Master's Thesis

An Improved TreeKEM protocol with stronger PCS

Tomoko Otsuru

Supervisor Eiichiro Fujisaki

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

March, 2025

Abstract

Secure Messaging protocols enable end-to-end secure communication over untrusted network and server infrastructure. They are used in major application services that provide secure message exchange between users, such as Signal, Facebook Messenger, etc. Their sessions may be long-lived and users may be offline, so they should guarantee Forward Secrecy (FS) and Post-compromise Security (PCS). Forward Secrecy satisfies that the past session keys remain secret even if a user is compromised at some point, while Post-compromise Security enables a user's session key to be secure again after some type of key updates (ideally, after any key update). TreeKEM is a continuous group key agreement (CGKA) protocol whose security has recently been analyzed by Alwen et al. (CRYPTO 2018). And it is at the core of the secure group messaging protocol discussed in the IETF MLS working group. In this paper, we focus on PCS of TreeKEM and provide better PCS security to TreeKEM with a simple modification. Our modification makes it significantly easier for the protocol to recover from a compromise.

Contents

List of Figures

Chapter 1

Introduction

Secure Messaging protocols enable end-to-end secure communication over untrusted network and server infrastructure. They are used in major application services that provide secure message exchange between users, such as Signal [1], Facebook Messenger [2] and etc. In 2018, an official working group was formed at the IETF and MLS was published as RFC 9420 in 2023 [4]. Google and other companies, including AWS, Cisco, Cloudflare, Meta, Wire, and Matrix, came out in support of MLS [3].

TreeKEM is a continuous group key agreement (CGKA) protocol and is at the core of the Secure Group Messaging (SGM) protocol in the IETF MLS working group. Alwen et al. [5] have first analyzed the security of TreeKEM and are followed by several papers [7, 8, 9, 10]. In [5], they have claimed that the original TreeKEM does not satisfy forward secrecy (FS) and proposed a modification that satisfies FS. Their modification is very simple: They have replaced the public-key encryption used in the original TreeKEM with updatable public-key encryption (hereafter referred to as TreeKEM with updatable PKE). As for PCS, [5] has claimed that the original TreeKEM is sound against the post-compromise attacks. However, their definition of PCS is not well deployed in the case of TreeKEM because it is defined in a general CGKA protocol. In TreeKEM, each user keeps plural secrets corresponding to each node in "the tree" of TreeKEM, a part of which is used to obtain the new group session key I. In Alwen's definition, the key compromise should always reveal the whole inner states of a compromised user. They do not consider "partial reveal". In this paper, we consider the node-by-node key compromise, which would be more suitable for TreeKEM. In addition, we introduce an extra key derived from the root secret. With this modification, we can significantly increase the case of key updates that the protocol can recover from compromise.

In Chapter 3, we explain TreeKEM with updatablePKE. In Chapter 4,

we describe the details of the node-by-node key compromise. In Chapter 5 we propose our modified protocol based on Alwen et al.'s protocol (i.e., TreeKEM with updatablePKE). We introduce a common extra key, k, derived from the root secret in a simple way.

1.1 Contributions

First, we analyze the security of TreeKEM with updatablePKE [5] in the context of node-by-node key compromise. We observe that compromising a single node key can lead to a chain reaction, causing further compromise of other node keys and update secrets when the compromised node lies on the update co-path — the sequence of sibling nodes from the update node to the root. In other words, update secrets I, from which users derive session keys, remain secure unless the node is part of the update co-path. However, the tree is potentially unsafe because users retain compromised keys.

Second, we propose an improvement to TreeKEM with updatablePKE. We introduce a common extra key, k, shared among all users, in a straightforward manner. In TreeKEM with the updatablePKE protocol, users compute the node key of a node v in the epoch t based on its value in epoch $t-1$, or the key remains unchanged. This dependency is the root cause of chain compromise and the retention of compromised keys. In our protocol, users incorporate k alongside the node key when encrypting or decrypting the path secrets s, which users use to compute update secrets I, and they update k in every epoch. Therefore, even if the attacker compromises both a node key and k, these keys become invalid after any user update, except for users possessing the sibling of the compromised node. Consequently, our protocol provides stronger PCS.

1.2 Related Work

Karthikeyan Bhargavan et al. proposed TreeKEM [6] and since then, many studies have been conducted on this topic [5, 7, 8, 9, 10, 11]. Notably, Joel Alwen et al.analyzed its security in detail [5]. They precisely analyze the security of TreeKEM within the CGKA framework and highlight that TreeKEM does not provide sufficient forward secrecy (FS). To address this limitation, they propose a modification to TreeKEM that incorporates updatablePKE, inspired by the work of Jost et al. [12]. In their revised protocol, the public and secret node keys are appropriately updated during encryption and decryption, respectively. This modification achieves optimal FS, ensuring

that after decryption, the node key reveals no information about the original message.

Chapter 2

Preliminaries

This section provides an overview of basic concepts related to binary trees as well as definitions of pseudorandom generators (PRGs), CPA-secure public-key encryption, and CGKA scheme as described in [5].

2.1 Binary trees

We denote a binary tree as τ and each node in the tree has either 0 or 2 unique children. The height of τ is defined as the length of the longest path from the root to any leaf. The node of height 0 is the root, a node with no children is called a leaf, and all the other nodes are called internal. We call a tree a full binary tree $\mathtt{FT}_h$ whose height is h and has 2^h leaves. Given two leaf nodes l and l' in a tree, let $\mathtt{LCA}(l, l')$ represent their least common ancestor.

2.2 Pseudorandom Generators

A pseudorandom generator is $\mathtt{prg} : \{0,1\}^n \to \{0,1\}^m$ with $n < m$. $\mathtt{prg}(U)$ for uniformly random $U \in \{0,1\}^n$ is indistinguishable from U' for uniformly random $U' \in \{0,1\}^m$. The advantage of an attacker at distinguishing between these two distributions is denoted by $\mathtt{Adv}^{\mathtt{prg}}_{\mathrm{prg}}(\mathcal{A})$.

Definition 1 (Pseudorandom Generator) *A pseudorandom generator* $\mathtt{prg}$ *is* (t, ε)*-secure if for all* t*-attackers* $\mathcal{A}$*,*

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

2.3 Public-key Encryption

Definition 2 (Public-key Encryption) *A public-key encryption (PKE) scheme is a triple of algorithms* $\Pi = ($PKEG, Enc, Dec$)$ *as follows.*

- *A key generation algorithm* PKEG *takes a security parameter and outputs* (pk, sk)*.*
- *An encryption algorithm* Enc *is an algorithm that takes a message* m *along with a public key* pk *and outputs a ciphertext* c*.*
- *A decryption algorithm* Dec *is a deterministic algorithm that takes a ciphertext* c *along with a secret key* sk *and outputs a message* m*.*

PKE should satisfy the correctness condition. For any message m,

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

IND-CPA security for PKE. We consider the following security game.

- Compute (pk, sk) $\leftarrow$ PKEG and $b \leftarrow \{0, 1\}$
- The adversary $\mathcal{A}$ takes pk and outputs (m_0, m_1)
- The adversary $\mathcal{A}$ receives $c \leftarrow \mathtt{Enc}(\mathtt{pk}, m_b)$
- $b' \leftarrow \mathcal{A}\ (c)$

We define that $\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ is defined by $\mathtt{Adv}^{\Pi}_{\mathrm{cpa}}(\mathcal{A})$.

Definition 3 *A public-key encryption scheme* Π *is* (t, ε)*-CPA-secure if for all* t*-attackers* $\mathcal{A}$*,*

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

2.4 Updatable Public-key Encryption (UPKE)

Definition 4 (Updatable Public-key Encryption) *An updatable public-key encryption (UPKE) scheme is a tripe of algorithms* UPKE $=($PKEG, Enc, Dec$)$ *as follows.*

- *A key generation algorithm* PKEG *is a probabilistic algorithm that takes a uniformly random key* $\mathtt{sk}_0$ *and outputs a initial public key* $\mathtt{pk}_0 \leftarrow \mathtt{PKEG}(\mathtt{sk}_0)$*.*

- *An encryption algorithm* $\texttt{Enc}$ *is a probabilistic algorithm that takes a message* m *along with a public key* $\texttt{pk}$ *and outputs a ciphertext* c *and a new public key* $\texttt{pk}'$.
- *A decryption algorithm* $\texttt{Dec}$ *takes a ciphertext* c *along with a secret key* $\texttt{sk}$ *and outputs a message* m *and a new secret key* $\texttt{sk}'$.

IND-CPA security for UPKE. We consider the following security game.

