3. **Output**: If $\mathbf{tkm}(\mathbf{q}_1, \ldots, \mathbf{q}_q) = 0$, return 0, otherwise, if $b = b'$ or if the **win** condition is triggered, return 1, otherwise return 0.

[a] Here we define the hybrid before H_i^x depending on the value of x.

[b] Here the notation $\$ \rightarrow (x, y)$ denotes the fact that (x,y) is uniformlly random, while $(x, y) \xrightarrow{\mathsf{prg}} (x', y')$ denotes the fact that (x', y') is the output of the PRG on input seed y.

Fig. 11. Definition of hybrid experiments in the TreeKEM security proof.

Let τ^* be the ratchet tree consisting of the labels of all keys generated by the latest updates that have been processed by at least one user up to (and including) the challenge epoch t^*, and let d be its depth. We will use the notation $\mathsf{sk} \in \tau^*$ (resp. $\mathsf{sk} \notin \tau^*$) to denote the fact that sk is (resp. is not) part of the label of a node in τ^*.

We track the epochs in which the keys in every ratchet tree node of τ^* are created (as shown in the example of Figure 5): Let t_i be the epoch in which the key on the i-th leaf (from left to right) was defined in τ^*. If there is no user in the i-th leaf we set $t_i \leftarrow -\infty$, and for keys that were generated during initialization and present in epoch t^*, we set $t_i \leftarrow 0$. Then, for the j^{th} node at depth i, we define the value $t_{i,j}$ as follows: If the node is a leaf, we set we set $t_{i,j} = t_j$; otherwise, we define $t_{i,j}$ to be the maximum of the values of its children, i.e., $t_{i,j} := \max\{t_{i+1,2j-1}, t_{i+1,2j}\}$.

Following the intuition laid out in Section 5.3, consider hybrid experiments

$$H_d^{\mathsf{c}}, H_d^{\mathsf{p}}, H_{d-1}^{\mathsf{c}}, H_{d-1}^{\mathsf{p}}, \ldots, H_1^{\mathsf{p}}, H_0^{\mathsf{c}} \ ,$$

where H_d^{c} is the original CGKA game and where the remaining hybrids are defined in Figure F.1. Note that the difference on the above hybrids comes from faking either a sequence of PRGs or a sequence of encryptions under the CPA-secure PKE scheme. A pictorial description of the hybrids can be found in Figure 12.

Before proving indistinguishability between the hybrids we prove a useful lemma that relates τ^* to TreeKEM's key graph.

Lemma 7. *Let $(\mathcal{V}, \mathcal{E}) \leftarrow \mathsf{KG}(\mathbf{q}_1, \ldots, \mathbf{q}_q)$ be the key graph with respect to the adversarial queries $(\mathbf{q}_1, \ldots, \mathbf{q}_q)$. For every $\mathsf{sk} \in \mathcal{V}$, sk' in τ^*, if $\mathsf{sk} \notin \tau^*$ then there is no path from sk to sk' in $(\mathcal{V}, \mathcal{E})$.*

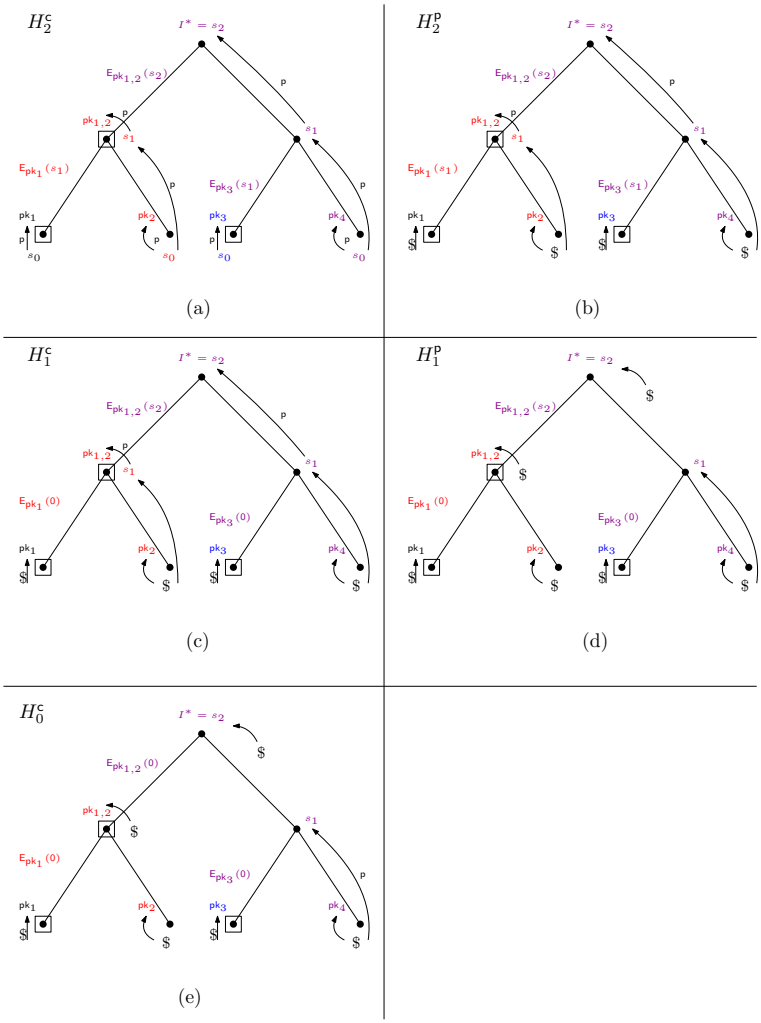

Fig. 12. In this figure we depict the sequence of hybrids for $n = 4$ users with ids 1,2,3,4, assuming they have issued updates from the left to the right. For each public key, the subscript denotes the user id(s) in ranges, that know the corresponding secret key. The first hybrid, H_2^c, is the original CGKA security game in which all values are generated via the PRG (this is denoted by arrows labeled by "p") and the ciphertexts encrypt the actual path secrets. In hybrid H_2^p we substitute all the values generated by the application of the PRG to the values s_0 (s_0 is uniform and independent for each user), with uniformly random values (this is denoted by arrows labeled by "\$"). In hybrid H_1^c all ciphertexts that encrypt path secrets under the public keys of the nodes at depth 2, are substituted by ciphertexts encrypting the zero message. The remaining hybrids are similar.

