Continuous Group Key Agreement with Flexible Authorization and Its Applications

Kaisei Kajita[1], Keita Emura[2], Kazuto Ogawa[2], Ryo Nojima[2], and Go Ohtake[1]

[1] Japan Broadcasting Corporation. Setagaya-ku, Tokyo, Japan.
[2] National Institute of Information and Communications Technology (NICT). Koganei, Tokyo, Japan.

Abstract. Secure messaging (SM) protocols allow users to communicate securely over an untrusted infrastructure. The IETF currently works on the standardization of secure group messaging (SGM), which is SM done by a group of two or more people. Alwen et al. formally defined the key agreement protocol used in SGM as continuous group key agreement (CGKA) at CRYPTO 2020. In their CGKA protocol, all of the group members have the same rights and a trusted third party is needed. On the other hand, some SGM applications may have a user in the group who has the role of an administrator. When the administrator as the group manager (GM) is distinguished from other group members, i.e., in a one-to-many setting, it would be better for the GM and the other group members to have different authorities. We achieve this flexible authorization by incorporating a ratcheting digital signature scheme (Cremers et al. at USENIX Security 2021) into the existing CGKA protocol and demonstrate that such a simple modification allows us to provide flexible authorization. This one-to-many setting may be reminiscent of a multicast key agreement protocol proposed by Bienstock et al. at CT-RSA 2022, where GM has the role of adding and removing group members. Although the role of the GM is fixed in advance in the Bienstock et al. protocol, the GM can flexibly set the role depending on the application in our protocol. On the other hand, in Alwen et al.'s CGKA protocol, an external public key infrastructure (PKI) functionality as a trusted third party manages the confidential information of users, and the PKI can read all messages until all users update their own keys. In contrast, the GM in our protocol has the same role as the PKI functionality in the group, so no third party outside the group handles confidential information of users and thus no one except group members can read messages regardless of key updates. Our proposed protocol is useful in the creation of new applications such as broadcasting services.

Keywords: Continuous Group Key Agreement · Secure Group Messaging · Ratcheting Digital Signatures.

1 Introduction

1.1 Background

Secure messaging (SM) is a technology that ensures secure communication between user terminals even in an untrusted communication infrastructure. SM has several characteristics that differ from general secure communication protocols such as TLS and SSH. First, SM is designed for message exchange in an off-line state where the user is cut off from communication, and also for long-term communication sessions (e.g., from the time a user buys a smartphone to the time he or she throws it away). Therefore, it is necessary to consider stronger security against the leakage of the user's confidential information than general secure communication protocols. For this reason, forward secrecy (FS) and post-compromised security (PCS) should be satisfied as the security requirements of SM. FS means that when a user's key is compromised, messages past that point are kept secret, and PCS means that when a user's key is compromised, the security is revived by continuing the protocol.

Two-party secure messaging. The double ratchet protocol recently proposed by Marlinspike and Perrin [23] has led to the comprehensive study of designs and analysis of two-party SM protocols. The double ratchet protocol is an encryption method utilized in many messenger applications such as Signal Protocol [23]. It is designed for two parties, so in a group with more than two users, the protocol must be executed on every pair of users. It means the computational complexity for updating the key information increases in proportion to n^2, where n is the group size. It's known that double ratcheting is thus inefficient and difficult to implement in environments with a large number of users [3].

Secure group messaging. SM, when there are more than n users ($n \geq 2$), is called *secure group messaging* (SGM). Currently, the Internet Engineering Task Force (IETF) has established the Messaging Layer Security (MLS) working group to standardize SGM as the MLS standard of the same name of the working group. The core technology of the MLS standard is the Continuous Group Key Agreement (CGKA) protocol, called TreeKEM [3], which is used to share keys for secure messaging among multiple users. In the MLS standard, the SGM consists of the CGKA protocol, a pseudo-random function and pseudo-random number generator (PRF-PRNG), an n-party forward-secure group authenticated encryption with associated data (FS-GAEAD), and a digital signature. In this paper, we focus on the CGKA protocol, which is crucial for the security and efficiency of MLS and, through its variants, may lead to a wide range of applications. In the CGKA protocol, a group key is asynchronously generated by dynamic group members in a processing unit called an *epoch.* The new group key is used to encrypt higher-layer application messages (e.g., chat text). While there are many secure messaging applications, including those with a group setting function, they are typically based on two parties, and group keys need to be created all over again every time a user is added to or removed from the group, thus limiting the group size. The CGKA protocol, in contrast, can update group keys efficiently and is expected to be particularly effective when the group size is large, such as in broadcasting services.

1.2 Our contribution

We discuss the following two issues of the Alwen et al's CGKA protocol. First, the CGKA protocol allows anyone to add or remove other users, and members of the group all have the same privileges, making it difficult to configure a system with flexible authorization. Thus, the CGKA protocol would be better to have a flexible authorization, where one of the group members is appointed as a group manager (GM) and the others are divided into other users by adding an authentication function. If a single user is permitted to have more powerful privileges, we can construct a CGKA protocol with a one-to-many group structure, which is suitable for broadcasting services. In this paper, we focus on the CGKA protocol developed by Alwen et al. [3] for SGM applications and propose *CGKA with flexible authorization* (CGKA-FA). We instantiate CGKA-FA as a concrete protocol $\mathsf{TreeKEM}^{*}_{\Sigma}$ (Fig. 3) based on the RTreeKEM proposed by Alwen et al. [3]. We achieve this by incorporating a ratcheting digital signature scheme [15] into the CGKA protocol and demonstrate that such a simple modification allows us to provide flexible authorization depending on which algorithm is given a ratcheting digital signature that controls the authority of group members. The ratcheting signature scheme proposed by Cremers et al. [15] is generically constructed from a signature scheme providing existential unforgeability against chosen message attack (EUF-CMA). The ratcheting signature scheme can update a pair of a secret key and a public key with an update message. The reason for applying the ratcheting signature scheme, besides giving each group member the option of authentication, is to improve on the security requirements of SGM, FS, and PCS. If a user is compromised, the CGKA protocol guarantees FS and PCS by updating the shared key, but the authentication is still compromised. That is, it is still possible to impersonate a compromised user. Impersonation is a crucial issue in the context of SGM since an adversary can easily obtain a new shared key by adding a new user. We formalize the security as FS with authentication update (FSAU) and PCS with authentication update (PCSAU). Cremers et al. devised the ratcheting signature scheme to account for cross-group security in multiple groups, but the idea of applying ratchet signatures to recover authentication in a single group may be an independent contribution.

Second, the CGKA protocol discussed in the MLS working group assumes the public key infrastructure (PKI) is a trusted third party to generate the primary keys used by group members. It is noted that the role of the PKI used by CGKA and our CGKA-FA is different from the usual one, such as issuing certifications. In this paper, we refer to the role of PKI as *the PKI functionality* that issues initial keys and stores private keys of all users, as required by the CGKA. The secret keys of all users are recorded in the PKI until all users update their secret keys. Therefore, if any confidential information in the PKI is leaked, the messages of all users can be decrypted. Although this vulnerability is unlikely to be a threat in actual operation and has not been considered in the MLS

Table 1. Comparison of CGKA-FA, the original CGKA, BE, and MKE. Let FSAU be FS with authentication update, and PCSAU be PCS with authentication update. add is an algorithm to add a new user to a group, and rem is an algorithm to remove a user from a group. Note that the PKI here is a trusted third party (TTP) with PKI functionality that manages initial and private key, and does not manage certifications.

	BE[18]	MKA[10]	CGKA[3]	CGKA-FA
PKI functionality	GM	GM	TTP	GM
FS/PCS or FSAU/PCSAU	–	FS/PCS	FS/PCS	FSAU/PCSAU
Initial sk	distributed by GM	distributed by GM	inquire with PKI	inquire with PKI
Updater of sk	GM	each user (including GM)	each user	each user (including GM)
Executor of add	GM	GM	each user	GM or each user
Executor of rem	GM	GM	each user	GM or each user

working group since the secret keys recorded in the PKI are updated soon after the group is generated, it is desirable not only for system security but also for the E2E security concepts of SGM to assume the nonexistence of a trusted third party who can decrypt users' messages. We let the GM manage the keys of group members, which were previously managed by the PKI, and each group member asks the GM for the key instead of the PKI. Assuming that the GM will not be compromised in the setup phase and follows the protocol, we can construct the SGM without assuming the existence of the trusted third party outside the group who has the PKI functionality by having the GM take over it. The fact that there is no entity "outside the group" that can decrypt the message is a significant advantage in the system security of SGM. Since CGKA-FA consists of such simple modifications from CGKA, we can directly use the current IETF MLS implementation.

1.3 Related works

Our proposed CGKA-FA protocol is similar to broadcast encryption (BE) [18] in that it has a GM, but unlike them, each user can update the group key, thus satisfying the FS and PCS security requirements for SGM. Moreover, our proposed CGKA-FA protocol does not need a trusted third party that has the PKI functionality. In conventional CGKA, PKI is used to generate the primary key used by the group members until group members execute an update operation. In the (symmetric key) BE proposed by Fiat and Naor [18], unlike SGM, a trusted group manager (GM) distributes not only the message contents but also all the secret keys. Moreover, the GM can add or revoke users at will, so users do not need to update their private keys themselves. In the public-key based BE introduced by Dodis and Fazio [16], anyone can distribute content, but the group management of users is done by a trusted third party. Except for some schemes (e.g. [25]), BE does not consider the security requirements of FS and PCS.