- Pick up $\texttt{sk}_0 \leftarrow \{0,1\}^\kappa$, and compute $\texttt{pk}_0 \leftarrow \texttt{PKEG}(\texttt{sk}_0)$, and $b \leftarrow \{0,1\}$.
- The adversary $\mathcal{A}$ takes $\texttt{pk}_0$ and for $i = 1, ..., q$, $\mathcal{A}$ outputs m_i and receives $(c_i, \texttt{pk}_i, r_i)$ such that $(c_i, \texttt{pk}_i) \leftarrow \texttt{Enc}(\texttt{pk}_{i-1}, m_i; r_i)$, for uniformly random r_i. Compute $(m_i, \texttt{sk}_i) \leftarrow \texttt{Dec}(\texttt{sk}_{i-1}, c_i)$.
- The adversary $\mathcal{A}$ outputs (m_0^*, m_1^*) and receives $(c^*, \texttt{pk}^*) \leftarrow \texttt{Enc}(\texttt{pk}_q, m_b^*; r_{q+1})$ and $\texttt{sk}^*$, $(\cdot, \texttt{sk}^*) \leftarrow \texttt{Dec}(\texttt{sk}_q, c^*)$.
- $b' \leftarrow \mathcal{A}(\texttt{pk}^*, \texttt{sk}^*, c^*)$

We define that $\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ is defined by $\mathsf{Adv}_{\text{cpa}}^{\texttt{UPKE}}(\mathcal{A})$.

Definition 5 *An updatable public-key encryption scheme* $\texttt{UPKE}$ *is* (t, ε)*-CPA-secure if for all* t*-attackers* $\mathcal{A}$*,*

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

2.5 Continuous Group Key Agreement

2.5.1 CGKA Security

Continuous Group Key Agreement (CGKA) enables group members to continuously share fresh secret random values, which they use to update their key material. In this section, we describe the main oracles of the CGKA security game, following the work of Alwen et al. [5].

Definition 6 (CGKA) *CGKA scheme* $\texttt{CGKA}$ *=* ($\texttt{init}$, $\texttt{create}$, $\texttt{add}$, $\texttt{rem}$, $\texttt{upd}$, $\texttt{proc}$) *consists of the following algorithms.*

- *A initialization algorithm* $\texttt{init}$ *takes an ID* $\texttt{ID}$ *and outputs an initial states* γ.

- *A group creation algorithm* `create` *takes a state* γ *and a list of IDs* $\mathbf{G}$ $= (\mathtt{ID}_1, \ldots, \mathtt{ID}_n)$*, then outputs a new state* γ' *and a control message* W*.*
- *An add algorithm* `add` *takes a state* γ *and an ID* $\mathtt{ID}'$*, then outputs a new state* γ' *as well as control messages* W *and* T*.*
- *A remove algorithm* `rem` *takes a state* γ *and an ID* $\mathtt{ID}'$ *then outputs a new state* γ' *and a control message* T*.*
- *An update secret random values algorithm* `upd` *takes a state* γ *then outputs a new state* γ' *and a control message* T*.*
- *A process algorithm* `proc` *takes a state* γ *and a control message* T*, then outputs a new state* γ' *and an update secret* I*.*

In CGKA game, there are group members and the server. Users update secret random values epoch by epoch. When a user takes an action except for process, it takes a current state γ and parameters and outputs a new state γ' and a control message. Control messages are stored in the server and other users process the messages at any time in order. The oracles are formalized through the security game described below. The attacker is granted access to various oracles to control the execution of a CGKA protocol. However, the attacker's capabilities and the restrictions on the order in which it may invoke the oracles are designed based on how a CGKA protocol would be used within a higher-level protocol. Most importantly, the attacker is not permitted to modify or inject any control messages.

CGKA game

init

$b \leftarrow \$\{0,1\}$
$\forall \texttt{ID} : \gamma[\texttt{ID}] \leftarrow \texttt{init}(\texttt{ID})$
$\texttt{lead}[\cdot], \mathbf{I}[\cdot], \mathbf{G}[\cdot] \leftarrow \varepsilon$
$\texttt{ep}[\cdot], \texttt{ctr}[\cdot] \leftarrow 0$
$D[\cdot] \leftarrow \texttt{true}$
$\texttt{chall}[\cdot] \leftarrow \texttt{false}$
$\textbf{pub}M[\cdot] \leftarrow \varepsilon$

add-user$(\texttt{ID}, \texttt{ID}')$

$t \leftarrow \texttt{ep}[\texttt{ID}]$
req$t > 0 \wedge \texttt{ID}' \notin \mathbf{G}[t]$
$c \leftarrow {+}{+}\,\texttt{ctr}[\texttt{ID}]$
$(\gamma[\texttt{ID}], W, T) \leftarrow \texttt{add}(\gamma[\texttt{ID}], \texttt{ID}')$
$M[t+1, \texttt{ID}, \texttt{ID}', c] \leftarrow (W, T)$
for$\tilde{\texttt{ID}} \in \mathbf{G}[t]$
$M[t+1, \texttt{ID}, \tilde{\texttt{ID}}, c] \leftarrow T$
$\mathbf{G}[t+1, \texttt{ID}, c] \leftarrow \mathbf{G}[t] \cup \{\texttt{ID}'\}$

deliver$(t, \texttt{ID}, \texttt{ID}', c)$

req $\texttt{lead}[t] \in \{\varepsilon, (\texttt{ID}, c)\}$
$\wedge (t = \texttt{ep}[\texttt{ID}'] + 1, \texttt{added}(t, \texttt{ID}, \texttt{ID}', c))$
$T \leftarrow M[t, \texttt{ID}, \texttt{ID}', c]$
$(\gamma[\texttt{ID}'], I) \leftarrow \texttt{proc}(\gamma[\texttt{ID}'], T)$
if $\texttt{lead}[t] = \varepsilon$
$\texttt{lead}[t] \leftarrow (\texttt{ID}, c)$
$\mathbf{I}[t] \leftarrow I$
$\mathbf{G}[t] \leftarrow \mathbf{G}[t, \texttt{ID}, c]$
else if $I \neq \mathbf{I}[t]$
win
if $\texttt{rem}(t, \texttt{ID}')$
$\texttt{ep}[\texttt{ID}'] \leftarrow -1$
else
$\texttt{ep}[\texttt{ID}'] + +$
$\texttt{ctr}[\texttt{ID}'] \leftarrow 0$

create-group$(\texttt{ID}_0, \texttt{ID}_1, ..., \texttt{ID}_n)$

$t \leftarrow \texttt{ep}[\texttt{ID}]$
req$t = 0$
$c \leftarrow {+}{+}\,\texttt{ctr}[\texttt{ID}_0]$
$(\gamma[\texttt{ID}_0], W)$
$\leftarrow \texttt{create}(\gamma[\texttt{ID}_0], \texttt{ID}_1, ..., \texttt{ID}_n)$
for$i = 0, ..., n$
$M[t+1, \texttt{ID}_0, \texttt{ID}_i, c] \leftarrow W$
$\mathbf{G}[t+1, \texttt{ID}, c] \leftarrow \{\texttt{ID}_0, \texttt{ID}_1, ..., \texttt{ID}_n\}$

remove-user$(\texttt{ID}, \texttt{ID}')$

$t \leftarrow \texttt{ep}[\texttt{ID}]$
req$t > 0 \wedge \texttt{ID}' \in \mathbf{G}[t]$
$c \leftarrow {+}{+}\,\texttt{ctr}[\texttt{ID}]$
$(\gamma[\texttt{ID}], T) \leftarrow \texttt{rem}(\gamma[\texttt{ID}], \texttt{ID}')$
for$\tilde{\texttt{ID}} \in \mathbf{G}[t]$
$M[t+1, \texttt{ID}, \tilde{\texttt{ID}}, c] \leftarrow T$
$\mathbf{G}[t+1, \texttt{ID}, c] \leftarrow \mathbf{G}[t] \setminus \{\texttt{ID}'\}$

corrupt$(\texttt{ID})$

return $\gamma[\texttt{ID}]$

reveal(t)

req $\mathbf{I}[t] \notin \{\varepsilon, \perp\} \wedge \neg\texttt{chall}[t]$
$\texttt{chall} \leftarrow \texttt{true}$
return $\mathbf{I}[t]$

send-update$(\texttt{ID})$

$t \leftarrow \texttt{ep}[\texttt{ID}]$
req$t > 0$
$c \leftarrow {+}{+}\,\texttt{ctr}[\texttt{ID}]$
$(\gamma[\texttt{ID}], T) \leftarrow \texttt{upd}(\gamma[\texttt{ID}])$
for$\tilde{\texttt{ID}} \in \mathbf{G}[t]$
$M[t+1, \texttt{ID}, \tilde{\texttt{ID}}, c] \leftarrow T$
$\mathbf{G}[t+1, \texttt{ID}, c] \leftarrow \mathbf{G}$

no-del$(\texttt{ID})$

$D[\texttt{ID}] \leftarrow \texttt{false}$

chall(t)

req $\mathbf{I}[t] \notin \{\varepsilon, \perp\} \wedge \neg\texttt{chall}[t]$
$I_0 \leftarrow \mathbf{I}[t]$
$I_1 \leftarrow \mathcal{K}$
$\texttt{chall}[t] \leftarrow \texttt{true}$
return I_b

Epochs. First, the attacker creates a group with a list of IDs in epoch 1. Thereafter, any group member may add new parties, remove existing members, or perform an update. The four oracles **create-group**, **add-user**, **remove-user**, and **send-update** initiate new epochs, and **deliver** ensuring that parties transition to the next epoch. If multiple parties attempt to initiate a new epoch, the attacker selects a single operation to define the new epoch; the corresponding sender is referred to as the *leader* of the epoch.

