Multicast Key Agreement, Revisited

Alexander Bienstock[1⋆], Yevgeniy Dodis[1⋆⋆], and Yi Tang[2]

[1] New York University
[2] University of Michigan
{abienstock,dodis}@cs.nyu.edu, yit@umich.edu

Abstract. Multicast Key Agreement (MKA) is a long-overlooked natural primitive of large practical interest. In traditional MKA, an omniscient *group manager* privately distributes secrets over an untrusted network to a dynamically-changing set of group members. The group members are thus able to derive shared group secrets across time, with the main security requirement being that only current group members can derive the current group secret. There indeed exist very efficient MKA schemes in the literature that utilize symmetric-key cryptography. However, they lack formal security analyses, efficiency analyses regarding dynamically changing groups, and more modern, robust security guarantees regarding user state leakages: *forward secrecy* (FS) and *post-compromise security* (PCS). The former ensures that group secrets prior to state leakage remain secure, while the latter ensures that after such leakages, users can quickly recover security of group secrets via normal protocol operations.

More modern Secure Group Messaging (SGM) protocols allow a group of users to asynchronously and securely communicate with each other, as well as add and remove each other from the group. SGM has received significant attention recently, including in an effort by the IETF Messaging Layer Security (MLS) working group to standardize an eponymous protocol. However, the group key agreement primitive at the core of SGM protocols, Continuous Group Key Agreement (CGKA), achieved by the TreeKEM protocol in MLS, suffers from bad worst-case efficiency and heavily relies on less efficient (than symmetric-key cryptography) public-key cryptography. We thus propose that in the special case of a group membership change policy which allows a *single* member to perform all group additions and removals, an upgraded version of classical Multicast Key Agreement (MKA) may serve as a more efficient substitute for CGKA in SGM.

We therefore present rigorous, stronger MKA security definitions that provide increasing levels of security in the case of both user and group manager state leakage, and that are suitable for modern applications, such as SGM. We then construct a formally secure MKA protocol with strong efficiency guarantees for dynamic groups. Finally, we run experiments which show that the left-balanced binary tree structure used in TreeKEM can be replaced with red-black trees in MKA for better efficiency.

1 Introduction

Multicast Key Agreement. Multicast Key Agreement (MKA) is a natural primitive of large practical interest that has not received much attention recently. In MKA, there is an omniscient *group manager* that privately distributes secrets through *control messages* over an untrusted network to a dynamically-changing set of group members. The secrets delivered via these control messages allow group members to derive shared group secrets across time. The main security requirement is that users added to the group are not able to derive old group secrets while users removed from the group are not able to derive new group secrets. Traditionally, these derived group secrets could then be used in some higher-level application, such as digital rights management (DRM) for PayTV, in which the group manager could broadcast content privately to users in a unidirectional manner.

There are several works in the literature that propose and study practical MKA protocols [36, 38, 20, 15, 25, 34], the best of which achieve $O(\log n_{\max})$ worst-case communication complexity, where $n_{\max}$ is the maximum number of users ever in the group. Moreover, all of these protocols utilize efficient *symmetric-key* cryptography for the control messages of each operation.

However, prior practical multicast works did *not* (i) provide formal security models or proofs, (ii) resolve efficiency issues regarding dynamically changing groups, nor, importantly, (iii) achieve more

⋆ Partially supported by a National Science Foundation Graduate Research Fellowship
⋆⋆ Partially supported by gifts from VMware Labs and Google, and NSF grants 1619158, 1319051, 1314568.

modern, robust security notions, such as privacy surrounding leakages of user secret states. With respect to a user secret state leakage, optimal security requires:

- *Forward Secrecy* (FS): All group secrets that were generated before the leakage should remain secure.
- *Post-Compromise Security* (PCS): Privacy of group secrets should be quickly recovered as a result of normal protocol operations.

Most of these prior schemes base off of the Logical Key Hierarchy (LKH) from [36, 38]. In these schemes, symmetric keys are stored at the nodes of a tree in which users are assigned to leaf keys and the group secret is the key at the root. The tree invariant is that users only know the keys at the nodes along the path from their leaf to the root. Thus in its simplest form, if, for example, a user is removed, the group manager simply removes their leaf, refreshes the other keys at the nodes along the path from their leaf to the root, and encrypts these keys to the children of the corresponding nodes on the path. This strategy thus achieves $O(\log n_{\max})$ communication and computational complexity.

We state the efficiency in terms of $n_{\max}$, since these schemes never address exactly how the tree should be balanced if many users are added or removed from the group. In order to achieve optimal efficiency of the scheme according to the lower bound of Micciancio and Panjwani [24], i.e., have communication and computational complexity of $O(\log n_{\text{curr}})$, where n_{curr} is the current number of users in the group, the tree must remain balanced after such operations. However, this complicates the tree invariant: how should it be maintained so that security is achieved? Users should never occupy interior nodes nor retain information for old leaf-to-root paths. The first column of Table 1 summarizes the properties of traditional practical MKA constructions.

New Applications: Single Administrator Secure Group Messaging. We observe that today, MKA has several more modern applications, including:

1. Adapting the classical use case of DRM to more modern purposes, such as live streaming on the internet (e.g., twitch), or
2. Communicating with resource-constrained IoT devices [29], or
3. Enabling *Secure Group Messaging* (SGM) among the group members.

Here, we focus on SGM and first provide some background: End-to-end Secure Messaging protocols allow two parties to exchange messages over untrusted networks in a secure and asynchronous manner. The famous double ratchet algorithm of the Signal protocol [27] achieves great security and efficiency in this setting, and is now the backbone of several popular messaging applications (e.g., Signal, WhatsApp, Facebook Messenger Secret Conversations, etc.) which are used by billions of people worldwide.

However, allowing *groups* of users to efficiently exchange messages over untrusted networks in a secure manner introduces a number of nontrivial challenges. Such SGM protocols have begun to receive more attention recently: e.g., the IETF has launched the message-layer security (MLS) working group, which aims to standardize an eponymous SGM protocol [5]. In the SGM setting, users share evolving group secrets across time which enable them to asynchronously and securely communicate with each other. Furthermore, they are allowed to asynchronously add and remove other users from the group. However, in practice, groups typically utilize some stable policy to determine membership change permissions.

The key agreement primitive at the center of SGM is called Continuous Group Key Agreement (CGKA) [1, 3, 35] and is achieved by the TreeKEM protocol in MLS [9], among others (e.g., [3, 37]). Much like MKA, CGKA requires end-to-end security of the evolving group secrets which can only be known by current group members, but also FS and PCS with respect to user state leakages. Furthermore, asynchronous dynamic operations can be performed by any user. It is a hard problem to design CGKA protocols from minimal security assumptions that are efficient. All current practical CGKA constructions require $\Omega(n_{\text{curr}})$ communication in the worst-case, due to complications regarding user-performed dynamic operations.[3] A definitively bad situation is immediately following the creation of a CGKA group: if many users remain offline for a long time after creation, communication complexity may remain $\Omega(n_{\text{curr}})$ during this time (for example, if the creator performs updates), since only a few users will have contributed secrets to the group. Additionally, all current CGKA protocols heavily rely on public-key cryptography (which is of course less efficient than symmetric-key cryptography).

[3] Most constructions informally claim to have "fair-weather" $O(\log n_{\max})$ communication complexity [2], i.e., in some (undefined) *good* conditions, they have $O(\log n_{\max})$ communication.

	Traditional MKA	CGKA	Our MKA
Efficiency	$O(\log n_{\max})$	$\Omega(n_{\text{curr}})$ ("fair-weather" $O(\log n_{\max})$)	$O(\log n_{\text{curr}})$
Heavy Use of Public-Key Cryptography	no	yes	no
Add/Remove	group mgr	anybody[1]	group mgr
Members PCS/FS	no/no	yes/yes	yes/yes
Group Mgr PCS/FS	no/no	N/A	weak[2]/yes
Group Mgr Local Storage	$\Omega(n_{\text{curr}})$	N/A	$O(1)$

Table 1. A comparison of some properties of the traditonal MKA construction in the literature, CGKA constructions, and our optimal MKA construction. The top half showcases the efficiency advantages of both traditional and our MKA, while the bottom showcases the functionality advantages of CGKA, and the added security and efficiency of our MKA with respect to traditional MKA.

[1] Although in formal definitions of CGKA constructions, any group member can perform add and remove operations, in practice, a stable policy is needed.

[2] PCS against group manager corruptions in our construction requires every group member at the time of corruption to be either removed or updated.

Although CGKA is usually the best way to distribute keys amongst group members in the SGM setting, due to its strong functionality guarantees, it clearly suffers from the above deficiencies. Therefore, more efficient alternatives should be provided for special cases. One such realistic setting is when the policy for group membership changes only permits a single group member (administrator) to add and remove users. In this case, we posit that a great substitute for CGKA is (a stronger version of) MKA. Therefore, a main contribution of this paper, upon which we elaborate in the next subsection of the introduction, is providing a strong formal model of MKA that is suitable for modern uses such as SGM, and then constructing a protocol which efficiently achieves the security of this model. This protocol does not heavily rely on public-key cryptography as in CGKA, and has much more efficient $O(\log n_{\text{curr}})$ communication for operations as opposed to $\Omega(n_{\text{curr}})$ communication in the worst-case for CGKA. Table 1 compares the properties of traditional MKA in the literature with CGKA and our more secure, and efficient, MKA protocol.

We emphasize that the group manager in MKA (who is the single member of the SGM group with membership change privileges) only generates the secrets of the group and the control messages by which they are distributed, and can (should) be viewed as completely divorced from the central delivery server that is usually a part of SGM protocols such as MLS. Therefore, for example, the group manager does not need to be always online and the group secrets remain private from the delivery server, thus providing proper end-to-end security for SGM. Furthermore, despite the fact that the group manager performs all membership change operations, actual SGM communications remain asynchronous by using the group secret from MKA for the Application Key Schedule, as is usually done with CGKA in MLS [5].

Group Authenticated Key Exchange. MKA and CGKA are closely related to the setting of Group Authenticated Key Exchange (GAKE) (e.g., [23, 13, 12, 14]). In GAKE, several (possibly overlapping) groups of users work together to establish independent group keys across time. Formal models, constructions, and proofs for GAKE indeed exist that consider FS and PCS. However, all such constructions involve a large amount of interaction, requiring many parties to be online, which is undesirable for the MKA and SGM settings.

1.1 Contributions

Multicast Key Agreement with optimal user security. This paper provides a more rigorous study of practical MKA constructions by first providing a formal security definition (based on that of [1] for CGKA) which guarantees correctness and privacy of the protocol, as well as PCS and FS for users. Moreover, our definition allows for partially active security, meaning the delivery service can send control messages to

users in arbitrarily different orders (with arbitrary delays), but cannot inject or modify control messages. We additionally allow for *adaptive* adversaries, achieving security proofs with quasipolynomial loss in the standard model, and polynomial loss in the random oracle model, using the proofs of Tainted TreeKEM (a CGKA protocol) in [3] as a template.

We then provide a secure and efficient construction of the primitive based on LKH that instead utilizes *generic* tree structures. This protocol moreover utilizes (dual) pseudorandom functions (dPRFs) [6] and an *Updatable Symmetric Key Encryption* (USKE) primitive [7] (which is the symmetric analog to Updatable Public Key Encryption (UPKE) [22, 1, 17]) to allow for optimal FS by refreshing keys in the tree as soon as they are used as an encryption secret key.

Furthermore, we show that if the tree structure used is a left-balanced binary tree (LBBT), as in most CGKA constructions, then the computational and communication complexity of operations besides creation are $O(\log n_{\max})$. However, we show that if the protocol uses 2-3 trees or red-black trees (RBTs), this complexity improves to $O(\log n_{\text{curr}})$, thus achieving optimal communication complexity according to the lower bound of [24]. Indeed, we implement our construction which shows empirically that this theoretical difference between $n_{\max}$ and n_{curr} makes RBTs more efficient than LBBTs in practice.[4] In section 5.4, we illustrate that in the average case, where updates are performed more often than additions and removals (which occur at the same rate), RBTs are the most efficient out of the three trees. Furthermore, in an event in which group membership drastically decreases, LBBTs are much less efficient than both RBTs and 2-3 trees after this decrease. Even if group membership gradually decreases over time, RBTs are still the most efficient of the three.

Adding security for group manager corruptions. We then add to the security definition of MKA to allow for corruptions of the group manager state. Despite these corruptions, our definition still demands immediate FS, i.e. all group secrets before the corruption should remain secure, and eventual PCS, i.e. after the group manager has updated or removed all of the members of the group at the time of the corruption, future group secrets should be secure. While our above construction already achieves this stronger definition, it relies on the group manager to store the keys at each node of the tree. Such a requirement assumes that the group manager has the capability to locally store such large ($\Omega(n_{\text{curr}})$) amounts of data, as well as handle availability, replication, etc. In an effort to reduce the local storage of the group manager and securely outsource storage of the tree to an untrusted remote server, we use the recent work of [10], who formalize the notion of *Forward Secret encrypted RAM* (FS eRAM) and apply it to MKA.[5] Intuitively, the security of FS eRAM only allows the adversary to obtain secrets currently at the nodes of the MKA tree upon corruption of the group manager, while past secrets at the nodes remain secure.

The work of [10] focuses on a lower bound for FS eRAM and only sketches its application to MKA, with similar limitations as the prior works on MKA (no formal analysis of security or dynamism). We therefore provide a similar result with a fully formal argument that achieves the security of the modified MKA definition, while retaining asymptotic communication and computational complexity of our original MKA scheme. Moreover, it enables the group manager to separate her state into $O(1)$ local storage and $O(n_{\text{curr}})$ remote storage. The third column of Table 1 shows the properties of our second construction.

Optimal PCS for group manager corruptions. The security notion described above allows us to retain the use of efficient symmetric-key encryption for control messages, while achieving good, but not optimal, eventual PCS for group manager corruptions. Indeed, we show in Section 7 that to obtain optimal PCS, i.e., security after one operation following a group manager corruption, public-key encryption is necessary. Then we show that achieving optimal PCS (and all the other properties described above) with public-key encryption is easy: In our construction, we simply replace USKE with UPKE (and the group manager only remotely, not locally, stores the public keys at the tree nodes).

[4] See `https://github.com/abienstock/Multicast-Key-Agreement` for the code.

[5] FS eRAM is also known in the literature as, e.g., "secure deletion" [28, 30, 31, 4, 32], "how to forget a secret" [16], "self-destruction" [19], and "revocability" [11].

2 Preliminaries

This section introduces some basic notation and concepts for trees. For more details on left-balanced binary trees, 2-3 trees, and left-leaning red-black trees, seee Appendix A. Also, for the definitions of PRGs, (d)PRFs and CPA-secure symmetric-key encryption, refer to Appendix B.

A tree τ is a connected undirected acyclic graph with a special node called the *root* of the tree. Every node in the tree has a unique identifier, for our purposes, some value $\ell \in \{0,1\}^\lambda$, where λ is the security parameter. For every node in a tree, there is a unique path from (and including) the node to the root of the tree, referred to as its *direct path*. The length of the direct path of a node is called the *depth* of that node. The *height* of the tree is the maximum depth of its nodes. The subgraph over the set of nodes whose direct paths contain node v (with v assigned to be the root) is called the *subtree* at v. Depth-1 nodes in the subtree at node v are the *children* of v, and v is their *parent*, and they are *siblings* of each other. The *degree* of a node v, denoted by $\deg(v)$ is its number of children. We call the children of a node v, $v.c_j$, for $j \in [\deg(v)]$, and its parent $v.p$. Nodes v such that $\deg(v) = 0$ are called *leaf node*s, and other nodes are called *internal node*s. The *maximum degree* of any node v of τ is denoted as $\deg(\tau)$. The set of siblings of all nodes along the direct path of node v is called the *copath* of v. We refer to a connected subgraph of the tree that includes the root as a *skeleton* of the tree. For any skeleton, its *frontier* consists of those nodes in the tree that are not in the skeleton but have edges from nodes in the skeleton.

3 Updatable Symmetric Key Encryption

In this section, we define and construct a symmetric analog to the well-known Updatable Public Key Encryption primitive in the Secure Messaging literature [22, 1, 17], which we call Updatable Symmetric Key Encryption (USKE). This notion and the following construction are quite similar to that of Bellare and Yee [7], except that we implicitly include the key update functionality directly in the encryption and decryption algorithms (whereas they provide a separate update algorithm): USKE schemes simply augment the syntax of the encryption algorithm of standard symmetric encryption to output a new key k' in addition to the ciphertext, and symmetrically, the decryption algorithm to output new key k', when decrypting that ciphertext. If the new key is exposed, then all plaintexts encrypted under previous versions of the key remain secure.

Definition 1 (Updatable Symmetric Key Encryption (USKE)). *A* USKE *scheme is a double of algorithms* $\mathsf{uske} = (\mathsf{UEnc}, \mathsf{UDec})$ *with the following syntax:*

- *Encryption:* UEnc *receives a key k and a message m and produces a ciphertext c and new key k'.*
- *Decryption:* UDec *receives a key k and a ciphertext c and produces a message m and new key k'.*

Correctness. A USKE scheme must satisfy the following correctness property. For any sequence of messages $m_1, \ldots, m_q$:

$$\Pr\left[\begin{matrix} k'_0 \leftarrow k_0; \text{For } i \in [q], (c_i, k_i) \leftarrow \mathsf{UEnc}(k_{i-1}, m_i); \\ (m'_i, k'_i) \leftarrow \mathsf{UDec}(k'_{i-1}, c_i) : m_i = m'_i \end{matrix}\right] = 1.$$

The following notion of CPA security that we define below is very similar to standard CPA-secure SKE:

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

(i) The challenger sets $k_0 \leftarrow_\$ \mathcal{K}$
(ii) For $i \in [q]$, $\mathcal{A}$ outputs m_i and receives back c_i such that $(c_i, k_i) \leftarrow \mathsf{UEnc}(k_{i-1}, m_i)$.
(iii) $\mathcal{A}$ sends challenge messages m_0^*, m_1^*.
(iv) For $i \in [q]$, the challenger computes $(m_i, k'_i) \leftarrow \mathsf{UDec}(k'_{i-1}, c_i)$, where $k'_0 = k_0$.
(v) Then the challenger computes $(c^*, k_{q+1}) \leftarrow \mathsf{UEnc}(k_q, m_b^*)$ for uniform $b \in \{0,1\}$, $(\cdot, k^*) \leftarrow \mathsf{UDec}(k'_q, c^*)$ and sends (c^*, k^*) to $\mathcal{A}$.
(vi) $\mathcal{A}$ sends bit $b' \in \{0,1\}$ to the challenger.

$\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ in the game is denoted by $\mathsf{Adv}^{\mathsf{uske}}_{\mathsf{cpa}^*}(\mathcal{A}) = |\Pr[b = b'] - 1/2|$.

Definition 2 (USKE security). *An updatable symmetric key encryption scheme* uske *is* (t, ε)-CPA*-secure *if for all $t-$attackers $\mathcal{A}$,*

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

$\mathsf{UEnc}((s,r),m)$:	$\mathsf{UDec}((s,r),c)$:
$c \leftarrow m \oplus r$	$m \leftarrow c \oplus r$
$(s',r') \leftarrow \mathsf{prg}(s)$	$(s',r') \leftarrow \mathsf{prg}(s)$
return $(c,(s',r'))$	**return** $(m,(s',r'))$

Fig. 1. USKE Construction from one-time pad and PRG.

Consistent USKE schemes. We call a USKE scheme *consistent* if encryption and decryption keys remain the same for all versions, i.e., for any sequence of messages $m_1,\ldots,m_q$, if $k'_0 \leftarrow k_0$, then for $i \in [q]$, $k'_i = k_i$, where $(c_i,k_i) \leftarrow \mathsf{UEnc}(k_{i-1},m_i); (m'_i,k'_i) \leftarrow \mathsf{UDec}(k'_{i-1},c_i)$. Our construction below is consistent, and throughout this work, we will assume that all USKE schemes are consistent.

3.1 Construction

In Figure 1, we give the details of a simple IND-CPA* secure USKE scheme that only relies on one-time pads and a PRG. It is easy to observe its correctness.

Theorem 1. *Assume* prg *is a* $(t_{\mathsf{prg}},\varepsilon_{\mathsf{prg}})$*-secure pseudorandom generator. Then the USKE scheme of Figure 1 is* $(t, q\cdot\varepsilon_{\mathsf{prg}})$*-CPA*-secure, where* $t \approx t_{\mathsf{prg}}$ *and* q *is the number of adversarial encryption queries (excluding the challenge).*

Proof. The proof simply works over $q+1$ hybrids $H_0,\ldots,H_q$, where H_0 is the original game and for each $i \in [q]$, H_i is the game in which the challenger computes the first i one-time pads uniformly at random during encryption, and the rest as the output of prg on the $(j-1)$-st seed s_{j-1}, for $j \in [i+1,q]$. Thus H_q is the game where both the one-time pad used to generate the challenge ciphertext c^*, and thus c^* itself, are uniformly random and independent of the key k^* sent to $\mathcal{A}$ and all other ciphertexts c_i sent to $\mathcal{A}$, regardless of the challenge bit b. Hence, $\mathsf{Adv}^{\mathsf{uske}}_{H_q}(\mathcal{A}) = 0$.

We will now show that for $i \in [q]$, $\mathsf{Adv}^{\mathsf{uske}}_{H_{i-1}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{uske}}_{H_i} + \varepsilon_{\mathsf{prg}}$. Assume towards contradiction that $\mathsf{Adv}^{\mathsf{uske}}_{H_{i-1}}(\mathcal{A}) > \mathsf{Adv}^{\mathsf{uske}}_{H_i} + \varepsilon_{\mathsf{prg}}$. Now, consider the reduction algorithm $\mathcal{B}^{\mathsf{prg}}_i$ which on input (s^*,r^*) from the prg challenger does the following:

- Let $s_0 \leftarrow \bot, r_0 \leftarrow_{\$} \mathcal{R}$.
- For $j \in [i-1]$, compute the j-th encyrption $\mathsf{UEnc}(k_{j-1},m_j)$ as normal, but let $s_j \leftarrow \bot, r_j \leftarrow_{\$} \{0,1\}^{\lambda}$.
- Then compute the i-th encryption as normal, except let $(s_i,r_i) \leftarrow (s^*,r^*)$.
- Compute everything else as normal, skipping step (iv) and using $k^* = k_{q+1}$.
- Finally, receive bit b' from $\mathcal{A}$ and output 0 iff $b' = b$.

Thus, if $\mathsf{Adv}^{\mathsf{uske}}_{H_{i-1}}(\mathcal{A}) > \mathsf{Adv}^{\mathsf{uske}}_{H_i} + \varepsilon_{\mathsf{prg}}$, then $\mathcal{B}^{\mathsf{prg}}_i$ breaks the security of the PRG game, reaching a contradiction. So, $\mathsf{Adv}^{\mathsf{uske}}_{\mathsf{cpa}^*}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{uske}}_{H_0}(\mathcal{A}) \leq q\cdot\varepsilon_{\mathsf{prg}}$. □

4 Multicast Key Agreement

A Multicast Key Agreement (MKA) scheme allows for a group manager to distribute secret random values to group members which they can use to obtain shared key material for some higher-level protocol. The manager (and only the manager) can make changes to the group, i.e. add and remove members, as well as help *individual* parties in the group update their secrets, if needed. There are many options for what a higher-level protocol can do with the shared group key material from the MKA scheme, as stated in the introduction.

In this section, we formally define the syntax of MKA schemes and introduce a security notion that captures correctness, key indistinguishability, as well as forward secrecy and post-compromise security for group members, even with a partially active and adaptive adversary that can deliver control messages in an arbitrary order for each user.

4.1 MKA Syntax

Before we formally define the syntax of MKA, observe that the group manager will often need to communicate secrets to users with whom she does not share any prior secrets (e.g., when adding them). Indeed, there are many primitives that require such delivery of secrets with no prior shared secrets – e.g., CGKA, Identity-Based Encryption (IBE), Broadcast Encryption (BE), etc. – and all do so using a secure channel (which may be compromised by an adversary on-demand). In modern SGM literature, this channel is usually implemented by a Public Key Infrastructure (PKI), since the parties anyway need a PKI to authenticate each other. For our setting, where all control messages are sent by a single party (much like IBE and BE), we take a more traditional route by not explicitly including a PKI for the secure channel used to deliver secret keys. Instead, we simply model separate *out-of-band* channels from the group manager to each user for such messages (which will indeed often be implemented via a PKI). In the definition, all *out-of-band values* k_{ID} output by group manager algorithms are *small* (security parameter size), and will be sent over the corresponding individual out-of-band channel from the group manager to user ID. Moreover, any out-of-band values input by the user process algorithm will have been retrieved from their corresponding out-of-band channel from the group manager.

Definition 3 (Multicast Key Agreement (MKA)). *An* MKA *scheme* $\mathsf{M} = (\mathsf{Minit}, \mathsf{Uinit}, \mathsf{create}, \mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{proc})$ *consists of the following algorithms:*[6]

- Group Manager Initialization*:* Minit *outputs initial state* Γ*.*
- User Initialization*:* Uinit *takes an ID* ID *and outputs an initial state* γ*.*
- Group Creation*:* create *takes a state* Γ *and a set of IDs* $\mathsf{G} = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$*, and outputs a new state* Γ'*, group secret* I*, control message* T*, and dictionary* $\mathsf{K}[\cdot]$*, which stores a small user out-of-band value for every* $\mathsf{ID} \in \mathsf{G}$*.*
- Add*:* add *takes a state* Γ *and an ID* ID *and outputs a new state* Γ'*, group secret* I*, control message* T*, and a small out-of-band value for* ID*,* k_{ID}*.*
- Remove*:* rem *takes a state* Γ *and an ID* ID *and outputs a new state* Γ'*, group secret* I*, and a control message* T*.*
- Update*:* upd *takes a state* Γ *and an ID* ID *and outputs a new state* Γ'*, group secret* I*, control message* T*, and a small out-of-band value for* ID*,* k_{ID}*.*
- Process*:* proc *takes a state* γ*, a control message* T *sent over the broadcast channel, and an (optional) out-of-band value* k*, and outputs a new state* γ' *and a group secret* I*.*

One uses a MKA scheme as follows: Once a group is established by the group manager using create, the manager may call the add, rem, or upd algorithms. We note that including an explicit upd algorithm for users enables better efficiency than the alternative – removing a user and immediately adding her back. After each operation performed by the group manager, a new *epoch* is instantiated with the corresponding group secret computed by the operation. It is the implicit task of a delivery server connecting the group manager with the members to relay the *control messages* to all current group members. The delivery server can deliver any control message output by the group manager to any user, at any time. Additionally, the group manager may (directly) send certain members secret values over the out-of-band channel. Whenever a group member receives a control message and the corresponding out-of-band secret from the group manager (if there is one), they use the proc algorithm to process them and obtain the group secret I for that epoch.