Since Marlinspike and Perrin [23] proposed Double ratcheting in 2016, based on the idea of the off-the-record (OTR) protocol [12], a lot of research has been done on secure messaging [2–4, 19, 27, 28, 13, 1]. Some formal analyses of the Double ratcheting protocol were provided [14, 2, 11]. Alwen et al. [2] formally defined the continuous key agreement (CKA) as the modular design of the double-ratchet algorithm and two-party key sharing for secure messaging, which had been ambiguous in the past. Alwen et al. introduced the CGKA protocol as a standalone primitive [3] and its modular design for SGM [4]. The authors analyzed *TreeKEM*, a core component in the scalable end-to-end secure group messaging (SGM) discussed in the Messaging Layer Security (MLS) working group of the Internet Engineering Task Force (IETF) [26]. The MLS protocol is currently under development by the eponymous working group of IETF [26]. Alwen et al. pointed out the forward secrecy vulnerability of TreeKEM in IETF MLS and introduced *RTreeKEM* using updatable public-key encryption (UPKE) as a means to address it. Recently,

There have been many studies in recent years on the composition and efficiency of SGM [1, 15, 28]. Weidner [28] proposed Casual TreeKEM, which is a general-purpose group key sharing protocol constructed by modifying ordinary TreeKEM for applications other than secure messaging. The motivation for the research by Weidner is similar to ours, but the major difference is that we consider a one-to-many construction. Alwen et al. proposed a server-aided CGKA that allows for concurrent updates [1]. As for the server-aided CGKA, Hashimoto et al. [20] and Alwen et al. [5] considered a more general (untrusted) delivery service and improved bandwidth. Klein et al. formalized and analyzed a variant of TreeKEM called Tainted TreeKEM [22]. Regarding security aspects, Cremers et al. [15] pointed out the problem of PCS in multi-party networks. Recently, Emura et al. [17] presented CGKA with membership privacy, and Alwen et al. [6] studied the insider security of CGKA.

As a group key sharing protocol for SGM with a one-to-many configuration, Bienstock et al. proposed a multicast key agreements (MKA) protocol [10], which is similar to our CGKA-FA in terms of research motivation and problem setting. In their group key agreement protocol, there is a GM in the group as in ours, and they achieved to speed up their key agreement protocol by constructing it based on symmetric keys, instead of CGKA based on public keys. However, they do not mention the security of GM impersonation. We guarantee the security against GM impersonation by defining a new impersonation resistance captured by PCSAU and FSAU. In addition, in Bienstock et al.'s protocol, the GM has full authority to add and remove group members, whereas in our protocol, the GM can flexibly set the authority depending on the application. In terms of implementation, our CGKA-FA has the advantage that it can be implemented simply by adding ratchet signatures to the MLS protocol, which is currently in the process of being standardized.

David Balvás focused on group management of secure group messaging [7]. He proposed an extension of CGKA called Administrated CGKA (A-CGKA), which, like our CGKA-FA, uses a digital signature scheme for group managers to distinguish between managers and non-managers in the group. They consider two digital signature schemes: one in which each manager has a signature key pair and the other in which all managers have the same signature key pair. The critical difference from our CGKA-FA is that A-CGKA uses general digital signatures in both cases, while we use ratchet signatures. Since digital signatures are an independent scheme from CGKA, an authentication update is required for post-compromised security and forward secrecy. A comparison among CGKA-FA, the original CGKA, BE, and MKA is provided in Table 1.

2 Applications

Our protocol can be applied to various applications such as online conferencing, enterprise chats, broadcasting services because it allows flexible authorization settings in accordance with requirements. For a small group, an extension of the widely used two-party secure messaging protocol is sufficient, but it becomes inefficient when the number of group members becomes large. Therefore, our protocol is particularly effective for services that have a one-to-many structure and a large number of users. For example, in a large-scale online conference, the GM is the conference organizer and the other users are the conference participants. This can be achieved by setting permissions so that only the organizer can generate groups and add or remove group members. In the example of a corporate chat with many employees, this can be achieved by appointing the GM as the administrator in the company, such as the information system department, and the users as the employees, and setting the permissions so that all users have the authority to create groups and add and remove group members. Our proposed CGKA-FA protocol can realize such applications with flexible authorization settings that are not envisioned by BE, CGKA, or MKA. If the authority setting is the same as that in the conventional CGKA protocol, there is an advantage that there is no third party with the PKI functionality.

Applications to the broadcasting field. We describe the application in more detail using a broadcasting service. In broadcasting services, there is a large-scale one-to-many relationship between the broadcaster and the viewers, and the broadcaster is required to manage the viewers, so there is a high affinity between our proposed protocol and the broadcasting field. Now consider remote TV program production such as music and sports programs where viewers or performers participate online in the production of the program and receive the video of the program recorded in real-time.

When considering the above application, Table 2 shows an example of the authorization settings. The authorities required by the viewer and the broadcaster are different: the broadcaster needs to be able to manage the group structure and remove unauthorized viewers, while the viewer needs to be able to invite (add) other viewers or leave the group. Here our CGKA-FA protocol $\mathsf{TreeKEM}^*_\Sigma$

Table 2. Example of application to the broadcasting field

Operation	Scheme	All users	GM
Initialization	TreeKEM*	✓	–
Group creation	TreeKEM$_\Sigma$	–	✓
Add	TreeKEM*	✓	–
Remove	TreeKEM$_\Sigma$	-	✓
Update	TreeKEM*	✓	–
Process	TreeKEM*, TreeKEM$_\Sigma$	✓	✓

is constructed by two protocol TreeKEM* and TreeKEM$_\Sigma$. For operations that can be performed by all users, TreeKEM* is used when it does not require a signature, and TreeKEM$_\Sigma$ is used for operations that can only be performed by the GM. Therefore, TreeKEM* is utilized by all users for Initialization, Add, and Update operations, and TreeKEM$_\Sigma$ is utilized for Group creation and Remove. Process is used for both, as it corresponds to the control messages generated by each operation.

MLS support. The latest MLS (as of the time of submission) is version 16 [9], but in this paper, our proposed CGKA-FA protocol TreeKEM$^*_\Sigma$ is based on version 7 [24]. All versions of MLS from 8 [8] onward utilize the propose-and-commit method. Considering the possibility that TreeKEM will be modified in IETF MLS in the future, and for the reasons of simplicity, we chose to focus on TreeKEM and RTreeKEM up to version 7, which has a simple form. We have confirmed that CGKA-FA can be easily constructed in version 11 MLS protocol, and here we briefly present the syntax of our idea applied to TreeKEM using the propose-and-commit method in version 11. As in CGKA-FA, all parties are divided into GM and other users, and each has its own dedicated syntax. The GM's syntax does not assume the existence of a third party that manages certifications, so it takes over the PKI functionality and adds signatures to the control messages it generates. Other users do not use signatures but interact with the GM instead of the PKI. In the CGKA syntax used in version 11 of the MLS protocol, initialization has been eliminated and interaction with PKI has been newly added. Therefore, it should be changed to interact with the GM instead of the PKI. As for group creation, it is the same as the CGKA-FA syntax, and only the syntax used by the GM incorporates signatures. Next, Add, Remove, and Update are improved to Issue proposals and Commit creation by the proposal-and-commit method, but each of them can be easily reconfigured by incorporating signatures only when the GM executes them as in CGKA-FA. As for Process, it has been changed to Processing commits and Joining, but it can be reconfigured by incorporating the signature verification function as in CGKA-FA. Since we have not yet confirmed the security and concrete construction of CGKA-FA in version 12 or later, we leave this to future work.

3 Preliminaries

Notation. For a positive integer $a \in \mathbb{N}$ we write $[a] := \{1, 2, \dots a\}$, and for $a, b \in \mathbb{N}$ with $a \leq b$ we write $[a, b] := \{a, a{+}1, \dots, b\}$. The concatenation for vectors $x = (x_1, \dots, x_a)$ and $y = (y_1, \dots, y_b)$ is denoted by $x||y := (x_1, \dots, x_a, y_1, \dots, y_b)$, and for a single element z we write $x||z := (x_1, \dots, x_a, z)$. We use $A[i] \leftarrow x$ and $y \leftarrow A[i]$ to denote assignment and retrieval between an arbitrary index space and element space, respectively. We denote the security parameter by λ. All our algorithms implicitly take as an argument 1^λ. For an algorithm A, we write $\mathsf{A}(\cdot; r)$ to denote that A is run with explicit randomness r.

3.1 Binary trees

In rooted binary trees, where every branch has two nodes, the nodes without child nodes are called leaf nodes, and the other nodes are called intermediate nodes. The height of a binary tree is the maximum path length from the root vertex to a leaf node. A binary tree is said to be complete if it has a height h and 2^h leaf nodes. Note that a binary tree with height $h = 0$ consists only of roots. If $h \geq 0$, we say that FT_h is a complete binary tree with height h. For some leaf nodes ℓ, ℓ', let $\mathsf{LCA}(\ell, \ell')$ be the nearest common ancestor of ℓ, ℓ', i.e., the node where the paths from each leaf node to the root intersect.

Left-balanced binary tree. A left-balanced binary tree (LBBT) with $n \in \mathrm{N}$ nodes LBBT_n is a binary tree satisfying the following properties. (1) LBBT_1 consists of a single node (root). (2) If $x = \mathsf{mp2}(n)$, where $\mathsf{mp2}(x) = \max(x : 2^x | n)$ is the maximum power of 2 among the factors of n, then the roots of LBBT_n have FT_x as the left child node and LBBT_{n-x} as the right child node. For $\mathsf{LBBT}_n = \tau$, let τ be the node whose ID ID is labeled with $\tau.\mathsf{ID}$. Let v be a node of τ, and let $v.\mathsf{pk}$ be the pk labeled to v.