Initialization. The init oracle **init** sets up the game and initializes all the parameters to track execution;

- b: A random bit which is used for challenges.
- γ: All user states that include ID of group members, public keys of all nodes, and secret keys of itself.

- `lead`: The leader of the epoch, whose operation defines the new epoch. In **deliver** the attacker determines it and its control message is processed by all group users.
- **I**: The update secret which is labeled at root. It is shared by all group members and used for key update.
- **G**: The group members list.
- `ep`: The epoch in which each user is currently in. Each user advances to the next epoch through process.
- `ctr`: The number of new operations which is initiated by a user within its current epoch, so we call this the local version number. Whenever a user moves to the next epoch, it resets.
- D: The flag indicates whether the user deletes its values or not. After **no-del** user does not delete its old values, instead it replaces them with new ones.
- `chall`: The flag indicates whether the attacker is allowed to issue a challenge for the epoch.
- M: All control messages which are stored in the server.

Leaders and local version number.
Control messages are stored in M with key ($t+1$, `ID`, $\texttt{ID}_i$, $\texttt{ctr}[\texttt{ID}]$), responding to the number of the next epoch, the sender, the recipient, and the local version number of the operation. The leader for epoch $t+1$ is the ID that is designed as the sender of the first control message delivered via **deliver** for that epoch. Additionally, the leader also sends a control message to itself, and the operation is completed when the control message returns to the leader and is processed in the same way as for all other users.

Group creation. When a user `ID` is in epoch 0, the oracle **create-group** allows `ID` to create a group with members $\{\texttt{ID}_0, ..., \texttt{ID}_n\}$. User `ID` calls the group creation algorithms and sends the resulting welcome messages to all users, including itself.

Adding and removing users and performing updates. Three oracles **add-user**, **remove-user**, and **send-update**, call the the corresponding CGKA algorithms, `add`, `rem`, and `upd` respectively, if the **req** statement is true and send the resulting control messages to the server.

Delivering control messages. The oracle **deliver** allows users to process the control messages stored in M. First, the **req** statement checks that (1) either there is no leader for epoch t yet or version c of $\texttt{ID}$ is the leader already and (2) the user $\texttt{ID}'$ is currently in epoch $t-1$ or a newly added group member. The predicate $\texttt{added}$ is defined by

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

If there is no leader for epoch t yet, the game selects a leader as explained above and stored the update secret in $\mathbf{I}$ for epoch t. The other case, whenever users get the update secret by $\texttt{proc}$, they checke it against $\mathbf{I}$ for t ensuring the correctness. Finally, the epoch counter is updated. If the process removes the user itself, the epoch counter is set to -1. The predicate $\texttt{removed}$ is defined as follows

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

Challenges and reveals. For each epoch, the attacker can call either **chall** or **reveal**. When calling **chall**(t) for some t, the oracle first checks that t indeed corresponds to an update epoch and that a leader already exists. The oracle **reveal** allows the attacker to know the update secret of an epoch.

Corruptions and deletions. The oracle **corrupt**($\texttt{ID}$) allows the attacker to learn all the inner states of user $\texttt{ID}$. The oracle **no-del**($\texttt{ID}$) causes user $\texttt{ID}$ to stop deleting old secrets values instead updating them with new ones. When the attacker call **corrupt**($\texttt{ID}$) after **no-del**($\texttt{ID}$), it gets all secret keys between the epochs.

Avoiding trivial attacks. To prevent the attacker from trivially winning the CGKA security game such as by challenging an epoch t's update secret and leaking some party's state in epoch t, the predicate **safe** is evaluated at the end of the game. This evaluation is performed on the set of queries $\mathbf{q_1},\ldots,\mathbf{q_q}$ to ensure that the execution was not susceptible to such trivial attacks. Specifically, the predicate checks whether the attacker could have directly computed the update secret in the challenge epoch t^* using the leaked state of a user $\texttt{ID}$ in some epoch t along with the control messages observed on the network.

The case is as follows.

1. $\texttt{ID}$ has not performed an update or been removed before the challenge epoch after corruption.

2. ID stopped deleting values at some point up to the challenge epoch and was corrupted.

The predicate is depicted below. The function $\mathtt{q2e}(\mathbf{q})$ returns the epoch corresponding to query $\mathbf{q}$. Specifically, for $\mathbf{q} \in \{$ **corrupt**(ID), **no-del**(ID) $\}$, if ID is in the group when a user calls $\mathbf{q}$, it returns the value of $\mathtt{ep}[\mathtt{ID}]$; otherwise, returns $\perp$. For $\mathbf{q} \in \{$**send-update**$(\mathtt{ID}),$ **remove-user**$(\mathtt{ID}, \mathtt{ID}')\}$, $\mathtt{q2e}(\mathbf{q})$ is the epoch for which any user initiates to process the operations. If $\mathbf{q}$ is not processed by any user, it returns $\perp$.

Safe Predicate

safe($\mathbf{q_1}$,...,$\mathbf{q_q}$)
  for (i,j) *s.t.*
   $\mathbf{q_i}$ = **corrupt**(ID) for some ID and $\mathbf{q_j}$ = **chall** (t^*) for some t^*
    if q2e $(\mathbf{q_i}) \leq t^*$ and $\nexists k$
      s.t. $0 <$ q2e $(\mathbf{q_i}) <$ q2e $(\mathbf{q_k}) \leq t^*$
      and $\mathbf{q_k} \in \{$**send-update**(ID), **remove-user**$(*,$ ID$)\}$
      return 0
    if q2e $(\mathbf{q_i}) > t^*$ and $\exists k$
      s.t. q2e $(\mathbf{q_k}) \leq t^*$ and $\mathbf{q_k}$ = **no-del** (ID)
      return 0
  return 1

Advantage. The attacker runs in time at most t, makes at most c challenges and never creates a group with more than n users (hereafter referred to as a $(t, c, n) - attacker$) and wins the CGKA security game if it correctly guesses the random bit b at the end and the safety predicate evaluates to `true`. We generally replace the predicate **safe** for any other predicate $\mathbf{P}$. The advantage of $\mathcal{A}$ with the safety predicate $\mathbf{P}$ against a CGKA scheme is defined by

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

Definition 7 (Non-adaptive CGKA security) *A CGKA protocol* CGKA *is non-adaptively $(t, c, n, \mathbf{P}, \varepsilon)$-secure if for all (t, c, n)-attackers,*

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

With non-adaptive security, an attacker is required to announce all corruptions at the beginning.

2.5.2 Forward Secrecy and Post-compromise Security

CGKA protocols provide forward secrecy and PCS, those are the basic properties.

Forward Secrecy (FS). It means leakage in the future does not cause leakage in the past. If the state of any group member is leaked at some point, all previous update secrets remain hidden from the attacker.

Post-compromise Security (PCS). It means recovery from past leakage. If the attacker knows the inner state of group members in the past, the update secrets become secret again after every group user whose state was leaked performs an update.

2.6 Single-Challenge Security to Multi-Challenge Security

For CGKA schemes, single-challenge non-adaptive security implies multi-challenge security, as shown by the following lemma. The proof of lemma 1 is given in [5].

Lemma 1 *Single-challenge to multi-challenge*
Assume that a CGKA protocol is (t, 1, n, $\mathbf{P}$*,* ε*)-secure. Then,* `CGKA` *is also (*t'*, c, n,* $\mathbf{P}$*,* ε'*)-secure for* $t' \approx t$ *and* $\varepsilon' = c\varepsilon$*.*

Chapter 3

TreeKEM with updatablePKE

3.1 Overview

In a TreeKEM RT, the group members are arranged in the leaves and all the nodes have an associated public-key encryption (PKE) pair, `pk` and `sk`, except for the root. Each user knows all secret keys on the nodes from the leaf to the root (hereafter referred to as *direct-path*). In order to perform an update and produce a new update secret I, a user first generates fresh key pairs on every node of its direct-path. Then, for every node v', the sibling of every node on the *direct-path*, it encrypts path secrets under the public key of v'. Each user in the subtree of v' can learn all new path secrets and keys from the parent of v' up to the root.