4.2 MKA Efficiency Measures

We introduce four measures of efficiency for MKA schemes. These efficiency measures are in terms of the number of group members at a certain epoch, n_{curr}, and the maximum number of group members throughout the execution of the protocol, $n_{\max}$. The first is that of worst-case space complexity of the

[6] We could easily allow for batch operations – as the latest MLSv11 draft does through the "propose-then-commit" framework [5] – which would be more efficient than the corresponding sequential execution of those operations, but we choose not to for simplicity. We also note that updating one user at a time in case of their corruption is of course much more efficient (and just as secure) than updating the whole group in case just that user was corrupted.

group manager state, i.e. the size of Γ, which we refer to as $s(n_{\text{curr}}, n_{\max})$. The second is that of worst-case communication complexity of control messages output by the group manager for add, rem, and upd operations, which we refer to as $c(n_{\text{curr}}, n_{\max})$. The third is that of worst-case time complexity of group manager add, rem, and upd operations, which we refer to as $t_1(n_{\text{curr}}, n_{\max})$. The fourth is that of worst-case time complexity of user proc process algorithms for control messages output by group manager, which we refer to as $t_2(n_{\text{curr}}, n_{\max})$. In our construction, using specific tree structures in Section 5, the worst-case space complexity of a user, i.e. the size of γ, is proportional to $t_2(n_{\text{curr}}, n_{\max})$. We will often refer to these measures without writing them explicitly as functions of n_{curr} and $n_{\max}$ (and will sometimes informally conflate n_{curr} and $n_{\max}$ into a single value, n, for simplicity).

4.3 MKA Security

We now introduce the formal security definition for MKA schemes that ensures optimal security for group members against adaptive and partially active adversaries. The basic properties any MKA scheme must satisfy for this definition are the following:

- *Correctness even with partially active adversaries*: The group manager and all group members output the same group secret in all epochs, once the group members have eventually received all control messages in order (with different messages possibly arbitrarily ordered in between).
- *Privacy*: The group secrets look random given the message transcript.
- *User Forward Secrecy (FS)*: If the state of any user is leaked at some point, all previous group secrets remain hidden from the attacker.
- *User Post-compromise Security (PCS)*: Once the group manager performs updates for every group member whose state was leaked, group secrets become private again.

All of these properties are captured by the single security game presented in Figure 2, denoted by user-mult. In the game the attacker is given access to oracles to drive the execution of the MKA protocol. We note that the attacker is not allowed to modify or inject any control messages, but again, they can deliver them in any arbitrary order. This is because we assume MKA will be used in a modular fashion, and thus, whichever higher-level application it is used within will provide authentication (as is also assumed for CGKA). However, we do assume that the attacker sees all broadcast control messages and can corrupt out-of-band messages.

Epochs. The main oracles to drive the execution of the game are the oracles for the group manager to create a group, add users, remove users, and update individual users' secrets, as well as the oracle to deliver control messages to users, i.e. **create-group**, **add-user**, **remove-user**, **update-user**, **deliver**. The first four oracles allow the adversary to instruct the group manager to initiate a new epoch, while the deliver oracle advances group members to the next epoch in which they are a group member, if in fact the message for that epoch is the one being delivered. The game forces the adversary to initially create a group, and also enables users to be in epochs which are arbitrarily far apart.

Initialization. The **init** oracle sets up the game and all variables needed to keep track of its execution. The game initially starts in epoch $t = 0$. Random bit b is used for real-or-random challenges, Γ stores the group manager's current state, and γ is a dictionary which keeps track of all of the users' states. The dictionary ep keeps track of which epoch each user is in currently (-1 if they are not in the group). Dictionaries $\mathbf{G}$ and $\mathbf{I}$ record the group members in each epoch, and the group secret of each epoch, respectively. Dictionary chall is used to ensure that the adversary cannot issue multiple challenges or reveals per epoch. Additionally, K records any out-of-band random values that the group manager needs to send to any of the group members for each epoch. Finally, M records all control messages that the group manager associates with group members for each epoch; the adversary has read access to M, as indicated by **pub**.

Operations. When the adversary calls any of the oracles to perform operations to define a new epoch t, the resulting control messages for group members ID are stored in M with the key (t, ID). Additionally, any associated out-of-band values for ID are stored in K with the key (t, ID). The oracle **create-group** causes the group manager to create a group with members $\mathsf{G} = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$, only if $t = 0$. Thereafter, the

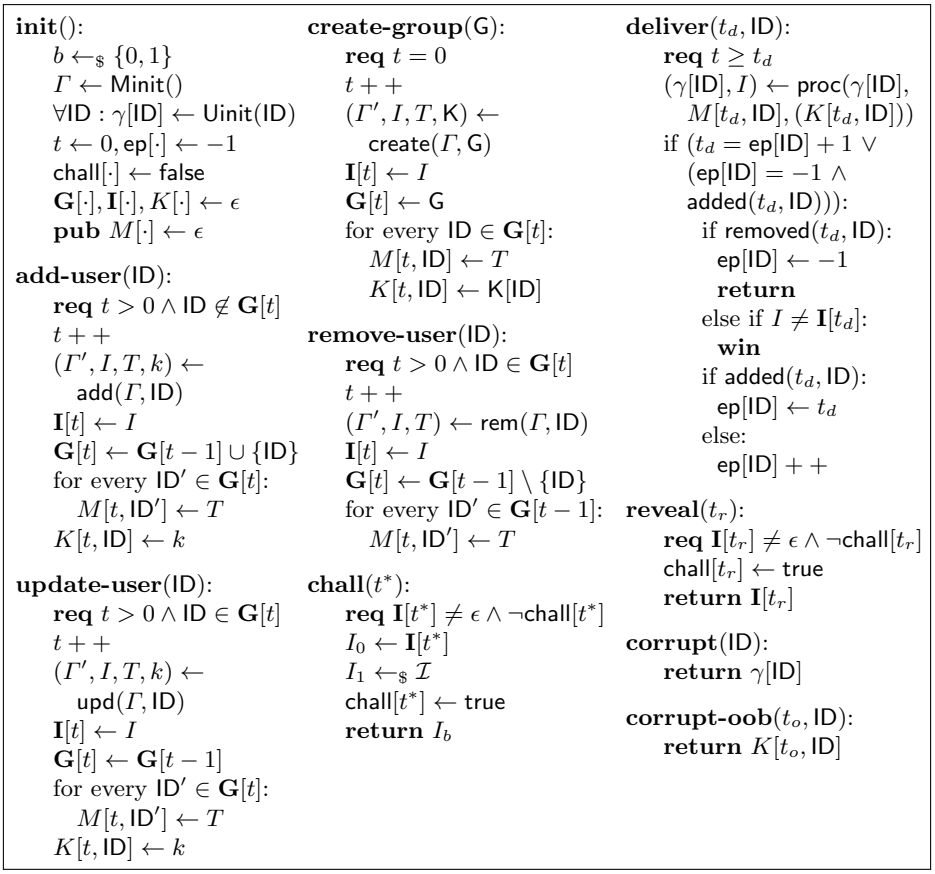

Fig. 2. Oracles for the MKA security game user-mult for a scheme $\mathsf{M} = (\mathsf{Minit}, \mathsf{Uinit}, \mathsf{create}, \mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{proc})$. Functions added and removed are explained in the text.

group manager calls the group creation algorithm and produces the resulting control message and out-of-band values for their respective users. In each of the oracles **add-user**, **remove-user**, and **update-user**, the **req** statements checks that the call is proper (e.g., that an added ID was not already in the group); exiting the call if not. Subsequently, the oracles call the corresponding MKA algorithms, and store the resulting control messages (and possibly out-of-band values) in M (and K).

Delivering Control Messages and out-of-band values. The oracle **deliver** is called with the same arguments (t_d, ID) that are used as keys for M and K. The associated control message (and possibly out-of-band value) is retrieved from M (and K) and run through proc on the current state of ID. The security game then checks that this is the next message for ID to process, i.e., either ID is in epoch $t_d - 1$, or was added to the group in epoch t_d, as checked by the function: $\mathsf{added}(t_d, \mathsf{ID}) := \mathsf{ID} \notin \mathbf{G}[t_d - 1] \wedge \mathsf{ID} \in \mathbf{G}[t_d]$. If so, the game first checks if ID is removed from the group in epoch t_d, as checked by the function: $\mathsf{removed}(t_d, \mathsf{ID}) := \mathsf{ID} \in \mathbf{G}[t_d - 1] \wedge \mathsf{ID} \notin \mathbf{G}[t_D]$. In this case, the game sets the epoch counter for ID to -1. Otherwise, the game requires that the group secret I which is output by $\mathsf{proc}(\gamma[\mathsf{ID}], M[t_d, \mathsf{ID}], (K[t_d, \mathsf{ID}]))$ is the same as the secret output by the group manager for that epoch. If it is not, the instruction **win** reveals the secret bit b to the attacker (this ensures correctness). Finally, the epoch counter for ID is set to t_d. Note: although the security game only requires correctness for control messages that are eventually delivered in order, the adversary can choose to deliver them in arbitrary order, and users must immediately handle them.

Challenges, corruptions, and deletions. In order to capture that group secrets must look random and independent of those for other rounds, the attacker is allowed to issue a challenge **chall**(t^*) for any epoch *or* use **reveal**(t_r) to simply learn the group secret of an epoch.

We model forward secrecy and PCS by allowing the adversary to learn the current state of any party by calling the **corrupt** oracle. Through such corruptions, the security game also prohibits the

```
user-safe(q_1, ..., q_q):
    for (i, j) s.t. q_i = corrupt(ID) ∨ (q_i = corrupt-oob(t, ID) ∧
      added(t, ID)) for some t, ID and q_j = chall(t*) for some t*:
        if q2e(q_i) < t* ∧ ∄l s.t. 0 < q2e(q_i) < q2e(q_l) ≤ t* ∧
          ((q_l = update-user(ID) ∧ ∄m > l s.t.
          q_m = corrupt-oob(q2e(q_l), ID)) ∨ q_l = remove-user(ID)):
            return 0
    return 1
```

Fig. 3. The **user-safe** safety predicate determines if a sequence of calls $(\mathbf{q}_1, \ldots, \mathbf{q}_q)$ allows the attacker to trivially win the MKA game.

phenomenon called *double joining*: In the specification of a protocol, users may delete certain secrets from their state, when they are deemed to be useless. However, a user could decide to act maliciously by saving all of these old secrets. Double joining is the phenomenon in which a user has been removed from the group in some epoch, but perhaps they have saved some secrets that allow them to still derive the group secret for future epochs. In fact, since the group manager handles all operations, this is the *only* way that users can act maliciously. The **corrupt** oracle on its own prohibits double joining because if the adversary corrupts some user ID in epoch t and removes them in some later epoch $t' > t$ without calling the update oracle before t', the group secret for epoch t' (and all later epochs for which ID is still not a group member) should be indistinguishable from random. This should hold even though the adversary can save all information ID can derive from their state at epoch t and control messages through time t'.

We also allow the adversary to corrupt out-of-band messages sent to a user in a given epoch by querying the **corrupt-oob** oracle. Note that since we do not explicitly require the out-of-band channel to be implemented in a particular way, the adversary can only corrupt individual messages. However, if for example a PKI is used without PCS after corruptions to implement the out-of-band channel, then the adversary in user-mult can simply repeatedly query the **corrupt-oob** oracle. Our modeling choice abstracts out the corruption model of the out-of-band channel, allowing the adversary to decide the consequences of corruption at a given time.

Avoiding trivial attacks. We prevent against trivial attacks by running the **user-safe** predicate at the end of the game on the queries $\mathbf{q}_1, \ldots, \mathbf{q}_q$ in order to determine if the execution had any such attacks. The predicate checks for every challenge epoch t^* if there is any $\mathsf{ID} \in \mathbf{G}[t^*]$ that was corrupted in epoch $t < t^*$; or who was added in epoch $t < t^*$ and whose corresponding out-of-band message for that add was corrupted. If so, the predicate checks if the user was not removed by the group manager or did not have her secrets updated by the group manager for an operation whose corresponding out-of-band message is not later corrupted, after t and before or during t^*.

If this were true, then the attacker could trivially compute the group secret in the challenge epoch t^* by using the state of corrupted user ID, or her corrupted out-of-band message, in epoch t and the broadcast control messages and/or future corrupted out-of-band messages. The predicate is depicted in Figure 3, which uses the function $\mathsf{q2e}(\mathbf{q})$, that returns the epoch corresponding to query q. Specifically if $\mathbf{q} = \mathbf{corrupt}(\mathsf{ID})$, if ID is a member of the group when $\mathbf{q}$ is made, $\mathsf{q2e}(\mathbf{q})$ corresponds to $\mathsf{ep}[\mathsf{ID}]$, otherwise, $\mathsf{q2e}(\mathbf{q})$ returns $\perp$.[7] If $\mathbf{q} = \mathbf{corrupt\text{-}oob}(t, \mathsf{ID})$, then $\mathsf{q2e}(\mathbf{q}) = t$. For $\mathbf{q} \in \{\mathbf{update\text{-}user}(\mathsf{ID}), \mathbf{remove\text{-}user}(\mathsf{ID})\}$, $\mathsf{q2e}(\mathbf{q})$ is the epoch defined by the operation.

Observe that this predicate achieves optimal security for users. For forward secrecy, if an adversary corrupts a user that is currently at epoch t, for every epoch $0 < t' \leq t$, the group secret is indistinguishable from random. For PCS, if the adversary corrupts a user that is currently at epoch t, or their out-of-band channel at epoch t if they are added in epoch t, once the group manager updates their secrets in some epoch $t' > t$ for which the corresponding out-of-band message is not corrupted (and does the same for every other corrupted user), the group secret is indistinguishable from random. Additionally, double-joins are prohibited, since once a user is removed from the group, even if their state is corrupted before the removal, the group secret of all future epochs in which the user is not a group member are indistinguishable from random.

[7] Any expression containing $\perp$ evaluates to false.

Advantage. In the following, a $(t, q_c, n_{\max})$-*attacker* is an attacker $\mathcal{A}$ that runs in time at most t, makes at most q_c challenge queries, and never produces a group with more than $n_{\max}$ members. The attacker wins the MKA security game user-mult if he correctly guesses the random bit b in the end *and* the safety predicate **user-safe** evaluates to true on the queries made by the attacker. The advantage of $\mathcal{A}$ against MKA scheme M is: $\mathsf{Adv}^{\mathsf{M}}_{\text{user-mult}}(\mathcal{A}) \coloneqq \left|\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}\right|$.

Definition 4 (MKA User Security). *A MKA scheme* M *is* $(s, c, t_1, t_2, t', q_c, n_{\max}, \varepsilon)$-secure in user-mult against adaptive and partially active attackers*, if for all* (t', q_c, n_{max})*-attackers,* $\mathsf{Adv}^{\mathsf{M}}_{\text{user-mult}}(\mathcal{A}) \leq \varepsilon$*, and the complexity measures of the group managers and users are* s, c, t_1, t_2*, as defined in Section 4.2.*

5 MKA Construction

Our MKA construction is similar to LKH [36, 38] described in the introduction, in which the group manager stores for each epoch, the whole tree with the group secret at at the root, keys at all other nodes, and n users at the leaves (with no extra leaves). We call this tree the *MKA tree* and it has $O(n)$ values in total for most commonly used trees. Observe that the group manager should be expected to store $\Omega(n)$ amount of information, since she must minimally be aware of the group members. Users only need to store in their state the secrets along their direct path, which we will show is size $O(\log(n))$ for most *balanced* trees.

5.1 MKA Trees

Below, we describe operations on MKA trees needed for our construction. In addition to modifying the tree, for the purposes of our MKA construction, each operation also returns a skeleton (refer back to Section 2 for a definition of skeleton and frontier). This skeleton consists of any new nodes added to the tree, as well as any nodes in the original tree whose subtree (rooted at that node) has been modified (i.e., a node has been removed from or added to it by the operation), and their edges to each other. Every edge in the skeleton is labeled with a color *green* or *blue* with the requirement that for each node v in the skeleton, at most one of its edges to its children can be colored green. When we specify the cryptographic details of our MKA scheme, if an edge is blue, it means that the secret at the parent will be encrypted under a key at the child, whereas if an edge is green, it means that the secret at the parent will be generated using a dPRF computation on a key at the child.[8] A tree has the following operations:

- *Initialize.* The $\tau_{\mathsf{mka}} \leftarrow \textsc{Init}(\ell_1, \ldots, \ell_n)$ operation initializes τ_{mka} with n leaves labeled by $(\ell_1, \ldots, \ell_n)$ (which will be user IDs).
- *Add.* The $\textsc{Add}(\tau_{\mathsf{mka}}, \ell)$ operation adds a leaf to the tree τ_{mka} labeled by ℓ (ID of added user).
- *Remove.* The $\textsc{Remove}(\tau_{\mathsf{mka}}, \ell)$ operation removes the leaf from the tree τ_{mka} with label ℓ (ID of user to be removed).

MKA Tree efficiency measures. We say a node in a MKA tree is *utilized* in a given epoch if it contains a secret. We assume w.l.o.g., that every interior node of a MKA tree is utilized, since otherwise, a needless efficiency decrease would result. We refer to a MKA tree τ_{mka} as a $(s_{tree}(n, m), s_{skel}(n, m), d(n, m), \deg(\tau_{\mathsf{mka}}))$-*tree* if it has degree of nodes bounded by $\deg(\tau_{\mathsf{mka}})$ and given that it contains at *maximum* m leaves throughout its existence: with n utilized leaves in any configuration it has depth $d(n, m)$, $s_{\text{tree}}(n, m)$ total nodes in the worst case and skeletons formed by Add or Remove operations with total number of nodes $s_{\text{skel}}(n, m)$ in the worst case. In terms of our MKA scheme, $m = n_{\max}$ is the maximum number of users in the group throughout the execution of the protocol and $n = n_{\text{curr}}$ is the number of users in the group at a given epoch. We will often refer to these measures without writing them explicitly as functions of m and n. It will become clear in our construction below that measure s_{tree} upper bounds MKA efficiency measure s and measure s_{skel} upper bounds MKA efficiency measures c, t_1, t_2 as described in Section 4.2 for constant-degree trees.

See Appendix A for the implementation details and efficiency of three types of trees that can be used in our MKA construction: LBBTs, 2-3 trees, and left-leaning red-black trees (LLRBTs) [33]. There we show that LBBTs are $(O(m), O(\log m), O(\log m), 2)$-trees, 2-3 trees are $(O(n), O(\log n), O(\log n), 3)$-trees, and LLRBTs are $(O(n),\ O(\log n), O(\log n),\ 2)$-trees. Figure 4 highlights this difference by demonstrating a removal of a leaf node from each type of tree starting in the same configuration.

[8] Where we use the standard PRF security of a dPRF dprf on the child's key (see Appendix B).

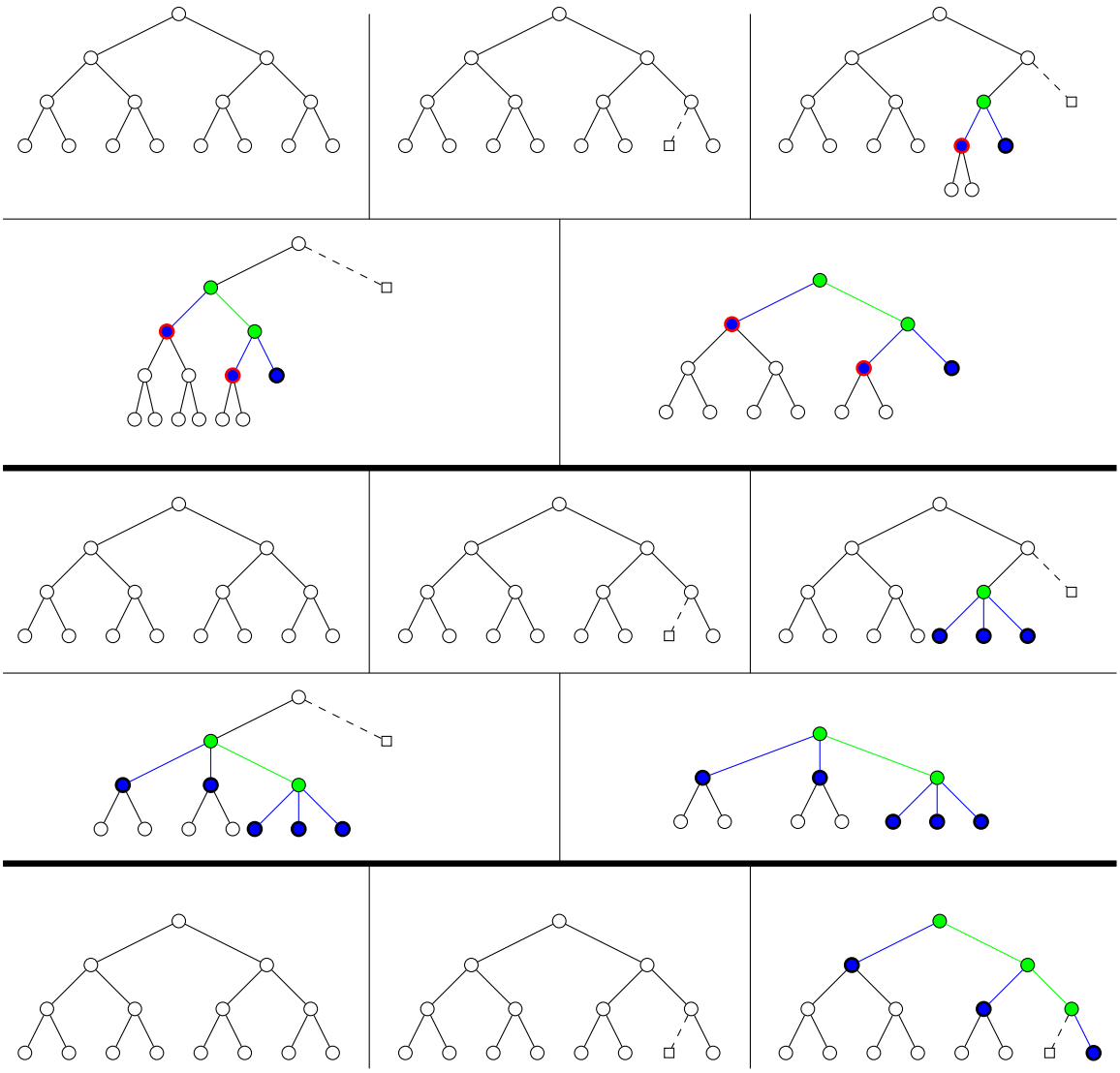

Fig. 4. The top part of the figure shows the deletion of a leaf from a LLRBT. As expected, black LLRBT nodes are drawn with black outline, while red LLRBT nodes are drawn with red outline. Observe that the LLRBT balances itself after the deletion, causing the rightmost leaf to have decreased depth from 3 to 2. The middle shows the deletion of the same leaf from a 2-3 tree with the same configuration. Observe that the height of the tree decreases from 3 to 2. The bottom shows the deletion of the same leaf from a LBBT starting with the same configuration. In the case of the LBBT, the deleted leaf is just marked as removed, and the structure of the tree does not change at all. We also show how the skeletons for the respective operations are constructed: skeleton nodes are colored green and frontier nodes are colored blue. For clarity, we also color edges within the skeleton as green, and edges from skeleton nodes to the frontier as blue to represent the corresponding dPRF computations and encryptions, respectively, in our MKA schemes. One can further observe that although the height of the 2-3 tree shrinks, the number of ciphertexts from the operation (5) is greater than that of the LBBT and LLRBT operations (3).

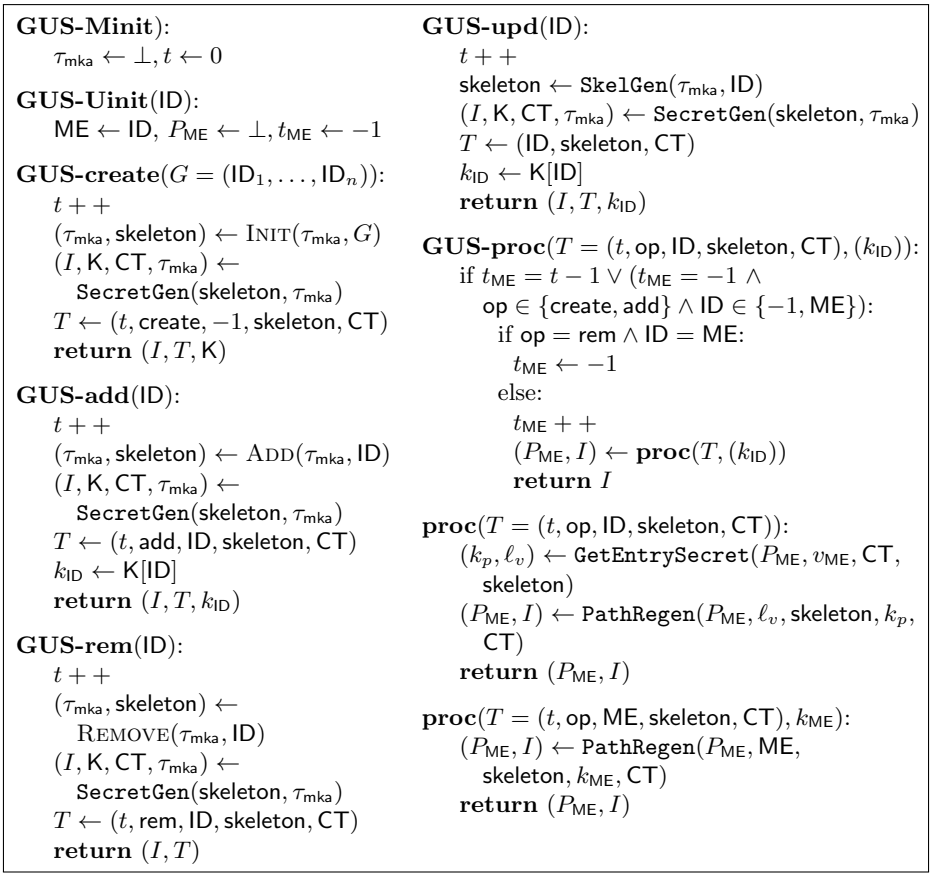

Fig. 5. Generic construction of multicast scheme GUS that achieves security in the user-mult security game.