It will be useful to consider the following types of paths:

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

3.2 Ratchet trees

We introduce some basic concepts pertaining to TreeKEM's ratchet tree (RT). Note that LBBT_n has exactly n leaves and that every internal node has two children. In an RT, nodes are *labeled* as follows.

- **Root:** The root is labeled by an update secret I.
- **Internal:** Internal nodes are labeled by a key pair $(\mathsf{pk}, \mathsf{sk})$ for the Updatable PKE (UPKE) scheme $\mathit{\Pi}$ [3].
- **Leaf nodes:** Leaf nodes are labeled like internal nodes, except that they additionally have an owner ID.

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

Resolutions and representatives. Crucial notions in ratchet trees are resolutions and representatives. Intuitively, the resolution of a node v is the smallest set of non-blank nodes that covers all leaves in v's subtrees. Each leaf ℓ' in the subtree τ' of some node v' has a representative in τ'.

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

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

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

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

3.3 Cryptographic tools

Ratcheting Digital signatures [15]. A ratcheting digital signature scheme Σ is proposed by [15] and given by five algorithms; KGen_Σ, Update_Σ, RcvUpd_Σ, Sign_Σ, and Vrfy_Σ. The key-generation algorithm KGen_Σ takes as input security parameter λ and outputs a pair of signing and verification keys, $\mathsf{sk}_{sig}, \mathsf{vk}_{sig}$. The update algorithm Update_Σ takes as input a secret and public key pair $(\mathsf{sk}_{sig}, \mathsf{vk}_{sig})$ and returns a new secret and public key pair $(\mathsf{sk}'_{sig}, \mathsf{vk}'_{sig})$ and an update message m_u. The RcvUpd_Σ is a deterministic algorithm which takes as input a public key vk_{sig} and an update message m_u, and returns the updated public key vk'_{sig}. The signing algorithm Sign_Σ takes as input $(\mathsf{sk}_{sig}, \mathsf{vk}_{sig})$ and a message $m \in \mathcal{M}_\lambda$ and produces a signature σ. The verification algorithm Vrfy_Σ takes as input vk_{sig}, m, σ, and outputs a verification result bit. For correctness, we require that

– For any sequence i, if $\left((\mathsf{sk}^{(i)}_{sig}, \mathsf{vk}^{(i)}_{sig}, m_u\right) \leftarrow \mathsf{Update}_\Sigma(\mathsf{sk}^{(i-1)}_{sig}, \mathsf{vk}^{(i-1)}_{sig})$ and $\mathsf{vk}'_{sig} \leftarrow \mathsf{RcvUpd}_\Sigma(\mathsf{vk}^{(i-1)}_{sig}, m_u)$, then $\mathsf{vk}^{(i)}_{sig} = \mathsf{vk}'_{sig}$.
– $\mathsf{Vrfy}_\Sigma(\mathsf{vk}_{sig}, m, \mathsf{Sign}_\Sigma(\mathsf{sk}_{sig}, m)) = 1$ for any $(\mathsf{sk}_{sig}, \mathsf{vk}_{sig})$ pair and for any $m \in \mathcal{M}_\lambda$.

Let the advantage be $\mathsf{Adv}_{cma}(\mathcal{A})$ that an adversary $\mathcal{A}$ wins the existentially unforgeable game against chosen message attack (EUF-CMA). Briefly, an adversary is allowed to access the signing oracle for any massage as in the usual EUF-CMA definition. In addition, the adversary calls the update oracle that updates the current key (precisely outputs an update message), and calls the corrupt oracle that returns the current signing key. The adversary outputting a pair of message and signature wins if they are valid under the current verification key, the signature is not an output of the signing oracle with the message, and the adversary has at least once used an update message returned by the update oracle after every compromise. For time t and the adversary's advantage ϵ, we say the Σ is (t, ϵ)-cma-secure, where $\mathsf{Adv}_{cma}(\mathcal{A}) \leq \epsilon$.

Pseudorandom generator. A pseudorandom generator $\mathsf{prg} : \mathcal{W} \to \mathcal{W} \times \mathcal{K}$ is a function that satisfies indistinguishability between $\mathsf{prg}(U)$ and U' for some uniformly random $U \in \mathcal{W}$ and $U' \in \mathcal{W} \times \mathcal{K}$.

Updatable public-key encryption [3]. An updatable public-key encryption (UPKE) scheme UPKE is given by a triple (UKGen, UEnc, UDec) of PPT Turing machines. The key-generation algorithm $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{UKGen}(n)$ takes as input security parameter n and outputs the initial public key pk_0 and the initial secret key sk_0. The encryption algorithm $(c, \mathsf{pk}') \leftarrow \mathsf{UEnc}(\mathsf{pk}, m)$ encrypts the message m by pk and outputs ciphertext c. At the same time, it updates the pk and outputs the updated public key pk'. The decryption algorithm $(m, \mathsf{sk}') \leftarrow \mathsf{UDec}(\mathsf{sk}, c)$ decrypts c by sk and outputs the original m. At the same time, it updates the sk and outputs the updated secret key sk'.

4 Continuous Group Key Agreement

This section describes the structure and security of the continuous group key agreement (CGKA) protocol introduced by Alwen et al. [3]. They first provided a formal definition and security model of TreeKEM, the core technology of the MLS protocol, as CGKA. Then, they developed RTreeKEM by improving the vulnerability of TreeKEM through a UPKE using the normal public key encryption. In this paper, we utilize RTreeKEM with UPKE applied as a CGKA protocol. A CGKA protocol aims at providing a steady stream of shared (symmetric) secret keys for a dynamically evolving set of parties. This aspect is tied together by epochs, where each epoch provides a timestamp role in asynchronous processing. CGKA schemes are non-interactive; that is, a party creates a new epoch by broadcasting a single message, which can then be processed by the other members to move along. Rather than relying on an actual broadcast scheme, CGKA schemes merely assume the existence of an untrusted (or partially trusted) delivery service. As multiple parties might try to initiate a new epoch simultaneously, the delivery service's main job is to determine the first one by picking an order. As a consequence, a party cannot immediately initiate a new epoch by itself. The MLS working group has improved on such stagnation and now considers the propose-and-commit method [8]. In this section, however, we present Alwen et al.'s (without propose-and-commit) CGKA protocol for simplicity. The basic structure of both schemes is the same, and we apply our ideas to Alwen et al.'s protocol from here onward. By applying our ideas to the propose-and-commit scheme, we can immediately construct CGKA-FA (as discussed in Section 5).

4.1 PKI functionality

The CGKA protocol requires the PKI functionality that issues initial keys and stores keys required to configure the Ratchet tree as follows.

- When any user ID may request a fresh (encryption) public key pertaining to some other user ID', the PKI functionality generates a fresh key pair $(\mathsf{pk}, \mathsf{sk})$ and returns pk to ID. Note that every public key is only used once.
- When any user ID' may request secret keys corresponding to public keys associated with them, if a triple $(\mathsf{pk}, \mathsf{sk}, \mathsf{ID}')$ is recorded, the PKI functionality returns sk to ID'.

Note that the PKI functionality defined above is different from a normal PKI that issues the certifications of public keys.

4.2 CGKA syntax

The continuous group key agreement protocol CGKA (**Initiation, Group creation, Add, Remove, Update, Process**) [3] consists of the following algorithms.

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

A state γ stores LBBT generated by each algorithm. The basic usage of a CGKA protocol is as follows. First, each user is initiated using $\mathsf{TK\text{-}init}$ before they join a group. Once a group is established using $\mathsf{TK\text{-}create}(\gamma, G)$, any group member, referred to as the *sender*, may utilize the algorithms to add or remove members or to perform updates. Each time, such a call results in a new epoch.

This is implicitly the task of a server connecting the parties, so the server then relays the resulting *control* messages to all current group members (including the sender). Note that there are two types of messages: a welcome message W, which is sent to parties joining a group, and a control message T, which is intended for parties already in the group. Whenever the server delivers a control message to a group member, they process it using proc. The algorithm proc outputs an update secret I, where the intention is that $I \neq \perp$ if and only if the control message corresponds to an update.

4.3 CGKA security

Here we first explain the security requirements and the security game of SGM and then define the CGKA security.

Security requirements. Any CGKA protocol must satisfy the following basic properties [3].

- **Correctness:** All group members obtain the same update secret I at each epoch.
- **Privacy:** The updated secret information I generated when a control message is received is indistinguishable from uniform randomness.
- **Forward secrecy:** When the state of a group member is compromised, the previously updated secret information is kept secret from the third party.
- **Post-compromise security:** When the state of a group member is compromised, the updated secret information is again kept secret when all the group members whose states were compromised update their states.

CGKA security game. The above properties are defined by a security game consisting of 10 oracles. This section gives a brief overview of the security game and describes some oracles (corrupt, chall, no-del,) which are necessary for the security definition. The other oracles are basically initialization oracles to prepare variables and oracles to execute each CGKA algorithm, so they are not described here. In the CGKA security game, the adversary is given access to various oracles to drive the execution of a CGKA protocol. The main oracles to drive the execution are the oracles to create groups, add users, remove users, and deliver control messages, i.e., $\mathsf{create\text{-}group}$, $\mathsf{add\text{-}user}$, $\mathsf{remove\text{-}user}$, $\mathsf{send\text{-}update}$, and $\mathsf{deliver}$. The first four oracles allow the adversary to instruct parties to initiate new epochs, whereas the deliver oracle makes parties actually proceed to the next epoch. The game forces the adversary to initially, i.e., in epoch 1, create a group. Thereafter, any group member may add new parties, remove current group members, or perform an update. The adversary may also corrupt any party at any point (thereby learning that party's secret state) and challenge the update secret in any epoch where an update operation was performed. These oracles are denoted as $\mathsf{corrupt}$ and chall, respectively. Furthermore, the adversary can instruct parties to stop deleting old secrets, whose oracle is denoted as $\mathsf{no\text{-}del}$. There will be restrictions checked at the end of the execution of the game to ensure that the adversary's challenge/corruption/no-deletion behavior does not lead to trivial attacks.