In this chapter, we explain the basic concepts around ratchet trees (RTs) and TreeKEM with updatablePKE in the same way as Alwen et al. [5]. The difference from the original TreeKEM is the use of UPKE. The pubic and secret node keys change suitably with encryption and decryption, respectively, and it provides optimal FS.

3.2 Rathcet trees

An RT in TreeKEM is a *left-balanced binary tree (LBBT)*. An LBBT has a maximal full binary tree as its left child and an LBBT on the remaining nodes as its right child.

Definition 8 (Left-Balanced Binary Tree) *For $n \in \mathbb{N}$, we denote a left-balanced binary tree with n nodes as* $\texttt{LBBT}_\texttt{n}$. *It is constructed as follows.*

- *The tree* $\texttt{LBBT}_\texttt{1}$ *is a single node.*

- *Let* $x = \texttt{mp2}(n)$. $\texttt{mp2}(n)$ *is the maximum power of two dividing* n. $\texttt{LBBT}_\texttt{n}$ *has the full subtree* $\texttt{FT}_\texttt{x}$ *as its left subtree and* $\texttt{LBBT}_{\texttt{n}-\texttt{x}}$ *as its right subtree.*

The nodes are labeled as follows. Labels are referred to using dot-notation (e.g., $v.\texttt{pk}$ is v's public key).

- The root is labeled with an update secret I.
- The internal nodes are labeled by a key pair ($\texttt{pk}$, $\texttt{sk}$) for the UPKE scheme.
- The leaf nodes are labeled as internal nodes, except that they also have an owner $\texttt{ID}$.

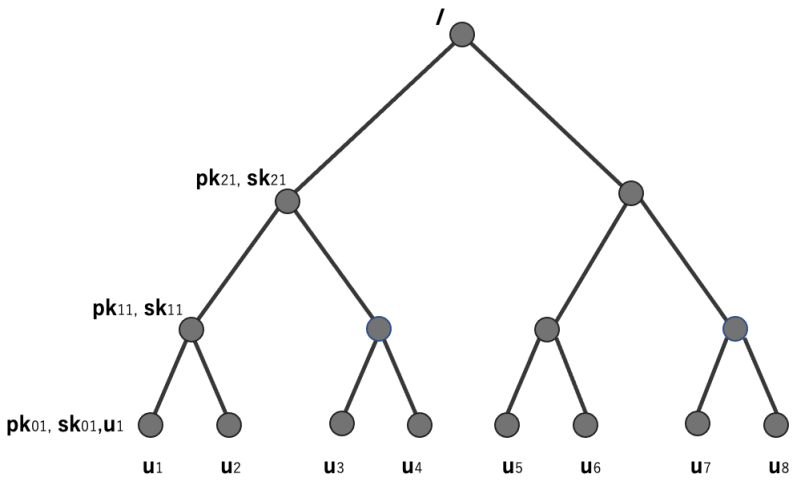

Figure 3.1: Left balanced binary tree

Direct-path. Direct-path is the path from a node v to the root.
Co-path. Co-path is the sequence of siblings of nodes on the direct-path.
Parent node. $\hat{v}$ is the parent of a node v.

Resolutions and representatives. Users use the public key of resolutions on the *co-path* when they compute the control messages for the subtree users, and use the secret key of the representative when they decrypt control messages. Intuitively, the resolution of a node v is the smallest set of non-blank nodes that covers all leaves in v's subtree.

Definition 9 (Resolution) *Let τ be a tree with node set V. The resolution* $\texttt{RES}(v)$ *of a node v is defined as follows.*

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

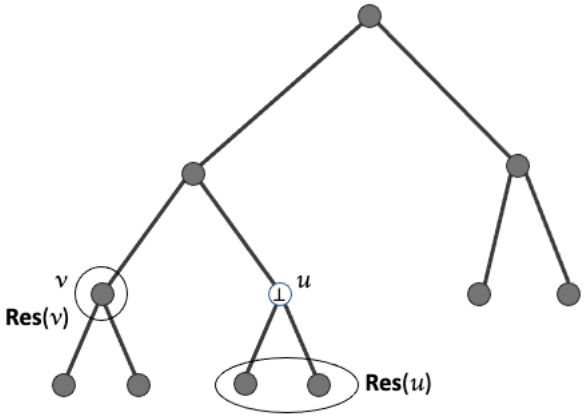

Figure 3.2: Resolution

Definition 10 (Representative) *Consider two leaf nodes l and l'.*

- *Assume l' is non-blank and in the subtree of v'. The representative* $\texttt{Rep}$ (v', l') *of l' in the subtree of v' is the first filled node on the path from v' down to l.*
- *Consider the least common ancestor $w = \texttt{LCA}(l, l')$ 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* $\texttt{Rep}$ (l, l') *of l' is defined as the representative* $\texttt{Rep}$ (v', l') *of l' in the subtree of v'.*

It is that $\texttt{Rep}\ (v', l') \in \texttt{RES}\ (v')$.

Subtree Users. We introduce the useful definition of Subtree users.

Definition 11 (Subtree Users) *Subtree Users of a node v and its sibling are as follows.*

- $\texttt{SU}$ (v) *are users arranged at leaves in the subtree of v.*
- $\texttt{SibSU}$ (v) *are users arranged at leaves in the subtree of v's sibling.*

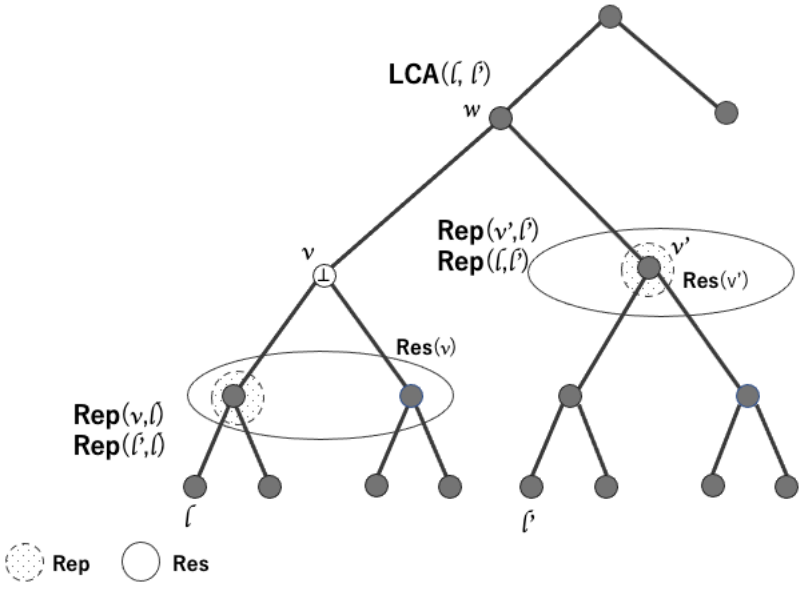

Figure 3.3: Representative

3.3 PKI

The TreeKEM protocol uses a public-key infrastructure (PKI) where parties can register ephemeral keys. The MLS documents [4, 13] explicitly describes how users can generate, authenticate, distribute, and verify each other's initialization keys. For simplicity as in [5], this work models the PKI as providing protocol algorithms and attackers can access to the PKI functionality. The PKI ensures that every public key is used only once.

- **get-pk**: Any user `ID` can request any user's fresh public key. When `ID` calls **get-pk** (`ID'`), the PKI generates a fresh pair of keys (`pk`, `sk`) and returns `pk` to user `ID`. The PKI also records the triple (`pk`, `sk`, `ID'`) and passes the information (`pk`, `ID'`) to the attacker.
- **get-sk**: Any user `ID'` can request secret keys corresponding to the public keys of itself. Specifically, when `ID'` calls **get-sk** (`pk`), if a triple (`pk`, `sk`, `ID'`) is recorded, the PKI returns `sk` to `ID'`.

3.4 TreeKEM with updatablePKE Protocol

In this section, we explain TreeKEM with updatablePKE protocol [5]. It makes use of a pseudorandom generator `prg` and a CPA-secure updatable

public-key encryption scheme UPKE.

Group creation. It creates a new ratchet tree with user lists $G = (\mathtt{ID}_1, ..., \mathtt{ID}_n)$. First, it creates a new pair of PKE keys ($\mathtt{pk}_0$,$\mathtt{sk}_0$) and fetches public keys $\mathbf{pk} = (\mathtt{pk}_1, ..., \mathtt{pk}_n)$, corresponding to the IDs in G from the PKI. Then, it makes the welcome message consists of G' and ($\mathtt{pk}_0$, **pk**).