5.2 GUS MKA protocol

We now describe the cryptographic details of an MKA scheme utilizing a generic tree structure that achieves security with respect to the user-mult game. We denote the scheme GUS (for Generic User Security). GUS makes (black-box) use of a dPRF dprf and an Updatable Symmetric Key Encryption (USKE) scheme uske = (UEnc, UDec).

We use USKE in GUS to obtain forward secrecy: Essentially, each node v in the MKA tree τ_{mka} for a given epoch t will store a key k which the manager can use to communicate to the users at the leaves of the subtree rooted at v information needed to derive the group secret. If the operation for t causes some information to be encrypted to some node key k, k must be refreshed (as in USKE): if k is not refreshed, in epoch $t+1$, an adversary can corrupt any user storing k and break FS by regenerating the group secret for epoch t.

GUS is depicted in Figure 5. Note that the group manager's state only consists of the current epoch t and the MKA tree for the current epoch t, which we denote as τ_{mka}. Each user ID stores the current epoch which they are in, t_{ME}, as well as the secrets of their direct path in τ_{mka} at t_{ME}, which we denote as P_{ID}, and refer to as their *secret path*.

GUS MKA Group Manager Operations. Here we define operations with generic trees for how the group manager creates the group, adds and removes members, and updates key material for members. In initialization, the group manager simply initializes her MKA tree to be empty, and her epoch $t \leftarrow 0$.

Skeleton Secret Generation. We first describe a procedure to generate the secrets and any corresponding ciphertexts for a skeleton. Refer to Figure 4 for a demonstration of the construction of the skeleton and edge coloring for a leaf removal in a LBBT, 2-3 tree, and LLRBT which start in the same configuration. The procedure $(I, \mathsf{K}, \mathsf{CT}, \tau'_{\mathsf{mka}}) \leftarrow \mathtt{SecretGen}(\mathsf{skeleton}, \tau_{\mathsf{mka}})$ first initializes sets $\mathsf{CT}[\cdot] \leftarrow \bot, \mathsf{K}[\cdot] \leftarrow \bot$ then recursively for each node v in skeleton starting with its leaves:

1. If v is a leaf of $\mathsf{skeleton}$ or a node for whom all edges to its children are labeled *blue* in $\mathsf{skeleton}$, the procedure samples a random value k_s and computes $(k_p||k'_{d,v}||k_{e,v}) \leftarrow \mathsf{dprf}(k_s, k_{d,v})$,[9] where $k_{d,v}$ is the current dPRF key at v (if it exists, $\perp$ otherwise), $k_{e,v}$ will be the new encryption key at v, $k'_{d,v}$ will be the new dPRF key at v, and k_p may be used for the parent of v. Then:
   (a) The group manager writes $(k'_{d,v}, k_{e,v}, \ell_v)$ to node v of τ_{mka}, where ℓ_v is either a new value if v is new, or its old value otherwise.
   (b) Additionally, if v was a leaf node of τ_{mka}, the manager sets $\mathsf{K}[\mathsf{ID}] \leftarrow k_s$, for the label ID of v.
2. Otherwise, it uses the key k_s (labeled as k_p above) obtained from the child whose edge from v is *green* and does the same.
3. For every child u of v that has a *blue* edge from v or is a *utilized* node in the frontier of $\mathsf{skeleton}$, it retrieves $(k_{d,u}, k_{e,u}, \ell_u)$ from node u, computes $(k'_{e,u}, c) \leftarrow \mathsf{UEnc}(k_{e,u}, k_s)$, $\mathsf{ct} \leftarrow (c, \ell_v, \ell_u)$, sets $\mathsf{CT}[u] \leftarrow \mathsf{ct}$, and lastly writes $(k_{d,u}, k'_{e,u}, \ell_u)$ to node u.
4. If v is the root, it sets $I \leftarrow \mathsf{dprf}(k_p, \perp)$.

Group Operations. To create a group $G = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$, the manager first increments t then calls $(\tau'_{\mathsf{mka}}, \mathsf{skeleton}) \leftarrow \textsc{Init}(\tau_{\mathsf{mka}}, G)$, which initializes the tree with the IDs in G at its leaves in τ_{mka} and returns the skeleton (which will be the whole tree). Then they call $(I, \mathsf{K}, \mathsf{CT}, \tau''_{\mathsf{mka}}) \leftarrow \mathtt{SecretGen}(\mathsf{skeleton}, \tau'_{\mathsf{mka}})$ to generate secrets and ciphertexts for $\mathsf{skeleton}$, set $T \leftarrow (t, \mathsf{create}, -1, \mathsf{skeleton}, \mathsf{CT})$ and return (I, T, K).

To add a user ID to the group, the manager first increments t then calls $(\tau'_{\mathsf{mka}}, \mathsf{skeleton}) \leftarrow \textsc{Add}(\tau_{\mathsf{mka}}, \mathsf{ID})$, which adds a leaf for ID to τ_{mka}. Then they call $(I, \mathsf{K}, \mathsf{CT}, \tau''_{\mathsf{mka}}) \leftarrow \mathtt{SecretGen}(\mathsf{skeleton}, \tau'_{\mathsf{mka}})$ to generate new secrets and ciphertexts for $\mathsf{skeleton}$, set $T \leftarrow (t, \mathsf{add}, \mathsf{ID}, \mathsf{skeleton}, \mathsf{CT})$ and oob value $k_{\mathsf{ID}} \leftarrow \mathsf{K}[\mathsf{ID}]$, and return (I, T, k_{ID}).

To remove a group member ID from the group, the group manager increments t then calls $(\tau'_{\mathsf{mka}}, \mathsf{skeleton}) \leftarrow \textsc{Remove}(\tau_{\mathsf{mka}}, \mathsf{ID})$, which removes the leaf for ID from τ_{mka}. Then they call $(I, \mathsf{K}, \mathsf{CT}, \tau''_{\mathsf{mka}}) \leftarrow \mathtt{SecretGen}(\mathsf{skeleton}, \tau'_{\mathsf{mka}})$ to generate new secrets and ciphertexts for $\mathsf{skeleton}$, set $T \leftarrow (t, \mathsf{rem}, \mathsf{ID}, \mathsf{skeleton}, \mathsf{CT})$, and return (I, T).

To update the secrets of a group member ID, the group manager first increments t, then, in the procedure $\mathsf{skeleton} \leftarrow \mathtt{SkelGen}(\tau_{\mathsf{mka}}, \mathsf{ID})$, forms the skeleton $\mathsf{skeleton}$, consisting of the nodes on the direct path of v_{ID} (the leaf occupied by ID) and its frontier being the copath of v_{ID}. They do so by traversing the direct path of v_{ID}, and for each node v besides v_{ID}, they color the edge to the child of v on the direct path as green. Then they call $(I, \mathsf{K}, \mathsf{CT}, \tau'_{\mathsf{mka}}) \leftarrow \mathtt{SecretGen}(\mathsf{skeleton}, \tau_{\mathsf{mka}})$ to generate new secrets and ciphertexts for $\mathsf{skeleton}$, set control message $T \leftarrow (t, \mathsf{up}, \mathsf{ID}, \mathsf{skeleton}, \mathsf{CT})$ and out-of-band value $k_{\mathsf{ID}} \leftarrow \mathsf{K}[\mathsf{ID}]$, and return (I, T, k_{ID}).[10]

GUS MKA User Operations. Here we define operations for how group members process changes that the group manager makes to the group, in an effort to recover the group secret. In initialization, users simply set $\mathsf{ME} \leftarrow \mathsf{ID}$ for input ID ID, $P_{\mathsf{ME}} \leftarrow \perp$, and $t_{\mathsf{ME}} \leftarrow -1$.

Path regeneration. When processing operations, users will have to regenerate secret pairs in P_{ID}. The procedure $(P_{\mathsf{ID}}, I) \leftarrow \mathtt{PathRegen}(P_{\mathsf{ID}}, \ell_w, \mathsf{skeleton}, k_s, \mathsf{CT})$:

1. First finds w corresponding to the label ℓ_w in $\mathsf{skeleton}$ and then traverses the direct path of w in $\mathsf{skeleton}$ to the root v_r, while at each node u:
   (a) Computes $(k_p||k'_{d,u}||k_{e,u}) \leftarrow \mathsf{dprf}(k_s, k_{d,u})$, where $k_{d,u}$ is the current encryption key at u (if it exists, $\perp$ otherwise) and writes $(k'_{d,u}, k_{e,u}, \ell_u)$ to u in P_{ID}.
   (b) Then, if the edge in $\mathsf{skeleton}$ from u to its parent v is blue, the procedure:
      i. Retrieves $\mathsf{CT}[u] = (c, \ell_v, \ell_u)$ then computes $(k'_p, k'_{e,u}) \leftarrow \mathsf{UDec}(k_{e,u}, c)$, and writes $(k'_{d,u}, k'_{e,u}, \ell_u)$ to u.
      ii. In this case, k'_p is used at v, i.e. $k_s \leftarrow k'_p$.
   (c) Otherwise, k_p is used at v, i.e. $k_s \leftarrow k_p$.
2. Lastly, it returns $I \leftarrow \mathsf{dprf}(k_p, \perp)$, where k_p was generated at the root v_r.

[9] We only need to use a dPRF at leaves of the MKA tree for updates in which the corresponding oob message is corrupted (we could use a PRG elsewhere), but we use one at all nodes, for all operations, for simplicity.

[10] Note: we do not actually have to include the skeleton, since the tree does not actually change after updates, but we do for ease of exposition. It should not affect the complexity of the scheme since its size should be proportional to that of CT.

Processing Control Messages. A user processing a control message $T = (t, \mathsf{op}, \mathsf{ID}, \mathsf{skeleton}, \mathsf{CT})$ first checks that either 1. $t_{\mathsf{ME}} = t-1$; or 2. $t_{\mathsf{ME}} = -1 \wedge \mathsf{op} \in \{\mathsf{create}, \mathsf{add}\} \wedge \mathsf{ID} \in \{-1, \mathsf{ME}\}$. If not, they stop processing. Otherwise, they first check if $\mathsf{op} = \mathsf{rem} \wedge \mathsf{ID} = \mathsf{ME}$; if so, they set $t_{\mathsf{ME}} \leftarrow -1$ and stop processing.

Otherwise, they increment t_{ME}. Then if $\mathsf{ID} \in \{-1, \mathsf{ME}\}$, that user needs to completely refresh their secret path P_{ME} as a result of processing the operation from which T was generated. Using the dPRF key k_{ME} (sent via an out-of-band channel) for the leaf v_{ME}, the user computes $(P_{\mathsf{ME}}, I) \leftarrow \mathtt{PathRegen}(P_{\mathsf{ME}}, \mathsf{ME}, \mathsf{skeleton}, k_{\mathsf{ME}}, \mathsf{CT})$ and returns I.

Otherwise, if $\mathsf{ID} \notin \{\mathsf{ME}, -1\}$, she needs only change part of her secret path P_{ME} as a result of processing the operation on ID from which T was generated. She:

1. First finds the node u that is on the direct path of her corresponding leaf v_{ME} and is in the frontier of $\mathsf{skeleton}$.
2. Then retrieves the ciphertext $(c, \ell_v, \ell_u) \leftarrow \mathsf{CT}[u]$.
3. Next, retrieves $(k_{d,u}, k_{e,u}, \ell_u)$ from u and computes $(k_p, k'_{e,u}) \leftarrow \mathsf{UDec}(k_{e,u}, c)$ to obtain the dPRF key k_p used at the parent v of u, then writes $(k_{d,u}, k'_{e,u}, \ell_u)$ to u.

We call this operation $(k_p, \ell_v) \leftarrow \mathtt{GetEntrySecret}(P_{\mathsf{ME}}, v_{\mathsf{ME}}, \mathsf{CT}, \mathsf{skeleton})$. They then compute $(P_{\mathsf{ME}}, I) \leftarrow \mathtt{PathRegen}(P_{\mathsf{ME}}, \ell_v, \mathsf{skeleton}, k_p, \mathsf{CT})$ and return I.

5.3 Security of GUS MKA protocol

Theorem 2 (security of GUS). *Let Q be the number of queries an adversary makes to the oracles of the* user-mult *game. Assume* dprf *is a $(t_{\mathsf{dprf}}, \varepsilon_{\mathsf{dprf}})$-secure pseudorandom generator,* uske *is a $(t_{\mathsf{cpa}*}, \varepsilon_{\mathsf{cpa}*})$-CPA*-secure USKE scheme, and τ_{mka} is a $(s_{tree}, s_{skel}, d, \deg(\tau_{\mathsf{mka}}))$-tree. Then,* GUS *is a $(s_{tree}, \deg(\tau_{\mathsf{mka}}) \cdot s_{skel}, \deg(\tau_{\mathsf{mka}}) \cdot s_{skel}, s_{skel}, t, q_c, n_{max}, \varepsilon)$-secure MKA protocol with respect to the* user-mult *security game for $\varepsilon \in \{\varepsilon_{\mathsf{S}}, \varepsilon_{\mathsf{RO}}\}$, where $t \approx t_{\mathsf{dprf}} \approx t_{\mathsf{cpa}*}$. In the standard model, $\varepsilon_{\mathsf{S}} = q_c(Q \cdot \varepsilon_{\mathsf{dprf}} + 2\varepsilon_{\mathsf{cpa}*} \cdot \deg(\tau_{\mathsf{mka}})) \cdot (2\deg(\tau_{\mathsf{mka}}))^d \cdot Q^{(\deg(\tau_{\mathsf{mka}}) \cdot d+1)}$. In the random oracle model, $\varepsilon_{\mathsf{RO}} = q_c(\varepsilon_{\mathsf{cpa}*} \cdot 2(s_{tree} \cdot Q)^2 + \mathsf{negl})$.*

The proof of Theorem 2 is provided in Appendix C.

Corollary 1.

1. *If the tree used in the* GUS *protocol is a LBBT, then* GUS *is a $(O(n_{max}), O(\log n_{max}), O(\log n_{max}), O(\log n_{max}), t, q_c, n_{max}, \varepsilon)$-secure MKA protocol, where $\varepsilon_{\mathsf{S}} = O(q_c n_{max}^2 \cdot Q^{2\log(n_{max})+1} \cdot (Q\varepsilon_{\mathsf{dprf}} + \varepsilon_{\mathsf{cpa}*}))$ and $\varepsilon_{\mathsf{RO}} = O(q_c \varepsilon_{\mathsf{cpa}*} \cdot (n_{max} Q)^2)$.*
2. *If the tree used in the* GUS *protocol is a 2-3 tree or LLRBT, then* GUS *is a $(O(n_{curr}), O(\log n_{curr}), O(\log n_{curr}), O(\log n_{curr}), t, q_c, n_{max}, \varepsilon)$-secure MKA protocol, where $\varepsilon_{\mathsf{S}} = O(q_c n_{curr}^3 \cdot Q^{2\log(n_{curr})+1} \cdot (Q\varepsilon_{\mathsf{dprf}} + \varepsilon_{\mathsf{cpa}*}))$ and $\varepsilon_{\mathsf{RO}} = O(q_c \varepsilon_{\mathsf{cpa}*} \cdot (n_{max} Q)^2)$.*

Proof. In Appendix A, we show that LBBTs are $(n_{\max}, \log n_{\max}, \log n_{\max}, 2)$-trees, 2-3 trees are $(n_{\text{curr}}, \log n_{\text{curr}}, \log n_{\text{curr}}, 3)$-trees and LLRBTs are $(n_{\text{curr}}, \log n_{\text{curr}}, \log n_{\text{curr}}, 2)$-trees. The results follow easily from this. □

5.4 Comparison of Trees in GUS

Here we compare the performance of GUS using LBBTs, 2-3 trees, and *restricted* LLRBTs empirically (see Appendix A for the details of these trees; we use restricted instead of normal LLRBTs simply because they give empirically slightly better performance).[11] We simulate GUS with the different trees on a randomized sequence of Add/Remove/Update operations, starting from a certain initial group size. The measure we adopt is the number of encryptions per operation, which encapsulates the goal of reducing communication for bandwidth-constrained devices.

Following [3] in the TreeKEM context, the scales of the simulations we conduct are represented by tree sizes $2^3, 2^4, \ldots, 2^{14}$; for simulation with tree size 2^i, the initial group size is 2^{i-1}, and the number of operations is $10 \cdot 2^i$, where each operation is sampled to be an Add/Remove/Update operation with ratio 1:1:8 (and the user to be removed/updated is chosen uniformly at random from current group members), as one expects updates to be more frequent.

[11] See `https://github.com/abienstock/Multicast-Key-Agreement` for the code.

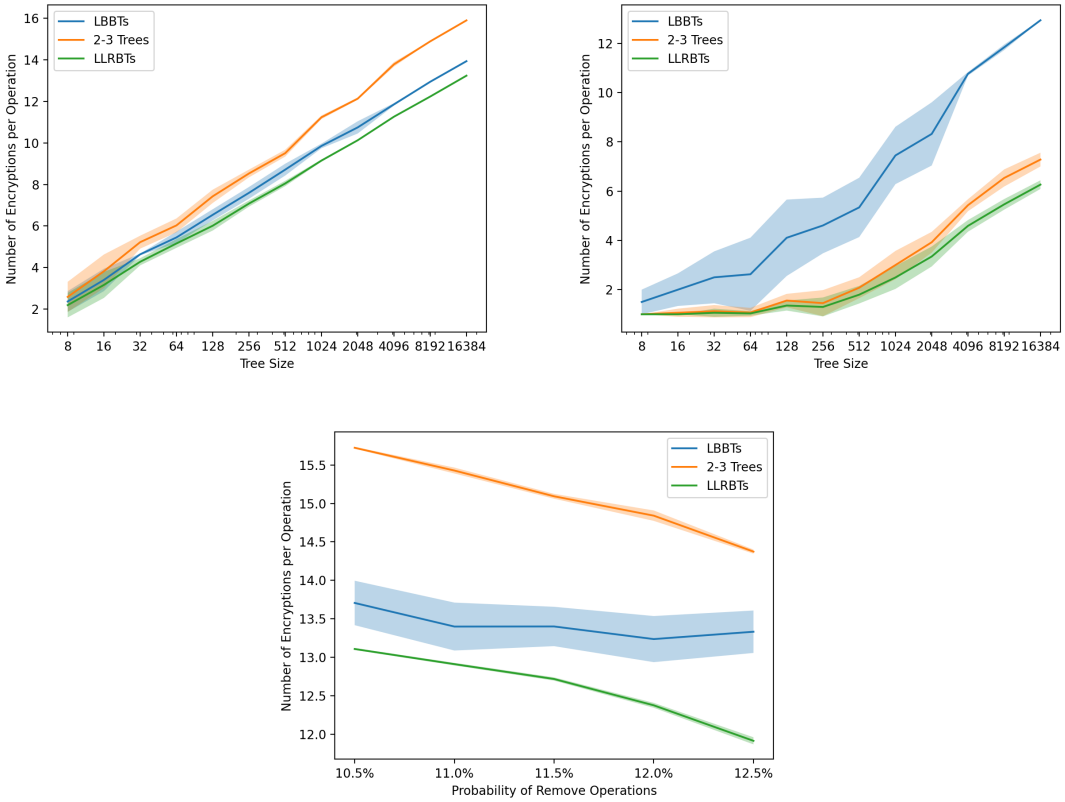

Fig. 6. Top Left: number of encryptions per operation of GUS using different trees in simulations with different initial tree sizes. Top Right: same as top left, except that at the beginning of simulations a random choice of 99% of the users are removed. Bottom: number of encryptions per operation of GUS using different trees in simulations with initial tree size 2^{14} and different probabilities of Remove operations. The bands illustrate one standard deviation under 8 independently repeated experiments.

As shown in the graph at the top left of Figure 6, although 2-3 trees have better worst-case complexity $O(\log n_{\text{curr}})$ than the complexity $O(\log n_{\max})$ of LBBTs, they empirically suffer, likely from the large overhead introduced by degree-3 nodes. Moreover, although restricted LLRBTs are isomorphic to 2-3 trees, they lead to dramatically improved efficiency in terms of number of encryptions, outperforming both 2-3 trees and LBBTs in a stable manner, likely due to improved asymptotic complexity and also retention of degree-2 nodes.

To see the difference between worst-case $O(\log n_{\text{curr}})$ and $(\log n_{\max})$ complexity, we repeated the simulations, but this time before performing the operations with 1:1:8 ratio, we first remove a random choice of 99% of the users from the group (and consequently the number of operations becomes $0.1 \cdot 2^i$). Removing a large number of users can happen in practice, e.g., after a popular event or livestream has concluded. As shown in the graph at the top right of Figure 6, this time both 2-3 trees and LLRBTs, which have worst-case complexity $O(\log n_{\text{curr}})$, have better performance in accordance with the decrease of the group size, while the performance of LBBTs, which have worst-case complexity $O(\log n_{\max})$, does not improve even once the group becomes 99% smaller.

To provide more evidence that $O(\log n_{\text{curr}})$ instead of $O(\log n_{\max})$ complexity makes a difference, we conduct additional simulations with initial tree size 2^{14} while increasing the probability of Remove operations from the previous 10% to as large as 12.5% (and decreasing the probability of Add operations so that the 80% probability of Update operations remains). Thus, in these experiments the group size

has a decreasing trend during the execution of the operations.[12] As can be observed in the graph at the bottom of Figure 6, both 2-3 trees and LLRBTs (complexity $O(\log n_{\text{curr}})$) benefit from the decreasing trend of group size, while LBBTs (complexity $O(\log n_{\max})$) hardly benefit. Besides, the variance when using LBBTs is dramatically larger than 2-3 trees and LLRBTs, which verifies that the performance of LBBTs is highly sensitive to the actual choices of removed users and cannot directly benefit from the decreasing trend of group size. Moreover, the performance of LLRBTs is stable and is better than that of LBBTs by roughly at least twice the huge standard deviation, which further verifies that LLRBTs outperform LBBTs in a stable manner.

6 Adding Security for Group Manager Corruptions

In this section, we introduce a security definition that requires both security with respect to user corruptions, and FS and eventual PCS for group manager corruptions: If the secret state of the group manager is leaked at some point, all previous group secrets remain hidden from the attacker. Furthermore, once every user has their secrets updated by the group manager or is removed from the group after her corruption, all future group secrets remain hidden.

Before we formally define this security, we observe that our GUS construction indeed already achieves it – the MKA tree only stores unused USKE keys and thus the adversary learns nothing about old group secrets from corrupting the group manager (eventual PCS also trivially follows). However, the group manager has large *local* storage in GUS (the whole MKA tree), which is undesirable. Therefore, in Section 8 we slightly modify our GUS construction to create our GMS (Generic Manager Security) construction, which has the above security, but $O(1)$ *local* storage and $O(n)$ *remote* storage for the group manager. We show how the group manager can use Forward Secret Encrypted RAM [10] (FS eRAM) in the GUS MKA scheme to achieve this. Intuitively, FS eRAM allows a client to securely outsource data storage to some remote server such that if the client's secret storage is leaked, in addition to the outsourced (encrypted) data, then all previously overwritten data remains secure. Canonical FS eRAM schemes allow for $O(1)$ local storage, $O(n)$ remote storage, and only introduce $O(\log n)$ overhead for Read() and Write() operations. Thus, GMS simply uses FS eRAM to outsource storage of the MKA tree, and does everything else as in GUS, so that corruptions of the group manager are forward secret (and eventual PCS trivially follows). Therefore, group manager local storage is $O(1)$, group manager remote storage is $O(n)$, computational complexity of the group manager becomes $O(\log^2 n)$,[13] and all other efficiency measures from GUS stay the same (namely $O(\log n)$ communication).

Although eventual PCS as defined in the security definition may be less than ideal, it allows us to use only symmetric-key encryption for control messages. We in fact show in Section 7 that public-key encryption is necessary for optimal PCS, i.e., security after one operation following a group manager corruption. Then, we show that achieving optimal PCS (and all the other properties described above) with public-key encryption is easy: In our construction, we simply replace USKE with UPKE (and the group manager only remotely stores the public keys at the tree nodes; with no local storage).

6.1 Group Manager State Separation and Efficiency Measures

For such added properties, we separate the state MKA group manager state Γ into two components: the secret (local) state, Γ_{sec}, which the attacker can only read upon state corruption of the group manager, and public (remote) state Γ_{pub}, for which the attacker *always* has read access. We will use $s_1(n_{\text{curr}}, n_{\max})$ to refer to the worst-case Γ_{sec} space complexity in a group with n_{curr} users currently and $n_{\max}$ maximum users across all epochs of the protocol execution. We will also use $s_2(n_{\text{curr}}, n_{\max})$ to refer to the corresponding worst-case Γ_{pub} space complexity.

6.2 MKA Security with Group Manager FS and Eventual PCS

To obtain security with respect to group manager corruptions, in addition to the security captured by user-mult, we create a new security game mgr-mult, in which we add to the user-mult game an additional

[12] Note that 12.5% is the threshold such that the group size does not approach zero in expectation, as the number of operations $10 \cdot 2^i$ is 20 times more than the initial group size 2^{i-1} and thus the difference between the probabilities of Remove and Add operations needs to be less than $1/20 = 5\%$.