Proof. Towards contradiction assume $\mathsf{sk} \in \mathcal{V}$, $\mathsf{sk} \notin \tau^*$, but there exists $\mathsf{sk}' \in \tau^*$ and a path from sk to sk' in $(\mathcal{V}, \mathcal{E})$. Clearly, a path from sk to sk' implies that sk was generated via an update operation before the challenge epoch and $\mathsf{sk} \notin \tau^*$ implies that sk was overwritten/erased by some protocol operation. We consider two cases. First, assume sk was overwritten by an update operation issued by the leaf node v and let v' be the node with secret key sk' in τ^*. By the protocol definition and the fact that sk' is recoverable from sk we have that v' lies in the direct path of node v, thus the update operation issued at node v should have also *overwritten* sk', implying that $\mathsf{sk}' \notin \tau^*$, reaching a contradiction.

For the second case, sk could have been blanked after adding or removing a user from/to the group at a leaf node, v. In that situation, all keys in the direct path of v are blanked, and by the protocol specification, those are the only keys recoverable by sk. Thus sk' cannot be such a key and the proof of the lemma is complete. □

In the following, we denote by nb_i the number non-blank nodes at depth i of τ^* and for $j \in [\mathsf{nb}_i]$ and a non-blank node v_j^i we set c_j^i to be the number of ciphertexts generated using public keys below depth i encrypting the path secret of v_j^i. Next, we prove indistinguishability between the hybrids:

Lemma 8. *For $i \in [d]$,*

1. $\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{c}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{p}}}(\mathcal{A}) + 2^i \cdot \varepsilon_{\mathsf{prg}}$.
2. $\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{p}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_{i-1}^{\mathsf{c}}}(\mathcal{A}) + \sum_{j=1}^{\mathsf{nb}_i} \mathsf{c}_j^i \cdot \varepsilon_{\mathsf{cpa}}$.

Proof. The proof is by contradiction. Let i be the maximum value in $[d]$ for which either of the above relations does not hold and in the sequence of hybrids $H_d^{\mathsf{c}}, H_d^{\mathsf{p}}, \ldots, H_1^{\mathsf{p}}, H_0^{\mathsf{c}}$, at least two adjacent hybrids are distinguishable. For all $j > i$ we assume both relations hold. We consider two cases:

(a) For the first case, assuming relation (*1.*) does not hold for the index i (but relation (*2.*) holds), we make a reduction to the security of the PRG with at most 2^i independent instances. Here 2^i is an upper bound on the number of PRG hybrids at depth i of τ^*. The actual number of faked PRGs at depth i is equal to the number of nodes (i, j) for which $t_{i,j} \notin \{0, -\infty\}$, i.e., to the number non-blank nodes at depth i, which we denote by nb_i. We define the adversary B_i^{p} for the PRG game as follows:

Algorithm B_i^{p}:

1. Query the oracle of the PRG experiment and receive the challenge $((\mathsf{sk}_1, s_1), \ldots, (\mathsf{sk}_{\mathsf{nb}_i}, s_{\mathsf{nb}_i}))$.
2. Let $t'_1, \ldots, t'_{\mathsf{nb}_i}$, be the epochs in which the secrets for the non-blank nodes at depth i are defined. Simulate H_i^{c} with the only difference that if the update query defines the epoch $t'_k = t_{i,j}$, for some $j \in [2^i]$, $k \in [\mathsf{nb}_i]$, set $(\mathsf{sk}_{\mathsf{d}-i}, s_{\mathsf{d}-i+1}) \leftarrow (\mathsf{sk}_k, s_k)$.[a]
3. For the rest of the execution simulate H_i^{c} and output the bit output by $\mathcal{A}$.

[a] Here we consider nb_i independent update operations.

The difference between H_i^{c} and H_i^{p}, is that the latter fakes one more level of PRGs for the update queries of the epochs t'_k, $k \in [\mathsf{nb}_i]$. In particular, in H_i^{c} at epoch $t'_k \notin \{0, -\infty\}$, we have $i^* = i + 1$, while in H_i^{p}, $i^* = i$, i.e., we substitute nb_i more PRG values with uniformly random and independent values. Now observe that if the challenge of the PRG game, $((\mathsf{sk}_1, s_1), \ldots, (\mathsf{sk}_{\mathsf{nb}_i}, s_{\mathsf{nb}_i}))$, consists of uniformly random values, by the definition of B_i^{p}, the update for the epoch t'_k is performed as follows:

$$\$ \to (\mathsf{sk}_0, s_1), \ldots, \$ \to (\mathsf{sk}_{\mathsf{d}-i-1}, s_{\mathsf{d}-i}), \$ \to (\mathsf{sk}_{\mathsf{d}-i}, s_{\mathsf{d}-i+1}) \xrightarrow{\mathsf{prg}} \ldots$$

which matches the execution in H_i^{p}. On the other hand if the challenge is the output of the PRG the computation, we have that

$$\$ \to (\mathsf{sk}_0, s_1), \ldots, \$ \to (\mathsf{sk}_{\mathsf{d}-i-1}, s_{\mathsf{d}-i}), \xrightarrow{\mathsf{prg}} (\mathsf{sk}_{\mathsf{d}-i}, s_{\mathsf{d}-i+1}) \xrightarrow{\mathsf{prg}} \ldots$$

and in that case B_i^{p} simulates H_i^{c}.