Adding a group member. When a user ID adds a new group member ID′, it first calls **get-pk**(ID′) and obtains pk′ of ID′ from the PKI. The welcome message for user ID′ simply consists of a public keys of the current RT, and the control messages for the remaining group members consist of the IDs of ID and ID′, and the public key of ID′.

Removing a group member. When a user ID removes user ID′, it first blanks all the keys on the *direct-path* of ID′. Then, the leaf node of the user ID′ is removed from the tree. The control messages consist of the user IDs of ID and ID′.

Performing an update. When a user ID at leaf node v performs an update, it computes new pairs of PKE keys on their direct-path and path secrets as follows.

- *Compute path secrets*: Consider $v_0 = v$, and $v_1, ..., v_d$ be the nodes on the direct-path of the user ID, from bottom to top. The user ID chooses a uniformly random s_0 and it computes

$$\mathtt{sk}_i \parallel s_{i+1} \leftarrow \mathtt{prg}(s_i), \ \textit{for } i = 0, ..., d-1.$$

- *Update RT labels*: For $i = 0, ..., d-1$, the user ID gets $\mathtt{pk}_i$ corresponding $\mathtt{sk}_i$ from the PKI and updates the label of v_i to ($\mathtt{pk}_i$,$\mathtt{sk}_i$).

$$\mathtt{pk}_i \leftarrow \mathtt{PKEG}(\mathtt{sk}_i)$$

- *Root node*: The user ID sets $I := s_d$.

Then, the user ID makes the update messages as follows.

- *Encrypt path secrets and update public keys*: Consider $v'_0, ..., v'_{d-1}$ be the nodes on the co-path of the user ID from bottom to top. For $i = 1, ..., d$, the user ID encrypts the path secrets s_i with the public key of every resolution of its child that is on update *co-path*. The user ID computes $(c_{ij}, \mathtt{pk}_{ij}) \leftarrow \mathtt{Enc}(v'_j.\mathtt{pk}, s_i)$ and sets the public key of v'_j to $\mathtt{pk}_{ij}$.

- *Output the update message*: All ciphertexts c_{ij} are concatenated to an overall ciphertext $\mathbf{c}$ and all keys $\mathtt{pk}_{ij}$ are stored in $\overline{\mathbf{PK}}$. Then, the user $\mathtt{ID}$ outputs the update messages consist of $\mathbf{PK}$, $\overline{\mathbf{PK}}$, and $\mathbf{c}$, where $\mathbf{PK} := (\mathtt{pk}_0, ..., \mathtt{pk}_{d-1})$.

Processing control messages. When a user $\mathtt{ID}_j$ processes a control message T, it first checks whether T is the output from itself operation. If so, they simply adopt the corresponding RT in $\tau'[\cdot]$. When T was the output from another user, it calls $\mathtt{proc}$ and process it as follows, depending on the type of control message.

- $T = (\mathtt{create}, G, \mathtt{pk})$: A user $\mathtt{ID}_j$ determines its position, j, in the G list, and calls **get-sk**$(\mathtt{pk}_j)$. Then it initializes the RT.
- $W = (\mathtt{wel}, \tilde{\tau})$: It simply adopts $\tilde{\tau}$ as the current RT and calls **get-sk** and sets the secret key.
- $T = (\mathtt{add}, \mathtt{ID}, \mathtt{ID}', \mathtt{pk}')$: A user $\mathtt{ID}_j$ adds the new user $\mathtt{ID}'$ to the RT and blank all nodes on the direct-path of user $\mathtt{ID}'$.
- $T = (\mathtt{rem}, \mathtt{ID}, \mathtt{ID}')$: A user $\mathtt{ID}_j$ blanks all nodes on the direct-path of user $\mathtt{ID}'$ and removes the leaf node of $\mathtt{ID}'$ from the RT.
- $T = (\mathtt{upd}, \mathtt{ID}, U)$: When a user $\mathtt{ID}_j$ at some leaf l' receives an update message, issued by the user $\mathtt{ID}$ at leaf v, it processes the update information as follows. Consider $w := \mathtt{Rep}(v, l')$. The user $\mathtt{ID}_j$ uses $w.\mathtt{sk}$ to decrypt c_{ij} and obtains s_i and $\mathtt{sk}'_w$. Then it computes the path secrets from s_i to s_d and their secret keys. Finally, it overrides the RT labels by the keys in $\mathbf{PK}$, $\overline{\mathbf{PK}}$ and the secret keys including those of updatablePKE.

3.4.1 Security

TreeKEM with updatablePKE protocol is non-adaptive security with a safety predicate **safe**. The proof of Theorem 1 is given in [5].

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

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

then, TreeKEM with UpdatablePKE is a $(t,\ c,\ n,$ **safe**$,\ \varepsilon)$*-secure CGKA protocol,* $\varepsilon = 2cn\ (\varepsilon_{\mathrm{prg}} + \varepsilon_{\mathrm{cpa}})$*, and* $t \approx t_{\mathrm{prg}} \approx t_{\mathrm{cpa}}$*.*

Chapter 4

Node-by-node Compromise

In this chapter, we analyze the node-by-node compromise and introduce a modified security game that is better suited for TreeKEM. In this case, the attacker does not get update secrets, but once the compromised node is on the update co-path, it triggers a chain reaction, compromising other node keys and update secrets I. Conversely, if the node is not on the update co-path, the update secrets remain secure and the attacker can compute update secrets when it is on the update co-path in the future. In other words, there appears a dangerous node in the tree when node-by-node compromise. Therefore we introduce PCS for node-by-node compromise which means recovery from dangerous keys.

4.1 Capturing Node key leakage

In TreeKEM with updatablePKE, users derive a node key of v in epoch $t+1$ from the corresponding key in epoch t or the key remains unchanged. When the attacker leaks $v_j.\mathtt{sk}$, it can compute the update secrets I and the node keys of $v_{j+1}, v_{j+2}, ...v_{d-1}$, which is the node from v_j to root, with any update of $\mathtt{SibSU}(v_j)$ until $\mathtt{SU}$ (v_j) updates. Conversely, with the other users' update, update secrets remain secret. However, once $\mathtt{SibSU}(v_j)$ performs an update, the attacker can compute update secrets because $v_j.\mathtt{sk}$ remains unupdated.

To clarify this issue, we first illustrate which secret keys are necessary for users to compute node keys and update secrets. For example, consider the security of $\mathtt{LBBT}_4$ (Fig.3.1) under the assumption that the attacker leaks $\mathtt{sk}_{11}$ in epoch t. In Fig.4.1, we depict the operations and the corresponding secret keys after epoch t. Each arrow indicates that users derive the right-hand side using the left-hand side and the control message. Consequently, even though the attacker initially leaks only one node key in epoch t, it can

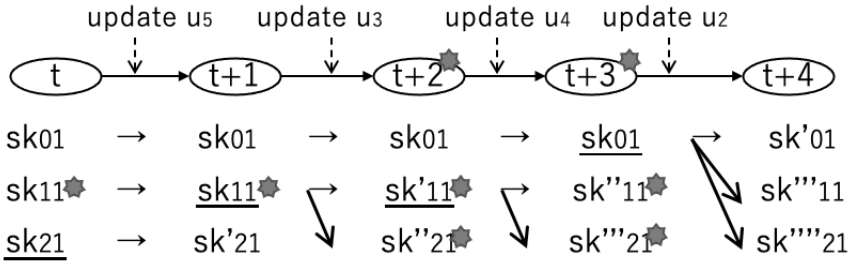

Figure 4.1: A node key compromise and subsequent leakage
Consider the attacker leaks $\mathtt{sk}_{11}$ in $\mathtt{LBBT}_4$ (Fig3.1) in epoch t.

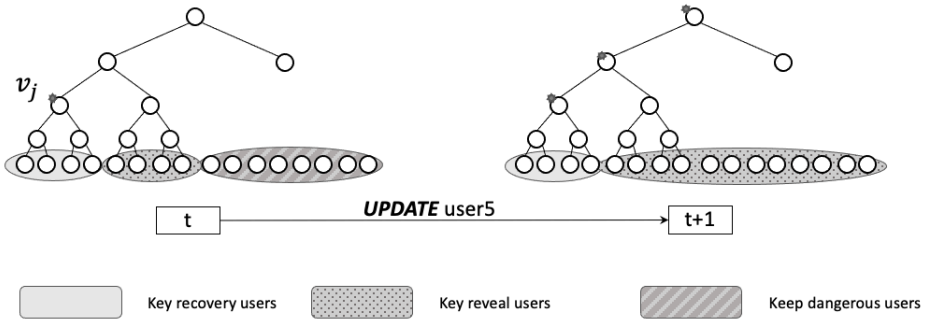

Figure 4.2: A node key compromise and three groups of users

compute additional node keys and update secrets in epoch $t+2$ and $t+3$.