[13] In Section 8, we also show how to retain GUS's $O(\log n)$ computational complexity.

oracle **mgr-corrupt**, which simply returns the secret state of the group manager to the adversary. Observe that there are now more trivial attacks which the adversary can use to win mgr-mult (e.g., corrupting the group manager and then challenging before every user has been updated or removed from the group). We therefore check a new **mgr-safe** predicate at the end of the game on the queries $\mathbf{q}_1, \ldots, \mathbf{q}_q$ in order to determine if the execution had any trivial attacks. In addition to that which **user-safe** checks for, **mgr-safe** also checks for every challenge epoch t^* if the group manager was corrupted in some epoch $t < t^*$, and there was any $\mathsf{ID} \in G[t]$ that did not have its secrets updated by the group manager (in an operation for which the oob to ID is not corrupted), and was not removed by the group manager, after the corruption, but before t^*.

mgr-corrupt():	**mgr-safe**($\mathbf{q}_1, \ldots, \mathbf{q}_q$):
return Γ_{sec}	for (i, j) s.t. $\mathbf{q}_i = \textbf{mgr-corrupt}()$, $\mathbf{q}_j = \textbf{chall}(t^*)$ for some t^*:
	if $\mathsf{q2e}(\mathbf{q}_i) < t^*$ and $\exists \mathsf{ID} \in G[\mathsf{q2e}(\mathbf{q}_i)]$ s.t. $\nexists l$ s.t.
	$0 < \mathsf{q2e}(\mathbf{q}_i) < \mathsf{q2e}(\mathbf{q}_l) \leq t^* \wedge ((\mathbf{q}_l = \textbf{update-user}(\mathsf{ID}) \wedge$
	$\nexists m$ s.t. $\mathbf{q}_m = \textbf{corrupt-oob}(\mathsf{q2e}(\mathbf{q}_l), \mathsf{ID})) \vee$
	$\mathbf{q}_l = \textbf{remove-user}(\mathsf{ID}))$:
	return 0
	return **user-safe**($\mathbf{q}_1, \ldots, \mathbf{q}_q$)

Fig. 7. Oracle and **mgr-safe** safety predicate introduced in the MKA security game mgr-mult for a scheme $\mathsf{M} = (\mathsf{Minit}, \mathsf{Uinit}, \mathsf{create}, \mathsf{add}, \mathsf{upd}, \mathsf{proc})$ to capture forward secrecy and eventual PCS for the group manager, as well as prevent trivial attacks.

Figure 7 denotes both of these changes. The predicate **mgr-safe** uses q2e() to additionally return the epoch corresponding to a query to $\mathbf{q} = \textbf{mgr-corrupt}()$. In this case, $\mathsf{q2e}(\mathbf{q})$ corresponds to the epoch t that the group manager is in when the query is made (i.e., the epoch defined by the most recent group manager operation **create-group**, **add-user**, **remove-user**, **update-user**).

Advantage. A $(t, q_c, n_{\max})$-*attacker* $\mathcal{A}$ and $\mathsf{Adv}^{\mathsf{M}}_{\mathsf{mgr\text{-}mult}}(\mathcal{A})$ are defined in the same manner as in Section 4.3.

Definition 5 (MKA Security with Group Manager FS and Eventual PCS). *A MKA scheme* M *is* $(s_1, s_2, c, t_1, t_2, t', q_c, n_{\max}, \varepsilon)$-secure in mgr-mult against adaptive, partially active attackers*, if for all* (t', q_c, n_{max})*-attackers,* $\mathsf{Adv}^{\mathsf{M}}_{\mathsf{mgr\text{-}mult\text{-}na}}(\mathcal{A}) \leq \varepsilon$*, and the efficiency measures of the group manager and users are* s_1, s_2, c, t_1, t_2 *as defined in Sections 4.2 and 6.1.*

7 Necessity of Public Key Cryptography for Optimal PCS of Group Manager Corruptions

In our mgr-mult security definition of Section 6, we only provide *eventual* PCS for group manager corruptions. We do so to allow the group manager to continue to use symmetric-key encryption for control messages. Indeed, in this section, we informally show that to obtain optimal PCS for group manager corruptions, i.e., security following a single operation after such a corruption, public-key encryption (PKE) is necessary. Moreover, we suggest a simple modification to GMS (of Section 8) to achieve optimal PCS by using UPKE instead of USKE.

Lemma 1 (Informal). *If optimal PCS is obtained for group manager corruptions in an MKA scheme, then a two-party key agreement protocol (and thus PKE) can be constructed from the algorithms of the manager.*

We first informally define a two-party key agreement:

Two-party key agreement (informal). A two-party key agreement between two parties, Alice and Bob, is a protocol in which Alice and Bob share no secret randomness upon initialization, but upon sampling secrets and communicating publicly with one another, they are able to agree on a shared secret. Moreover, any adversary, Eve, who can see their public communication only has negligible advantage in distinguishing the shared secret from random.

Proof (sketch). Assume that the size of the group of the MKA scheme is two. Refer to these two members as $\mathcal{A}$ and $\mathcal{B}$. From the group manager, $\mathcal{M}$, $\mathcal{A}$, and $\mathcal{B}$, we will construct key exchange algorithms $\mathcal{C}$ and $\mathcal{D}$:

1. $\mathcal{C}$ creates a group with $\mathcal{A}$ and $\mathcal{B}$, then sends the entire group manager state Γ to $\mathcal{D}$.
2. $\mathcal{D}$, upon reception of Γ, runs the update algorithm of $\mathcal{M}$ for $\mathcal{A}$ to obtain group secret I and control message T, which it sends to $\mathcal{C}$.
3. Finally, $\mathcal{C}$ processes T using $\mathcal{B}$'s secret state to obtain I.

By correctness of the protocol M, the secret I which $\mathcal{C}$ and $\mathcal{D}$ obtain is in fact the same. Moreover, if optimal PCS with respect to group manager corruptions is achieved by the MKA scheme, then even though the adversary sees Γ (and T), no attacker $\mathcal{A}$ has a non-negligible advantage of distinguishing I from random. □

To augment our GMS scheme (of Section 8) to additionally achieve optimal PCS for group manager corruptions (in addition to FS for group manager corruptions and optimal security with respect to user corruptions as presented in mgr-mult), we simply replace the USKE keys at each MKA tree node with UPKE key pairs as in [1]. The group manager then only stores the public keys of the tree nodes remotely and for each operation encrypts new secrets for the tree to the corresponding public keys as in GMS. This augmented scheme achieves the same asymptotic efficiency measures as GMS.

8 GMS MKA protocol

In this section, we give the formal details for our MKA construction GMS, secure with respect to the mgr-mult security game defined in Section 6, and with small group manager local storage. We start by formally defining FS eRAM and providing a secure and efficient scheme for it.

8.1 Forward Secret Encrypted RAM Definition

We formally define the syntax and security game for Forward Secret encrypted RAM, which one can use to securely store and retrieve outsourced data from a remote server in a forward secret manner.

Forward Secret encrypted RAM Syntax. The syntax allows a user to initialize, read, and write to the FS eRAM using a master key MK.

Definition 6 (Forward Secret Encrypted RAM). *A* Forward Secret encrypted RAM *scheme* $\mathsf{eram} = (\mathsf{eram\text{-}init}, \mathsf{eram\text{-}read}, \mathsf{eram\text{-}write})$ *consists of the following algorithms:*

- $(\mathrm{M}, \mathrm{MK}) \leftarrow \mathsf{eram\text{-}init}(1^\lambda)$*, which initializes the public RAM cells* M *and generates a master key* MK*.*
- $(\mathrm{M}', \mathrm{MK}', d) \leftarrow \mathsf{eram\text{-}read}(\mathrm{M}, \mathrm{MK}, i)$*, which returns data* d *of virtual cell* i*.*
- $(\mathrm{M}', \mathrm{MK}') \leftarrow \mathsf{eram\text{-}write}(\mathrm{M}, \mathrm{MK}, d, i)$*, which replaces the contents of virtual cell* i *with data* d*.*

It can be the case that $d = \perp$ for deletion when $\mathsf{eram\text{-}write}(\mathrm{M}, \mathrm{MK}, d, i)$ is used. For simplicity, we will often use $\mathsf{eram\text{-}read}(\mathrm{M}, \mathrm{MK}, i)$ and $\mathsf{eram\text{-}write}(\mathrm{M}, \mathrm{MK}, d, i)$ in a manner that *implicitly* changes M and MK.

Forward Secret encrypted RAM Efficiency Measures. We provide three measures of efficiency for forward secret encrypted RAMs. All three measures will be in terms of n_{curr}, the number of virtual cells which the user has written data other than $\perp$ to at a given point in time, and $n_{\max}$, the maximum number of such cells at any point in the protocol execution. The first is worst case space complexity of MK, which we refer to as $s_1(n_{\mathrm{curr}}, n_{\max})$. The second, $s_2(n_{\mathrm{curr}}, n_{\max})$, is the worst case space complexity of M, i.e. the total number of cells in M that do not contain $\perp$. The third, $t(n_{\mathrm{curr}}, n_{\max})$, is the worst case time complexity of eram-read and eram-write operations. We will often refer to these measures without writing them explicitly as functions of n_{curr} and $n_{\max}$.

Forward Secret encrypted RAM Correctness. An FS eRAM scheme is *correct* if for any sequence of operations:

$$\Pr[(\cdot,\cdot,d^*) \leftarrow \mathsf{eram\text{-}read}(\mathrm{M},\mathrm{MK},i) : d^* = d] = 1,$$

for any execution of $\mathsf{eram\text{-}read}(\mathrm{M},\mathrm{MK},i)$ in the sequence after an execution of $\mathsf{eram\text{-}write}(\mathrm{M},\mathrm{MK},d,i)$, with $d \neq \perp$ and before a subsequent execution of $\mathsf{eram\text{-}write}(\mathrm{M},\mathrm{MK},d',i)$, for some data d', in the sequence where the probability is over the random coin tosses of the protocol.

Forward Secret encrypted RAM Security. We define security with respect to the following game between a challenger and an adversary. We emphasize that the adversary has read-access to *all* of M (which is usually encrypted) on which the FS eRAM operates.

The challenger initially chooses $b \in \{0,1\}$ uniformly and runs $(\mathrm{M},\mathrm{MK}) \leftarrow \mathsf{eram\text{-}init}(1^\lambda)$. Then, the adversary has access to the following oracles:

- **write**(d,i), which computes $\mathsf{eram\text{-}write}(\mathrm{M},\mathrm{MK},d,i)$.
- **corrupt**(), which simply returns MK.
- **chall**(d_0,d_1,i), which computes $\mathsf{eram\text{-}write}(\mathrm{M},\mathrm{MK},d_b,i)$ with the requirement that $d_0 \neq \perp$, $d_1 \neq \perp$.

An adversary is not allowed to call **corrupt**() after a call to **chall**(d_0,d_1,i), without first using **write**(d,i) to overwrite the i-th virtual cell with some other data d, since otherwise they would trivially win. Observe that w.l.o.g. there is no read oracle since the adversary already knows the data in cells which they filled using **write**(), and should not know the data in cells filled via **chall**(). Further observe that this definition provides forward secrecy, since upon a corruption, any data previously written to any virtual cell should be hidden.

Advantage. In the following, a $(t', n_{\max})$-attacker is an attacker $\mathcal{A}$ that runs in time at most t' and never fills more than $n_{\max}$ virtual cells with data (not equal to $\perp$). The attacker wins the forward secret encrypted RAM security game if she correctly guesses the random bit b in the end. The advantage of $\mathcal{A}$ against a forward secret encrypted RAM scheme eram is defined by

$$\mathsf{Adv}^{\mathsf{eram}}_{\mathsf{enc\text{-}ram}}(\mathcal{A}) := |\Pr[\mathcal{A} \to 1 | b = 1] - \Pr[\mathcal{A} \to 0 | b = 1]|$$

Definition 7 (Forward secret encrypted RAM security). *A forward secret encrypted RAM protocol* R *is* $(s_1, s_2, t, t', n_{\max}, \varepsilon)$-secure*, where* s_1, s_2, t *are the efficiency measures discussed above, if for all* (t', n_{max})*-attackers,*

$$\mathsf{Adv}^{\mathsf{R}}_{\mathsf{enc\text{-}ram}}(\mathcal{A}) \leq \varepsilon.$$

8.2 FS eRAM Construction tRAM

We now formally construct a dynamic forward secret encrypted RAM scheme tRAM utilizing the following cryptographic primitives: a pseudorandom generator prg, and a CPA-secure symmetric key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

We use the same high-level idea as in the folklore constructions [10, 28, 30, 31, 4, 32, 16, 19, 11], wherein the public RAM cells M are arranged in a generic *encryption tree* τ_{eram}. Each of the interior nodes in M will hold an (encrypted) symmetric key which encrypts the keys (or data) of its two children, and is encrypted by its parent, so that the contents of all of the cells in M are ciphertexts encrypting their associated data.[14] The master key MK allows the protocol to decrypt the key at the root r of τ_{eram}. Figure 8 shows an example configuration of tRAM for $n_{\mathrm{curr}} = 4$.

The same Init, Add, and Remove operations as those defined in section 5.1 for MKA trees are also used for τ_{eram}, with the following modifications:

- they use a one-to-one correspondence of node labels with RAM cell identifiers, so that a node with label i is stored in cell i,
- Add and Remove operations return not only a new skeleton $\mathsf{skeleton}$, as before, but also an old skeleton $\mathsf{skeleton}'$ defined below, and

[14] We assume the forward secret encrypted RAM scheme can arrange these cells in accordance with the generic tree operations using unencrypted pointers, for example.

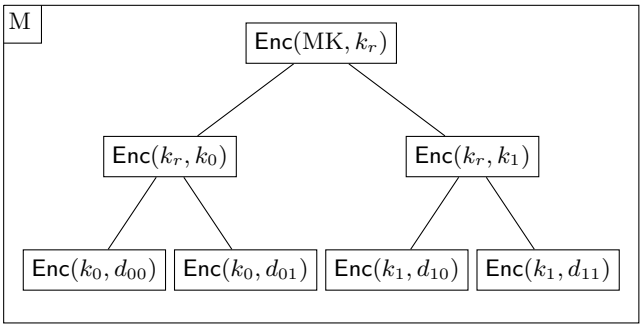

Fig. 8. An example of our forward secret encrypted RAM construction with four virtual cells. The four leaf cells of τ_{eram} hold the (encrypted) user data and the interior cells hold keys encrypted by the keys at their parents in M. Using MK, the protocol can decrypt down to the leaves to access the data.

– for ADD operations, the color of the edge in skeleton from the added leaf to its parent is always blue.

The skeleton skeleton$'$ consists of the old nodes in τ_{eram} before the operation that had edges to nodes in the frontier of skeleton, along with all of their ancestors, up to the root, and all edges between nodes. For the INIT operation, skeleton$'$ = skeleton. The existence of skeleton$'$ will allow the scheme to find and decrypt the keys of the frontier of skeleton in τ_{eram}, before encrypting them under the new keys of their parents. We note that all tree operations above are performed on the public cells M of the encrypted RAM.

Encryption Tree efficiency measures. To provide full flexibility for generic encryption trees, we allow the symbol ϵ to be written to nodes of τ_{eram} by the construction to indicate *empty* nodes that are not deleted from the tree, but do not hold any data.[15] We say a node in an encryption tree is *utilized* if it holds data $d \notin \{\perp, \epsilon\}$. We assume w.l.o.g., that every interior node of a MKA tree is utilized, since otherwise, a needless efficiency decrease would result.

Encryption trees have very similar efficiency measures to those of MKA trees, where here m is the *maximum* number of leaves used throughout an encryption tree's existence (with data other than $\perp$) and n refers to the number of utilized leaves in the tree, in any configuration, at a given time. Thus $s_{\text{tree}}(n, m)$ refers to the total number of nodes in the worst case with n utilized leaves in any configuration, and $s_{\text{skel}}(m, n)$ refers to the total number of nodes in skeleton formed by ADD and REMOVE operations, in the worst case. We also add measure $s'_{\text{skel}}(n, m)$, which refers to the total number of nodes in skeleton$'$ formed by ADD and REMOVE operations, in the worst case. Observe that $s'_{\text{skel}}(n, m) \approx s_{\text{skel}}(n, m)$ for many commonly used trees, including LBBTs, 2-3 trees, and LLRBTs. With these measures, we define $(s_{tree}, s_{skel}, s'_{skel}, \deg(\tau_{\mathsf{eram}}))$-*tree*s, where $\deg(\tau_{\mathsf{eram}})$ is a bound on the degree of nodes in τ_{eram}, and as usual, we do not explicitly write the measures as functions of m and n. In terms of our encrypted RAM scheme, $m = n_{\max}$ is the maximum number of virtual cells that are used to store data $d \neq \perp$ throughout the execution of the scheme and $n = n_{\text{curr}}$ is the number of utilized virtual cells at a given point in time.

Construction Specifications. In the specification, we will use the notation ct_v to refer to the ciphertext stored in M at the node v of τ_{eram}. The details of the scheme are as follows:

– $(\text{M}, \text{MK}) \leftarrow \mathsf{eram\text{-}init}(1^\lambda)$ initializes the public cells M with $\perp$, and $\text{MK} \leftarrow \perp$.
– $d \leftarrow \mathsf{eram\text{-}read}(\text{M}, \text{MK}, i)$ fetches the direct path of cell i from the server and performs the following:
  1. $k_r \leftarrow \mathsf{Dec}(\text{MK}, \mathsf{ct}_r)$, where r is the root of τ_{eram} (if τ_{eram} is a single node, then actually $d = k_r$ and we return d).
  2. Then recursively, $k_v \leftarrow \mathsf{Dec}(k_w, \mathsf{ct}_v)$ at the nodes v along the direct path of i, using k_w at parent w of v, until the key k_p for the parent p of cell i is obtained. If $\mathsf{ct}_v = \perp$ at any point, return $\perp$.
  3. Returns $d \leftarrow \mathsf{Dec}(k_p, \mathsf{ct}_i)$.
– $\mathsf{eram\text{-}write}(\text{M}, \text{MK}, d, i)$ performs the following:

[15] For example, a leaf node marked as `removed` in an encryption tree using a LBBT structure will hold data ϵ.

1. If $d = \perp$, executes $(\tau_{\mathsf{eram}}, \mathsf{skeleton}, \mathsf{skeleton}') \leftarrow$ $\textsc{Remove}(\tau_{\mathsf{eram}}, i)$.
2. Otherwise, if i is not already in τ_{eram}:
   - $(\tau_{\mathsf{eram}}, \mathsf{skeleton}, \mathsf{skeleton}') \leftarrow \textsc{Init}(\tau_{\mathsf{eram}}, i)$, if not done before.
   - Otherwise, it adds i to τ_{eram}: $(\tau_{\mathsf{eram}}, \mathsf{skeleton}, \mathsf{skeleton}') \leftarrow \textsc{Add}(\tau_{\mathsf{eram}}, i)$
3. Otherwise, it builds $\mathsf{skeleton} = \mathsf{skeleton}'$ itself, namely the skeleton consisting of the direct path of the cell i, where its frontier is the copath of i, and each edge within the skeleton is colored green (except blue to i).
4. Then it fetches $\mathsf{skeleton}$ and $\mathsf{skeleton}'$ from the server and computes $\mathsf{skeleton\text{-}modify}(\tau_{\mathsf{eram}}, \mathsf{skeleton}, \mathsf{skeleton}', d, i)$.

– $\mathsf{skeleton\text{-}modify}(\tau_{\mathsf{eram}}, \mathsf{skeleton}, \mathsf{skeleton}', d, i)$ sets $D[\cdot] \leftarrow \perp, D[i] \leftarrow d$, then:

1. If the corresponding $\mathsf{eram\text{-}write}()$ operation did not initialize τ_{eram}, decrypts the keys of nodes v on the frontier of the skeletons by executing:
   - $k_r \leftarrow \mathsf{Dec}(\mathrm{MK}, \mathsf{ct}_r)$, where r is the root of τ_{eram}.
   - Then recursively $D[v] \leftarrow \mathsf{Dec}(k_w, \mathsf{ct}_v)$ at the the nodes v in $\mathsf{skeleton}'$ and its frontier using k_w at the parent w of v, until the pre-existent data d_j, $j \in \deg[p]$, of cell i and its siblings are obtained. If $\mathsf{ct}_v = \perp$ for any v, set $D[v] \leftarrow \perp$.
2. Then recursively for each node v in $\mathsf{skeleton}$ (sans the new leaf in cell i):
   - If v is a leaf node of the skeleton, or for whom all edges to its children are labeled blue, the procedure samples a random seed s and computes $\mathsf{prg}(s) = (s'||k_v)$, where s' may be used by its parent to generate its key, and sets $D[v] \leftarrow k_v$.
   - Otherwise, it uses the seed s obtained from the child whose edge from v is green and does the same.
   - For every child u of v, it writes $\mathsf{Enc}(k_v, D[u])$ to node u in M.
3. The new seed s' that is generated at the root r of τ_{eram} is again used to generate a new master key $\mathrm{MK} \leftarrow \mathsf{prg}(s')$ (if τ_{eram} is a single node, we just sample s' uniformly at random).
4. Finally, $\mathsf{Enc}(\mathrm{MK}, k_r)$ is written to the root in M.

For the purposes of Theorem 3 below, the skeleton associated with computation $\mathsf{eram\text{-}read}(\mathrm{M}, \mathrm{MK}, i)$ contains those nodes on the direct path of cell i.

Security of tRAM Protocol.

Theorem 3 (Security of tRAM). *Assume that* prg *is a* $(t_{\mathsf{prg}}, \varepsilon_{\mathsf{prg}})$*-secure pseudorandom generator,* Π *is a* $(t_{\mathsf{cpa}}, \varepsilon_{\mathsf{cpa}})$*-CPA-secure symmetric-key encryption scheme, and* τ_{eram} *is a* $(s_{tree}, s_{skel}, s'_{skel}, \deg(\tau_{\mathsf{eram}}))$*-tree. Then,* tRAM *is a* $(O(1), s_{tree}, (s_{skel} + s'_{skel}) \cdot \deg(\tau_{\mathsf{eram}}), t, n_{max}, \varepsilon)$*-secure forward secret encrypted RAM protocol for* $\varepsilon = q s_{tree}(n_{max}, n_{max})(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}})$*, where* q *is the number of times* **write**() *or* **chall**() *is queried by* $\mathcal{A}$ *and* $t \approx t_{\mathsf{prg}} \approx t_{\mathsf{cpa}}$*.*

The proof of Theorem 3 is provided in Appendix D.

Corollary 2.

1. *For LBBT* τ_{eram}*,* tRAM *is a* $(O(1), O(n_{max}), O(\log n_{max}), t, n_{max}, \varepsilon)$*-secure forward secret encrypted RAM protocol, where* $\varepsilon = O(q n_{max}(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}}))$*.*
2. *For 2-3 tree or LLRBT* τ_{eram}*,* tRAM *is a* $(O(1), O(n_{curr}), O(\log n_{curr}), t, n_{max}, \varepsilon)$*-secure forward secret encrypted RAM protocol, where* $\varepsilon = O(q n_{max}(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}}))$*.*

Proof. In Appendix A, we show that LBBTs are $(n_{\max}, \log n_{\max}, 2)$-trees, 2-3 trees are $(n_{\mathrm{curr}}, \log n_{\mathrm{curr}}, 3)$-trees, and LLRBTs are $(n_{\mathrm{curr}}, \log n_{\mathrm{curr}}, 2)$-trees. It is also easy to show that they are indeed $(n_{\max}, \log n_{\max}, \log n_{\max}, 2)$-, $(n_{\mathrm{curr}}, \log n_{\mathrm{curr}}, \log n_{\mathrm{curr}}, 3)$, and $(n_{\mathrm{curr}}, \log n_{\mathrm{curr}}, \log n_{\mathrm{curr}}, 3)$-trees, respectively. The results follow easily from this. □

8.3 GMS Construction

In this section, we discuss how to compose our GUS MKA protocol, secure with respect to the user-mult game, with FS eRAM to obtain a new MKA scheme, GMS, secure with respect to the mgr-mult security game. Such a composition results in $\Gamma_{\mathsf{sec}} = \mathrm{MK}$, $\Gamma_{\mathsf{pub}} = \mathrm{M}$ for GMS, where MK is the master key and M is the public cells of the FS eRAM scheme.

In GMS, the virtual cells of the FS eRAM scheme that the group manager interacts with contain the nodes of τ_{mka}. Now, when retrieving and writing values at a node v of τ_{mka} in GUS, the group manager executes $d \leftarrow \mathsf{eram\text{-}read}(\mathrm{M}, \mathrm{MK}, v)$, $\mathsf{eram\text{-}write}(\mathrm{M}, \mathrm{MK}, d, v)$, respectively, in GMS. Observe that this change does not affect the view of the users and so they do not have to change their behavior at all. However, one must also observe that the computational complexity of all operations is increased, as the $\mathsf{eram\text{-}read}(), \mathsf{eram\text{-}write}()$ encrypted RAM operations to access and write data to the RAM have their own non-negligible computational complexity. In the case of LBBTs, 2-3 trees, and LLRBTs, if we use our tRAM encrypted RAM scheme, computational complexity of add, update, and remove operations become $O(\log^2 n)$, where n is $n_{\max}$ for LBBTs, or n_{curr} for the latter two. On the other hand, communication complexity of $O(\log n_{\max})$ or $O(\log n_{\mathrm{curr}})$ is preserved.

Remark 1. One can optimize the above composition of GUS and tRAM by overlaying the two trees used in the constructions as done in [10]. Doing so will recover $O(\log n)$ computational complexity of $\mathsf{add}, \mathsf{rem}, \mathsf{upd}$ MKA operations, instead of $O(\log^2 n)$, where $n = n_{\max}$ or n_{curr}, depending on what type of tree is used. Additionally, security and all other efficiency measures are preserved. We conjecture the security proof to be almost identical to that of Appendix E.

Composition Theorem. We will argue that our composed protocol GMS is secure with respect to the mgr-mult game. Intuitively, from the perspective of users, nothing changes since their state is identical to that which it would have been in GUS, so corrupting them gives no more advantage than it did in the user-mult game. In terms of forward secrecy of the group manager, both the encrypted RAM protocol and GUS are forward secure, so no old data can be accessed upon corruption. Finally, in terms of eventual PCS for the group manager, once the group manager removes or updates all of the users that were in the group at the time of the corruption, the adversary no longer possesses any keys that can help her derive the group secret; every cell in Γ_{pub} has changed.