Now we formally argue that the simulation described above is correct. By assumption we have that the hybrids $H_d^{\mathsf{c}}, H_d^{\mathsf{p}}, \ldots, H_i^{\mathsf{c}}$ are computationally indistinguishable and we need to prove that $H_i^{\mathsf{c}} \approx H_i^{\mathsf{p}}$. Observe that the only way for the adversary to distinguish between between the two hybrids, is by distinguishing a uniformly random $(\mathsf{sk}_{\mathsf{d}-i}, s_{\mathsf{d}-i+1})$ from one that is output by the RPG on input $s_{\mathsf{d}-i}$. Since $\mathcal{A}$ is an admissible adversary we have that $\mathbf{tkm}(Q_1, \ldots, Q_q) = 1$, thus the adversary is not allowed to compromise any user whose state contains at least one key that enables the recovery of the update secret in the challenge epoch. Since all group members should be able to compute the update secret for the challenge epoch, this implies that no key of τ^* is ever leaked to the adversary and this also holds for the corresponding path secrets $s_1, \ldots, s_d$. Also, from Lemma 7 we have that no key in the key graph of $(\mathbf{q}_1, \ldots, \mathbf{q}_q)$ that does not belong to τ^*, enables the recovery of any key in τ^*. We conclude that the above simulation is correct, without B_i^{p} accessing the seed of the PRG game. Hence, if $\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{c}}}(\mathcal{A}) > \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{p}}}(\mathcal{A}) + \mathsf{nb}_i \cdot \varepsilon_{\mathsf{prg}}$, B_i^{p} breaks the security of the PRG in this multi-instance PRG game, reaching a contradiction. We conclude that

$$\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{c}}}(\mathcal{A}) < \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{p}}}(\mathcal{A}) + \mathsf{nb}_i \cdot \varepsilon_{\mathsf{prg}} \leq \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}_i^{\mathsf{p}}}(\mathcal{A}) + 2^i \varepsilon_{\mathsf{prg}}.$$

(b) For the second case we make a reduction to the CPA security of the encryption scheme. Let v_j, $j \in [\mathsf{nb}_i]$, be the non-blank nodes at depth i. For each v_j, let $\bar{v}_1^j, \ldots, \bar{v}_{k_j}^j$, be the descendants of v_j that under their public keys the path secret of v_j was encrypted, and let $\bar{t}_{l,j}$, be the epoch in which the keys of $\bar{v}_l^j$ were introduced. Also, let $\hat{t}_{l,j}$ be the epoch in which the path secret of v_j is encrypted under the public key of $\bar{v}_l^j$.

We define the adversary B_i^{c} for the CPA game as follows:

Algorithm B_i^{c}:

1. For $j \in [\mathsf{nb}_i]$, receive k_j public keys $(\mathsf{pk}_{1,j}^*, \ldots, \mathsf{pk}_{k_j,j}^*)$ from the challenger of the CPA game.
2. Execute H_{i+1}^{p} with the following differences:
   (a) For $j \in [\mathsf{nb}_i]$, $l \in [k_j]$, in the update for the epoch $\bar{t}_{l,j}$, set the public key of the node $\bar{v}_l^j$ to $\mathsf{pk}_{l,j}^*$ and the secret key to ϵ. If $\bar{t}_{l,j} = 0$, introduce $\mathsf{pk}_{l,j}^*$ during the **create-group**(G) operation.
   (b) For $j \in [\mathsf{nb}_i]$, $l \in [k_j]$, set $M_0^{l,j} \leftarrow s_{\mathsf{d}-i+1}$ (here $s_{\mathsf{d}-i+1}$ is as it is computed by H_{i+1}^{p}) and $M_1^{l,j} \leftarrow \mathsf{0}$.[a]
   (c) Set $\mathbf{M}_0$ (resp. $\mathbf{M}_1$) to be the concatenation of $M_0^{l,j}$ (resp. $M_1^{l,j}$). Send $\mathbf{M}_0$, $\mathbf{M}_1$, to the CPA challenger, and receive the challenge ciphertexts $c_{l,j}^*$, $j \in [\mathsf{nb}_i]$, $l \in [k_j]$.
   (d) For $j \in [\mathsf{nb}_i]$, $l \in [k_j]$, in the update for the epoch $\hat{t}_{l,j}$, publish encryptions of $s_1, \ldots, s_{\mathsf{d}-i} \leftarrow \mathsf{0}$, and $s_{\mathsf{d}-i+2}, \ldots, s_{\mathsf{d}}$ (as they are computed by H_{i+1}^{p}), and for the index $\mathsf{d}-i+1$ publish $c_{l,j}^*$.
3. Output the bit output by $\mathcal{A}$.

[a] Note that for distinct (l, j), $s_{\mathsf{d}-i+1}$ are independent since we refer to distinct epochs.

The hybrids H_{i+1}^{p} and H_i^{c} are only different for the **send-update** queries for the epochs $\hat{t}_{l,j}$. In H_{i+1}^{p} and in epoch $\hat{t}_{l,j}$ we have $i^* = i+1$, while in H_i^{c}, $i^* = i$, i.e., we fake at most $n-1$ more independent ciphertexts. H_{i+1}^{p} in epoch $\hat{t}_{l,j}$ publishes encryptions of $s_1, \ldots, s_{\mathsf{d}-i} \leftarrow \mathsf{0}$, $s_{\mathsf{d}-i+1}, \ldots, s_{\mathsf{d}}$ while, H_i^{c} publishes encryptions of $s_1, \ldots, s_{\mathsf{d}-i+1} \leftarrow \mathsf{0}$, $s_{\mathsf{d}-i+2}, \ldots, s_{\mathsf{d}}$. Clearly, if for all (l, j) $c_{l,j}^*$ encrypts $M_0^{l,j} = s_{\mathsf{d}-i+1}$, B_i^{c} simulates H_{i+1}^{p}, and if $c_{l,j}^*$ encrypts $M_1^{l,j} = \mathsf{0}$, B_i^{c} simulates H_i^{c}.