– **corrupt**. The corrupt oracle corrupt takes user ID ID as an input and returns the dictionary of ID's state $\gamma[ID]$, where the dictionary γ keeps track of the users' states.
– **chall**. The challenge oracle chall takes an epoch t as an input. The oracle first checks that t corresponds to an update epoch. Two challenges I_0 and I_1 are made from the dictionary of update secret (i.e., shared secret key) of the epoch t and random space, respectively. Then the oracle returns I_b, where b is a random bit $b \leftarrow \{0, 1\}$.
– **no-del**. The no-deletion oracle no-del takes a party ID as inputs and stop deleting old values of ID. The dictionary D keeps track of which parties delete their old values and which do not.

In order to ensure that the adversary does not win the CGKA security game with trivial attacks (e.g., challenging an epoch t's updated secret and leaking some party's state in epoch t), at the end of the game, the predicate **safe** (shown in Fig. 1) is run on the queries $q_1, \ldots, q_q$ in order to determine whether the execution was devoid of such attacks. The predicate uses the function q2e, which returns the epoch corresponding to query q. Specifically, for $q \in \{\mathsf{corrupt(ID)}, \mathsf{no\text{-}del(ID)}\}$, if ID is a member of the group when q is made, q2e is the value of ep[ID], otherwise, q2e returns $\perp$.

```
safe $(q_1, \ldots, q_q)$
    for $(i, j)$ s.t. $q_i = \mathsf{corrupt(ID)}$ and $q_j = \mathsf{chall}(t^*)$
    if $\mathsf{q2e}(q_i) \le t^*$ and $\nexists k$ s.t. $0 < \mathsf{q2e}(q_i) \le \mathsf{q2e}(q_k) \le t^*$
    and $q_k \in \{\mathsf{send\text{-}update(ID)}, \mathsf{remove\text{-}user}(*, \mathsf{ID})\}$
        return 0
    if $\mathsf{q2e}(q_i) > t^*$ and $\exists k$ s.t. $\mathsf{q2e}(q_k) \le t^*$
    and $q_k = \mathsf{no\text{-}del(ID)}$
        return 0
```

Fig. 1. Safety predicate: **safe**. The safety predicate determines whether a sequence of oracle calls $(q_1, \ldots, q_q)$ allows the adversary to trivially win the CGKA security game for some ID ID and epoch t^*.

For the CGKA protocol, we introduce the following two lemmas 1 and 2 from [3]; Lemma 1 describes the relation of predicates **pcs**, **fsu**, and **tkm**, where **pcs** and **fsu** are predicates that define PCS and FS with update (FSU), respectively, snf **tkm** describes the security game of TreeKEM, and lemma 2 shows that single-challenge (non-adaptive) security implies multi-challenge security.

Lemma 1 ([3]). *For any sequence of queries Q, if* $\mathbf{pcs(Q) = 1}$ *or* $\mathbf{fsu(Q) = 1}$*, then* $\mathbf{tkm(Q) = 1}$*, where* **pcs** *and* **fsu** *are predicates that define PCS and FS with update (FSU), respectively.*

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

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

Lemma 2 ([3]). *Let* $\mathsf{P} \in \{\mathbf{tkm}, \mathbf{pcsau}, \mathbf{fsau}\}$ *and assume that a CGKA protocol* CGKA *is* $(\tilde{t}, 1, n, \mathsf{P}, \tilde{\varepsilon})$*-secure. Then,* CGKA*is also* $(\tilde{t'}, c, n, \mathsf{P}, \tilde{\varepsilon}')$*-secure for* $\tilde{t'} \approx \tilde{t}$ *and* $\tilde{\varepsilon}' = c\tilde{\varepsilon}$.

Definition 1 (Non-adaptive CGKA security). A continuous group key-agreement protocol CGKA is non-adaptively $(t, c, n, \mathsf{P}, \varepsilon)$-secure if, for all (t, c, n)-attackers,

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

5 CGKA with Flexible Authorization

With Alwen et al.'s CGKA protocol RTreeKEM as a basis, we propose a *continuous group key agreement with flexible authorization* (CGKA-FA) by applying a ratcheting digital signature to authenticate users. We appoint one user in the group as *the group manager (GM)* and the others as *other users.* This allows us to achieve a one-to-many broadcast-type CGKA protocol that securely provides other users with the authority to perform specific operations only. In addition, while TreeKEM uses the PKI to generate the keys for RT configuration, as the PKI functionality, our CGKA-FA has a one-to-many configuration with GMs, which means we can modify the protocol so that GMs manage the keys that PKI used to manage, and each group member queries GMs for the keys. In actual operation, the CGKA-FA protocol is incorporated into the SGM protocol, as well as CGKA, along with PRF-PRNG, FS-GAEAD, and digital signatures. In general, since authentication for key distribution is required to use digital signatures, a trusted third party is necessary after all, but this requires a normal PKI that does not issue initial keys or store private keys, and our concept of PKI in CGKA-FA has security advantages.

5.1 CGKA-FA Syntax

We introduce the syntax of the CGKA-FA protocol $\mathsf{CGKA\text{-}FA}$ as the same components of CGKA. $\mathsf{CGKA\text{-}FA}$ consists of two protocols: one that replaces CGKA's queries to the PKI with GM, and another that adds ratchet signatures. The basic structure is the same as that of CGKA. Therefore, we can add the FA functionality to CGKA easily. In the following syntax, we provide the ratcheting signature to Add, Remove, and Update algorithms. We remark that which algorithm will contain the signature is flexibly decided to depend on the application. The syntax of CGKA-FA is shown below.

- **Initiation:** $\gamma \leftarrow \mathsf{TK\text{-}init}^*/\mathsf{TK\text{-}init\text{-}sig}\ (\mathsf{ID})$. $\mathsf{TK\text{-}init}^*$ and $\mathsf{TK\text{-}init\text{-}sig}$ take a user's ID ID and outputs an initial state γ. Additionally, $\mathsf{TK\text{-}init\text{-}sig}$ generates a key pair for a ratcheting signature.
- **Group creation:** $(\gamma', W, \sigma) \leftarrow \mathsf{TK\text{-}create}^*/\mathsf{TK\text{-}create\text{-}sig}\ (\gamma,\ G)$. $\mathsf{TK\text{-}create}^*$ and $\mathsf{TK\text{-}create\ \text{-}sig}$ take a state γ and a list of IDs $\mathsf{G} = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$ and outputs a new state γ' and a welcome message W. Additionally, $\mathsf{TK\text{-}create\text{-}sig}$ generates a ratcheting signature σ.
- **Add:** $(\gamma', (W, T), \sigma) \leftarrow \mathsf{TK\text{-}add}^*/\mathsf{TK\text{-}add\text{-}sig}(\gamma\ , \mathsf{ID}')$. $\mathsf{TK\text{-}add}^*$ and $\mathsf{TK\text{-}add\text{-}sig}$ take a state γ and an ID ID', and output a new state γ' as well as welcome messages W and control messages T. Additionally, $\mathsf{TK\text{-}add\text{-}sig}$ generates a ratcheting signature σ.
- **Remove:** $(\gamma', T, \sigma) \leftarrow \mathsf{TK\text{-}rem}^*/\mathsf{TK\text{-}rem\text{-}sig}(\gamma, \mathsf{ID}')$. $\mathsf{TK\text{-}rem}^*$ and $\mathsf{TK\text{-}rem\ \text{-}sig}$ take a γ and an ID ID' and outputs a new state γ' and a control message T. Additionally, $\mathsf{TK\text{-}rem\text{-}sig}$ generates a ratcheting signature σ.
- **Update:** $(\gamma', T, \sigma) \leftarrow \mathsf{TK\text{-}upd}^*/\mathsf{TK\text{-}upd\text{-}sig}$. $\mathsf{TK\text{-}upd}^*$ and $\mathsf{TK\text{-}upd\text{-}sig}$ update the state and outputs a new state γ' and a control message T. Additionally, $\mathsf{TK\text{-}upd\text{-}sig}$ generates a ratcheting signature σ.
- **Process:** $(\gamma', I) \leftarrow \mathsf{TK\text{-}proc}^*/\mathsf{TK\text{-}proc\text{-}sig}(\gamma, T, W, \sigma)$. $\mathsf{TK\text{-}\ proc}^*$ and $\mathsf{TK\text{-}proc\ \text{-}sig}$ take a state γ, a control message T, and a welcome message W if it exists. A signature σ is an input only as needed for $\mathsf{TK\text{-}proc\text{-}sig}$ and verified. It then outputs a new state γ' and an update secret I.

5.2 CGKA-FA security

In this section, we describe the security of CGKA-FA. We consider the case that Add, Remove, and Update algorithms contain ratcheting signatures. The only difference in security between CGKA and CGKA-FA is the presence or absence of a digital signature. The difference in security between CGKA and CGKA-FA is the presence or absence of ratchet signatures. Since the GM takes over the PKI functionality in CGKA-FA, it is necessary to clarify some conditions for the GM. First, since the GM creates initial keys for group members, the GM is assumed to be uncompromised until the first update is performed by each group member. Then, after each group member performs an update, the GM can perform an update at any time. This means that we do not assume an adversary who will not even allow an update to be performed. These conditions are implicitly assumed for all group members in CGKA. The CGKA-FA security game can be played in much the same way as the CGKA security oracle. However, since CGKA-FA uses ratcheting signatures, we define here the oracle vrfy that verifies the signatures. vrfy does not have to be an Oracle but is defined here as a new security oracle because it simplifies the game by defining it according to a security game such as [3].