Theorem 4 (security of GMS). *Let Q be the number of queries an adversary makes to the oracles of the* mgr-mult *game. Assume* dprf *is a $(t_{\mathsf{dprf}}, \varepsilon_{\mathsf{dprf}})$-secure dPRF,* uske *is a $(t_{\mathsf{cpa}*}, \varepsilon_{\mathsf{cpa}*})$-CPA*-secure USKE scheme,* eram *is a $(s_1^{\mathsf{eram}}, s_2^{\mathsf{eram}}, t_1^{\mathsf{eram}}, t_2^{\mathsf{eram}}, m_{max}^{\mathsf{eram}}, \varepsilon_{\mathsf{eram}})$-secure forward secret encrypted RAM scheme, and τ_{mka} is a $(s_{tree}(n, m), s_{skel}, \deg(\tau_{\mathsf{mka}}))$-tree. Then* GMS *is a $(s_1^{\mathsf{eram}}(s_{tree}(n_{curr}, n_{max}), s_{tree}(n_{max}, n_{max})), s_2^{\mathsf{eram}}(s_{tree}(n_{curr}, n_{max}), s_{tree}(n_{max}, n_{max})), \deg(\tau_{\mathsf{mka}}) \cdot s_{skel}, \deg(\tau_{\mathsf{mka}}) \cdot s_{skel} \cdot t_1^{\mathsf{eram}}, s_{skel}, t, q_c, n_{max}, \varepsilon)$-secure MKA scheme with respect to the* mgr-mult *security game, for $\varepsilon \in \{\varepsilon_{\mathsf{S}}, \varepsilon_{\mathsf{RO}}\}$, where $t \approx t_{\mathsf{dprf}} \approx t_{\mathsf{cpa}*}$. In the standard model, $\varepsilon_{\mathsf{S}} = (\varepsilon_{\mathsf{eram}} + Q \cdot \varepsilon_{\mathsf{dprf}} + 2\varepsilon_{\mathsf{cpa}*} \cdot \deg(\tau_{\mathsf{mka}})) \cdot (2\deg(\tau_{\mathsf{mka}}))^{d+1} \cdot Q^{(\deg(\tau_{\mathsf{mka}}) \cdot d + 2)}$. In the random oracle model, $\varepsilon_{\mathsf{RO}} = (\varepsilon_{\mathsf{eram}} + \varepsilon_{\mathsf{cpa}*}) \cdot 2 \cdot \deg(\tau_{\mathsf{mka}}) \cdot ((s_{tree} + 1) \cdot Q)^2 + \mathsf{negl}$.*

The proof of Theorem 4 is provided in Appendix E.

Corollary 3.

1. *If we use our* GUS *MKA protocol and our forward secret encrypted RAM protocol* tRAM*, both with LBBTs, then* GMS *is a $(O(1), O(n_{max}), O(\log n_{max}), O(\log^2 n_{max}), O(\log n_{max}), t, q_c, n_{max}, \varepsilon)$-secure MKA protocol with respect to* mgr-mult*, where $\varepsilon = O(q_c n_{max}^2 \cdot Q^{2\log(n_{max})+2} \cdot (\varepsilon_{\mathsf{tRAM}} + Q\varepsilon_{\mathsf{dprf}} + \varepsilon_{\mathsf{cpa}*}))$ in the standard model and $\varepsilon = O(q_c(\varepsilon_{\mathsf{tRAM}} + \varepsilon_{\mathsf{cpa}*}) \cdot (n_{max} \cdot Q)^2)$ in the random oracle model.*
2. *If we use our* GUS *MKA protocol and our forward secret encrypted RAM protocol* tRAM*, both with 2-3 trees or LLRBTs, then* GMS *is a $(O(1), O(n_{curr}), O(\log n_{curr}), O(\log^2 n_{curr}), O(\log n_{curr}), t, q_c, n_{max}, \varepsilon)$-secure MKA protocol with respect to* mgr-mult*, where $\varepsilon = O(q_c n_{curr}^2 \cdot Q^{2\log(n_{curr})+2} \cdot (\varepsilon_{\mathsf{tRAM}} + Q\varepsilon_{\mathsf{dprf}} + \varepsilon_{\mathsf{cpa}*}))$ in the standard model and $\varepsilon = O(q_c(\varepsilon_{\mathsf{tRAM}} + \varepsilon_{\mathsf{cpa}*}) \cdot (n_{max} \cdot Q)^2)$ in the random oracle model.*

Proof. This follows directly from Theorem 4, and Corollaries 1 and 2. □

References

1. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2020)
2. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Heidelberg, Germany, Durham, NC, USA (Nov 16–19, 2020)
3. Alwen, J., Capretto, M., Cueto, M., Kamath, C., Klein, K., Markov, I., Pascual-Perez, G., Pietrzak, K., Walter, M., Yeo, M.: Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In: 2021 IEEE Symposium on Security and Privacy (SP). IEEE (2021)
4. Bajaj, S., Sion, R.: Ficklebase: Looking into the future to erase the past. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). pp. 86–97. IEEE
5. Barnes, R., Beurdouche, B., Millican, J., Omara, E., Cohn-Gordon, K., Robert, R.: The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-11, Internet Engineering Task Force (Dec 2020), work in Progress
6. Bellare, M., Lysyanskaya, A.: Symmetric and dual PRFs from standard assumptions: A generic validation of an HMAC assumption. Cryptology ePrint Archive, Report 2015/1198 (2015), `https://eprint.iacr.org/2015/1198`
7. Bellare, M., Yee, B.S.: Forward-security in private-key cryptography. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 1–18. Springer, Heidelberg, Germany, San Francisco, CA, USA (Apr 13–17, 2003)
8. Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM J. Comput. 18(4), 766–776 (Aug 1989), `https://doi.org/10.1137/0218053`
9. Bhargavan, K., Barnes, R., Rescorla, E.: TreeKEM: Asynchronous decentralized key management for large dynamic groups (2018), published at `https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8`
10. Bienstock, A., Dodis, Y., Yeo, K.: Forward secret encrypted ram: Lower bounds and applications. In: TCC 2021 (2021)
11. Boneh, D., Lipton, R.J.: A revocable backup system. In: USENIX Security Symposium. pp. 91–96 (1996)
12. Bresson, E., Chevassut, O., Pointcheval, D.: Provably authenticated group Diffie-Hellman key exchange – the dynamic case. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 290–309. Springer, Heidelberg, Germany, Gold Coast, Australia (Dec 9–13, 2001)
13. Bresson, E., Chevassut, O., Pointcheval, D.: Dynamic group Diffie-Hellman key exchange under standard assumptions. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 321–336. Springer, Heidelberg, Germany, Amsterdam, The Netherlands (Apr 28 – May 2, 2002)
14. Bresson, E., Chevassut, O., Pointcheval, D.: Security proofs for an efficient password-based key exchange. In: Jajodia, S., Atluri, V., Jaeger, T. (eds.) ACM CCS 2003. pp. 241–250. ACM Press, Washington, DC, USA (Oct 27–30, 2003)
15. Canetti, R., Garay, J., Itkis, G., Micciancio, D., Naor, M., Pinkas, B.: Multicast security: a taxonomy and some efficient constructions. In: IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320). vol. 2, pp. 708–716 vol.2 (1999)
16. Di Crescenzo, G., Ferguson, N., Impagliazzo, R., Jakobsson, M.: How to forget a secret. In: Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science. p. 500–509. STACS'99, Springer-Verlag, Berlin, Heidelberg
17. Dodis, Y., Karthikeyan, H., Wichs, D.: Updatable public key encryption in the standard model. In: TCC 2021 (2021)
18. Fuchsbauer, G., Kamath, C., Klein, K., Pietrzak, K.: Adaptively secure proxy re-encryption. In: Lin, D., Sako, K. (eds.) PKC 2019, Part II. LNCS, vol. 11443, pp. 317–346. Springer, Heidelberg, Germany, Beijing, China (Apr 14–17, 2019)
19. Geambasu, R., Kohno, T., Levy, A.A., Levy, H.M.: Vanish: Increasing data privacy with self-destructing data. In: USENIX Security Symposium. vol. 316 (2009)
20. Harney, H., Muckenhirn, C.: Rfc2093: Group key management protocol (gkmp) specification (1997)
21. Jafargholi, Z., Kamath, C., Klein, K., Komargodski, I., Pietrzak, K., Wichs, D.: Be adaptive, avoid overcommitting. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 133–163. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2017)
22. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 159–188. Springer, Heidelberg, Germany, Darmstadt, Germany (May 19–23, 2019)
23. Katz, J., Yung, M.: Scalable protocols for authenticated group key exchange. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 110–125. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2003)

24. Micciancio, D., Panjwani, S.: Optimal communication complexity of generic multicast key distribution. In: Cachin, C., Camenisch, J. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 153–170. Springer, Heidelberg, Germany, Interlaken, Switzerland (May 2–6, 2004)
25. Mittra, S.: Iolus: A framework for scalable secure multicasting. In: Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 277–288. SIGCOMM '97, Association for Computing Machinery, New York, NY, USA (1997)
26. Panjwani, S.: Tackling adaptive corruptions in multicast encryption protocols. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 21–40. Springer, Heidelberg, Germany, Amsterdam, The Netherlands (Feb 21–24, 2007)
27. Perrin, T., Marlinspike, M.: The double ratchet algorithm (2016), `https://signal.org/docs/specifications/doubleratchet/`
28. Peterson, Z.N., Burns, R.C., Herring, J., Stubblefield, A., Rubin, A.D.: Secure deletion for a versioning file system. In: FAST. vol. 5 (2005)
29. Porambage, P., Braeken, A., Schmitt, C., Gurtov, A., Ylianttila, M., Stiller, B.: Group key establishment for enabling secure multicast communication in wireless sensor networks deployed for iot applications. IEEE Access 3, 1503–1511 (2015)
30. Reardon, J., Basin, D., Capkun, S.: Sok: Secure data deletion. In: 2013 IEEE symposium on security and privacy. pp. 301–315. IEEE (2013)
31. Reardon, J., Ritzdorf, H., Basin, D., Capkun, S.: Secure data deletion from persistent media. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 271–284 (2013)
32. Roche, D.S., Aviv, A., Choi, S.G.: A practical oblivious map data structure with secure deletion and history independence. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 178–197. IEEE (2016)
33. Sedgewick, R.: Left-leaning red-black trees (2008)
34. Sherman, A.T., McGrew, D.A.: Key establishment in large dynamic groups using one-way function trees. IEEE Transactions on Software Engineering 29(5), 444–458 (2003)
35. Tselekounis, Y., Coretti, S., Alwen, J., Dodis, Y.: Modular design of secure group messaging protocols and the security of mls. In: ACM CCS 2021 (2021)
36. Wallner, D., Harder, E., Agee, R.: Rfc2627: Key management for multicast: Issues and architectures (1999)
37. Weidner, M., Kleppmann, M., Hugenroth, D., Beresford, A.R.: Key agreement for decentralized secure group messaging with strong security guarantees. Cryptology ePrint Archive, Report 2020/1281 (2020), `https://eprint.iacr.org/2020/1281`
38. Wong, C.K., Gouda, M., Lam, S.S.: Secure group communications using key graphs. In: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 68–79. SIGCOMM '98, Association for Computing Machinery, New York, NY, USA (1998)

A Balanced Trees

Here we describe different balanced trees that one can use in our MKA construction, including left-balanced binary trees, 2-3 trees, and left-leaning red-black trees. We compare the performance of GUS using these different trees in Section 5.4.

A.1 Left-Balanced Binary Trees

We first recall some definitions regarding binary trees. A *binary tree* is a tree where all internal nodes are degree-2. For a degree-2 node, its two children are often referred to as the *left child* and the *right child*. A binary tree is said to be *full* if its leaf nodes share the same depth.

Definition 8 (Left-Balanced Binary Tree). *A tree is a (non-empty)* left-balanced binary tree *(LBBT) if it satisfies either of the following conditions:*

- *the tree is a single node;*
- *the root of the tree is a degree-2 node, where the left child is a* full *binary tree, the right child is a LBBT, and the height of the left child is no less than the height of the right child.*

Initializing Tree. Given labels for the leaf nodes $\ell_1, \ldots \ell_n$, the INIT operation for a LBBT proceeds as follows:

- if $n = 1$, create the tree τ as a single (leaf) node;
- otherwise, let m be the largest power of 2 below n (exclusive);
- create a degree-2 node whose left child is a full binary tree with m leaf nodes and whose right child is a LBBT with $n - m$ leaf nodes as the root of τ.

Note that this construction is unique because of the uniqueness of number m that is a power of 2 below n and satisfies $m \geq n - m$.

The skeleton skeleton that will be returned consists of the whole initialized tree, where for each interior node, the edge to its left child is colored green and the edge to its other child is colored blue.

Adding Leaf Node to Tree. The ADD operation for a LBBT proceeds as follows: If there exists leaf node marked as `removed`, then simply replace the marked leaf node with the new leaf node labeled with ℓ. Otherwise, append the new leaf node v to the tree τ as follows:

- if τ is full, create a degree-2 node whose children are (τ, v) as the new root;
- if τ is not full, append v to the right child of τ (which by definition is also a LBBT) recursively.

The skeleton skeleton that will be returned consists of the nodes on the direct path of node newly labeled ℓ. For each node in skeleton, if it has a child that is also in skeleton, the edge to this child is colored green.

Removing Leaf Node from Tree. The REMOVE operation for a LBBT proceeds as follows:

- Mark the leaf node v_ℓ corresponding to the label ℓ to be removed as `removed`.
- Recursively replace the parent of the rightmost leaf v in τ with the (left) sibling of v until the rightmost leaf node in τ is not marked as `removed`.

The skeleton skeleton that will be returned consists of the nodes on the direct path of v_ℓ (not including v_ℓ itself) that are still in the tree (or just the root r if all such nodes have been removed). For each node in skeleton, if it has a child that is also in skeleton, the edge to this child is colored green.

Efficiency measures of LBBTs for MKA. Due to the mechanics of the remove operation for LBBTs, namely lazily marking nodes as `removed`, the total number of nodes can remain relatively unchanged even after several removals of leaves. For example, if at some point all of the leaves that will be removed are in the subtree of the left child of the root, the number of nodes can only increase. Figure 4 shows how the deletion of a leaf that is not the rightmost leaf of the tree does not change the structure of the LBBT.

We note that if a given leaf is marked as `removed` in a LBBT being used as a MKA tree, it will not have any associated user or secrets and will thus *not* be considered utilized. Since this is the worst case, we must state the efficiency measures of LBBTs used as MKA trees in terms of m, the maximum number of leaves used throughout its existence. Therefore, it can easily be seen that LBBTs are $(m, \log m, \log m, 2)$-trees.

A.2 2-3 Trees

Definition 9 (2-3 Tree). *A tree is a (non-empty)* 2-3 tree *if it satisfies either of the following conditions:*

- *the tree is a single node;*
- *the root of the tree is a node of degree 2 or 3, where the children are 2-3 trees of the same height.*

Initializing Tree. Given labels for the leaf nodes $\ell_1, \ldots, \ell_n$, the INIT operation for a 2-3 tree proceeds as follows:

- if $n = 1$, create the tree τ as a single (leaf) node;
- otherwise, let h be the integer such that $3^{h-1} < n \leq 3^h$ (explicitly, $h = \lceil \log_3 n \rceil$);
- if $n > 2 \cdot 3^{h-1}$, create a degree-3 node whose children are 2-3 trees with respectively m, $m + r_1$, $m + r_2$ leaf nodes[16] as the root of τ, where $m = \lfloor n/3 \rfloor$, and $r_1 = r_2 = 0$ if $n \equiv 0 \pmod 3$, $r_1 = 0$, $r_2 = 1$ if $n \equiv 1 \pmod 3$ and $r_1 = r_2 = 1$ if $n \equiv 2 \pmod 3$;
- otherwise ($n \leq 2 \cdot 3^{h-1}$), create a degree-2 node whose children are 2-3 trees with respectively $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ leaf nodes as the root of τ.

The correctness of this construction derives from the following arithmetic facts:

- if $2 \cdot 3^{h-1} < n \leq 3^h$, then $3^{h-2} < 2 \cdot 3^{h-2} \leq \lfloor n/3 \rfloor \leq \lceil n/3 \rceil \leq 3^{h-1}$;
- if $3^{h-1} < n \leq 2 \cdot 3^{h-1}$, then $3^{h-2} < 3^{h-1}/2 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq 3^{h-1}$.

The skeleton $\mathsf{skeleton}$ that will be returned consists of the whole initialized tree, where for each interior node, the edge to its leftmost child is colored green and the edge to its other child(ren) is (are) colored blue.

Adding Leaf Node to Tree. The ADD operation for a 2-3 tree adds a new leaf node v with label ℓ_v to the tree τ as follows:

- set $\mathsf{skeleton} = \{v\}$;
- Choose a position to add v i.e. choose a future sibling u of v;[17]
- add v as a new child of the parent p of u;
- if the degree of p becomes invalid i.e. reaches 4, then
  - remove u and v from the children of p and create a degree-2 node p' whose children are (u, v);
  - set $\mathsf{skeleton} = \mathsf{skeleton} \cup \{p, p'\}$ and color the edge from p' to u as blue, and the edge from p' to v green;
  - if p is the root of τ, then create a degree-2 node r whose children are (p, p') as the new root and set $\mathsf{skeleton} = \mathsf{skeleton} \cup \{r\}$, color the edge to p blue and the edge to p' green;
  - otherwise, add p' recursively (with future sibling p).
- otherwise, set $\mathsf{skeleton} = \mathsf{skeleton} \cup \{p, v_1, \ldots, v_l\}$, where $v_l, \ldots, v_l$ are on the direct path of p, and color the edges of these nodes to their children in the skeleton green.

The skeleton $\mathsf{skeleton}$ that will be returned consists of the direct path of v and any nodes p that had children removed from them.

Removing Leaf Node from Tree. The REMOVE operation for a 2-3 tree proceeds as follows: Remove the leaf node v with label ℓ_v from the tree τ as follows:

- set $\mathsf{skeleton} = \emptyset$;
- remove v from the children of its parent p;
- if the degree of p becomes invalid i.e. reaches 1, then
  1. if p is the root of τ, then use the only child of p as the new root;
  2. otherwise, if there exists sibling p' of p whose degree is 3, then move one child from p' to p. Additionally, for the nodes $p, v_1, \ldots, v_l$ on the direct path of p and the node p', set $\mathsf{skeleton} = \mathsf{skeleton} \cup \{p, v_1 \ldots, v_l, p'\}$, then for all of these nodes, except for v_1, the parent of p and p', color edges to their children in the skeleton green. For v_1, color its edge to p green and its edge to p' blue;
  3. otherwise (all siblings of p are degree-2), move the only child of p to one of its siblings, p', set $\mathsf{skeleton} = \mathsf{skeleton} \cup \{p'\}$, color the edge from p' to its child in the skeleton green (if it exists), and finally remove p recursively.
- otherwise, for the nodes $p, v_1, \ldots, v_l$ on the direct path of p, set $\mathsf{skeleton} = \mathsf{skeleton} \cup \{p, v_1, \ldots, v_l\}$, and for all of these nodes, color edges to their children in the skeleton green.

The skeleton $\mathsf{skeleton}$ consists of any parents p' from step (3.), as well as the parent p' and the direct path of p from step (2.), if the step is taken, OR the direct path of p in the last case otherwise.

[16] In order to ensure that the children have the same height, when recursively constructing the children, h should not be re-calculated independently and $h \leftarrow h - 1$ should be used.

[17] Randomly choosing such a position suffices, but one can make more sophisticated choices by, for example, greedily choosing the lowest non-full subtree to insert into.

Efficiency measures of 2-3 trees for MKA. Since all leaves of a 2-3 tree are at the same depth, a 2-3 tree shrinks and grows proportionally to the number of leaves it contains. Indeed, the height of a 2-3 tree is always $O(\log n)$, where n is the current number of leaves. Figure 4 shows that the height of a 2-3 tree decreases when the same leaf as discussed above in an identically configured LBBT is removed, however the number of ciphertexts from the operation is larger than that of the LBBT operation, due to degree-3 nodes. These degree-3 nodes render 2-3 trees at times less efficient than LBBTs, despite smaller height.

We note that all leaves in a 2-3 tree being used as a MKA tree are always utilized, and therefore n corresponds exactly to the number of utilized leaves in the tree, as defined in Section 5.1. Therefore, 2-3 trees are $(n, \log n, \log n, 3)$-trees.

A.3 Left-Leaning Red-Black Trees

The red-black tree is a widely used balanced tree data structure. The left-leaning red-black tree [33] is a restricted variant of red-black tree that preserves the favorable properties of red-black tree while simplifies the associated algorithms such as addition and removal of leaf nodes.

Definition 10 (Left-Leaning Red-Black Trees). *A tree is a (non-empty)* left-leaning red-black tree *(LLRBT) if it is binary, has every node colored either red or black, and satisfies either of the following conditions:*

- *the tree is a single black node;*
- *the root of the tree is a black node, and the children of every interior node match with either of the following color patterns:*
  1. *both children are black, or*
  2. *left child is red and internal, both grandchildren of the left child are black, and right child is black, or*
  3. *both children are red and internal, and all of the four grandchildren are black,*

  and every leaf node in the tree has the same black depth*, i.e., the number of black nodes on the direct path of that leaf node.*

It can be observed that LLRBTs are isomorphic to *2-3-4 trees*, the generalization of 2-3 trees that also allows degree-4 internal nodes (while both 2-3 and 2-3-4 trees are special cases in a family called *B trees*), by "contracting" the red nodes (which are always internal and non-root). More specifically, by contracting the red nodes, color pattern 1 remains a degree-2 node, color pattern 2 becomes a degree-3 node, and color pattern 3 becomes a degree-4 node, while the uniform black depth ensures that the leaf nodes has the same depth after the contraction. This contraction process is invertible, by "inserting" red nodes to form color pattern 1, 2, or 3 for degree-2, -3, or -4 nodes accordingly, and hence is an isomorphism between LLRBTs and 2-3-4 trees. Note that due to the correspondence between color patterns and degrees, by forbidding color pattern 3, the restricted LLRBTs become exactly isomorphic to 2-3 trees.

The isomorphism between normal/restricted LLRBTs and 2-3-4/2-3 trees induces the addition and removal (as well as initialization) processes for LLRBTs; specifically, the induced processes map to the isomorphic 2-3-4/2-3 trees, apply the corresponding process there, and map back to get the resulting normal/restricted LLRBTs. The induced processes indeed yield results that match with certain canonical processes for LLRBTs. One can come up with efficient algorithms that yield the same results as the induced processes but do not explicitly carry out the mapping forth and back, while we omit the detailed algorithms here for simplicity. In particular, the algorithms can be as efficient as those for 2-3-4/2-3 trees, and therefore LLRBTs are $(n, \log n, \log n, 2)$-trees.

It is worth mentioning that although there exists isomorphism between normal/restricted LLRBTs and 2-3-4/2-3 trees, it does not mean that they lead to the same performance when applied in GUS, for they have different organizations of nodes, which are the critical units for storing keys in GUS. Indeed, Figure 4 shows that after deleting the same leaf from an identically configured tree as above for LLBTs and 2-3 trees, the LLRBT remains balanced, and the number of ciphertexts is low due to having only degree-2 nodes.

B Basic Definitions

We introduce the cryptographic primitives that our protocols will make use of in this paper. We often make use of their *multi-instance* versions, in which the security of multiple independent instantiations of the primitive are together considered secure.

B.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 the attacker $\mathcal{A}$ at distinguishing between these two distributions is denoted by $\mathsf{Adv}^{\mathsf{prg}}_{\mathrm{prg}}(\mathcal{A})$; the attacker is parameterized by its running time t.

Definition 11. *A pseudorandom generator* prg *is* (t, ε)*-secure if for all* t*-attackers* $\mathcal{A}$*,*

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

B.2 Dual Pseudorandom Functions

A *Pseudorandom Function Family (PRF)* is a function family $\mathsf{prf} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ that is secure with respect to the following security game against some adversary $\mathcal{A}$ with running time t:

- The challenger chooses b uniformly from $\{0, 1\}$.
- If $b = 1$, then the challenger samples $k \leftarrow_{\$} \mathcal{K}$ and sets $\mathsf{F} \leftarrow \mathsf{prf}(k, \cdot) : \mathcal{X} \to \mathcal{Y}$.
- Otherwise, the challenger samples random function $\mathsf{F} : \mathcal{X} \to \mathcal{Y}$.
- $\mathcal{A}$ adaptively sends queries $x_1, \ldots, x_q \in \mathcal{X}$ and receives back $\mathsf{F}(x_1), \ldots, \mathsf{F}(x_q)$ after each such query, for q polynomial in the security parameter.
- $\mathcal{A}$ sends bit $b' \in \{0, 1\}$ to the challenger.

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

Definition 12. *A pseudorandom function family* prf *is* (t, ε)*-secure if for all* t*-attackers* $\mathcal{A}$*,*

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

For a function family $\mathsf{F} : \mathcal{K}_1 \times \mathcal{K}_2 \to \mathcal{Y}$, let $\mathsf{F}^{\mathrm{swap}} : \mathcal{K}_2 \times \mathcal{K}_1 \to \mathcal{Y}$ be defined by $\mathsf{F}^{\mathrm{swap}}(k_1, k_2) = \mathsf{F}(k_2, k_1)$. A *dual Pseudorandom Function Family (dPRF)* is a function family $\mathsf{dprf} : \mathcal{K}_1 \times \mathcal{K}_2 \to \mathcal{Y}$ such that both dprf and $\mathsf{dprf}^{\mathrm{swap}}$ are PRFs [6].