Now we formally argue that the simulation described above is correct. As in case 1, since $\mathcal{A}$ is an admissible adversary we have that $\mathbf{tkm}(\mathbf{q}_1, \ldots, \mathbf{q}_q) = 1$, thus no key of τ^* is ever leaked to the adversary and by Lemma 7, there is no compromised key in the key graph that leads to any key in τ^*. Consequently, setting the public key, $\mathsf{pk}_{l,j}^*$, of $\bar{v}_l^j$ without knowing the corresponding secret key is not an issue as there will be no corruptions under $\bar{v}_l^j$, before at least one honest update overwrites its keys. In addition, despite the fact that B_i^{c} is sending encryptions of $\mathsf{0}$, the correct keys for all nodes of the ratchet tree as well as the update secret are hardcoded to the private states of users on message delivery.

At depth i the ratchet tree has nb_i non-blank nodes and for the j-th node we need to fake c_j^i ciphertexts, thus we have $\sum_{j=1}^{\mathsf{nb}_i} \mathsf{c}_j^i$ ciphertexts in total and the proof is concluded. □

Total security loss due to the CPA hybrids. We upper bound the quantity $\mathsf{t}_{\mathsf{cpa}} := \sum_{i=1}^{d-1} \sum_{j=1}^{\mathsf{nb}_i} \mathsf{c}_j^i$,[12] which gives us the total security loss due to the CPA hybrids. For starters, we consider the case of static groups with n users, i.e., there are no blank nodes and the ratchet tree is of depth $\log n$. In this case it is not hard to see that at depth i we have $\mathsf{nb}_i = 2^i$ non-blank nodes, where for each one of them its path secret is encrypted under a single public key from depth $i+1$, i.e., $\mathsf{c}_j^i = 1$ for $j \in \mathsf{nb}_i$ (cf. Figure 13). Thus, $\mathsf{t}_{\mathsf{cpa}} = \sum_{i=0}^{d-1} 2^i = 2^d - 1 = n - 1$.

For the general case in which blank nodes might exist, it is not hard to see that for every node v in τ^*, the protocol generates at most one ciphertext that encrypts a path secret s under pk_v, and s allows the recovery of keys in τ^* (e.g., in Figure 13 each node can have at most one incoming dashed edge). Towards contradiction assume that for some node v at depth d_v, the protocol computes $\mathsf{Enc}(\mathsf{pk}_v, s)$ and $\mathsf{Enc}(\mathsf{pk}_v, s')$, and s, s' are the path secrets of nodes u, u', in depths d_u, $d_{u'}$. Clearly, $d_v > d_u, d_{u'}$ and the non-blank nodes u, u' are along the direct path of v. If $d_u > d_{u'}$, we have that the keys of node u where introduced before the keys of node u'. This implies that during the update operation that defines the keys of u' in τ^*, the protocol should

[12] The outer sum goes up to $d-1$ because H_d^{c} is the original game.

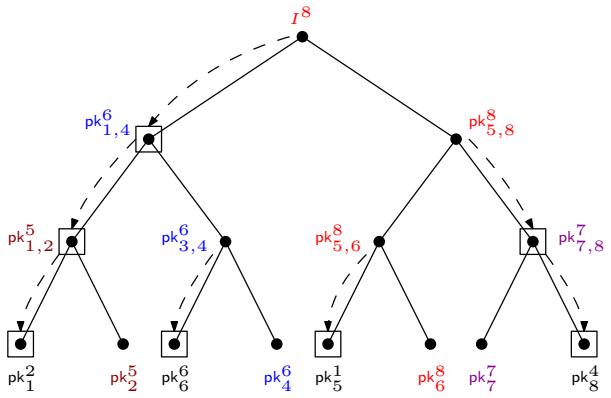

Fig. 13. An example of the ratchet tree with $n = 8$ users performing updates that define 8 epochs. The order in which the 8 ids issue the updates is $(5, 1, 3, 8, 2, 4, 7, 6)$, i.e., the first update is by id 5, the second by id 1 and so forth. A dashed arrow from a node u to a node v denotes the fact that due to an update operation the path secret of node u is encrypted under the public key of node v.

have encrypted s' under pk_u contradicting the fact that s' is encrypted under pk_v (a similar argument holds if $d_u < d_{u'}$). Thus, we can have at most one ciphertext for each node in τ^* (besides the root), which gives us a bound of at most $2n$ CPAs.[13]

Total security loss due to the PRG hybrids. By lemma 8 we have that at depth i the security loss between H^{p}_i and the previous hybrid is at most $2^i\varepsilon_{\mathsf{prg}}$. Thus the total security loss for all PRG hybrids is $\varepsilon_{\mathsf{prg}} \cdot \sum_{i=1}^{d} 2^i = \varepsilon_{\mathsf{prg}}(2^{d+1} - 2) \leq 2n\varepsilon_{\mathsf{prg}}$.

Protocol correctness. It is not hard to see that protocol correctness holds. The adversary is not allowed to modify messages transmitted over the network and the oracles of the CGKA security game (cf. Figure 1) do not allow group splitting: all users move to epoch t after an update operation by processing part of the same message, encrypting path secrets s_i that enable the computation of the update secret consistently. The same argument holds for every epoch defined by an add/remove user operation as users process the same message. This implies that users have a consistent view of group membership.