- **vrfy.** For a user ID ID, the verification oracle vrfy takes the ratcheting signature σ_{ID}, verification key vk_{sig}, and the update massage m_u as an inputs. Then the oracle runs $\mathsf{vk}'_{sig} \leftarrow \mathsf{RcvUpd}_{\Sigma}(\mathsf{vk}_{sig}, m_u)$ and returns the result of $\mathsf{Vrfy}_{\Sigma}(\mathsf{vk}'_{sig}, \sigma_{\mathsf{ID}})$ according to the verification of the ratcheting signature.

The safety predicate used is the same as **safe** in Fig. 1.

Security requirements. CGKA-FA satisfies the properties of correctness and privacy as discussed in section 4.3. We consider not only FS and PCS but also the GM impersonation resistance because we manage a one-to-many structure where a GM has the PKI functionality. Alwen et al. defined FS with updates considering vulnerabilities in "key" updates. We define FS with authentication update (FSAU) and PCS with authentication update (PCSAU) to consider vulnerabilities in "authentication", respectively, and GM impersonation resistance.

- **Forward secrecy with authentication update (FSAU):** When the state of a group member is compromised, the previously updated secret information and the key pair of signatures are kept secret from the third party.
- **Post-compromise security with authentication update (PCSAU):** When the state of a group member is compromised, the updated secret information and the key pair of signatures are again kept secret.

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

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

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

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

6 TreeKEM$^*_{\Sigma}$

In this section, we describe all the algorithms involved in the protocol and explain the concrete TreeKEM$^*_{\Sigma}$ protocol in detail. We instantiate the CGKA-FA as TreeKEM$^*_{\Sigma}$, as shown in Fig. 2 and 3. Our proposed protocol TreeKEM$^*_{\Sigma}$ differs from Alwen et al.'s CGKA protocol RTreeKEM [3,4] in two ways. First, by combining the ratcheting signature scheme Σ with RTreeKEM, we add a function to generate the ratcheting signature σ for each control message output by each algorithm, and the processing proc verifies the signature first. In this way, the group members can be sure that the GM has executed the algorithm by adding the signature only to the GM's control message. Second, in Alwen et al.'s algorithms of group creation and adding a group member, the pk generated by the PKI is queried, but in our version, the pk generated by the GM is queried, and in the same way in proc, the sk is queried and obtained from the GM. In this way, the system can be configured without trusted third-party entities.

6.1 Sub-algorithms

We introduce the sub-algorithms of our TreeKEM$^*_\Sigma$; Init, AddID, Blank, Pub, RemID, UpGen, UpPro:

Init$(G, \mathsf{pk}, j, \mathsf{sk}_j)$:

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

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

AddID$(\tau, \mathsf{ID}, \mathsf{pk})$:

Given an RT τ, set the labels of the first blank leaf of τ to $(\mathsf{ID}, \mathsf{pk}, \perp)$ and then output the resulting tree, τ'. If there is no blank leaf in the tree $\tau = \mathsf{LBBT}_n$, method AddLeaf(τ) is called.

AddLeaf(τ):

Add a leaf z to the RT $\tau = \mathsf{LBBT}_n$, resulting in a new tree τ' =AddLeaf(τ):

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

Blank(τ, ID):

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

Pub(τ):

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

$\textsf{RemID}(\tau, \mathsf{ID})$:

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

$\textsf{Trunc}(\tau)$:

The recursive procedure $\textsf{Trunc}(v)$ is called on the rightmost leaf v of τ and outputs the following τ'.

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

$\textsf{UpGen}(\tau, \mathsf{ID})$:

Having computed the new keys on its direct path, a user ID proceeds as follows.

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

The control message for this operation simply consists of ME and U.

$\textsf{PropUp}(\tau, v, s_0)$:

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

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

$\textsf{UpPro}(\tau, \mathsf{ID}, \mathsf{ID}', U)$:

Given control message $T = (\mathsf{up}, \mathsf{ID}, U)$, a user ID' at some leaf ℓ' receiving $U = (\mathsf{pk}, \mathbf{c})$, issued by the user with id ID at leaf v, recovers the update information as follows.

- Let $w = \mathsf{Rep}(v, \ell')$.
- The user with ID' uses $w.sk$ to decrypt c_{ij} and obtain s_i.
- Update the ratchet tree by overriding the public-key labels on the v-root path by the keys in PK and then producing a new tree $\tau' \leftarrow \textsf{PropUp}(\tau, \mathsf{LCA}(v,\ell'), s_i)$.

TreeKEM*

```
TK-init*(ID)
    $\mathsf{ME} \leftarrow \mathsf{ID}$
    $\tau \leftarrow \perp$
    $\mathsf{ctr} \leftarrow 0$
    $\tau'[\cdot], \mathsf{conf}[\cdot] \leftarrow \perp$

TK-create*(G)
    $\mathsf{ctr}++$
    $\mathsf{ID}_0 \leftarrow \mathsf{ME}$
    $(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{UKGen}$
    for $i = 1, \ldots, |G|$
        $\mathsf{pk}_i \leftarrow \mathsf{GMget\text{-}pk}(|G|.i)$
    $G' \leftarrow (\mathsf{ID}_0, G)$
    $\mathsf{pk}' \leftarrow (\mathsf{pk}_0, \mathsf{pk})$
    $\tau'[\mathsf{ctr}] \leftarrow \textsf{Init}(G', \mathsf{pk}', 0, \mathsf{sk}_0)$
    $W \leftarrow (\mathsf{create}, G', \mathsf{pk}')$
    $\mathsf{conf}[\mathsf{ctr}] \leftarrow W$
    return $W$

TK-add*(ID')
    $\mathsf{ctr}++$
    $\mathsf{pk}' \leftarrow \mathsf{GMget\text{-}pk}(\mathsf{ID}')$
    $\tau'[\mathsf{ctr}] \leftarrow \textsf{AddID}(\tau, \mathsf{ID}', \mathsf{pk}')$
    $\tau'[\mathsf{ctr}] \leftarrow \textsf{Blank}(\tau'[\mathsf{ctr}], \mathsf{ID}')$
    $W \leftarrow (\mathsf{wel}, \textsf{Pub}(\tau'[\mathsf{ctr}]))$
    $T \leftarrow (\mathsf{add}, \mathsf{ME}, \mathsf{ID}', \mathsf{pk}')$
    $\mathsf{conf}[\mathsf{ctr}] \leftarrow T$
    return $(W, T)$

TK-rem*(ID')
    $\mathsf{ctr}++$
    $\tau'[\mathsf{ctr}] \leftarrow \textsf{Blank}(\tau, \mathsf{ID}')$
    $\tau'[\mathsf{ctr}] \leftarrow \textsf{RemID}(\tau'[\mathsf{ctr}], \mathsf{ID}')$
    $T \leftarrow (\mathsf{rem}, \mathsf{ME}, \mathsf{ID}')$
    $\mathsf{conf}[\mathsf{ctr}] \leftarrow T$
    return $T$

TK-upd*
    $\mathsf{ctr}++$
    $(\tau'[\mathsf{ctr}], U) \leftarrow \textsf{UpGen}(\tau, \mathsf{ME})$
    $T \leftarrow (\mathsf{up}, \mathsf{ME}, U)$
    $\mathsf{conf}[\mathsf{ctr}] \leftarrow T$
    return $T$

TK-proc*(T, W)
    if $\exists j : T = \mathsf{conf}[j]$
        $\tau \leftarrow \tau'[j]$
    else
        $\mathsf{proc}^*(T, W)$
    $\mathsf{ctr} \leftarrow 0$
    $\tau'[\cdot], \mathsf{conf}[\cdot] \leftarrow \perp$
    return $(\tau.I)$
```

```
proc*(W=(create, G, pk))
    let $j$ s.t. $G.\mathsf{ID}_j = \mathsf{ME}$
    $\mathsf{sk}_j \leftarrow \mathsf{GMget\text{-}sk}(\mathsf{pk}.j)$
    $\tau \leftarrow \textsf{Init}(G, \mathsf{pk}, j, \mathsf{sk}_j)$

proc*(T=(add, ID, ID', pk'))
    $\tau \leftarrow \textsf{AddID}(\tau, \mathsf{ID}', \mathsf{pk}')$
    $\tau \leftarrow \textsf{Blank}(\tau, \mathsf{ID}')$

proc*(W = (wel, τ'))
    $\tau \leftarrow \tau'$
    $\tau.\mathsf{ME}.\mathsf{sk} \leftarrow \mathsf{GMget\text{-}sk}(\tau.\mathsf{ME}.\mathsf{pk})$

proc*(T = (rem, ID, ID'))
    $\tau \leftarrow \textsf{Blank}(\tau, \mathsf{ID}')$
    $\tau \leftarrow \textsf{RemID}(\tau, \mathsf{ID}')$

proc*(T = (up, ID, U))
    $\tau \leftarrow \textsf{UpPro}(\tau, \mathsf{ID}, \mathsf{ME}, U)$
```

Fig. 2. TreeKEM*. Once any ID ID calls $\mathsf{GMget\text{-}pk}(\mathsf{ID}')$, $(\mathsf{pk}, \mathsf{sk})$ is generated by GM and returns pk to ID and GM records $(\mathsf{pk}, \mathsf{sk}, \mathsf{ID}')$. ID' can inquire to GM by $\mathsf{GMget\text{-}sk}(\mathsf{pk})$. After they are recorded, it returns sk. The only difference between TreeKEM* and RTreeKEM is to use $\mathsf{GMget\text{-}sk}(\mathsf{pk}.j)$ to get sk_j from GM and $\mathsf{GMget\text{-}pk}(\mathsf{ID}')$ to get pk' corresponding to ID' from GM instead of $\mathsf{get\text{-}sk}$ and $\mathsf{get\text{-}pk}$ in RTreeKEM.