Definition 13. *A dual pseudorandom function family* dprf *is* (t, ε)*-secure if both* dprf *and* dprf^{swap} *are* (t, ε)*-secure PRFs.*

B.3 Symmetric-Key Encryption

Definition 14. *A* symmetric-key encryption (SKE) *scheme is a triple of algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with the following synatx:*

- Key generation*:* Gen *receives (implicitly) a security parameter and outputs a fresh key* $k \leftarrow \mathsf{Gen}$*.*
- Encryption*:* Enc *receives a key* k *and a message* m *and produces a ciphertext* c*.*
- Decryption*:* Dec *receives a key* k *and a ciphertext* c *and produces a message* m*.*

Correctness. A SKE scheme must satisfy the following correctness property. For any message m,

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

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

- The challenger runs $k \leftarrow \mathsf{Gen}$.
- $\mathcal{A}$ adaptively sends encryption queries for messages $m_1, \ldots, m_{q_1}$ and receives back $\mathsf{Enc}(k, m_1), \ldots, \mathsf{Enc}(k, m_{q_1})$ after each such query, where q_1 is polynomial in the security parameter.
- $\mathcal{A}$ sends challenge messages m_0, m_1.
- The challenger chooses b uniformly from $\{0, 1\}$, computes $c = \mathsf{Enc}(k, m_b)$ and sends c to $\mathcal{A}$.
- $\mathcal{A}$ adaptively sends encryption queries for messages $m'_1, \ldots, m'_{q_2}$ and receives back $\mathsf{Enc}(k, m'_1), \ldots, \mathsf{Enc}(k, m'_{q_2})$ after each such query, where q_2 is polynomial in the security parameter.
- $\mathcal{A}$ sends bit $b' \in \{0, 1\}$ to the challenger

$\mathcal{A}$ 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 15. *A symmetric-key encryption scheme Π is (t, ε)-CPA-secure if for all t-attackers $\mathcal{A}$,*

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

C GUS Security Proofs

In this section, we provide two security proofs of our construction GUS against the adaptive, partially active adversary in the user-mult game – one in the standard model achieving Quasi-polynomial security loss, and another in the Random Oracle model achieving polynomial security loss. We thus prove Theorem 2 of the main body. We use the proofs of Tainted TreeKEM (a CGKA protocol) in [3] as a template.

We will prove security against an adversary that issues a *single* challenge query. Using a standard hybrid argument,[18] we can show that if the protocol is $(s_{\mathrm{tree}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, s_{\mathrm{skel}}, t, 1, n_{\max}, \varepsilon)$-secure then it is $(s_{\mathrm{tree}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, s_{\mathrm{skel}}, t, q_c, n_{\max}, \varepsilon')$-secure, where $\varepsilon' = q_c \cdot \varepsilon$. We omit it for brevity.

C.1 Challenge Graph

We will argue for the security of GUS in the framework of Jafargholi et al. [21], with which we will assume familiarity throughout this section. To do so, we need to view the user-mult game for GUS as a game on a graph, then define the *challenge graph* for group key I^* as a (modified) subgraph of the whole GUS graph.

Intuitively, each node i in the GUS-user-mult graph is associated with:

1. dPRF key k_s^i that is sampled randomly in Step 1 of the `SecretGen` procedure of the group manager for a node u in τ_{mka}, or computed via a dPRF computation on key k_s^j and the old dPRF key $k_d^{j,\mathrm{old}}$ at a child $u.c$ of u (corresponding to node j in the GUS-user-mult graph) in Step 2 of `SecretGen`;
2. new dPRF key k_d^i computed from k_s^i and $k_d^{i,\mathrm{old}}$; and
3. USKE keys $k_{e,0}^i, \ldots, k_{e,v_i}^i$, where v_i is the number of times GUS encrypts to u before removing or refreshing it (i.e., before u is removed from τ_{mka} or appears in a subsequent skeleton for an operation).

However, nodes i in the graph corresponding to keys for leaves ℓ_{ID} in τ_{mka}, generated starting with an update for some ID, do not contain k_s^i if the out-of-band message (that contains k_s^i) for the update was corrupted.

The edges of the graph are induced by 1. encryptions of k_s^j to $k_{e,l}^i$ for some $l \in [v_i]$; 2. dPRF computations in which the PRF security of dprf is used to compute k_s^j with key k_s^i; 3. dPRF computations in which the PRF security of $\mathsf{dprf}^{\mathrm{swap}}$ is used to compute k_d^j with key k_d^i (c.f. Appendix B.2); and 4. dPRF computations in which the PRF security of $\mathsf{dprf}^{\mathrm{swap}}$ is used to compute k_s^j with key k_d^i. Observe that these edges (when viewed with respect to τ_{mka}) are in the opposite direction that standard tree definitions use. Unless otherwise stated, we will throughout this section refer to the child of a node u in the GUS-user-mult graph as the node to which u has an edge and that is closer to the root than u, and the parents of u as those nodes from which u has an edge and that are closer to the leaves than u. Formally:

[18] For example, one very similar to Lemma 6 from [1].

Definition 16 (GUS-user-mult graph). *The* GUS-user-mult graph $\mathcal{G}_{\text{user-mult}} = (\mathcal{V}_{\text{user-mult}}, \mathcal{E}_{\text{user-mult}})$ *is implicitly generated by the adaptive actions of an adversary* $\mathcal{A}$ *against* GUS *in the* user-mult *MKA game. For each key* k_s^i *that is sampled randomly in Step 1 or computed via a dPRF computation in Step 2 of the* `SecretGen` *procedure of the group manager for some node* u *in* τ_{mka}*, and is encrypted to* v_i *times before being removed from* τ_{mka} *or appearing in some* skeleton *of a subsequent operation, where* $(\cdot, k_d^i, k_{e,0}^i) \leftarrow \mathsf{dprf}(k_s^i, \cdot), (\cdot, k_{e,l}^i) \leftarrow \mathsf{UEnc}(k_{e,l-1}^i, \cdot)$ *for* $l \in [v_i]$*:*

1. *Node* $i = \{k_d^i, k_{e,0}^i, k_{e,1}^i, \ldots, k_{e,v_i}^i\} \in \mathcal{V}_{\text{user-mult}}$ *if* u *is some leaf* ℓ_{ID} *in* τ_{mka} *such that* k_s *is sampled during query*
   update-user(ID) *of epoch* t *and* ***corrupt-oob***(t, ID) *is later queried; or*
2. *Node* $i = \{k_s^i, k_d^i, k_{e,0}^i, k_{e,1}^i, \ldots, k_{e,v}^i\} \in \mathcal{V}_{\text{user-mult}}$ *otherwise.*

Edge $(i, j) \in \mathcal{E}_{\text{user-mult}}$ *if*

1. GUS *creates a ciphertext* $(\mathsf{ct}, \cdot) \leftarrow \mathsf{UEnc}(k_{e,l}^i, k_s^j)$ *for some* $l \in [v_i]$*; or*
2. GUS *computes* $\mathsf{dprf}(k_s^i, \cdot) \rightarrow (k_s^j, \cdot, \cdot)$ *(Note: only if* $k_s^i \in i$*); or*
3. GUS *computes* $\mathsf{dprf}(\cdot, k_d^i) \rightarrow (\cdot, k_d^j, \cdot)$*, where* j *replaces* i *at leaf* ℓ_{ID} *in* τ_{mka} *during query* ***update-user***(ID) *of epoch* t*, and* ***corrupt-oob***(t, ID) *is later queried; or*
4. GUS *computes* $\mathsf{dprf}(\cdot, k_d^i) \rightarrow (k_s^j, \cdot, \cdot)$*, where* j *replaces the parent of* ℓ_{ID} *which* i *corresponds to in* τ_{mka} *(in the standard tree sense) during query* ***update-user***(ID) *of epoch* t*, and* ***corrupt-oob***(t, ID) *is later queried.*

The challenge graph for I^* is intuitively the subgraph of the GUS-user-mult graph $\mathcal{G}_{\text{user-mult}}$ induced on the nodes from which I^* is trivially reachable (and also trivially computable using their keys). Therefore, each vertex i does not include the dPRF key k_d^i computed for it, nor those USKE key versions which cannot be used to trivially derive I^* (i.e., keys that may be corrupted at challenge epoch t^* or later but cannot be used to compute I^* using normal dPRF or decryption operations).

Definition 17 (GUS-user-mult-challenge Graph). *The* GUS-user-mult-challenge Graph $\mathcal{G}^*_{\text{user-mult}} = (\mathcal{V}^*_{\text{user-mult}}, \mathcal{E}^*_{\text{user-mult}})$ *includes only the nodes which have a path to* I^* *in* $\mathcal{G}_{\text{user-mult}}$*, and the corresponding edges on those paths. Moreover, for each* $i \in \mathcal{V}^*_{\text{user-mult}}$ *which was node* $i_{\mathcal{V}_{\text{user-mult}}} \in \mathcal{V}_{\text{user-mult}}$ *that corresponds to a* τ_{mka} *node* u *at time* t^**, or the version of epoch* $t < t^*$ *at* u *if* u *is leaf* ℓ_{ID} *and the most recent* ***update-user***(ID) *query before* t^* *in epoch* t *is such that* ***corrupt-oob***(t, ID) *is later queried:*

- *If in* $\mathcal{G}_{\text{user-mult}}$*,* $i_{\mathcal{V}_{\text{user-mult}}}$ *has an ingoing path* $(j_1, j_2), (j_2, j_3), \ldots, (j_{c_i}, j_{c_i+1} = i_{\mathcal{V}_{\text{user-mult}}})$ *of type 3 edges of length* $c_i \geq 1$*, where* $j_1 \in \mathcal{V}_{\text{user-mult}}$ *has no ingoing edges and* $j_c \in \mathcal{V}_{\text{user-mult}}, c \in \{2, 3, \ldots, c_i + 1\}$ *have only one ingoing edge which is of type 3, then,* $i = i_{\mathcal{V}_{\text{user-mult}}} \cup_{c \in [c_i]} j_c$*. Therefore also,* $j_c \notin \mathcal{V}^*_{\text{user-mult}}$ *and* $(j_c, j_{c+1}) \notin \mathcal{E}^*_{\text{user-mult}}$*, for* $c \in [c_i]$*.*
- *Otherwise, let* $v_i^* + 1 \in [v_i]$ *be the corresponding version of the USKE key for* $i_{\mathcal{V}_{\text{user-mult}}}$ *that is generated at time* t^**, the challenge epoch. Then,* $i = i_{\mathcal{V}_{\text{user-mult}}} \setminus \{k_d^i, k_{e,v_i^*+1}, \ldots, k_{e,v_i}\}$*.*

Thus $\mathcal{G}^*_{\text{user-mult}}$ (roughly) corresponds to τ_{mka} at the challenge epoch t^*, plus each leaf ℓ_{ID} absorbs the old keys that were at old versions of the leaf until the most recent version corresponding to an Update of ID for which the out-of-band channel was not corrupted. Thus, we have the following lemma which stipulates that none of the keys in $\mathcal{G}^*_{\text{user-mult}}$ are leaked to the adversary via corruption, according to predicate **user-safe**.

Lemma 2. *For queries* $\mathbf{q}_1, \ldots, \mathbf{q}_Q$ *made by an adversary* $\mathcal{A}$ *in* user-mult*, if* **user-safe**$(\mathbf{q}_1, \ldots, \mathbf{q}_Q) \rightarrow$ true*, it holds that none of the keys contained in nodes of* $\mathcal{G}^*_{\text{user-mult}}$ *are leaked to* $\mathcal{A}$ *via corruption.*

Proof. Intuitively, **user-safe** ensures that for all ID that were in the group up to and including the challenge operation op^* at t^*, if 1. ID was previously corrupted, or 2. ID was added to the group in epoch t_{ID} for which **corrupt-oob**$(t_{\mathsf{ID}}, \mathsf{ID})$ is later queried by $\mathcal{A}$, then either ID has since been removed or has been updated in an operation such that the out-of-band message is not corrupted. Thus, since only (updated and since uncorrupted) users that are in the group at time t^* know any secrets at the nodes of τ_{mka} at this time (or any old version of encryption keys, or leaf keys that were updated with out-of-band corrupted keys), no information on these keys are leaked to the adversary via corruption.

Formally, assume that some key k in some $i \in \mathcal{V}^*_{\text{user-mult}}$ is leaked to $\mathcal{A}$ via corruption. We will show that **user-safe**$(\mathbf{q}_1, \ldots, \mathbf{q}_Q) \rightarrow$ false, a contradiction. There are three cases:

1. k is in some $i \in \mathcal{V}^*_{\mathsf{user\text{-}mult}}$ corresponding to a leaf node ℓ_{ID} in τ_{mka} at epoch t^* that is in $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$, and k was generated for the oldest version of ℓ_{ID} that was absorbed by i in $\mathcal{V}^*_{\mathsf{user\text{-}mult}}$. In this case, either when ID was most recently added to the group in epoch $t \leq t^*$, $\mathcal{A}$ queried **corrupt-oob**(t, ID); or $\mathcal{A}$ queried **corrupt**(ID) while ID was still in some epoch $t \leq t^*$ during which k was in the version of i at ℓ_{ID}. Moreover, by the definition of $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$ if $\mathcal{A}$ queried (possibly multiple times) **update-user**(ID) in some epoch $t < t' \leq t^*$, then $\mathcal{A}$ later queried **corrupt-oob**(t', ID). However, it is clear that $\mathcal{A}$ violates **user-safe** in this case.
2. k is in some $i \in \mathcal{V}^*_{\mathsf{user\text{-}mult}}$ corresponding to a leaf node ℓ_{ID} in τ_{mka} at epoch t^*, and k was generated for some more recent version of ℓ_{ID} that was absorbed by i in $\mathcal{V}^*_{\mathsf{user\text{-}mult}}$. Then it must be that the dPRF key of the version before this one in i was known to $\mathcal{A}$, from which we can inductively apply this argument until we (possibly) reach the first case and apply the above argument; or $\mathcal{A}$ queried **corrupt**(ID) while ID was still in some epoch $t \leq t^*$ during which i was the version of ℓ_{ID}. And moreover as above, all later update out-of-band messages were corrupted by $\mathcal{A}$. In either case, **user-safe** is violated.
3. k is in some $i \in \mathcal{V}^*_{\mathsf{user\text{-}mult}}$ corresponding to an interior node u in τ_{mka}. Then there must exist some user ID at one of the leaves in the subtree rooted at u such that one of the two above arguments hold. □

C.2 Security Proof for GUS in the Standard Model

For security of GUS in the standard model, we use the framework of Jafargholi et al. [21]. In user-mult, the adversary needs to execute queries that result in **user-safe** evaluating to true, and distinguish a challenge group key I^* from a uniformly random value. We will first consider the *selective* user-mult game, where the adversary needs to schedule its queries all at once at the beginning of the game. We will call the two possible executions of the game as the *real* and *random* user-mult games and prove their indistinguishability through a sequence of hybrid games. Using the framework of [21], we will define these hybrid games via the *reversible black pebbling game*, introduced by Bennett [8], where, starting with a directed acyclic graph with a unique sink and no pebbles on any nodes (in our case, $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$ with sink I^*), in each step one can put or remove one pebble on a node following certain rules. The goal is to reach the point in which only the sink has a pebble. Each pebbling configuration $\mathcal{P}_\ell$ then uniquely defines a hybrid game H_ℓ: a node i in $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$ being pebbled means that in this hybrid game, whenever $\mathsf{dprf}(k^i_s, k^{i,\mathrm{old}}_d) \to (k_p||k^i_d||k^i_{e,0})$ is computed (for the final time, for i corresponding to a leaf), we instead sample the output $(k_p||k^i_d||k^i_{e,0})$ randomly in the simulation. All other nodes and edges are simulated as in the real user-mult game. Thus the real game H_{real} is the empty pebbling configuration $\mathcal{P}_0$ and the random game H_{random} is the final configuration $\mathcal{P}_L$ where only the sink node is pebbled (where L is the length of the pebbling sequence).

Definition 18 (Reversible black pebbling). *A reversible pebbling of a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with unique sink* sink *is a sequence $(\mathcal{P}_0, \ldots, \mathcal{P}_L)$ with $\mathcal{P}_\ell \subseteq \mathcal{V}$ ($\ell \in [0, L]$), such that $\mathcal{P}_0 = \emptyset$ and $\mathcal{P}_L = \{\mathsf{sink}\}$, and for all $\ell \in [L]$ there is a unique $v \in \mathcal{V}$ such that: 1. $\mathcal{P}_\ell = \mathcal{P}_{\ell-1} \cup \{v\}$ or $\mathcal{P}_\ell = \mathcal{P}_{\ell-1} \setminus \{v\}$; and 2. for all $u \in \mathrm{parents}(v) : u \in \mathcal{P}_{\ell-1}$.*

By Lemma 2, we know that none of the keys in the challenge graph are leaked to the adversary throughout the entire game. This will allow us to prove indistinguishability of consecutive hybrid games from dPRF security and USKE security.

Lemma 3. *Let $(\mathcal{P}_0, \ldots, \mathcal{P}_L)$ be a valid pebbling sequence on the challenge graph. If* dprf *is a $(t_{\mathsf{dprf}}, \varepsilon_{\mathsf{dprf}})$-secure dPRF and* $\mathsf{uske} = (\mathsf{UEnc}, \mathsf{UDec})$ *is a $(t_{\mathsf{cpa}*}, \varepsilon_{\mathsf{cpa}*})$-CPA*-secure USKE scheme, then any two consecutive hybrid games $H_\ell, H_{\ell+1}$ are $(t, Q\varepsilon_{\mathsf{dprf}} + 2\varepsilon_{\mathsf{cpa}*} \cdot \deg(\tau_{\mathsf{mka}}))$-indistinguishable for $t \approx t_{\mathsf{dprf}} \approx t_{\mathsf{cpa}*}$.*[19]

Proof. Let $H_\ell, H_{\ell+1}$ be two consecutive hybrid games. We assume that $\mathcal{P}_{\ell+1}$ has one additional pebble on node v^* that $\mathcal{P}_\ell$ does not have. The case where $\mathcal{P}_{\ell+1}$ is obtained from $\mathcal{P}_\ell$ by removing one pebble can be proved in a similar way.

[19] For many pairs of consecutive hybrid games (those where a pebble is neither being added nor removed from a leaf), the extra Q factor is not needed.

First, consider the case where v^* is a leaf. Let $k_d^{v^*,c^{v^*}}$ be the (final) key at v^* which is used to compute k_s^u for the child u of v^* in $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$ (the parent in τ_{mka}). In this case, $k_d^{v^*}$ is first computed via a sequence of dPRF computations of length c^{v^*}. Let hybrids $H_\ell := H_0, H_1, \ldots, H_{c^{v^*}} := H_{\ell+1}$ be such that for hybrid H_i, H_i is defined similarly to H_{i-1} except that the i-th dPRF output is replaced by a uniformly random value. For the first dPRF computation, uniformly random key $k_s^{v^*}$, which is not leaked to the adversary by Lemma 2, is used to compute $k_d^{v^*,1}$. Thus, we can construct reduction algorithm $\mathcal{B}_0$, where we rely on the PRF security of dprf using key $k_s^{v^*}$ to output real or random $k_d^{v^*,1}$. For all subsequent dPRF computations, $k_d^{v^*,c}$, which is not leaked to the adversary by Lemma 2, is used to compute $k_d^{v^*,c+1}$, for $c \in [c^{v^*}]$. Thus, we can construct reduction algorithm $\mathcal{B}_c$, where we rely on the PRF security of $\mathsf{dprf}^{\mathrm{swap}}$ (c.f. Appendix B.2) using key $k_d^{v^*,c}$ to output real or random $k_d^{v^*,c+1}$. For $c \in [0, c^{v^*}]$, $\mathcal{B}_c$ can perfectly simulate the rest of the hybrids, as by Lemma 2, all preceding keys are never leaked to $\mathcal{A}$, and $\mathcal{B}$ can simply embed $k_d^{v^*c}$ into the state of the corresponding ID during the operation that creates $k_d^{v^*c}$. Thus, any advantage ε of an adversary in distinguishing any H_{c-1}, H_c, $c \in [c^{v^*}]$ leads to the same advantage for $\mathcal{B}_c$ in the PRF game on dprf (or $\mathsf{dprf}^{\mathrm{swap}}$ for $\mathcal{B}_0$). Therefore, H_ℓ and $H_{\ell+1}$ are $(c^{v^*} \cdot \varepsilon_{\mathsf{dprf}})$-indistinguishable, and thus $(Q \cdot \varepsilon_{\mathsf{dprf}})$-indistinguishable.

Otherwise, observe that v^* has $0 \leq m_e \leq \deg(\tau_{\mathsf{mka}})$ ingoing encryption edges $(u_1, v^*), \ldots, (u_{m_e}, v^*)$, and at most one ingoing dPRF edge (u', v^*). We prove indistinguishability of hybrids $H_\ell := H_{\ell,0}, \ldots, H_{\ell,2m_e+1} := H_{\ell+1}$, where the intermediate hybrids are defined as follows:

- $H_{\ell,m}$ for $m \in [m_e]$ is defined similarly to $H_{\ell,m-1}$ except that the encryption $\mathsf{UEnc}(k_{e,v^*_{u_m}}, k_s^{v^*})$ is replaced by an encryption of a uniformly random key.
- H_{ℓ,m_e+1} is defined similarly to H_{ℓ,m_e+1} except that instead of $(k_p || k_d^{v^*} || k_{e,0}^{v^*})$ being the output of a dPRF computation on $k_s^{v^*}$ and $k_s^{v^*,\mathrm{old}}$, it is uniformly sampled.
- H_{ℓ,m_e+1+m} for $m \in [m_e]$ is defined similarly to $H_{\ell,m_e+1+m-1}$, except that the encryption of a uniformly random key is replaced by $\mathsf{UEnc}(k_{e,v^*_{u_m}}, k_s^{v^*})$. Note that we indeed have $H_{\ell,2m_e+1} = H_{\ell+1}$.

For all $m \in [m_e]$ $\varepsilon_{\mathsf{cpa}*}$-indistinguishability of $H_{\ell,m-1}$ and $H_{\ell,m}$ follows from the CPA* security of uske and the fact that by pebbling rules, u_m must be pebbled so that in game $H_{\ell,m}$, $k_{e,0}^{u_m}$ is sampled uniformly at random: A reduction algorithm $\mathcal{B}$ against a CPA* challenger can simply query the challenger on all messages (keys k_s^w for some parent w – in the traditional tree sense – of u_m in τ_{mka}) encrypted to u_m up to $k_s^{v^*}$ and include the output ciphertexts c in the corresponding control messages as usual. Then, $\mathcal{B}$ can issue challenge query on $k_s^{v^*}$ and uniformly random key k and include the output ciphertext c^* in the corresponding control message as usual. $\mathcal{B}$ can perfectly simulate the rest of the hybrid, as by Lemma 2 all USKE keys $k_{e,0}^{u_m}$ until $k_{e,v_{u_m}}^{u_m}$ are never leaked to $\mathcal{A}$, then $\mathcal{B}$ can use the key $k_{e,v^*_{u_m}+1}^{u_m}$ returned by the challenge query as usual to encrypt later messages to u_m. Moreover, $\mathcal{B}$ can simply embed all keys k_s^w into the state of any ID that would normally have it when they process the corresponding control message for the operations that encrypt k_s^w. Thus any advantage ε of an adversary in distinguishing $H_{\ell,m-1}, H_{\ell,m}$ leads to the same advantage of $\mathcal{B}$ in the CPA* game on uske.

$\varepsilon_{\mathsf{dprf}}$-indistinguishability of $H_{\ell,m_e}, H_{\ell,m_e+1}$ follows from the pseudorandomness of dprf: If v^* has parent u' in $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$, by pebbling rules u' must be pebbled, in game H_{ℓ,m_e}, so $k_s^{v^*}$ (resp. $k_d^{u'}$ if v^* corresponds to the child of u' which is in a leaf node ℓ_{ID} in τ_{mka} whose last update up to and including t^* had its oob message corrupted) is sampled uniformly at random instead of being the output of a dPRF computation. It is also independent of all ciphertexts generated by GUS, as in game H_{ℓ,m_e}. Therefore, a reduction algorihtm $\mathcal{B}$ against a dPRF challenger (using the PRF security of dprf (resp. $\mathsf{dprf}^{\mathrm{swap}}$, c.f. Appendix B.2) with key $k_s^{v^*}$ (resp. $k_d^{u'}$) can simply query the dPRF challenger on the input that normally computes $(k_p || k_d^{v^*} || k_{e,0}^{v^*})$ (either the old version $k_d^{u'}$ or $\perp$ (resp. $k_s^{v^*}$)) to get $(k'_p || k'_d || k'_{e,0})$. $\mathcal{B}$ can then use k'_p in place of k_p, and similarly for the other output keys as needed. $\mathcal{B}$ can perfectly simulate the rest of the hybrid, as by Lemma 2, $k_s^{v^*}$ (resp. $k_d^{u'}$) is never leaked to $\mathcal{A}$, and $\mathcal{B}$ can simply embed $k'_p, k'_d, k'_{e,0}$ into the state of any ID that would normally have them after processing the corresponding control message for the operation that creates them. Thus any advantage ε of an adversary in distinguishing H_{ℓ,m_e} and H_{ℓ,m_e+1} leads to the same advantage for $\mathcal{B}$ in the PRF game on dprf (resp. $\mathsf{dprf}^{\mathrm{swap}}$).

$\varepsilon_{\mathsf{cpa}*}$-Indistinguishability of $H_{\ell,m_e+1,m-1}, H_{\ell,m_e+1+m}$ is identical to the case of $H_{\ell,m-1}, H_{\ell,m}$, following from the CPA* security of uske.

It is clear that in the reductions, the running time remains essentially the same. Thus, the lemma is proved. □

By choosing a trivial pebbling sequence of the challenge graph, the above already implies *selective* user-mult security of GUS. However, in the adaptive setting, the challenge graph is not known to the reduction until the adversary makes her query challenge, at which point it is too late for the reduction to embed any dPRF or USKE challenges, since some of the keys may have already been used in answering previous queries of the adversary. Thus, to simulate a hybrid H_ℓ, the reduction needs to guess *some* of the adaptive choices the adversary makes. This would naively result in exponential security loss; however, the framework of Jafargholi et al. [21] allows us to achieve quasi-polynomial loss:

Theorem 5 (Framework for proving adaptive security, informal [21]). *Let $G_{\mathsf{real}}, G_{\mathsf{random}}$ be two adaptive games, and H_{real}, H_{random} be their respective selective versions, where the adversary schedules its queries all at once at the beginning of the game. Furthermore, let $H_{\mathsf{real}} := H_0, H_1, \ldots, H_L := H_{\mathsf{random}}$ be a sequence of hybrid games such that each pair of subsequent games can be simulated and proven (t, ε)-indistinguishable by guessing only M bits of information on the adversary's choices. Then $G_{\mathsf{real}}, G_{\mathsf{random}}$ are $(t, \varepsilon \cdot L \cdot 2^M)$-indistinguishable.*

Now, the problem of proving user-mult security of GUS reduces to finding a sequence of indistinguishable hybrids such that each hybrid can be simulated using only a small amount of random guessing. Defining hybrid games via pebbling configurations as above and using the space-optimal pebbling sequence for directed acyclic graphs, described in [18, Algorithm 1], which uses $L = (2\deg(\tau_{\mathsf{mka}}))^d$ steps and only $(\deg(\tau_{\mathsf{mka}}) + 1) \cdot d$ pebbles, where d is the depth of τ_{mka}, implies a security reduction for GUS with only quasipolynomial loss in security.