Next, we illustrate how specific user updates influence the compromise of update secrets or the persistence of compromised nodes. When an attacker leaks a node key, users are categorized into three groups: users whose update ensures the secrecy of update secrets and the attacker needs another corruption to get update secrets, whose update leads to reveal update secrets, and whose update leave the node key unchanged and the attacker can derive the update secrets once the second group user updates. We refer to these groups as *Key recovery users*, *Key reveal users*, and *Keep dangerous users*, respectively. Specifically, when the attacker leaks $v_j.\mathtt{sk}$, *Key recovery users* are $\mathtt{SU}(v_j)$, while *Key reveal users* are $\mathtt{SibSU}(v_j)$ (Fig. 4.2).

Leakage of Multiple Node Keys. When the attacker leaks multiple node keys, it can compute update secrets with any update of `SibSU`. The compromised node heals when `SU` of itself performs an update. The tree heals as a whole when the users have updated all the keys of compromised nodes.

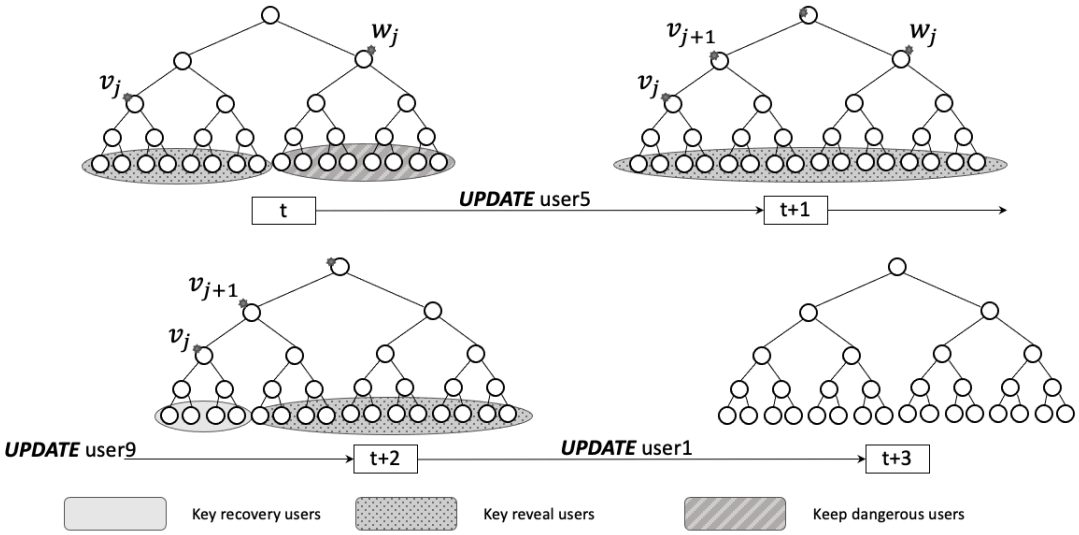

Figure 4.3: Two node keys compromise and three groups of users

However, if compromised nodes persist, the attacker can compute the update secrets again when `SibSU` of the node performs an update.

4.2 PCS for node-by-node compromise

In the CGKA game, **corrupt** means that the attacker obtains all the secrets the users have, including the update secrets. PCS ensures that after every group user whose state was leaked performs an update, the update secrets become secret again (Sect.2.5.2). However, in the context of the node-by-node compromise, which is more relevant for TreeKEM, we consider corruption as the leakage of a node key while the update secrets remain secure. In this case, corruption results in *Key reveal users* appearing in the tree instead of the direct exposure of update secrets. In other words, there is a dangerous key which allows the attacker to derive update secrets once *Key reveal user* performs an update. Therefore, we define PCS for nod-by-node compromise as follows.

Post-compromise security for node-by-node compromise. After any `SU` of every corrupted node performs an update, there are no dangerous keys in the tree.

Chapter 5

Our proposal

As seen above, in TreeKEM with updatablePKE, only the SU of corrupted user can perform an update and update dangerous keys. Without such recovery, the attacker can compute update secrets, or the tree retains dangerous keys. In this chapter, we propose a new protocol that allows recovery through any user updates except *Key reveal users*.

5.1 Add extra key

We introduce a common extra key, k, which is derived from a pseudorandom generator prg with s_d at the root as its input. In this new protocol, the attacker can compute the update secrets by combining the node key of v_j, the extra key, and an update of $\texttt{SibSU}(v_j)$ in the same epoch. However, since users update k in every epoch, the protocol ensures that the attacker cannot exploit the extra key across epochs.

5.2 Our protocol

We introduce a common extra key, k, for all users in a simple way. Whenever a user ID performs an update, it computes $k \leftarrow \texttt{prg}(s_d)$. Other operations are identical to those of TreeKEM with updatablePKE.

Performing an update. A user ID at leaf node v performs an update as follows.

- *Compute path secrets*: Consider $v_0 = v$, and $v_1, ..., v_d$ be the nodes on the direct-path of the user ID from bottom to top. Direct-path is the

path from a node v to the root (in Sect.3.2). The user $\texttt{ID}$ chooses a uniformly random s_0. Then it computes

$$\texttt{sk}_i \parallel s_{i+1} \leftarrow \texttt{prg}(s_i),\ \ for\ i = 0, ..., d$$

- *Update RT labels*: For $i = 0, ..., d-1$, the user $\texttt{ID}$ gets $\texttt{pk}_i$ from the PKI and updates the label of v_i to $(\texttt{pk}_i, \texttt{sk}_i)$.

$$\texttt{pk}_i \leftarrow \texttt{PKEG}(\texttt{sk}_i)$$

- *Root node*: The user $\texttt{ID}$ computes $I := s_{d+1}$.

Then, the user $\texttt{ID}$ sends the update messages as follows.

- *Encrypt path secrets and update public keys*: Consider $v'_0, ..., v'_{d-1}$ be the nodes on the co-path of the user $\texttt{ID}$ from bottom to top. Co-path is the sequence of siblings of nodes on the direct-path (in sect.3.2). For $i = 1, ..., d$, the user $\texttt{ID}$ encrypts path secrets s_i with the public key of every resolution of its child that is on update co-path. Resolution is defined at **Def.9**.

  For every value s_i and every node $v_j \in \texttt{RES}\ (v'_{i-1})$, the user $\texttt{ID}$ computes

$$(ct_{ij}, \texttt{pk}_{ij}) \leftarrow \mathsf{ENC}(v_j.\texttt{pk}, k, s_i; r_i),$$

$$v_j.\texttt{pk} \leftarrow \texttt{pk}_{ij}$$

- *Output the update message*: All ciphertexts ct_{ij} are concatenated to an overall ciphertext $\mathbf{c}$ and all keys $\texttt{pk}_{ij}$ are stored in $\overline{\mathbf{PK}}$. Then, the user $\texttt{ID}$ outputs the update message consists of $\mathbf{PK}$, $\overline{\mathbf{PK}}$, and $\mathbf{c}$, where $\mathbf{PK} := (\texttt{pk}_0, ..., \texttt{pk}_{d-1})$.

- *Update an extra key*: The user $\texttt{ID}$ replaces k with a new extra key.

$$k \leftarrow \texttt{sk}_d$$

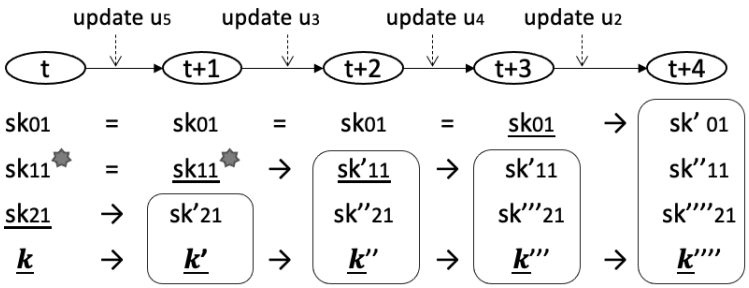

Figure 5.1: Our protocol: A node key compromise and subsequent leakage
Consider the case of $\texttt{LBBT}_4$ (Fig.3.1) under the assumption that the attacker leaks the node key of $\texttt{sk}_{11}$ in epoch t. The update secrets keep secret, even if the user3 performs an update in epoc $t+2$. This is because the update messages are encrypted both with the node key and the extra key. For the same reason and through the use of UPKE, $\texttt{sk}_{11}$ heals by user3's update in epoch $t+2$. In contrast, in TreeKEM with UpdatablePKE, the attacker can compute the update secrets in epoch $t+2$ (Fig.4.1) .