Putting things together. The final hybrid is H^{c}_0. In that hybrid, the update secret I^* is uniform and all information published in the network is independent of it, as all ciphertexts encrypting information related to the update secret has been substituted with ciphertexts encrypting the zero message. Thus, $\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}^{\mathsf{c}}_0}(\mathcal{A}) = 1/2$ and given the above we derive,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{cgka-na}}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}^{\mathsf{c}}_{\mathrm{d}}}(\mathcal{A}) &\leq \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}^{\mathsf{c}}_0}(\mathcal{A}) + 2n\varepsilon_{\mathsf{cpa}} + (2n-2)\varepsilon_{\mathsf{prg}} \\ &\leq \frac{1}{2} + 2n(\varepsilon_{\mathsf{cpa}} + \varepsilon_{\mathsf{prg}}).\end{aligned}$$

□

F.2 Adaptive Security for TreeKEM

As for many cryptographic primitives, proving *adaptive security* for the TreeKEM protocol is quite challenging, since it requires guessing of the adaptive choices to be made by the adversary during the protocol execution. In this section we argue that TreeKEM is adaptively secure using the framework of [14], which establishes a generic connection between adaptive security proofs for the *Generalized Selective Decryption* (GSD) problem [20] and *graph pebbling*.

[13] Note that the bound of $n - 1$ CPAs also holds for the general case, still it's more intuitive to formally argue for the bound $2n$.

GSD and graph pebbling. Let $(\mathsf{Enc}, \mathsf{Dec})$ be a CPA-secure symmetric encryption scheme. In the GSD game the challenger initially samples n uniformly random keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$, and during the game execution the adversary is allowed to make three types of queries: (i) encryption queries of the form $(\mathbf{enc}, i, j)$, after which it receives $\mathsf{Enc}(\mathsf{sk}_i, \mathsf{sk}_j)$, (ii) corruption queries of the form $(\mathbf{cor}, i)$ in which the challenger reveals sk_i, and (iii) a single challenge query $(\mathbf{chall}, i)$, in which the adversary receives a real-or-random challenge $\mathsf{sk}^* := \mathsf{sk}_i$. As the game proceeds, the queries made by the adversary build a directed graph over the set of vertices $\{1, \ldots, n\}$ (i.e., we have one vertex for each key) as for every query $(\mathbf{enc}, i, j)$ a directed edge (i, j) is added to the graph. We say that the key i is corrupted if there exists a query $(\mathbf{cor}, i)$, or i is reachable in the GSD graph from an already corrupted vertex (key). The goal is to prove that the adversary cannot distinguish if the challenge key sk^* is the real one, or uniformly random and independent, with the restriction that there are no cycles in the constructed graph, and the challenge vertex is a non-corrupted sink.

In [14] the authors relate adaptive proofs of security for GSD, with the problem of finding "good" (see below) graph pebbling strategies over the GSD graph. In particular, a pebble on the edge (i, j) means that instead of answering the query $(\mathbf{enc}, i, j)$ with the "real" value $\mathsf{Enc}(\mathsf{sk}_i, \mathsf{sk}_j)$, we answer it with a "fake" value $\mathsf{Enc}(\mathsf{sk}_i, r)$, for a uniformly random r. The goal is to move from a hybrid with no pebbles, to one with pebbles on all the incoming edges of the sink node (the challenged key), while complying with the following *pebbling rule*: one can put/remove a pebble on the edge (i, j) only if all incoming edges to node i are pebbled. The goal here is not only to pebble the graph, but also to minimize the maximum number of pebbles, p, used at any stage of the pebbling process, since the security loss (due to guessing) is proportional to n^p.

In [14] (cf. Corollary 2) the authors show that for DAGs of depth d, the security loss is $O(n^{8d})$.

The GSD game w.r.t. TreeKEM. The result of [14] seems to almost immediately apply to our setting, as the TreeKEM protocol constructs DAGs of logarithmic depth.[14] However, there are two points that need to be addressed before deriving adaptive security for TreeKEM:

1. We are in the public key setting.
2. Some keys might be computable by other keys via a sequence of one or more PRG operations (this relates to the way the update operation works in TreeKEM).

Addressing point 1 is straightforward by adapting the interpretation of an encryption edge to the PKE setting: an encryption edge (i, j) in our setting means that the key sk_j is encrypted under the public key pk_i, where pk_i is the public key that corresponds to secret key sk_i, generated by executing $\mathsf{pk}_i \leftarrow \mathsf{PKEG}(\mathsf{sk}_i)$. To address point 2, we introduce a second type of edges, namely PRG edges: a PRG edge (i, j) means that sk_j is in the output of $\mathsf{prg}(\mathsf{sk}_i)$;[15] Encryption, corruption and challenge queries work as in the original GSD game.

We now show that the framework of [14] can be easily extended to our setting. In particular, the notion of placing a pebble to an edge (i, j) is interpreted as follows: if (i, j) is an encryption edge then instead of answering the query $(\mathbf{enc}, i, j)$ with $\mathsf{Enc}(\mathsf{pk}_i, \mathsf{sk}_j)$, we answer it with $\mathsf{Enc}(\mathsf{pk}_i, r)$, for uniformly random r; if (i, j) is a PRG edge, then we replace the output of the PRG, sk_j, with a uniformly random value. The key observation here is that placing a pebble (or faking) an edge corresponds to either a PRG or a CPA reduction, and in order to place a pebble on edge (i, j) we require all incoming edges to node i to be already pebbled. This rule makes sk_i independent of any other information, enabling a reduction to IND-CPA security.

From the above and Corollary of 2 of [14] we derive that TreeKEM is adaptively secure with a security loss factor of $O(n^{\log n})$.

F.3 Proof of Non-Adaptive Security for Modified TreeKEM

Theorem 2 (Non-adaptive security of Modified TreeKEM). *Assume that*

- prg *is a* $(t_{\mathsf{prg}}, \varepsilon_{\mathsf{prg}})$*-secure pseudo-random generator,*
- Π *is a* $(t_{\mathsf{cpa}}, \varepsilon_{\mathsf{cpa}})$*-CPA-secure updatable public-key encryption scheme.*

Then, the protocol of Section 6.3 is a $(t, c, n, \mathbf{safe}, \varepsilon)$*-secure CGKA protocol, for* $\varepsilon = 2cn(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}})$ *and* $t \approx t_{\mathsf{prg}} \approx t_{\mathsf{cpa}}$.