TreeKEM$_\Sigma$

```
TK-init-sig(ID)
    TK-init$^*$(ID)
    ctr$' \leftarrow 0$
    conf$'[\cdot] \leftarrow \perp$
    $(\mathsf{sk}_{sig}, \mathsf{vk}_{sig}) \leftarrow \mathsf{KGen}_\Sigma$
    return $\mathsf{vk}_{sig}$
TK-upd-sig
    TK-upd$^*$
    ctr$'$ + +
    $\sigma \leftarrow \mathsf{Sign}(T, \mathsf{sk}_{sig})$
    $((\mathsf{sk}'_{sig}, \mathsf{vk}'_{sig}), m_u)$
    $\leftarrow \mathsf{Update}_\Sigma(\mathsf{sk}_{sig}, \mathsf{vk}_{sig})$
    conf$'$[ctr$'$] $\leftarrow m_u$
    return $(T, \sigma, m_u)$
TK-rem-sig(ID$'$, $\mathsf{sk}_{sig}$)
    TK-rem$^*$(ID$'$)
    $\sigma \leftarrow \mathsf{Sign}(T, \mathsf{sk}_{sig})$
    return $(T, \sigma)$
TK-create-sig(G,$\mathsf{sk}_{sig}$)
    TK-create$^*(G)$
    $\sigma \leftarrow \mathsf{Sign}_\Sigma(W, \mathsf{sk}_{sig})$
    return $(W, \sigma)$
TK-add-sig(ID$'$,$\mathsf{sk}_{sig}$)
    TK-add$^*$(ID$'$)
    $\sigma \leftarrow \mathsf{Sign}((W, T), \mathsf{sk}_{sig})$
    return $((W, T), \sigma)$
TK-proc-sig($T, W, \sigma, \mathsf{vk}_{sig}, m_u$)
    if $\exists j \;\; m_u = \mathsf{conf}'[j]$
        $1/0 \leftarrow \mathsf{Vrfy}_\Sigma(\sigma, \mathsf{vk}_{sig})$
    else
        $\mathsf{vk}'_{sig} \leftarrow \mathsf{RcvUpd}_\Sigma(\mathsf{vk}_{sig}, m_u)$
        $1/0 \leftarrow \mathsf{Vrfy}_\Sigma(\sigma, \mathsf{vk}'_{sig})$
    TK-proc$^*(T, W)$
```

Fig. 3. TreeKEM$_\Sigma$. TreeKEM$^*_\Sigma$ combines TreeKEM$_\Sigma$ and TreeKEM*. The differences between TreeKEM$_\Sigma$ and RTreeKEM are to use a digital signature scheme and GMget-pk and GMget-sk, as mentioned.

6.2 TreeKEM$^*_\Sigma$ protocol

We explain the TreeKEM$^*_\Sigma$ protocol in Fig. 2 and 3 in this section. First, in TreeKEM$^*_\Sigma$, group members are divided into GMs and other users, each running a different algorithm. Thus, TreeKEM$^*_\Sigma$ is composed of TreeKEM* in Fig. 2 and TreeKEM$_\Sigma$ in Fig. 3. TreeKEM* replaces the role of PKI in RTreeKEM with GM. Specifically, get-pk and get-sk are algorithms that query the PKI for public and private keys in RTreeKEM, respectively, but they are replaced by GMget-pk and GMget-sk in TreeKEM*. On the other hand, TreeKEM$_\Sigma$ is a protocol that incorporates the processes of signature generation, verification, and update of signature keys into TreeKEM*. Note that neither TreeKEM* nor TreeKEM$_\Sigma$ fully corresponds to GM and other users. Both GMs and other users use TreeKEM* and TreeKEM$_\Sigma$. TreeKEM$^*_\Sigma$ is a combination of TreeKEM* and TreeKEM$_\Sigma$, with the application deciding which algorithms to run for GM and other users. The application-specific details were explained in section 2. Since TreeKEM* operates in the same way as RTreeKEM except that the query to the PKI is changed to a query to GM, we omit the explanation of TreeKEM* here and describe the operation of TreeKEM$_\Sigma$ as follows.

Initialization. The initialization procedure TK-init* and TK-init-sig expect as input an ID ID and initialize several state variables: Variable ME remembers the ID of the party running the protocol and τ will keep track of the RT used.

The other variables are used to keep track of all the operations (creates, adds, removes, and updates) initiated with ME. Specifically, each time a party performs a new operation, it increases ctr and stores the potential next state in $\tau'[\mathsf{ctr}]$. Moreover, $\mathsf{conf}[\mathsf{ctr}]$ will store the control message as confirmation that the operation was accepted. These variables are reset each time proc processes a control message (which can either be one of the messages in conf or a message sent by another party). Finally, in the case of $\mathsf{TK\text{-}init\text{-}sig}$ of $\mathrm{TreeKEM}_{\Sigma}$, the key pair $(\mathsf{sk}_{\mathsf{sig}}, \mathsf{pk}_{\mathsf{sig}})$ of signature scheme Σ is additionally generated by KGen_{Σ}.

Group creation. Given lists of users $G = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$, The group creation procedure $\mathsf{TK\text{-}create}^*$ and $\mathsf{TK\text{-}create\text{-}sig}$ generate a new UPKE key pair $(\mathsf{pk}_0, \mathsf{sk}_0)$ and $(\mathsf{pk}_i, \mathsf{sk}_i)$ corresponding to the IDs in G by calling $\mathsf{GMget\text{-}pk}(|G|.i)$ for $i \in [n]$. Then they initialize a new ratchet tree by calling $\textsc{Init}$ with $G' = (\mathsf{ID}_0, G)$ and $\mathsf{pk}' = (\mathsf{pk}_0, \mathsf{pk})$ as well as 0 and sk_0. The welcome message W simply consists of G' and pk'. Finally, in the case of $\mathsf{TK\text{-}create\text{-}sig}$ of $\mathrm{TreeKEM}_{\Sigma}$, the ratcheting signature σ is additionally output with W.

Adding a group member. To add a new group member ID', $\mathsf{TK\text{-}add}^*$ and $\mathsf{TK\text{-}add\text{-}sig}$ first obtain a corresponding public key pk' from the GM (itself if GM performs) by calling $\mathsf{GMget\text{-}pk}(\mathsf{ID}')$ and then updates the RT by calling $\textsc{Add}\mathsf{ID}$ followed by $\textsc{Blank}$, which removes all keys from the new party's leaf up to the root. This ensures that the new user does not know any secret keys used by the other group members before he or she has joined. The welcome message for the new user simply consists of a public copy of the current RT (specifically, $\textsc{Pub}$ sets the sender's secret-key label to $\perp$), and the control message for the remaining group members of the IDs of the sender and the new user as well as the latter's public key. Finally, in the case of $\mathsf{TK\text{-}add\text{-}sig}$, the ratcheting signature σ is additionally output with control and welcome messages.

Removing a group member. $\mathsf{TK\text{-}rem}^*$ and $\mathsf{TK\text{-}rem\text{-}sig}$ call $\textsc{Blank}$ and remove a group member ID' removed by blanking all the keys from the leaf node of ID' to the root. This prevents parties from using keys in the future. User ID' is subsequently removed from the tree by calling $\textsc{Rem}\mathsf{ID}$. The control message for the remaining group members consists of the IDs of the sender and the removed user. Finally, in the case of $\mathsf{TK\text{-}rem\text{-}sig}$, the ratcheting signature σ is additionally output with control message T.

Performing an update. $\mathsf{TK\text{-}upd}^*$ and $\mathsf{TK\text{-}upd\text{-}sig}$ perform an update of a user ID by choosing new key pairs on their direct path by calling $\textsc{UpGen}$. The control message T for this operation consists of ME and U. The ratcheting signature σ is additionally output in the case of $\mathsf{TK\text{-}upd\text{-}sig}$.

Processing control messages. When processing control messages T and W, in the case of $\mathsf{TK\text{-}proc\text{-}sig}$ of $\mathrm{TreeKEM}_{\Sigma}$ a user first verifies the received ratcheting signature σ. If it is not verified, the process aborts. Then both of $\mathsf{TK\text{-}proc}^*$ and $\mathsf{TK\text{-}proc\text{-}sig}$ check whether the control messages correspond to the operation they initiated. If so, they simply adopt the corresponding RT in $\tau'[\cdot]$. Depending on the type of control message, proc^* operates as follows.

```
tkm $(q_1, \dots, q_q)$
    $(V, E) \leftarrow \mathsf{KG}(q_1, \dots, q_q)$
    for $(i, j)$ s.t. $q_i = \mathsf{corrupt}(\mathsf{ID})$, $t = \mathsf{q2e}(q_i)$, $q_j = \mathsf{chall}(t^*)$
    if $\mathbf{I}[t^*] \in \mathcal{K}^t_{\mathsf{ID}}$
        return 0
    return 1
```

Fig. 4. Safety predicate: **tkm** for some ID ID and epoch t^*.