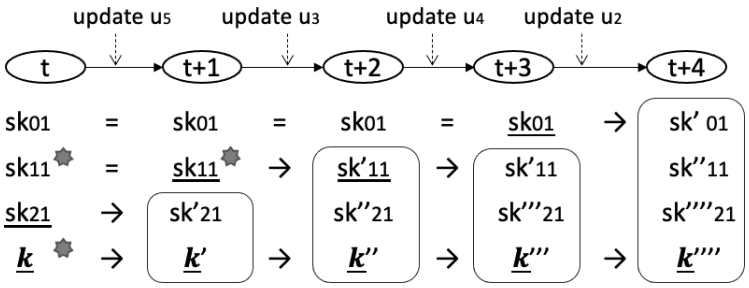

Figure 5.2: Our protocol: Compromise of a node key and an extra key
The extra key becomes secret again in epoc $t+1$ because user5 which is $\texttt{SibSU}(\hat{v})$ performs an update. The tree heals as a whole in epoch $t+2$ due to the use of UPKE.

5.3 Comparison

In our protocol, for the attacker to get update secrets it must possess the node key of v_j, the extra key, and an update of $\texttt{SibSU}(v_j)$ in the same epoch. In contrast, in previous protocols, the attacker could compute the update secrets only by obtaining the node key of v_j and an update of $\texttt{SibSU}(v_j)$ before an update of $\texttt{SU}(v_j)$. To highlight the improvements of our protocol, we present three comparative examples between the new and previous protocols. The definitions of SU, SibSU, and $\hat{v}$ are in Sect.3.2.

case1 When the attacker knows the node key of v_j and $\texttt{SibSU}(\hat{v_j})$ performs an update in epoch t:

- In TreeKEM with updatablePKE, $\texttt{SibSU}(\hat{v_j})$ is *Keep dangerous users* and *Key revel users* remain in epoch $t+1$, because the node key of v_j is not on the update co-path and unupdated.
- In our protocol, there are no *Key reveal users* in epoch t and $t+1$. Because an attacker needs a node key and an extra key to get update secrets.

case2 When the attacker knows the node key of v_j and $\texttt{SibSU}(v_j)$ performs an update in epoch t:

- In TreeKEM with UpdatablePKE, $\texttt{SibSU}(v_j)$ is *Key reveal users* and the attacker can compute the update secrets, because the one of the path secret, that is on the parent node of v_j, is encrypted with the node key of v_j.
- In our protocol, there are no *Key reveal users* in epoch t and $t+1$ for the same reason as case1. The difference from case1 is that the update $\texttt{SibSU}(v_j)$ leads no compromised nodes in the tree due to the use of updatablePKE. This provides a higher level of security compared to our scenario where no dangerous keys remain but a compromised key persists. This implies a much stronger than our PCS (PCS for node-by-node compromise); however, we leave a detailed exploration of this aspect for future work.

case3 When the attacker knows both the node key of v_j and an extra key, and $\texttt{SibSU}(\hat{v_j})$ performs an update in epoch t:

- In TreeKEM with updatablePKE, $\texttt{SibSU}(\hat{v_j})$ is *Keep dangerous users* and *Key reveal users* remain in epoch $t+1$ for the same reason as in case1.

- In our protocol, there are more *Key recovery users* in epoch t compared to TreeKEM with updatablePKE because $\texttt{SibSU}(\hat{v})$ is a part of them. Because the node key of v_j is not on the co-path but the extra key is updated, the update prevents the attacker from computing the new extra key and update secrets.

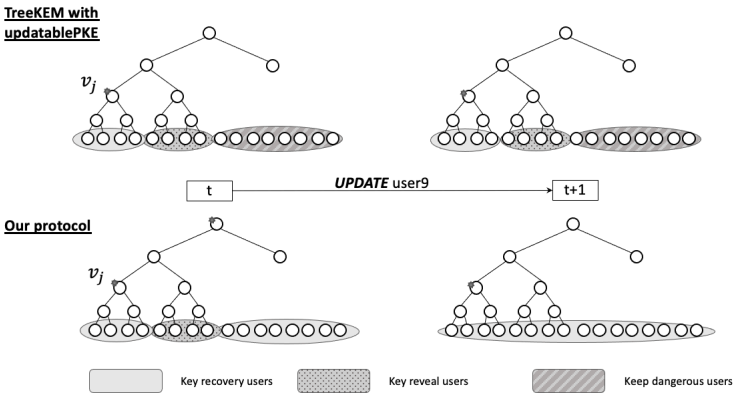

Figure 5.3: Comparison case3: Leakage of a node key and an extra key, and update of $\texttt{SibSU}(\hat{v})$

However, if there is leakage both of the node key and of the extra key, and $\texttt{SibSU}(v)$ performs an update in the same epoch, there is no difference between ours and previous protocols.

The first and second cases result from the use of two keys for encryption and decryption. The attacker cannot compute the update secrets only by obtaining a node key. The third case is attributed to the extra key, which users update in every epoch. Even if the the user does not update the node key via updatablePKE in epoch t, users consistently updates the extra key. Consequently, there are more *Key recovery users* in epoch t and no *Key reveal users* in epoch $t+1$ in the tree (Fig.5.3).

Chapter 6

Security

In our protocol, we introduce an extra key, which we use to encrypt and decrypt path secrets in addition to node keys. In this chapter, we explain this scheme, which we refer to as Updatable SKE with Separate Keys.

6.1 Secret-Key Encryption (SKE)

We first recall a secret-key encryption scheme $\mathsf{SKE} = (\mathbf{E}, \mathbf{D})$ as follows.

- Key Generation: Choose a security parameter κ and pick up a random secret-key $k \leftarrow \{0,1\}^\kappa$.
- An encryption algorithm $\mathbf{E}$ is an algorithm that takes a message $m \in \mathcal{M}$ along with secret key k and outputs a ciphertext $c \leftarrow \mathbf{E}(k, m)$ where $\mathcal{M}$ is a message space.
- A decryption algorithm $\mathbf{D}$ is a deterministic algorithm that takes a ciphertext c along with a secret key k and outputs $m = \mathbf{D}(k, c)$.
- A new secret key: $k' \leftarrow \{0,1\}^\kappa$.

SKE should satisfy the correctness condition as follows. For any sufficiently large κ and any message $m \in \mathcal{M}$,

$$\Pr[k \leftarrow \{0,1\}^\kappa; c \leftarrow \mathbf{E}(k, m) : \mathbf{D}(k, c) = m] = 1.$$

One-Time CPA security for SKE. We consider the following security game.

- Choose $k \leftarrow \{0,1\}^\kappa$.

- The adversary $\mathcal{A}$ chooses (m_0, m_1).
- Choose $b \leftarrow \{0,1\}$; compute $c \leftarrow \mathbf{E}(k, m_b)$, and feed it to $\mathcal{A}$.
- The adversary $\mathcal{A}$ finally outputs bit $b' \leftarrow \mathcal{A}(c)$.

We define that $\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ is defined by

$$\mathsf{Adv}^{\text{OT-CPA}}_{\mathcal{A},\mathsf{SKE}}(1^\kappa) := \left|\Pr[b = b'] - \frac{1}{2}\right|.$$

Definition 12 *A secret-key encryption algorithm* SKE *is* (t, ε)*-OT-CPA secure if for all* t*-time attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\text{OT-CPA}}_{\mathcal{A},\mathsf{SKE}}(1^\kappa) \leq \varepsilon.$$

6.2 Updatable SKE with separate keys (USKE)

We define an updatable secet-key encryption scheme with separate keys USKE = (KGen, ENC, DEC) as follows.

- A key generation algorithm KGen is a probabilistic algorithm that takes uniformly random secret keys, $\mathtt{sk}, k \in \{0,1\}^\kappa$, and outputs $(\mathtt{pk}, (\mathtt{sk}, k))$.
- An encryption algorithm ENC is a probabilistic algorithm that takes a message $m \in \mathcal{M}$ along with $(\mathtt{pk}, k)$ and outputs a ciphertext and a new partial public-key $(ct, \mathtt{pk}') \leftarrow \mathsf{ENC}(\mathtt{pk}, k, m)$ where $\mathcal{M}$ is a message space (possibly depending on κ).
- A decryption algorithm DEC takes a ciphertext ct along with $(\mathtt{sk}, k)$ and outputs m and an new partial secret-key $\mathtt{sk}'$.