[14] The update operation generates $\log n$ fresh keys (secrets) and the secret at depth i is encrypted under keys that lie in depth $j > i$.

[15] Assume $(\mathsf{sk}'_i, \mathsf{sk}''_i) \leftarrow \mathsf{prg}(\mathsf{sk}_i)$ as in TreeKEM.

Hybrid H_i^x, $i \in [d]$, $x \in \{\mathsf{c}, \mathsf{p}\}$:

1. *Initialization*: Execute **init**, **create-group**(G) and set $S[\cdot] \leftarrow \epsilon$. If $x = \mathsf{p}$, define $H := H_i^{\mathsf{c}}$, otherwise, $H := H_{i+1}^{\mathsf{p}}$.[a]
2. *Protocol execution*: Compute $(\mathbf{q}_1, \ldots, \mathbf{q}_q) \leftarrow \mathcal{A}$. For $r \in [q]$:
   (a) If $\mathbf{q}_r$ is a query for the oracles **corr**, **chall**, **add-user**, **remove-user**, **no-del**, process it as in H.
   (b) If $\mathbf{q}_r =$ **send-update** (ID), then, if the query is not processed by any user, process it as in H. Otherwise, let t be the epoch defined by such a query a. If for all $w \in [l]$, $t \neq t_w$, process the queries as in H. Otherwise, let d' be the depth of the leaf in τ^*, for which $t_w = t \notin \{\perp, 0\}$, for some $w \in [l]$. Compute:
      i. Let $X := \{x \mid x \geq i,\ t_{x,j} = t,\ j \in [2^x]\}$.
         – If $X = \emptyset$ then process everything as in H and go the next query.
         – Otherwise, let $i' \leftarrow \min_x X$ and let j' be such that $t_{i',j'} = t$ for $j' \in \left[2^{i'}\right]$.
            • If $x = \mathsf{p}$, $(i^*, j^*) \leftarrow (i', j')$.
            • Otherwise, if $i' > i$, $i^* \leftarrow i'$, otherwise $i^* \leftarrow i + 1$.

         Set $\mathsf{d} \leftarrow d - d' + 1$.
      ii. (**PRG to uniform**): Consider a modified version of the UpGen operation, UpGen′, that computes the path secrets as follows:
      $$\$ \rightarrow (\mathsf{sk}_0, s_1), \$ \rightarrow \ldots, \$ \rightarrow (\mathsf{sk}_{\mathsf{d}-i^*}, s_{\mathsf{d}-i^*+1}) \xrightarrow{\mathsf{prg}} \ldots \xrightarrow{\mathsf{prg}} (\mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}})^b$$
      iii. (**Faking ciphertexts**):
         – Set $s_1^*, \ldots, s_{\mathsf{d}-i^*+1}^* \leftarrow \mathsf{0}$.
         – If $x = \mathsf{p}$, publish encryptions of $s_1^*, \ldots, s_{\mathsf{d}-i^*}^*, s_{\mathsf{d}-i^*+1}, \ldots, s_{\mathsf{d}}$.
         – If $x = \mathsf{c}$, publish encryptions of $s_1^*, \ldots, s_{\mathsf{d}-i^*+1}^*, s_{\mathsf{d}-i^*+2}, \ldots, s_{\mathsf{d}}$.
         – Set $S[\mathsf{ID}] \leftarrow (\mathsf{sk}_1, \ldots, \mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}})$.
   (c) If $\mathbf{q}_r =$ **deliver**$(t, \mathsf{ID}, \mathsf{ID}', c)$:
      – If for all $w \in [l]$, $t \neq t_w$, execute **deliver** normally.
      – Otherwise, $(\mathsf{sk}_1, \ldots, \mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}}) \leftarrow S[\mathsf{ID}]$. Let $v = \mathsf{LCA}(\mathsf{ID}, \mathsf{ID}')$ and let d'' be its depth. Set the keys from node v to the root to $\mathsf{sk}_{d'-d''}, \ldots, \mathsf{sk}_{\mathsf{d}-1}, s_{\mathsf{d}}$, for ID'.
3. **Output**: If $\mathbf{tkm}(\mathbf{q}_1, \ldots, \mathbf{q}_q) = 0$, return 0, otherwise, if $b = b'$ or if the **win** condition is triggered, return 1, otherwise return 0.

[a] Here we define the hybrid before H_i^x depending on the value of x.

[b] Here the notation $\$ \rightarrow (x, y)$ denotes the fact that (x,y) is uniformlly random, while $(x, y) \xrightarrow{\mathsf{prg}} (x', y')$ denotes the fact that (x', y') is the output of the PRG on input seed y.

Fig. 14. The hybrids for the proof of Theorem 2.

Proof. The proof is along the lines the proof of Theorem 1, thus we present the parts in which this proof deviates from it and we explain why.

Let τ^* be the ratchet tree in epoch t^*, consisting of the labels all keys generated by the latest updates that have been processed by at least one user up to t^*, and let d be its depth.

The variables $t_{i,j}$. The variables $t_{i,j}$ are defined as in the proof of Theorem 1. The difference here is that those variables now store the epoch in which the *initial* keys $(\mathsf{pk}_0, \mathsf{sk}_0)$ of the UPKE scheme are generated. In the current scheme the keys of a node v change not only due to an update of a leaf node u such that v is in the direct path of u, but also whenever a path secret is encrypted under pk_v. However, in order to fake ciphertexts and prove indistinguishability between hybrids, we need to introduce in the execution the initial keys of the UPKE scheme.