- $T = (\mathsf{create}, G, \mathsf{pk})$: Determine the position j of ID in the G list, retrieve the appropriate secret key sk_j from the GM, and initialize the RT via Init.
- $W = (\mathsf{wel}, \tau')$: Adopt τ' as the current RT τ and set the secret key at ID's node to the key $\mathsf{GMget\text{-}sk}$ retrieved from the GM.
- $T = (\mathsf{add}, \mathsf{ID}, \mathsf{ID}', \mathsf{pk}')$: Add the new user ID' to the RT and blank all nodes in the direct path of the new user.
- $T = (\mathsf{rem}, \mathsf{ID}, \mathsf{ID}')$: Blank all nodes on the direct path of user ID' and remove ID' from the RT.
- $T = (\mathsf{up}, \mathsf{ID}, U)$: A user ID' at some leaf ℓ' receiving $U = (\mathsf{PK}, c)$, issued by the user with ID ID at leaf v recovers the update information via UpPro.

After processing, the variables pertaining to keeping track of ID's unconfirmed operations are reset.

6.3 Security of TreeKEM$^*_\Sigma$

This section presents the security of the TreeKEM$^*_\Sigma$ protocol. As with CGKA, the CGKA-FA protocol satisfies the safety predicate **safe**. Furthermore, the security is defined by **tkm, fsau**, and **pcsau**. We first illustrate the security predicate **tkm** of the RTreeKEM protocol in Fig. 4. Let (V, E) be the key graph KG defined by executing a sequence of operations of the RTreeKEM protocol. For a user with ID ID and an epoch t, $\mathcal{K}^t_{\mathsf{ID}}$ consists of the following elements:

1 the private keys in the state of ID in epoch t, and
2 the private keys in V that are reachable from the above keys in the key graph (V, E).

Having defined the key graph, adversaries are now captured via the predicate **tkm** in Fig. 4. This predicate essentially makes sure that the adversary does not learn any keys from which a challenged update secret is reachable.

The predicates **pcsau** and **fsau** in Fig. 5 capture PCSAU and FSAU, respectively. PCSAU is achieved by excluding corruptions after any challenge in the predicate **pcsau**. The notion of FSAU captured when the state of a party ID is leaked, all keys before the most recent update by ID remain secret in the predicate **fsau**.

```
pcsau ($q_1, \ldots, q_q$)
    if $(i, j)$ s.t. $q_i = \mathsf{corrupt}(\mathsf{ID})$, $q_j = \mathsf{chall}(t^*)$, $t^* < \mathsf{q2e}(q_i)$
        return 0
    if $t^* > \mathsf{q2e}(q_i)$
    and $\nexists k$ s.t. $q_k = \mathsf{send\text{-}update}(\mathsf{ID})$ s.t. $\mathsf{q2e}(q_i) \leq q_k < t^*$
    and $q_\ell = \mathsf{vrfy}(\sigma_{\mathsf{ID}}) \neq 1$
        return 0
    return $\mathsf{safe}(q_1, \ldots, q_q)$
fsau ($q_1, \ldots, q_q$)
    for $(i,j)$ s.t. $q_i = \mathsf{corrupt}(\mathsf{ID})$, $q_j = \mathsf{chall}(t^*)$
    if $t^* < \mathsf{q2e}(q_i)$
    and $\nexists$ s.t. $q_k = \mathsf{send\text{-}update}(\mathsf{ID})$ s.t. $t^* < \mathsf{q2e}(q_k) \leq \mathsf{q2e}(q_i)$
    and $q_\ell = \mathsf{vrfy}(\sigma_{\mathsf{ID}}) \neq 1$
        return 0
    return $\mathsf{safe}(q_1, \ldots, q_q)$
```

Fig. 5. Safety predicate: **pcs** and **fsu** for some ID ID and epoch t^*.

Moreover, TreeKEM$^*_\Sigma$ satisfies PCSAU and FSAU when a ratcheting digital signature scheme Σ is $(t_{cma}, \epsilon_{cma})$-secure. Concretely, the following theorem holds.

Theorem 1. *Assume that*

- prg *is a* $(t_{\mathsf{prg}}, \varepsilon_{\mathsf{prg}})$*-secure pseudo-random generator,*
- Π *is a* $(t_{\mathsf{cpa}}, \varepsilon_{\mathsf{cpa}})$*-CPA-secure updatable public-key encryption scheme, and*
- Σ *is a* $(t_{\mathsf{cma}}, \epsilon_{cma})$*-CMA-secure ratcheting digital signature scheme.*

TreeKEM$^*_\Sigma$ *is then* $(t, c, n, \boldsymbol{P}, \epsilon)$*-adaptive secure CGKA-FA for* $\mathsf{P} \in \{$***tkm**, **pcsau**, **fsau***$\}$. *Note that* $\epsilon = 2cn(\epsilon_{\mathsf{prg}} + \epsilon_{\mathsf{cpa}}) + (c-1)\epsilon_{\mathsf{cma}}$, $t \approx t_{\mathsf{prg}} \approx t_{\mathsf{cpa}} \approx t_{\mathsf{cma}}$ *are held.*

Proof of Theorem. To prove the above theorem, we introduce the helpful lemmas. The following lemma deals with the relationships among the **tkm**, **pcsau**, and **fsau** predicates.

Lemma 3. *When Lemma 1 holds for any sequence of queries Q, if* $\mathbf{pcsau}(\mathbf{Q}) = \mathbf{1}$ *or* $\mathbf{fsau}(\mathbf{Q}) = \mathbf{1}$*, then* $\mathbf{tkm}(\mathbf{Q}) = \mathbf{1}$.

Proof of Lemma 3. The only differences between **pcs** and **pcsau** is to verify the ratcheting signature. Therefore, there are no corruptions in epochs after t^* for both **pcs** and **pcsau**. If $\mathbf{pcsau}(\mathbf{Q}) = \mathbf{1}$, then $\mathbf{pcsau}(\mathbf{Q}) = \mathbf{pcs}(\mathbf{Q}) = \mathbf{safe}(\mathbf{Q}) =$

1. For the FSAU predicate, the authentication (i.e., the ratcheting keys) update are performed at the same time of $\mathsf{send\text{-}update}$. Therefore, If $\mathbf{fsu}(\mathbf{Q}) = \mathbf{1}$ then $\mathbf{tkm}(\mathbf{Q}) = \mathbf{1}$ holds, then immediately $\mathbf{fsau}(\mathbf{Q}) = \mathbf{1}$ then $\mathbf{tkm}(\mathbf{Q}) = \mathbf{1}$ holds.

□

For the CGKA-FA protocol, single-challenge (non-adaptive) security implies multi-challenge security, as shown by the following lemma.

Lemma 4. *Let* $\mathsf{P} \in \{\mathbf{tkm}, \mathbf{pcsau}, \mathbf{fsau}\}$ *and assume that a CGKA-FA protocol* $\mathsf{CGKA\text{-}FA}$ *is* $(t, 1, n, \mathsf{P}, \varepsilon)$*-secure. Then,* $\mathsf{CGKA\text{-}FA}$ *is also* $(t', c, n, \mathsf{P}\ ,\varepsilon')$*-secure for* $t' \approx t$ *and* $\varepsilon' = (c-1)\varepsilon$*.*

Proof of Lemma 4. We assume that the advantage of the adversary $\tilde{\mathcal{A}}'$ of CGKA with single challenge be $\tilde{\varepsilon}$ and the advantage of the adversary $\tilde{\mathcal{A}}$ of CGKA with multi challenges be $\tilde{\varepsilon'}$. According to the result of Lemma 2,

$$\tilde{\varepsilon}' = c\tilde{\varepsilon} \tag{1}$$

holds. Now we assume that the advantage of the adversary $\mathcal{A}'$ of $\mathsf{CGKA\text{-}FA}$ with single challenge be ε and the advantage of the adversary $\mathcal{A}$ of $\mathsf{CGKA\text{-}FA}$ with multi challenges be ε'. The difference between advantages of CGKA and $\mathsf{CGKA\text{-}FA}$ is to verify the ratcheting signature or not. Therefore,

$$\varepsilon_{cma} + \varepsilon = \tilde{\varepsilon}', \tag{2}$$

$$\tilde{\varepsilon} = \varepsilon' + \varepsilon_{cma} \tag{3}$$

hold. From equations (1), (2), (3), we get

$$\varepsilon_{cma} + \varepsilon = c(\varepsilon' + \varepsilon_{cma})$$
$$\therefore \varepsilon = c\varepsilon' + (c-1)\varepsilon_{cma}$$

□

Now, we resume the proof of the Theorem 1. We assume that the adversary $\mathcal{A}$ above lemma 4 will compromise the GM after they all perform update process because the GM has the initial keys of all other users. In addition, $\mathcal{A}$ can not stop updating for GM. This condition allows us to consider a security game similar to a regular game of CGKA.

The theorem is proved w.r.t. **tkm** and by considering an adversary $\mathcal{A}$ that makes only a single challenge query. The final result is obtained by applying Lemma 3 and 4.