Theorem 6. *If* dprf *is a $(t_{\mathsf{dprf}}, \varepsilon_{\mathsf{dprf}})$-secure dPRF,* uske $=$ (UEnc, UDec) *is a $(t_{\mathsf{cpa}*}, \varepsilon_{\mathsf{cpa}*})$-CPA*-secure USKE scheme, and τ_{mka} is a $(s_{tree}, s_{skel}, d, \deg(\tau_{\mathsf{mka}}))$-tree, then* GUS *is $(s_{tree}, \deg(\tau_{\mathsf{mka}}) \cdot s_{skel}, \deg(\tau_{\mathsf{mka}}) \cdot s_{skel}, s_{skel}, t, q_c, n_{max}, \varepsilon)$-*user-mult *secure for $\varepsilon = (Q \cdot \varepsilon_{\mathsf{dprf}} + 2\varepsilon_{\mathsf{cpa}*} \cdot \deg(\tau_{\mathsf{mka}})) \cdot (2\deg(\tau_{\mathsf{mka}}))^d \cdot Q^{\deg(\tau_{\mathsf{mka}}) \cdot d + 1}$.*

Proof. Observe that the challenge graph is a directed acyclic graph of degree $\deg(\tau_{\mathsf{mka}})$ and depth d, and let $(\mathcal{P}_0, \ldots, \mathcal{P}_L)$ be the recursive pebbling strategy for directed acyclic graphs from [18, Algorithm 1] which uses $L = (2\deg(\tau_{\mathsf{mka}}))^d$ steps and at most $(\deg(\tau_{\mathsf{mka}}) + 1) \cdot d$ pebbles. We will prove that each pebbling configuration $\mathcal{P}_\ell$ can be represented using $M = (\deg(\tau_{\mathsf{mka}}) \cdot d + 1) \cdot (\log Q)$ bits. The theorem then follows from Lemma 3 and Theorem 5.

We use the following property of the pebbling strategy: For all $\ell \in [0, L]$, there exists a leaf in τ_{mka} such that all pebbled nodes lie either on the path from that leaf to the sink or on the copath. Furthermore, the subgraph on this set of potentially pebbled nodes contains at most $\deg(\tau_{\mathsf{mka}}) \cdot d + 1$ nodes. Throughout the game, the reduction always knows in which position in the binary tree a node ends up, but it does not know which of the up to Q versions of the node will end up in the challenge tree. Note: we do not need to guess the number of USKE keys at each node u in the challenge graph, nor, if it is a leaf, how many dPRF keys it absorbs. This is because we can just fake encryptions (resp. dPRF computations) until the dPRF key corresponding to the version of the child which we guess to be in the challenge graph is encrypted (resp. computed) under u. So, the reduction needs to only guess for at most $\deg(\tau_{\mathsf{mka}}) \cdot d + 1$ nodes, which of the up to Q versions of that node will be in the challenge graph. This proves the security loss in the claim.

It is obvious that correctness holds for the protocol. The adversary is not allowed to modify messages transmitted over the network and thus the protocol does not allow the group to split: all users process the same message output by the group manager at each epoch t that allows them to obtain the group secret.

It is clear that the worst-case space complexity of the group manager state is always s_{tree}, since she just stores the whole tree and its secrets. The worst-case communication and time complexity of the group manager for add, remove, and update operations is $\deg(\tau_{\mathsf{mka}}) \cdot s_{\text{skel}}$ since for each operation, the group manager creates a skeleton of size s_{skel} and generates secrets and ciphertexts for at most all of the edges from each node in skeleton, including all edges to the frontier, whose number is bounded by $\deg(\tau_{\mathsf{mka}})$. The time complexity of users per add, remove, and update operations is s_{skel}, since the `PathRegen` subroutine traverses along a path in skeleton for each operation whose size is indeed bounded by s_{skel}. □

C.3 Security Proof for GUS in the ROM

To show the security of GUS in the ROM we use and adapt the results of Alwen et al. [3] on Generalized Selective Decryption (GSD) to the USKE setting for our purposes:

Definition 19 (Generalized Selective Decryption (GSD), adapted from [3, 26]). *Let* $(\mathsf{UEnc}, \mathsf{UDec})$ *be a USKE scheme with secret key space* $\mathcal{K}$ *and message space* $\mathcal{M}$ *such that* $\mathcal{K} \subseteq \mathcal{M}$*. The GSD game (for USKE schemes) is a two-party game between challenger* C *and an adversary* $\mathcal{A}$*. On input an integer* N*, for each* $v \in [N]$ *the challenger* C *picks a uniformly random initial key* k_0^v *and intializes the key graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}) := ([N], \emptyset)$ *and the set of corrupt users* $\mathcal{C} = \emptyset$*.* $\mathcal{A}$ *can adaptively issue the following queries:*

- $(\textbf{\textit{encrypt}}, u, v)$*: On input two nodes* u *and* v*,* C *returns an encryption* $(k_{i+1}^u, c) \leftarrow \mathsf{UEnc}(k_i^u, k_0^v)$ *of the* 0*-th version of the key at node* v*,* k_0^v*, under the current version of the key at node* u*,* k_i^u*, adds the directed edge* (u, v) *to* $\mathcal{E}$*, and replaces the current version of* k_i^u *at* u *with new version* k_{i+1}^u*. Each pair* (u, v) *can only be queried at most once.*
- $(\textbf{\textit{corrupt}}, v)$*: On input a node* v*,* C *returns* k_i^v*, the current version of the key at node* v *and adds* v *to* $\mathcal{C}$*.*
- $(\textbf{\textit{chall}}, v)$*, single access: On input a challenge node* v*,* C *samples* $b \leftarrow_\$ \{0, 1\}$ *and returns* k_i^v *(the current version of the key at node* v*) if* $b = 0$*, otherwise it outputs a uniformly random* $k \in \mathcal{K}$*. In the context of GSD we denote the* challenge graph *as the graph induced by all nodes from which the challenge node* v *is reachable. We require that none of the nodes in the challenge graph are in* $\mathcal{C}$*, that* $\mathcal{G}$ *is acyclic and that the challenge node* v *is a sink – thus* $k_i^v = k_0^v$*.*

Finally, $\mathcal{A}$ *outputs bit* b' *and it wins the game if* $b' = b$*. We call the encryption scheme* (t, ε)*-adaptive GSD-secure if for any adversary* $\mathcal{A}$ *running in time* t *it holds*

$$\mathsf{Adv}_{\mathsf{GSD}}(\mathcal{A}) := |\Pr[\mathcal{A} \to 1 | b = 1] - \Pr[\mathcal{A} \to 1 | b = 0]| \leq \varepsilon.$$

We use the following result of [3], which they prove for a public key encryption version of GSD, and can be easily adapted to our setting of USKE GSD.

Theorem 7. *For any USKE scheme* $\mathsf{uske} = (\mathsf{UEnc}, \mathsf{UDec})$ *and hash function* H*, let the encryption scheme* $\mathsf{uske}' = (\mathsf{UEnc}', \mathsf{UDec}')$ *be defined as follows: 1. It samples* k_s, k_d *randomly for its initial key* $k_0' = (k_s, k_d)$*. 2. Then, the first time it uses* UEnc' *or* UDec'*, it computes* $k_0 \leftarrow \mathsf{dprf}(k_s, k_d)$*, and uses* k_0 *as the first input to* $\mathsf{UEnc}, \mathsf{UDec}$*, with the corresponding message or ciphertext. 3. Then, for the next use of* UEnc' *(resp.* UDec'*), it uses the key* k_1 *(resp.* k_1'*) output by* UEnc *(resp.* UDec*) in (2) above as input to* UEnc *(resp.* UDec*) again; and proceeds like this for all subsequent computations.*

If uske *is* $(t, \varepsilon_{\mathsf{cpa}*})$*-CPA* secure and* dprf *is modelled as a random oracle* H*, then* uske' *is* $(t, \varepsilon_{\mathsf{GSD}})$*-adaptive GSD secure, where* $\varepsilon_{\mathsf{GSD}} = 2N^2 \cdot \varepsilon_{\mathsf{cpa}*} + \frac{mN}{2^{\ell-1}}$*, with* N *the number of nodes,* m *the number of oracle queries to* H*,* ℓ *the length of dPRF keys.*

The theorem can be proved almost identically to that of [3, Theorem 3]. We provide a sketch below for exposition.

Proof (Sketch). We prove GSD security by a sequence of hybrids inerpolating between the *real* game GSD_0 where the challenge query is answered with real key $k_0^v (= (k_s^v, k_d^v))$ and the *random* game GSD_1 where it is answered with an independent uniformly random key in $\mathcal{K}^2$ (where we assume the dPRF key space is the same as the USKE key space).

- Define $G_0 := \mathsf{GSD}_0$, the real GSD game.
- Let $k' \in \mathcal{K}^2$ and v be the challenge node. For $1 \leq i \leq \mathsf{indeg}(v)$ we define the hybrid game G_i as follows: The game is similar to G_{i-1} except that the i-th query of the form $(\textbf{encrypt}, u, v)$ is answered by $\mathsf{UEnc}(k_i^u, k' (= (k_s', k_d'))$.

Observe that the game $G_{\mathsf{indeg}(v)}$ is distributed exactly the same as GSD_1. Therefore, in this case, for any GSD-adversary $\mathcal{A}$ with advantage ε, the advantages of $\mathcal{A}$ in distinguishing hybrid games G_{i-1} from G_i sum up to at least ε. Since two subsequent hybrid games differ in exactly one encryption edge, we will use this distinguishing advantage to solve a CPA* challenge. To simulate the game G_i, the reduction simply guesses the challenge node v as well as the source node u of the i-th encryption edge to v. We denote these guesses v^* and u^*, respectively. However, this simulation is only possible if $\mathcal{A}$ does not query its oracle H on any of the pairs of dPRF keys corresponding to the parents of v^* (in the traditional graph-theoretic sense, i.e., those that nodes that have an edge to v^*), since otherwise $\mathcal{A}$ can trivially distinguish G_0 from $G_{\mathsf{indeg}(v)}$. Alwen et al. [3] encompass this issue by the following (more general) event.

– Event E: $\mathcal{A}$ queried a pair of dPRF keys (k_s, k_d) to the random oracle which correspond to the initial uske' key for a node that was not corrupted and is not reachable by any corrupted node, and no challenge query was issued for it.

Intuitively, event E is true if $\mathcal{A}$ queried the random oracle H on some pair of dPRF keys which it does not trivially know and which is associated with a node that might end up in the challenge graph. Note that if E is not true, we do not need to recursively fake any encryptions of k^{u^*} to fake the encryption query of v^* under u^*, because the first uske key $k_{e,0}$ that is generated at u^* is done so via an H computation on a pair of dPRF keys such that (at least one of them) is not known to $\mathcal{A}$. Thus for any reduction algorithm $\mathcal{B}$ that needs to answer an encryption query of adversary $\mathcal{A}$ with target u^*, $\mathcal{B}$ can just assign a random uske' key k' to u^* and encrypt k' instead of k^{u^*}. As long as $\mathcal{A}$ never queries the random oracle on k^{u^*}, then using k' in the encryption is information-theoretically indistinguishable to $\mathcal{A}$ from using k^{u^*}.

In fact, whether or not an adversary $\mathcal{A}$ is successful in triggering event E in games GSD_0 and GSD_1, Alwen et al. [3] show how to obtain an reduction algorithm with advantage $> \frac{\varepsilon}{2N^2} - \frac{m}{N2^\ell}$ against the CPA security of a PKE scheme. Using almost the exact same techniques (except querying the USKE encryption oracle when needed for non-challenge encryptions, as opposed to the reduction algorithm generating them itself using the public key given to it by the challenger), we can obtain a reduction algorithm with the same advantage against the CPA* security of uske. □

Now, as [3] does, we can adapt the above theorem to show a polynomial time reduction for GUS in the ROM. Intuitively, the $\mathsf{GUS\text{-}user\text{-}mult}$ graph $\mathcal{G}_{\mathsf{user\text{-}mult}}$ corresponds to a GSD graph in the above sense (i.e., for the transformed uske', where dprf is replaced by the random oracle H), with two differences:[20]

1. There are additional edges corresponding to initial keys that are derived from each other via H computations; and
2. A given leaf ℓ_{ID} of the challenge graph $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$ may contain x sequences of USKE keys, where x is the number of consecutive times ID is updated before the challenge epoch t^* (and without any subsequent updates after the x updates and before t^*) such that the oob message for the update is corrupted by $\mathcal{A}$.

The following Theorem shows that this differences do not impact security.

Theorem 8. *If the USKE scheme in* GUS *is* $(t, \varepsilon_{\mathsf{cpa}*})$*-secure and* dprf *is modelled as a random oracle* H*, then* GUS *is* (t, Q, ε)-$\mathsf{user\text{-}mult}$*-secure, where* $\varepsilon = \varepsilon_{\mathsf{cpa}*} \cdot 2(s_{tree} \cdot Q)^2 + \mathsf{negl}$.[21]

Proof (Sketch). In order to adapt the proof of Theorem 7 to the $\mathsf{GUS\text{-}user\text{-}mult}$ graph $\mathcal{G}_{\mathsf{user\text{-}mult}}$, we begin by accounting for the first case above: i.e., keys can be derived from each other via H computations. This can be handled easily, since in the GSD game, the reduction can just sample all initial keys randomly and independently as before, and when nodes u get corrupted, it can just program H to ensure consistency for keys k_0^v, which are generated via a computation of H on a key at corrupted nodes u. In the case where event E from Theorem 7 happens with low probability (cf. Lemma 6 of [3]), we still obtain indistinguishability, since as before the computation of H on keys at u occurs with low probability. Furthermore, in the case where event E happens with large probability (cf. Lemma 5, Corollary 2 of [3]), the main observation that Alwen et al. make is that it is still sufficient to simulate the GSD game correctly until E happens.

To account for the second case above: it is easy to see that when faking encryptions of the challenge node v^* to leaf ℓ_{ID}, we do so as normal to the latest USKE key of the most recent sequence of USKE keys in ℓ_{ID}, and indistinguishability still follows from the fact that the 0-th version of this most recent sequence was generated via an H computation on a key which is not leaked to $\mathcal{A}$ via Lemma 2. Thus, we can reach the same conclusions of event E as usual.

We conclude by observing that the $\mathsf{GUS\text{-}user\text{-}mult}$ graph $\mathcal{G}_{\mathsf{user\text{-}mult}}$ has at most $N = s_{\text{tree}}Q$ nodes and that by Lemma 2, none of the keys in the challenge graph $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$ are leaked if **user-safe** evaluates to true. □

[20] Also, corruptions have small differences due to the presence of **corrupt-oob** and **corrupt**, but we omit this due to simplicity and the fact that by Lemma 2, all nodes in the challenge graph are not *fully* leaked to $\mathcal{A}$ (i.e., at least one of k_s, k_d is not leaked).

[21] The efficiency of the protocol is the same as in the standard model theorem.

Hybrid H_j^p, $j \in \{0, 1, \ldots, d\}$ is as follows:

1. Initialize $(\tau_{\mathsf{eram}}, \mathrm{MK}) \leftarrow \mathsf{eram\text{-}init}(1^\lambda)$ as usual; set $\mathsf{Chall} \leftarrow \emptyset$. Define $H := H_{j+1}$.
2. For each call $\mathbf{corrupt}()$, process it as in H.
3. For each call $\mathbf{write}(d, i)$:
   (a) If $i \in \mathsf{Chall}$, set $\mathsf{Chall} \leftarrow \mathsf{Chall} \setminus \{i\}$.
   (b) Proceed as in H except that for each $l \in \mathsf{Chall}$:
   - Perform the generation of seeds and keys for the nodes $v \in \mathsf{skeleton}$ at depths $j' > j$ along the direct path of l (not including l) as in H.
   - For the node v at depth j, if $v \in \mathsf{skeleton}$ instead of using a randomly sampled seed s, or one obtained from its child, to compute $\mathsf{prg}(s) = (s'||k)$, simply uniformly at random sample $\$ \rightarrow (s'||k)$.
   - Encrypt its children as in H.
4. For each call $\mathbf{chall}(d_0, d_1, i)$: Set $\mathsf{Chall} \leftarrow \mathsf{Chall} \cup \{i\}$ and proceed as in step (b) of 3. above.

Hybrid H_j^c, $j \in \{0, 1, \ldots, d\}$ is as follows:

1. Initialize $(\tau_{\mathsf{eram}}, \mathrm{MK}) \leftarrow \mathsf{eram\text{-}init}(1^\lambda)$ as usual; set $\mathsf{Chall} \leftarrow \emptyset, D'[\cdot] \leftarrow \epsilon$. If $j = 0$, define $H = H_0^p$. Otherwise, define $H := H_{j-1}^c$.
2. For each call $\mathbf{corrupt}()$, process it as in H.
3. For each call to $\mathbf{write}(d, i)$:
   (a) If $i \in \mathsf{Chall}$, set $\mathsf{Chall} \leftarrow \mathsf{Chall} \setminus \{i\}$.
   (b) Proceed as in H except that for each $l \in \mathsf{Chall}$:
   - Replace all seeds and keys for the nodes $v \in \mathsf{skeleton}$ along the direct path of l with uniformly random values and perform all encryptions for those nodes v of depth $j' < j$ as in H.
   - Replace the data of the node v at depth j with instead $\mathsf{Enc}(k, 0)$, where $k = \mathrm{MK}$ if $j = 0$ and $k = k_{v.p}$ for the parent $v.p$ of v otherwise.
   - Set $D'[v] \leftarrow D[v]$.
   - If the key for a node w whose cell contains an encryption of 0 is needed, retrieve it from D', i.e. $k_w \leftarrow D'[w]$.
4. For each call $\mathbf{chall}(d_0, d_1, i)$: Set $\mathsf{Chall} \cup \{i\}$ and proceed as in step (b) of 3. above.

Fig. 9. Definition of hybrid experiments in the security proof of the encrypted RAM scheme.

D Proof of Theorem 3

Theorem 3 (Security of tRAM). *Assume that* prg *is a* $(t_{\mathsf{prg}}, \varepsilon_{\mathsf{prg}})$*-secure pseudorandom generator,* Π *is a* $(t_{\mathsf{cpa}}, \varepsilon_{\mathsf{cpa}})$*-CPA-secure symmetric-key encryption scheme, and* τ_{eram} *is a* $(s_{tree}, s_{skel}, s'_{skel}, \deg(\tau_{\mathsf{eram}}))$*-tree. Then,* tRAM *is a* $(O(1), s_{tree}, (s_{skel} + s'_{skel}) \cdot \deg(\tau_{\mathsf{eram}}), t, n_{max}, \varepsilon)$*-secure forward secret encrypted RAM protocol for* $\varepsilon = q s_{tree}(n_{max}, n_{max})(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}})$*, where* q *is the number of times* $\mathbf{write}()$ *or* $\mathbf{chall}()$ *is queried by* $\mathcal{A}$ *and* $t \approx t_{\mathsf{prg}} \approx t_{\mathsf{cpa}}$*.*

Proof. We consider the following experiments:

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

where H_d^p is the original encrypted RAM game and the remaining hybrids are defined in Figure 9. The difference between the p hybrids is in faking one more depth of PRG computations, while the difference between the c hybrids is in faking one more depth of encryptions under the CPA-secure SKE scheme.

Now we prove indistinguishability between hybrids. As in the proof of Theorem 2, we let $v_{i,j}$, $i \in [h], j \in [\mathsf{nb}_i]$ be the j-th node at depth i of τ_{eram}, where nb_i is the number of nodes at depth i.

Lemma 4. *For* $j \in [d]$*:*

1. $\mathsf{Adv}^{\mathsf{tRAM}}_{H_j^p}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{tRAM}}_{H_{j-1}^p}(\mathcal{A}) + q \deg(\tau_{\mathsf{eram}})^{j-1} \varepsilon_{\mathsf{prg}}$.
2. $\mathsf{Adv}^{\mathsf{tRAM}}_{H_j^c}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{tRAM}}_{H_{j+1}^c}(\mathcal{A}) + q \deg(\tau_{\mathsf{eram}})^{j+1} \varepsilon_{\mathsf{cpa}}$.
   Also, $\mathsf{Adv}_{H_0^p}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{tRAM}}_{H_0^c}(\mathcal{A}) + q\varepsilon_{\mathsf{cpa}}$,

where q *is the number of queries* $\mathcal{A}$ *makes to* $\mathbf{write}()$ *and* $\mathbf{chall}()$*.*

Proof. The proof is by contradiction. Let j_1 be the minimum value in $[d]$ for which relation (1.) does not hold, or if relation (1.) holds for all such $j \in [d]$, let $j_2 = 0$ if $\mathsf{Adv}_{H_0^p}(\mathcal{A}) > \mathsf{Adv}^{\mathsf{tRAM}}_{H_0^c}(\mathcal{A}) + \varepsilon_{\mathsf{cpa}}$, or otherwise the maximum value in $[d]$ for which (2.) does not hold, and in the sequence of hybrids, at least two adjacent hybrids are distinguishable. For all previous values of j_1 and/or j_2, we assume the relations hold. We consider two cases:

Case a) Assuming relation (1.) does not hold, for index j, we make a reduction to the security of the PRG. We define the adversary $\mathcal{B}_j^p$ in Figure 10.

Algorithm $\mathcal{B}_j^p$:

1. Query the oracle of the PRG experiment and receive the challenge

$$\left(s^1_{v_{j-1,1}}||k^1_{v_{j-1,1}}\right), \ldots, \left(s^{m_{v_{j-1,1}}}_{v_{j-1,1}}||k^{m_{v_{j-1,1}}}_{v_{j-1,1}}\right), \ldots, \left(s^1_{v_{j-1,\mathsf{nb}_{j-1}}}||k^1_{v_{j-1,\mathsf{nb}_{j-1}}}\right),$$

$$\ldots, \left(s^{m_{v_{j-1,\mathsf{nb}_{j-1}}}}_{v_{j-1,\mathsf{nb}_{j-1}}}||k^{m_{v_{j-1,\mathsf{nb}_{j-1}}}}_{v_{j-1,\mathsf{nb}_{j-1}}}\right),$$

   where $m_{v_{j-1,g}}$, $g \in [\mathsf{nb}_{j-1}]$ is the number of queries to **write**() or **chall**() in which the PRG computation at g-th node of depth $j-1$ is replaced with a uniformly random value in H^p_{j-1}.
2. During the h-th such query for each node $v_{j-1,g}$, simulate H^p_j, with the only difference that the (seed, key) pair for the node at depth $j-1$ is set to $\left(s^h_{v_{j-1,g}}||k^h_{v_{j-1,g}}\right)$.
3. For the rest of the execution simulate H^p_j and output 1 if and only if $\mathcal{A}$ outputs $b = b'$.

Fig. 10. Reduction algorithm $\mathcal{B}_j^p$.

The difference between H^p_j and H^p_{j-1} is that in each query to **write**() or **chall**(), the latter may fake one more level of PRG computations. In particular, in H^p_j, only the PRG computations up to depth j are faked, while in H^p_{j-1}, those at depth $j-1$ is also faked. Now observe that if the challenge of the PRG game consists of uniformly random values, $\mathcal{B}^p_j$, simulates the execution of H^p_{j-1}. On the other hand, if the challenge is output by the PRG, $\mathcal{B}^p_j$, simulates the execution of H^p_j.

We formally argue that the simulation described above is correct. By assumption, we have that the hybrids $H^p_d, \ldots, H^p_j$ are computationally indistinguishable and we need to prove that $H^p_j \approx H^p_{j-1}$. Observe that the only way for the adversary to distinguish between the two hybrids is by distinguishing a uniformly random $(s'||k)$ from one that is output by the PRG on some uniformly random seed s. Since $\mathcal{A}$ cannot corrupt the encrypted RAM before overwriting any cells $i \in \mathsf{Chall}$, no key along the direct path of any cell i before it is overwritten is leaked to the adversary. Thus, we conclude that the above simulation is correct, without $\mathcal{B}^p_j$ accessing the seed of the PRG game. Hence, if $\mathsf{Adv}^{\mathsf{tRAM}}_{H^p_j}(\mathcal{A}) > \mathsf{Adv}^{\mathsf{tRAM}}_{H^p_{j-1}}(\mathcal{A}) + q\deg(\tau_{\mathsf{eram}})^{j-1}\varepsilon_{\mathsf{prg}}$, $\mathcal{B}^p_j$ breaks the security of the PRG in the game, reaching a contradiction. We conclude that

$$\mathsf{Adv}^{\mathsf{tRAM}}_{H^p_j}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{tRAM}}_{H^p_{j-1}}(\mathcal{A}) + q\deg(\tau_{\mathsf{eram}})^{j-1}\varepsilon_{\mathsf{prg}}.$$

Case b) For the second case, we make a reduction to the CPA security of the encryption scheme. We define the adversary $\mathcal{B}^c_j$ in Figure 11.

Algorithm $\mathcal{B}^c_j$:

1. For any query to **write**() or **chall**(), simulate H^c_j (or H^p_0) with the only differnce being:
   - For the parents p of nodes $v_{j+1,g}$, $g \in [\mathsf{nb}_{j+1}]$ such that $v_{j+1,g}$ is on the direct path of some cell $i \in \mathsf{Chall}$, disregard its sampled key k_p.
   - For any siblings w of such nodes $v_{j+1,g}$ that are themselves not along the direct path of any such i, query the encryption oracle for node $v_{j+1,g}$ on $D[w]$, the data of w, receive ciphertext $c_{v_{j+1,g}}$ and write $c_{v_{j+1,g}}$ to $v_{j+1,g}$).
   - Set $M_0 \leftarrow D[v_{j+1,g}]$, and $M_1 \leftarrow \mathsf{0}$, and send them to the CPA challenger for $v_{j+1,g}$.
   - Write the returned ciphertext c^* to $v_{j+1,g}$).
2. For the rest of the execution simulate H^c_j (or H^p_0) and output 1 if and only if $\mathcal{A}$ outputs $b = b'$.