The hybrids are identical to those of Theorem 1. The main difference is the use of UPKE instead of PKE. For the ease of exposition, we revisit the hybrids below.

Hybrid H_d^{c}. This hybrid is the original CGKA experiment with respect to $\mathcal{A}$.

Hybrid H_i^x, $i \in [d]$, $x \in \{\mathsf{c}, \mathsf{p}\}$: Presented in Figure 14.

By nb_i we denote the number non-blank nodes at depth i of τ^* and for $j \in [\mathsf{nb}_i]$ and a non-blank node v_j we set c_j^i to be the number of ciphertexts generated using public keys below depth i encrypting the path secret of v_j.

As in the proof of Theorem 1 we present the following lemma.

Lemma 9. *For $i \in [d]$,*

1. $\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}^{\mathsf{c}}_{\mathsf{i}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}^{\mathsf{p}}_{\mathsf{i}}}(\mathcal{A}) + 2^i \cdot \varepsilon_{\mathsf{prg}}$.
2. $\mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}^{\mathsf{p}}_{\mathsf{i}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{CGKA}}_{\mathrm{H}^{\mathsf{c}}_{\mathsf{i}-1}}(\mathcal{A}) + \sum_{j=1}^{\mathsf{nb}_i} \mathsf{c}^i_j \cdot \varepsilon_{\mathsf{cpa}}$.

Proof. The case of PRGs (case 1) is identical to the one in the proof of Theorem 1. This is due to the fact that we can have UPKE schemes in which the randomness for updating the UPKE public and secret key pair is independent of the encrypted message, which in our case is a path secret s. Thus, when faking ciphertexts we substitute s by zero, but we don't have to fake the information that is transmitted in the ciphertext and enables UPKE updates. The scheme that we present in Section 7 has this property.

For the second case we make a reduction to the CPA security of the UPKE encryption scheme. Let v_j, $j \in [\mathsf{nb}_i]$, be the non-blank nodes at depth i. For each v_j, let $\bar{v}^j_1, \ldots, \bar{v}^j_{k_j}$, be the ancestors of v_j, that we used their public keys to encrypt the path secret of v_j, and let $\bar{t}_{l,j}$, be the epoch in which the *initial* keys of $\bar{v}^j_l$ were introduced and $\hat{t}_{l,j}$ the epoch in which the path secret of v_j is encrypted under the public key of $\bar{v}^j_l$.

We define the adversary B^{c}_i for the CPA game as follows:

Algorithm B^{c}_i:

1. For $j \in [\mathsf{nb}_i]$, receive k_j public keys $(\mathsf{pk}^0_{1,j}), \ldots, \mathsf{pk}^0_{k_j,j})$[a] from the challenger of the CPA game.
2. Execute H^{p}_{i+1} with the following differences:
   (a) For $j \in [\mathsf{nb}_i]$, $l \in [k_j]$, in the update for the epoch $\bar{t}_{l,j}$, set the public key of the node $\bar{v}^j_l$ to $\mathsf{pk}^0_{l,j}$ and the secret key to ϵ. If $\bar{t}_{l,j} = 0$, introduce $\mathsf{pk}^0_{l,j}$ during the **create-group**(G) operation.
   (b) For $j \in [\mathsf{nb}_i]$, $l \in [k_j]$, let $q_{l,j}$ be the number of path secrets encrypted under the public key of $\bar{v}^j_l$ up to epoch $\hat{t}_{l,j} - 1$. For $z \in [q_{l,j}]$, send the path secret s_z to the challenger and receive $(c_z, \mathsf{pk}_z, r_z)$. Set the public key of node $\bar{v}^j_l$ to pk_z.
   (c) For $j \in [\mathsf{nb}_i]$, $l \in [k_j]$, set $M^{l,j}_0 \leftarrow s_{\mathsf{d}-i+1}$ (here $s_{\mathsf{d}-i+1}$ is as it is computed by H^{p}_{i+1}) and $M^{l,j}_1 \leftarrow \mathsf{0}$.[b]
   (d) Set $\mathbf{M}^*_0$ (resp. $\mathbf{M}^*_1$) to be the concatenation of $M^{l,j}_0$ (resp. $M^{l,j}_1$). Send $\mathbf{M}^*_0$, $\mathbf{M}^*_1$, to the CPA challenger and receive $c^*_{l,j}$, $\mathsf{pk}^*_{l,j}$, $\mathsf{sk}^*_{l,j}$, $j \in [\mathsf{nb}_i]$, $l \in [k_j]$.
   (e) For $j \in [\mathsf{nb}_i]$, $l \in [k_j]$, in the update for the epoch $\hat{t}_{l,j}$,
      – Publish encryptions of $s_1, \ldots, s_{\mathsf{d}-i} \leftarrow \mathsf{0}$, and $s_{\mathsf{d}-i+2}, \ldots, s_{\mathsf{d}}$ (as they are computed by H^{p}_{i+1}), and for the index $\mathsf{d} - i + 1$ publish $c^*_{l,j}$.
      – Set the public and secret key pair of node $\bar{v}^j_l$ to $(\mathsf{pk}^*_{l,j}, \mathsf{sk}^*_{l,j})$.
3. Output the bit output by $\mathcal{A}$.

[a] In the UPKE CPA game definition the index of the key (here 0) is a subscript but here we use it as a superscript.
[b] Note that for distinct (l, j), $s_{\mathsf{d}-i+1}$ are independent since we refer to distinct epochs.