Recall that an update operation by a node at depth d produces, for a uniformly random s_0, the values

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

where $I = s_d$ is the update secret. Moreover, the update operation encrypts each s_i under the corresponding keys on the resolution of the co-path nodes by using the CPA-secure UPKE,

$$s_1^* \xrightarrow{\mathsf{UEnc}} s_1, s_2^* \xrightarrow{\mathsf{UEnc}} s_2, \cdots, s_{d-1}^* \xrightarrow{\mathsf{UEnc}} s_{d-1}.$$

CGKA-FA then generates at most one ratcheting signature for each control message that is generated for each operation performed. Consider a hybrid argument where pseudorandom number generation, updatable encryption, and ratcheting signature generation are all replaced by a random string. Let the advantage of original game be $\tilde{\varepsilon}_0 = \tilde{\varepsilon}$ and let $\tilde{\varepsilon}_d$ the advantage which all hybrids are applied for RT of depth d. Since the CPA-secure UPKE is used for all nodes of RT from layer 1 to $d-1$, the total cpa hybrids is

$$\Sigma_{i=0}^{d-1} 2^i \cdot \varepsilon_{cpa} = (2^d - 1)\varepsilon_{cpa} = (n-1)\varepsilon_{cpa} \geq n\varepsilon_{cpa}. \tag{4}$$

Since the pseudorandom generator is used for all nodes of RT from layer 2 to d, the total prg hybrids is

$$\Sigma_{i=1}^{d} 2^i \cdot \varepsilon_{\mathsf{prg}} = (2^{d+1} - 1)\varepsilon_{\mathsf{prg}} = (2n-2)\varepsilon_{\mathsf{prg}} \geq 2n\varepsilon_{\mathsf{prg}}. \tag{5}$$

The CMA-secure ratcheting signature is used once, the cma hybrid is ε_{cma}. From equations (4) and (5) and cma hybirid, we get

$$\begin{aligned}\tilde{\varepsilon}_d &\leq \tilde{\varepsilon}_0 + n\varepsilon_{cpa} + 2n\varepsilon_{\mathsf{prg}} + \varepsilon_{cma} \\ &\leq \frac{1}{2} + 2n(\varepsilon_{cpa} + \varepsilon_{\mathsf{prg}}) + \varepsilon_{cma}.\end{aligned}$$

Therefore,

$$\varepsilon := \left| \Pr[\mathcal{A} \text{ wins}] - \frac{1}{2} \right| = \left| \tilde{\varepsilon}_d - \frac{1}{2} \right| \leq 2n(\varepsilon_{cpa} + \varepsilon_{\mathsf{prg}}) + \varepsilon_{cma}.$$

Now from Lemmas 3 and 4, we get

$$\epsilon = 2cn(\epsilon_{\mathsf{prg}} + \epsilon_{\mathsf{cpa}}) + (c-1)\epsilon_{\mathsf{cma}}.$$

□

7 Evaluation

In this section, we show how much overhead is incurred for our proposed protocol against existing RTreeKEM and explain how many specific cryptographic operations are required in Table 3. One of the advantages of our protocol is that it brings significant advantages over RTreeKEM, a scheme currently being standardized in the IETF MLS, with a minimal change: the adoption of ratcheting digital signatures which can be generically constructed from a signature scheme. Here, we employ ECDSA[21] as the underlying signature scheme for constructing the ratcheting signature scheme. kP denotes multiple of points of elliptic curve. ℓ denotes modular inverse. The main costs incurred in the cryptographic operations of ECDSA are as follows: Key generation of ECDSA requires one multiple of points of elliptic curve kP. Signature generation of ECDSA require one multiple of points of elliptic curve and one modular inverse (kP + ℓ). Verification of ECDSA requires two multiples of points of elliptic curve and one modular inverse $(2k\mathrm{P}+\ell)$. In our protocol, we also let GM generate UPKE keys for the members in the group to answer queries of GMget-pk, which is a cost that GM has the PKI functionality. We estimate this cost by employing the UPKE scheme based on Computational Diffie-Hellman (CDH) under the random oracle model [3]. The key generation of the CDH-based UPKE scheme requires $k\mathrm{P}$ as the computational cryptographic operation cost.

Our CGKA-FA scheme TreeKEM$^*_\Sigma$ consists of two components, TreeKEM* and TreeKEM$_\Sigma$, each of which can be used with or without a signature on the output. It is important to note that no overhead is incurred in TreeKEM* compared to the original RTreeKEM. We then describe the additional costs involved in TreeKEM$_\Sigma$.

First, during setup, there is an overhead of one ratchet signature and $(n-1)$ UPKE key generation, where n denotes the number of group member. The total estimated computational cost during setup is $k\mathrm{P} + (n-1)k\mathrm{P} = nk\mathrm{P}$. This overhead will not be a bottleneck because the cost of PKI functionality is only necessary for the first time and can be prepared offline in advance.

The sender side computational cost is for one ratchet signature and one update of the ratchet signature key. The update of the ratchet signing key actually consists of key generation and its signature, so the total overhead is two ratchet signatures and one key generation. Thus, the computational cost with ECDSA is $2(k\mathrm{P}+\ell) + k\mathrm{P} = 3k\mathrm{P} + 2\ell$.

Next, the computational cost on the receiver side is one verification of the ratchet signature and one RcvUpd_Σ of the ratchet signature key. Since the RcvUpd_Σ algorithm runs the verification algorithm internally, the computational cost using ECDSA is $2(2k\mathrm{P}+\ell) = 4k\mathrm{P} + 2\ell$.

Finally, regarding the communication cost, there is one ratchet signature for each control message and one update message m_u related to the update of the ratchet signature as overhead. Since m_u actually consists of a new public key and its signature, there is a total additional costs of $2|\sigma| + |\mathsf{vk}_{sig}|$ involved in the communication. From the above, our proposed protocol can be implemented at a very low additional cost, both in terms of computation and communication.

Table 3. Additional costs of our protocol against existing RTreeKEM per single operation. We employ ECDSA as an underlying signature scheme of a ratcheting digital signature and the CDH-based UPKE scheme as a UPKE scheme.

	Setup cost	Comp. cost (Sender)	Comp. cost (Receiver)	Comm. cost
TreeKEM*	0	0	0	0
TreeKEM$_\Sigma$	$n\mathsf{kp}$	$3\mathsf{kp}+2\ell$	$4\mathsf{kp}+2\ell$	$2\lvert\sigma\rvert + \lvert\mathsf{vk}_{sig}\rvert$

8 Conclusion

We achieve the CGKA-FA protocol by incorporating a ratcheting digital signature scheme into the existing CGKA protocol. In addition, we analyze the security of CGKA-FA, where the secret information is not managed by a third party. In our CGKA-FA protocol, it is possible to set the privileges of users in a group, which is highly compatible with the field of broadcasting and may lead to the development of new broadcasting services.

References

1. Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzyzstof Pietrzak, and Michael Walter. CoCoA: Concurrent Continuous Group Key Agreement. In EUROCRYPT. 2022. Springer, pp. 815–844.
2. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In EUROCRYPT. 2019. Springer, pp. 129–158.
3. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security Analysis and Improvements for the IETF MLS Standard for Group Messaging. In CRYPTO. 2020. Springer, pp. 248–277.
4. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. In ACM CCS. 2021. ACM, pp. 1463–1483.
5. Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. Cryptology ePrint Archive, Report 2021/1456 (2021).
6. Joël Alwen, Daniel Jost, and Marta Mularczyk. On the Insider Security of MLS. In CRYPTO. 2022. Springer, pp. 34–68.
7. David Balbás, Collins Daniel, and Serge Vaudenay. 2022. Cryptographic Administration for Secure Group Messaging. Cryptology ePrint Archive, Report 2022/1411 (2022).
8. Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-8. 2019. Internet Engineering Task Force. `https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-8`.
9. Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-16. 2022. Internet Engineering Task Force. `https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/16`. Work in Progress.
10. Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. Multicast Key Agreement, Revisited. In Cryptographers' Track at the RSA Conference. 2022. Springer, pp. 1–25
11. Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A More Complete Analysis of the Signal Double Ratchet Algorithm. In CRYPTO. 2022. Springer, pp. 782–811.
12. Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communica- tion, or, why not to use PGP. In Proceedings of the 2004 ACM workshop on Privacy in the electronic society. 2004. ACM, pp. 77–84.

13. Melissa Chase, Trevor Perrin, and Greg Zaverucha. The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable En- cryption. In ACM CCS. 2020. ACM, pp. 1445–1459.
14. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. In IEEE EuroS&P. 2017. IEEE, pp. 451–466.
15. Cas Cremers, Britta Hale, and Konrad Kohbrok. The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter. In USENIX Security. 2021. USENIX, pp. 1847–1864.
16. Yevgeniy Dodis and Nelly Fazio. Public key broadcast encryption for stateless receivers. In ACM Workshop on Digital Rights Management. 2002. Springer, pp. 61–80.
17. Keita Emura, Kaisei Kajita, Ryo Nojima, Kazuto Ogawa, and Go Ohtake. Membership privacy for asynchronous group messaging. In WISA 2022 to appear. Available at Cryptology ePrint Archive 2022/046.
18. Amos Fiat and Moni Naor. Broadcast encryption. In CRYPTO. 1993. Springer, pp. 480–491.
19. Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An Efficient and Generic Construction for Signal' s Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable. In PKC. 2021. Springer, pp. 410–440.
20. Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In ACM CCS. 2001. ACM, pp. 1441–1462.
21. Don Johnson, Menezes Alfred, and Vanstone Scott. The elliptic curve digital signature algorithm (ECDSA). International journal of information security 1.1. 2001. pp. 36–63.
22. Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In IEEE S&P. 2021. IEEE, pp. 268–284.
23. T. Perrin M. Marlinspike. 2016. The Signal Protocol. Technical Report. Working Draft, Technical Report, November 2016. `https://signal.org/docs/specifications`.
24. J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-7. 2019. Internet Engineering Task Force. `https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-7`.
25. Kazuto Ogawa, Goichiro Hanaoka, and Hideki Imai. Traitor tracing scheme secure against adaptive key exposure and its application to anywhere TV service. IEICE transactions on fundamentals of electronics, communications and computer sciences 90, 5 (2007), pp. 1000–1011.
26. Nick Sullivan and Sean Turner. Message layer security (mls) working group. `https://datatracker.ietf.org/wg/mls/about/`.
27. Nihal Vatandas, Rosario Gennaro, Bertrand Ithurburn, and Hugo Krawczyk. On the Cryptographic Deniability of the Signal Protocol. In ACNS. 2020. Springer, pp. 188–209.
28. Matthew Weidner. Group messaging for secure asynchronous collaboration. Master thesis, University of Cambridge, 2019.