Fig. 11. Reduction Algorithm $\mathcal{B}^c_j$.

The difference between hybrids H^c_j, H^c_{j+1} (resp. H^p_0, H^c_0) is that the latter may fake one more level of encryption for each query to **write**() or **chall**(). In H^c_j (resp. H^p_0), we have that the ciphertexts down to depth j (resp. none of the ciphertexts) are faked, while in H^c_{j+1}, we also fake the ciphertexts of the

nodes at depth $j+1$ (resp. depth 0). H_j^c writes into the RAM the real encryptions of $D[v_{j+1,g}]$ at cells for nodes $v_{j+1,g}$, c_0^*, while H_{j+1}^c writes into the RAM the fake encryptions of $\mathsf{0}$, c_1^*. Clearly, if they are c_0^*, $\mathcal{B}_j^c$ simulates H_j^c (resp. H_0^p), while if they are c_1^*, $\mathcal{B}_j^c$ simulates H_{j+1}^c (resp. H_0^c).

Now we formally argue that the above simulation is correct. Again, since $\mathcal{A}$ cannot corrupt the encrypted RAM before overwriting any cells $i \in \mathsf{Chall}$, no key along the direct path of any cell i is leaked to the adversary before it is overwritten. In addition, despite the fact that $\mathcal{B}_j^c$ is writing encryptions of $\mathsf{0}$ to the RAM, the real data d' are stored in D' and retrieved if necessary. Moreover, $\mathcal{B}_j^c$ can equivalently simulate the view of $\mathcal{A}$ at any point in which $\mathsf{Chall} = \emptyset$, because all nodes will have been overwritten with real encryptions at this point, since only nodes on the direct path of challenge cells ever contain encryptions of $\mathsf{0}$. Thus, if $\mathsf{Adv}^{\mathsf{tRAM}}_{H_j^c}(\mathcal{A}) > \mathsf{Adv}^{\mathsf{tRAM}}_{H_{j+1}^c}(\mathcal{A}) + q\deg(\tau_{\mathsf{eram}})^{j+1}\varepsilon_{\mathsf{cpa}}$ (resp. $\mathsf{Adv}_{H_0^p}(\mathcal{A}) > \mathsf{Adv}^{\mathsf{tRAM}}_{H_0^c}(\mathcal{A}) + q\varepsilon_{\mathsf{cpa}}$), $\mathcal{B}_j^c$ breaks the CPA-security of the SKE game, reaching a contradiction. We conclude that

$$\mathsf{Adv}^{\mathsf{tRAM}}_{H_j^c}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{tRAM}}_{H_{j+1}^c}(\mathcal{A}) + q\deg(\tau_{\mathsf{eram}})^{j+1}\varepsilon_{\mathsf{cpa}},$$

$$\mathsf{Adv}_{H_0^p}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{tRAM}}_{H_0^c}(\mathcal{A}) + q\varepsilon_{\mathsf{cpa}}.$$

Thus, the proof of the lemma is concluded. □

Total security loss. From the above, we have that the total security loss for all hybrids is at most

$$\sum_{j=0}^{d} q\deg(\tau_{\mathsf{eram}})^j(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}}) \leq q s_{\mathrm{tree}}(n_{\max}, n_{\max})(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}}).$$

Correctness. Correctness is obvious since the scheme stores MK, which can decrypt the root, and every node in the tree decrypts its children, down to the data at the leaves. So at any time, data at any cell can be retrieved correctly.

Protocol Efficiency. It is clear that the space complexity of MK and τ_{eram} are $O(1)$ and s_{tree}, respectively. It is also clear that the time complexity of $\mathsf{eram\text{-}read}$, $\mathsf{eram\text{-}write}$ operations is $O(s_{\mathrm{skel}} + s'_{\mathrm{skel}}) \cdot \deg(\tau))$ since the $\mathsf{skeleton\text{-}modify}$ operation visits each node in both of the skeletons as well as their frontier at most twice, and performs a constant number of operations.

Conclusion. The final hybrid is H_d^c. In that hybrid, the contents of each challenged cell i (for each time they are challenged), is independent of d_b since it has been replaced by an encryption of the zero message, and moreover all other information in the encrypted RAM is independent of d_b. Thus $\mathsf{Adv}^{\mathsf{tRAM}}_{H_d^c}(\mathcal{A}) = 0$ and given the above:

$$\mathsf{Adv}^{\mathsf{tRAM}}_{\mathsf{enc-RAM}}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{tRAM}}_{H_d^p}(\mathcal{A}) \leq q s_{\mathrm{tree}}(n_{\max}, n_{\max})(\varepsilon_{\mathsf{prg}} + \varepsilon_{\mathsf{cpa}}).$$

Now, the complete theorem has been proved. □

E GMS Security Proofs

In this section, we provide two security proofs of our construction GMS against the adaptive, partially active adversary in the $\mathsf{mgr\text{-}mult}$ game – one in the standard model achieving Quasi-polynomial security loss, and another in the Random Oracle model achieving polynomial security loss. Thus, we prove Theorem 4. We again use the proofs of Tainted TreeKEM in [3] as a template.

We will prove security against an adversary that issues a *single* challenge query. Using a standard hybrid argument,[22] we can show that if the protocol is $(s_{\mathrm{tree}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, s_{\mathrm{skel}}, t, 1, n_{\max}, \varepsilon)$-secure then it is $(s_{\mathrm{tree}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, \deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}, s_{\mathrm{skel}}, t, q_c, n_{\max}, \varepsilon')$-secure, where $\varepsilon' = q_c \cdot \varepsilon$. We omit it for brevity.

[22] For example, one very similar to Lemma 6 from [1].

Challenge Graph As in Section C, we will argue for the security of GMS in the framework of Jafargholi et al. [21]. To do so, we need to view the mgr-mult game for GMS as a game on a graph, then define the *challenge graph* for group key I^* as a subgraph of the whole GMS graph.

We really only need to address how the use of Forward Secret encrypted RAM in GMS affects the GUS-user-mult graph $\mathcal{G}_{\text{user-mult}}$ of Definition 16 and the GUS-user-mult-challenge graph $\mathcal{G}^*_{\text{user-mult}}$ of Definition 17. In short, there is not much effect: The Forward Secret encrypted RAM eram generates a sequence of master keys $\text{MK}_0, \ldots, \text{MK}_l$, each of which may be used to recover some of the keys of the MKA tree τ_{mka} in a given epoch. Thus we introduce to $\mathcal{G}_{\text{user-mult}}$ one node $u_{\text{MK}} = \{\text{MK}_0, \ldots, \text{MK}_l\}$ and edges from u_{MK} to all other nodes in $\mathcal{G}_{\text{user-mult}}$. Formally:

Definition 20 (GMS-mgr-mult graph). *The* GMS-mgr-mult graph $\mathcal{G}_{\text{mgr-mult}} = (\mathcal{V}_{\text{mgr-mult}}, \mathcal{E}_{\text{mgr-mult}})$ *is the same as* $\mathcal{G}_{\text{user-mult}} = (\mathcal{V}_{\text{user-mult}}, \mathcal{E}_{\text{user-mult}})$ *of Definition 16, with the additions*

- $\mathcal{V}_{\text{mgr-mult}} = \mathcal{V}_{\text{user-mult}} \bigcup \{u_{\text{MK}}\}$*; and*
- $\mathcal{E}_{\text{mgr-mult}} = \mathcal{E}_{\text{user-mult}} \bigcup_{u \in \mathcal{V}_{\text{user-mult}}} \{(u_{\text{MK}}, u)\}$.

Additionally for the challenge graph, we ensure that the FS eRAM master key node, u'_{MK}, only contains those FS eRAM master keys $\text{MK}_i \in \{\text{MK}_0, \ldots, \text{MK}_l\}$ such that for some key k that is contained in some node u of $\mathcal{V}^*_{\text{user-mult}}$ and is written to virtual cell w of eram (corresponding to node w of τ_{mka} that is present in the challenge epoch), at the point in which MK_i was generated, the write containing k was the most recent write to virtual cell w of eram (including deletions – $\perp$).

Definition 21 (GMS-mgr-mult-challenge graph). *The* GMS-mgr-mult-challenge graph $\mathcal{G}^*_{\text{mgr-mult}} = (\mathcal{V}^*_{\text{mgr-mult}}, \mathcal{E}^*_{\text{mgr-mult}})$ *is the same as* $\mathcal{G}^*_{\text{user-mult}} = (\mathcal{V}^*_{\text{user-mult}}, \mathcal{E}^*_{\text{user-mult}})$ *of Definition 17, with the additions*

- $\mathcal{V}^*_{\text{mgr-mult}} = \mathcal{V}^*_{\text{user-mult}} \bigcup \{u'_{\text{MK}}\}$*, where* $u'_{\text{MK}} =$[23]

$$\left\{ \begin{array}{c} \text{MK}_i \in \{\text{MK}_0, \ldots, \text{MK}_l\} : \exists u \in \mathcal{V}^*_{\text{user-mult}}, k \in u, w, d, k \in d, j \leq i. \\ (\cdot, \text{MK}_j) \leftarrow \text{eram-write}(\cdot, \text{MK}_{j-1}, d, w) \wedge \forall r \in [j, i], \\ \cdot, (\text{MK}_r) \leftarrow \text{eram-write}(\cdot, \text{MK}_{r-1}, d', w'), w' \neq w \end{array} \right\};$$

  and
- $\mathcal{E}^*_{\text{mgr-mult}} = \mathcal{E}^*_{\text{user-mult}} \bigcup_{u \in \mathcal{V}^*_{\text{user-mult}}} \{(u'_{\text{MK}}, u)\}$.

Thus, those MK_i in u'_{MK} correspond to those master keys which by the correctness of eram can read some key k that is contained in a node of $\mathcal{V}^*_{\text{user-mult}}$ from virtual cell w of eram. Thus, we have an analog of Lemma 2 for $\mathcal{G}^*_{\text{mgr-mult}}$, which stipulates that none of the keys in the graph can be leaked to the adversary, according to predicate **mgr-safe**.

Lemma 5. *For queries* $\mathbf{q}_1, \ldots, \mathbf{q}_Q$ *made by an adversary* $\mathcal{A}$ *in* mgr-mult*, if* **mgr-safe**$(\mathbf{q}_1, \ldots, \mathbf{q}_Q) \rightarrow$ true*, it holds that none of the keys contained in nodes of* $\mathcal{G}^*_{\text{mgr-mult}}$ *are leaked to* $\mathcal{A}$ *via corruption.*

Proof. Intuitively, in addition to the usual conditions of **user-safe**, **mgr-safe** also ensures that if the group manager was corrupted before the challenge operation op^* at t^*, then all users ID in the group at that time were since removed or updated in an operation such that the oob message is never corrupted. After these operations, the master key MK which is corrupted can only be used to derive old versions of keys in τ_{mka}, since all virtual cells in eram will have been overwritten during the operations to remove or update the users in the group at the time of the group manager corruption. Thus, no information on the keys in the graph is leaked to the adversary via corruption.

Formally, assume that some key k in some $i \in \mathcal{V}^*_{\text{mgr-mult}}$ is leaked to $\mathcal{A}$ via corruption. We will show that **mgr-safe**$(\mathbf{q}_1, \ldots, \mathbf{q}_Q) \rightarrow$ false, a contradiction. There are two cases:

1. $k = \text{MK} \in u'_{\text{MK}}$ is some eram master key that by definition of $\mathcal{G}^*_{\text{mgr-mult}}$ can be used to derive some key k that is contained in a node $u \in \mathcal{V}^*_{\text{user-mult}}$ and data d written to virtual cell w of eram, corresponding to node w of τ_{mka}. In this case, $\mathcal{A}$ queried **mgr-corrupt**() in some epoch $t \leq t^*$, in which d was most recently written to w in eram. However, it is thus clear that before the challenge epoch t^* and after epoch t, w was never overwritten during some query **update-user**(ID) or **remove-user**(ID) by $\mathcal{A}$, for some ID that has w on its direct path in τ_{mka}. Thus $\mathcal{A}$ clearly violates **mgr-safe**.

[23] $k \in d$ means that k is included in the data d written to virtual cell w.

2. Otherwise, k is in some $i \in \mathcal{V}^*_{\mathsf{user\text{-}mult}}$. Since from above it must be that k was not leaked to $\mathcal{A}$ through some eram operation using a master key MK that was leaked to $\mathcal{A}$ via a **mgr-corrupt**() query: It must be that k was leaked to $\mathcal{A}$ through some **corrupt**() or **corrupt-oob**() operation. The lemma thus follows from Lemma 2, since **mgr-safe**$(\mathbf{q}_1, \ldots, \mathbf{q}_Q) \to \mathsf{true}$ implies **user-safe**$(\mathbf{q}_1, \ldots, \mathbf{q}_Q) \to \mathsf{true}$. □

Security Proof for GMS in the Standard Model We will again use the framework of Jafargholi et al. [21] for security of GMS in the standard model. We will first consider the selective mgr-mult game, where the adversary needs to schedule its queries all at once at the beginning of the game. We will call the two possible executions of the game as the real and random mgr-mult games and prove their indistinguishability through a sequence of hybrid games. We will again define these hybrids via the reversible black pebbling game of Bennett [8] described in Definition 18, but this time on $\mathcal{G}^*_{\mathsf{mgr\text{-}mult}}$. If a node i in $\mathcal{G}^*_{\mathsf{mgr\text{-}mult}}$ which is also in $\mathcal{G}^*_{\mathsf{user\text{-}mult}}$ is pebbled, then in this hybrid game, $(k_p||k^i_d||k^i_{e,0})$ is sampled uniformly at random as before. However, if $i = u'_{\mathrm{MK}}$, then in this hybrid game, all FS eRAM writes $(\mathrm{M}', \mathrm{MK}') \leftarrow \mathsf{eram\text{-}write}(\mathrm{M}, \mathrm{MK}, d, w)$, for d containing some key k that is in a node u of $\mathcal{V}^*_{\mathsf{user\text{-}mult}}$, are replaced with $(\mathrm{M}', \mathrm{MK}') \leftarrow \mathsf{eram\text{-}write}(\mathrm{M}, \mathrm{MK}, 0, w)$.

By Lemma 5, we know that none of the keys in the challenge graph are leaked to the adversary throughout the entire game. This will allow us to prove indistinguishability of consecutive hybrid games from dPRF security, USKE security, and FS eRAM security.

Lemma 6. *Let $(\mathcal{P}_0, \ldots, \mathcal{P}_L)$ be a valid pebbling sequence on the challenge graph. If* dprf *is a $(t_{\mathsf{dprf}}, \varepsilon_{\mathsf{dprf}})$-secure dPRF,* $\mathsf{uske} = (\mathsf{UEnc}, \mathsf{UDec})$ *is a $(t_{\mathsf{cpa}*}, \varepsilon_{\mathsf{cpa}*})$-CPA*-secure USKE scheme, and* eram *is a $(t_{\mathsf{eram}}, \varepsilon_{\mathsf{eram}})$-secure FS eRAM protocol, then any two consecutive hybrid games $H_\ell, H_{\ell+1}$ are $(t, \varepsilon_{\mathsf{eram}} + Q\varepsilon_{\mathsf{dprf}} + 2\varepsilon_{\mathsf{cpa}*} \cdot \deg(\tau_{\mathsf{mka}}))$-indistinguishable for $t \approx t_{\mathsf{dprf}} \approx t_{\mathsf{cpa}*}$.*[24]

Proof. Let $H_\ell, H_{\ell+1}$ be two consecutive hybrid games. We assume that $\mathcal{P}_{\ell+1}$ has one additional pebble on node v^* that $\mathcal{P}_\ell$ does not have. The case where $\mathcal{P}_{\ell+1}$ is obtained from $\mathcal{P}_\ell$ by removing one pebble can be proved in a similar way.

For the cases where $v^* \neq u'_{\mathrm{MK}}$, indistinguishability of H_ℓ, H_{ℓ_1} follows from the same argument as Lemma 3 and the fact that Lemma 5 ensures that none of the keys in the challenge graph are leaked to $\mathcal{A}$, so that the corresponding reductions $\mathcal{B}$ can perfectly simulate their hybrids.

When $v^* = u'_{\mathrm{MK}}$, $\varepsilon_{\mathsf{eram}}$-indistinguishability of $H_\ell, H_{\ell+1}$ follows from the FS eRAM security of eram: We can construct a reduction algorithm $\mathcal{B}$ against the FS eRAM challenger that simply queries the **write**() oracle of the challenger on all data d that it normally writes to eram and that does not contain any keys k that are contained in a node $u \in \mathcal{V}^*_{\mathsf{user\text{-}mult}}$. Then, for GMS writes of data d^* to eram that do contain some key k that is contained in a node $u \in \mathcal{V}^*_{\mathsf{user\text{-}mult}}$, $\mathcal{B}$ can query the **chall**() oracle of the FS eRAM challenger on d^* and 0. $\mathcal{B}$ can perfectly simulate the rest of the hybrid, since by Lemma 5, none of the master keys MK that can read any challenge data d^* from eram by correctness of eram are leaked to $\mathcal{A}$ via **mgr-corrupt**() corruptions. Moreover, it can just keep track of the keys in τ_{mka} itself, so that it can use them when needed, and ignore the contents of the public cells M of eram. Thus any advantage ε of an adversary in distinguishing $H_\ell, H_{\ell+1}$ leads to the same advantage of $\mathcal{B}$ in the FS eRAM security game on eram.

It is clear that in the reductions, the running time remains essentially the same. Thus, the lemma is proved. □

As for the user-mult security of GUS, with a trivial pebbling sequence of the challenge graph, the above already implies selective mgr-mult security of GMS. However, in the adaptive setting, the reduction that simulates a hybrid H_ℓ needs to guess some of the adaptive choices the adversary makes. We again use the framework of Jafargholi et al. [21], given in Theorem 5 and the space-optimal pebbling sequence for directed acyclic graphs, described in [18, Algorithm 1], which uses $L = (2\deg(\tau_{\mathsf{mka}}))^{d+1}$ steps and only $(\deg(\tau_{\mathsf{mka}})+1)\cdot(d+1)$ pebbles, to achieve quasi-polynomial loss in security. Note: we use $d+1$ to account for the extra FS eRAM master key node u'_{MK}.

[24] For many pairs of consecutive hybrid games (those where a pebble is neither being added nor removed from a leaf), the extra Q factor is not needed. Also, for many pairs of consecutive hybrid games (those where a pebble is neither being added nor removed from u'_{MK}), the extra $\varepsilon_{\mathsf{eram}}$ term is not needed.

Theorem 9. *If* dprf *is a* $(t_{\mathsf{dprf}}, \varepsilon_{\mathsf{dprf}})$*-secure dPRF,* uske = (UEnc, UDec) *is a* $(t_{\mathsf{cpa}*}, \varepsilon_{\mathsf{cpa}*})$*-CPA*-secure USKE scheme,* eram *is a* $(s_1^{\mathsf{eram}}, s_2^{\mathsf{eram}}, t_1^{\mathsf{eram}}, t_2^{\mathsf{eram}}, m_{max}^{\mathsf{eram}}, \varepsilon_{\mathsf{eram}})$*-secure forward secret encrypted RAM scheme, and* τ_{mka} *is a* $(s_{tree}(n, m), s_{skel}, \deg(\tau_{\mathsf{mka}}))$*-tree, then* GMS *is* $(s_1^{\mathsf{eram}}(s_{tree}(n_{curr}, n_{max}), s_{tree}(n_{max}, n_{max})), s_2^{\mathsf{eram}}(s_{tree}(n_{curr}, n_{max}), s_{tree}(n_{max}, n_{max})), \deg(\tau_{\mathsf{mka}}) \cdot s_{skel}, \deg(\tau_{\mathsf{mka}}) \cdot s_{skel} \cdot t_1^{\mathsf{eram}}, s_{skel}, t, q_c, n_{max}, \varepsilon)$*-secure, for* $\varepsilon = (\varepsilon_{\mathsf{eram}} + Q \cdot \varepsilon_{\mathsf{dprf}} + 2\varepsilon_{\mathsf{cpa}*} \cdot \deg(\tau_{\mathsf{mka}})) \cdot (2\deg(\tau_{\mathsf{mka}}))^{d+1} \cdot Q^{(\deg(\tau_{\mathsf{mka}}) \cdot d + 2)}$.

Proof. Observe that the challenge graph is a directed acyclic graph of degree $\deg(\tau_{\mathsf{mka}})$ and depth $d+1$, and let $(\mathcal{P}_0, \ldots, \mathcal{P}_L)$ be the recursive pebbling stratedgy for directed acyclic graphs from [18, Algorithm 1] which uses $L = (2\deg(\tau_{\mathsf{mka}}))^{d+1}$ steps and at most $(\deg(\tau_{\mathsf{mka}}) + 1) \cdot (d+1)$ pebbles. From the proof of Theorem 6, it is easy to see that each pebbling configuration $\mathcal{P}_\ell$ can be represented using $M = (\deg(\tau_{\mathsf{mka}}) \cdot d + 2) \cdot (\log Q)$ bits. This is because the reductions need to guess the same information on the nodes $\mathcal{V}^*_{\mathsf{user\text{-}mult}}$ as the reductions for GUS in user-mult did, plus whether or not the master key node u'_{MK} is pebbled. This proves the security loss in the claim.

The correctness of the composed protocol GMS trivially follows from the correctness of GUS and eram.

It is clear that the space complexity of Γ_{sec} is $s_1^{\mathsf{eram}}(s_{\mathrm{tree}}(n_{\mathrm{curr}}, n_{\mathrm{max}}), s_{\mathrm{tree}}(n_{\mathrm{max}}, n_{\mathrm{max}}))$ and the space complexity of Γ_{pub} is $s_2^{\mathsf{eram}}(s_{\mathrm{tree}}(n_{\mathrm{curr}}, n_{\mathrm{max}}), s_{\mathrm{tree}}(n_{\mathrm{max}}, n_{\mathrm{max}}))$, since the encrypted RAM holds the nodes of τ_{mka}. Thus the worst-case current and maximum number of encrypted RAM cells used by τ_{mka} are its total number of nodes currently, and its maximum number of nodes throughout the protocol execution. The communication complexity of group manager operations is still $\deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}}$, since for each operation of the group manager in GMS, the same exact ciphertexts are created as those in GUS. The time complexity of group manager operations is $\deg(\tau_{\mathsf{mka}}) \cdot s_{\mathrm{skel}} \cdot t_1^{\mathsf{eram}}$, since for each node that is in the skeleton or frontier of an operation, the group manager reads or writes to at most one cell of the encrypted RAM. Finally, the time complexity of user processes is the same as that of GUS, since the proc() algorithm does not change at all in GMS. □

Security Proof for GMS in the ROM To show the security of GMS in the ROM, we again use our Appendix C.3 adaptation of the results of Alwen et al. [3] on Generalized Selective Decryption (GSD) to the USKE setting.

We can just adapt our proof of the security of GUS against user-mult in the ROM model of Theorem 8 to show a polynomial time reduction for GMS in the ROM. First, we need to account for group manager corruptions. We then also have to account for the u_{MK} and u'_{MK} nodes in the GMS-mgr-mult graph $\mathcal{G}_{\mathsf{mgr\text{-}mult}}$ and challenge graph $\mathcal{G}^*_{\mathsf{mgr\text{-}mult}}$, respectively, and their outgoing edges in the two (which may correspond to encryptions of uske′ keys k_i for i not necessarily 0). These changes can be taken care of relatively easily:

Theorem 10. *If the USKE scheme in* GUS *is* $(t, \varepsilon_{\mathsf{cpa}*})$*-secure,* eram *is a* $(t, \varepsilon_{\mathsf{eram}})$*-secure FS eRAM, and* dprf *is replaced with a random oracle* H*, then* GMS *is* (t, Q, ε)-mgr-mult*-secure, where* $\varepsilon = (\varepsilon_{\mathsf{eram}} + \varepsilon_{\mathsf{cpa}*}) \cdot 2 \cdot \deg(\tau_{\mathsf{mka}}) \cdot ((s_{tree} + 1) \cdot Q)^2 + \mathsf{negl}$.[25]

Proof (Sketch). We first can make the same adaptations as Theorem 8 did to account for the change between our defined GSD game and what occurs in GUS against user-mult. Then, to also account for the first change above, when **mgr-corrupt**() is queried by $\mathcal{A}$, the GSD challenger C simply returns the most recent eram master key MK. Since by Lemma 5, MK is not in the challenge graph (nor are any of the other corrupted keys), this does not change anything.

To account for the second change above, in the sequence of hybrids $\mathsf{GSD}_0 := G_0, \ldots, G_{\mathsf{indeg}(v)} := \mathsf{GSD}_1$ defined in Theorem 7, we introduce hybrid games $G_{1,a}, \ldots, G_{\mathsf{indeg}(v),a}$; where each $G_{i,a}$ occurs before G_i in the new sequence. The hybrid games $G_{i,a}$ are defined the same as G_{i-1} except that when any uske′ keys for node v or u *in the challenge graph* (corresponding to the i-th query of the form $(\mathbf{encrypt}, u, v)$) are written to eram, we instead write 0. We thus again have that $G_{\mathsf{indeg}(v)} = \mathsf{GSD}_1$.

To show $\varepsilon_{\mathsf{eram}}$-indistinguishability of $G_{i,a}$ and G_i for each i, we can construct reduction algorithm $\mathcal{B}$ against the FS eRAM challenger that is very similar to that of Lemma 6, except that it only issues

[25] We do not believe we can use the optimized Lemma 6 of [3] due to the introduction of FS eRAM, which requires the reduction to guess GSD edges it will fake in a way that we believe is incompatible with Lemma 6; hence we use their Lemma 4 which introduces the extra $\deg(\tau_{\mathsf{mka}})$ factor in the security loss, which is constant in most cases. Note also: the efficiency of the protocol is the same as in the standard model theorem.

challenge queries to the FS eRAM challenger for USKE keys of node v and u in the challenge graph (again, corresponding to the i-th query of the form $(\mathbf{encrypt}, u, v)$); it issues write queries to the FS eRAM challenger for all other writes of GMS. Since by Lemma 5, none of the keys of the challenge graph are leaked to $\mathcal{A}$ (including useful eram master keys), $\mathcal{B}$ can perfectly simulate the rest of the hybrid as in Lemma 6. □