The hybrids H^{p}_{i+1} and H^{c}_i are only different for the **send-update** queries for the epochs $\hat{t}_{l,j}$. In H^{p}_{i+1} and in epoch $\hat{t}_{l,j}$ we have $i^* = i + 1$, while in H^{c}_i, $i^* = i$, i.e., we fake at most $n - 1$ more independent ciphertexts. H^{p}_{i+1} in epoch $\hat{t}_{l,j}$ publishes encryptions of $s_1, \ldots, s_{\mathsf{d}-i} \leftarrow \mathsf{0}$, $s_{\mathsf{d}-i+1}, \ldots, s_{\mathsf{d}}$ while, H^{c}_i publishes encryptions of $s_1, \ldots, s_{\mathsf{d}-i+1} \leftarrow \mathsf{0}$, $s_{\mathsf{d}-i+2}, \ldots, s_{\mathsf{d}}$. Clearly, if $c^*_{l,j}$, encrypts $M^{l,j}_0 = s_{\mathsf{d}-i+1}$, B^{c}_i simulates H^{p}_{i+1}, and if $c^*_{l,j}$ encrypts $M^{l,j}_1 = \mathsf{0}$, B^{c}_i simulates H^{c}_i.

Now we formally argue that the simulation described above is correct. Since $\mathcal{A}$ is an admissible adversary we have that $\mathbf{safe}(\mathbf{q}_1, \ldots, \mathbf{q}_q) = 1$, thus all corrupted users have either been removed from the group or they have updated their states, and also there is no user for which **no-del** has been enabled before the challenge epoch and the user is corrupted after the challenge epoch. Consequently, no key of τ^* is leaked to the adversary up to epoch t^*. Thus, setting the public key, $\mathsf{pk}^0_{l,j}$, of $\bar{v}^j_l$ without knowing the corresponding secret key is not an issue as there will be no corruptions under $\bar{v}^j_l$, up to epoch t^*. In addition, despite the fact that B^{c}_i is sending encryptions of $\mathsf{0}$, the correct keys for all nodes of the ratchet tree as well as the update secret are hardcoded to the private states of users on message delivery.

The main difference in the current proof is that now we allow corruptions after epoch t^*. However, by property of the updatable public key encryption, after epoch t^*, B_i^{c} learns the secret keys, $\mathsf{sk}^*_{l,j}$, for all nodes $\bar{v}_l^j$, and can perfectly simulate the view of the adversary from that point and on. □

The arguments about the total cost of PRGs and CPAs, as well as protocol correctness, are similar to the those in the proof of Theorem 1. □

F.4 Adaptive Security for Modified TreeKEM

By inspecting the proofs of non-adaptive security for TreeKEM and Modified Treekem, we observe that the sequence of hybrid arguments for the two protocols is identical, and the only difference is the use of UPKE encryption (used by Modified TreeKEM) in place of standard PKE (used by TreeKEM). As a consequence an update operation in Modified TreeKEM updates: (i) the keys along the direct path of the node that issued the update operation, and (ii) all keys in the resolution of the co-path nodes. Case (i) is identical to TreeKEM, thus we only need to show how case (ii) is captured in the GSD and pebbling games.

Consider a node v in the ratchet tree with UPKE keys $(\mathsf{pk}_i, \mathsf{sk}_i)$. After processing a ciphertext encrypted under pk_i, the keys of node v are updated to $(\mathsf{pk}_{i+1}, \mathsf{sk}_{i+1})$ and knowledge of sk_i enables the computation of sk_{i+1}. In order to capture this relation between private keys, and in addition to CPA and PRG edges (considered in TreeKEM), we consider a third type of edges, which we call *updatable encryption* (UE) edges. A UE edge $(i, i+1)$ means that sk_{i+1} is derived by sk_i via a key update operation of the UPKE scheme.

Having modeled all the Modified TreeKEM operations w.r.t. GSD graphs, it is tempting to conclude that the result of [14] directly applies in our setting, and security of Modified TreeKEM follows. However, there is one point that needs to be addressed: in Modified TreeKEM the adversary can issue polynomially-many update operations that update the key of a specific node of the ratchet tree, creating paths polynomial in length consisting exclusively of UE edges. Consequently, the resulting graph in not of logarithmic depth and the bound of [14] becomes useless (the security loss factor becomes $O(n^{\mathrm{poly}(n)})$).

In order to prove security for Modified TreeKEM, we need to effectively handle UE edges in the pebbling process. Let $\mathsf{sk}_0, \ldots, \mathsf{sk}_t$, be a path of UE edges in the GSD graph and assume that at some point in the game $\mathsf{Enc}(\mathsf{pk}_t, \mathsf{sk})$ is requested by the adversary and sk is challenged. Then, if we follow the original rules of the GSD game would require that in order to place a pebble on the edge $(\mathsf{sk}_t, \mathsf{sk})$, all incoming edges to sk_t should have also been pebbled, including the UE edge $(\mathsf{sk}_{t-1}, \mathsf{sk}_t)$. We argue that this is not the case: by the CPA security properties of UPKE, ciphertexts can be faked at any point in the computation, and the only requirement is that the *initial keys* $(\mathsf{pk}_0, \mathsf{sk}_0)$ are independent of any other value in the execution and sk_0 is safe. In the context of the pebbling game, this simply means that pebbling all incoming (PRG or CPA) edges to sk_0 suffices for faking ciphertexts that are computed over updated keys and no pebbling (faking) is required for UE edges. As a consequence, an adaptive proof in the UPKE setting requires (i) guessing if the incoming edges to an initial key sk_0, and which of them, are pebbled, and (ii) guessing the length of the UE path $\mathsf{sk}_0, \ldots, \mathsf{sk}_t$, or in other words, guessing the right version of the UPKE key, out of all keys. This requires guessing correctly logarithmically many values (keys and pebbling configurations), as any path consists of logarithmically many CPA/PRG edges. Note that any single node on the path might introduce polynomially many UE edges, but for each one of them we just need to guess its length (i.e. there is no guessing related to pebbling for edges in the UE path). Given the above, we believe that the $O(n^{\log n})$ bound of [14] applies in the Modified TreeKEM setting, but we leave it as an open problem to formally prove it.