USKE should satisfy the correctness condition. For any sufficiently large κ, all message $m \in \mathcal{M}$,

$$\Pr[\mathtt{sk}, k \in \{0,1\}^\kappa; (\mathtt{pk}, (\mathtt{sk}, k)) \leftarrow \mathsf{KGen}(\mathtt{sk}, k); \\ (ct, \mathtt{pk}') \leftarrow \mathsf{ENC}(\mathtt{pk}, k, m); (m', \mathtt{sk}') = \mathsf{DEC}(\mathtt{sk}, k, ct) : m = m'] = 1.$$

Leak One-Time CPA security for $\mathtt{USKE}$. We consider the following security game.

- Pick up $\mathtt{sk}_0, k_0 \leftarrow \{0,1\}^\kappa$, compute $(\mathtt{pk}_0, (\mathtt{sk}_0, k_0)) \leftarrow \mathsf{KGen}(\mathtt{sk}_0, k_0)$, and $b \leftarrow \{0,1\}$.
- The adversary $\mathcal{A}$ takes $\mathtt{pk}_0$ and for $i = 1, \ldots, q$, $\mathcal{A}$ outputs m_i and receives $(ct_i, \mathtt{pk}_i, r_i)$ such that $(ct_i, \mathtt{pk}_i) \leftarrow \mathsf{ENC}(\mathtt{pk}_{i-1}, k_{i-1}, m_i; r_i)$, for uniformly random (r_i, k_i).
- For $i = 1, \ldots, q$, compute $(m_i, \mathtt{sk}_i) = \mathsf{DEC}(\mathtt{sk}_{i-1}, k_{i-1}, ct_i)$.
- $\mathcal{A}$ requests either $\mathtt{sk}_i$ or k_i for each i. Let $\mathtt{leak}_i$ be $\mathtt{sk}_i$ or k_i that $\mathcal{A}$ has requested.
- $\mathcal{A}(\{\mathtt{pk}_i\}_{i\in\{0,[q]\}}, \{\mathtt{leak}_i\}_{i\in[q]})$ outputs (m_0^*, m_1^*).
- $(ct^*, \mathtt{pk}_{q+1}) \leftarrow \mathsf{ENC}(\mathtt{pk}_q, k_q, m_b^*; r_q)$ for uniformly random (r_q, k_q), $(\cdot, \mathtt{sk}_{q+1}) = \mathsf{DEC}(\mathtt{sk}_q, k_q, ct^*)$ and $k_{q+1} \leftarrow \{0,1\}^\kappa$.
- $\mathcal{A}$ requests either $\mathtt{sk}_{q+1}$ or k_{q+1}. Set $\mathtt{leak}_{q+1}$ to be $\mathtt{sk}_{q+1}$ or k_{q+1} that $\mathcal{A}$ has requested.
- $b' \leftarrow \mathcal{A}(\mathtt{pk}_q, \{\mathtt{leak}_i\}_{i\in[q+1]}, ct^*)$.

We define that $\mathcal{A}$ wins the game if $b = b'$. The advantage of $\mathcal{A}$ is defined by

$$\mathsf{Adv}^{\text{LK-OT-CPA}}_{\mathcal{A},\mathtt{USKE}}(1^\kappa) := \left|\Pr[b = b'] - \frac{1}{2}\right|.$$

Definition 13 *An updatable secret-key encryption algorithm with separate keys,* $\mathtt{USKE}$*, is* (t, ε)-LK-OT-CPA *secure if for all* t*-time attackers* $\mathcal{A}$*,*

$$\mathsf{Adv}^{LK\text{-}OT\text{-}CPA}_{\mathcal{A},\mathtt{USKE}}(1^\kappa) \leq \varepsilon.$$

Constructions of $\mathtt{USKE}$. If $\mathtt{UPKE}$ is IND-CPA secure and SKE is OT-CPA secure, then the following $\mathtt{USKE}$'s are LK-OT-CPA secure.

- (Construction I) Let $\mathtt{UPKE} = (\mathtt{PKEG}, \mathtt{Enc}, \mathtt{Dec})$ be an updatablePKE scheme and $\mathsf{SKE} = (\mathbf{E}, \mathbf{D})$ be a secret key encryption scheme with the common message space $\mathcal{M}(\kappa)$ for every security parameter κ.
- (Construction II) Let $\mathtt{UPKE} = (\mathtt{PKEG}, \mathtt{Enc}, \mathtt{Dec})$ be an updatablePKE scheme and $\mathsf{SKE} = (\mathbf{E}, \mathbf{D})$ be a secret key encryption scheme such that $\mathbf{E}(\mathcal{M}^{\text{SKE}}(\kappa)) \subset \mathcal{M}^{\text{UPKE}}(\kappa)$ where $\mathcal{M}^{\text{SKE}}$ and $\mathcal{M}^{\text{UPKE}}$ denote the message spaces of SKE and $\mathtt{UPKE}$, respectively.

6.3 Improved PCS for node-by-node compromise

As in Sections 5.2 and 5.3, in our protocol, there are more *Key recovery users*, whose update ensures the security of update secrets and prohibits the attacker from learning update secrets without another corruption. This means that no dangerous keys remain in the tree. Dangerous keys are users that allow the attacker to derive update secrets once *Key reveal user* performs an update. We define improved PCS as follows.

Improved Post-compromise Security for node-by-node compromise. After any user's update except for `SibSU` of compromised node, there is no dangerous keys in the tree.

Protocol	Conditions for Reveal Update Secretes	PCS: Heal from dangerous keys	PCS: Heal from compromised nodes (higher level of security)
TreeKEM with updatablePKE (original CGKA game)	• **corrupt** (ID)	-	-
TreeKEM with updatablePKE (node-by-node compromise)	• **corrupt node** (v) • SibSU(v)	• SU(v) performs an update. <PCS for node-by-node compromise>	• SU(v) performs an update.
Our protocol (node-by-node compromise)	• **corrupt node** (v) • **corrupt root** • SibSU (v) performs an update. The node key, the extra key, and SibSU (v')'s update are in the same epoch.	• Any user performs an update except for SibSU (v). <Improved PCS for node-by-node compromise>	• SU(v) performs an update. • Only node keys compromise: SU($\hat{v}$) performs an update. • Node keys and an extra key compromise: SibSU($\hat{v}$) performs an update and SU($\hat{v}$)

Figure 6.1: Conditions for reveal the update secrets and PCS

6.4 Modified CGKA game

We propose a modification to the CGKA security game and improved PCS to better suit TreeKEM protocols. Specifically, we introduce two new oracles, **corrupt node** and **corrupt root**, which replace the existing oracle **corrupt** [5].

Corruption of Node and Root. The attacker is allowed to learn the current state of a node and the root by calling the oracles **corrupt node** and **corrupt root**, respectively. These oracles return the secret key of the node

and the extra key.

Avoiding Trivial Attacks and Introducing Improved PCS Predicate. To ensure that the attacker may not win the modified CGKA security game with trivial attacks, we propose a modified predicate. As seen above, for the attacker to compute the update secrets, it must obtain a node key, an extra key, and an update of $\texttt{SibSU}(v)$ in the same epoch. The specific case is as follows:

- $\texttt{SibSU}(v)$ performs an update in the same epoch in which the attacker leaks both the node key of v and the extra key, and $\texttt{SU}(v)$ has not performed an update, nor have any users been removed before the challenge epoch.

improved pcs predicate

improved pcs predicate $(\mathbf{q_1}, ..., \mathbf{q_q})$
for (i, j)
$s.t.$ $\mathbf{q_i} =$ **corrupt node**(v)
for some node v and q2e $(\mathbf{q_i}) = \tilde{t}$,
$\mathbf{q_j}$=**chall**(t^*)

if $\tilde{t} < t^*$, $\nexists k, \exists l,$ and $\exists m$
$s.t.$
$\tilde{t} \leq$ q2e $(\mathbf{q_k}) <$ q2e $(\mathbf{q_l}) < t^*$
$\mathbf{q_k} =$ **send-update** $(\texttt{SU}(\hat{v}))$
$\mathbf{q_l} =$ **corrupt root**
$\mathbf{q_m} =$ **send-update** $(\texttt{SibSU}(v))$
q2e $(\mathbf{q_l}) =$ q2e $(\mathbf{q_m}) = \hat{t}$

if $\nexists n$
$s.t.$
$\hat{t} \leq$ q2e$(\mathbf{q_n}) < t^*$
$\mathbf{q_n} \in \{$**send-update**$(\texttt{SU}(v)),$ **remove-user**$(*, \texttt{SU}(v))\}$
return 0

return 1

return $\perp$

Our protocol is non-adaptive security with a safety predicate **improved pcs predicate**. As the CGKA game, for modified CGKA game the advantage of $\mathcal{A}$ is defined by

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

In this paper, we do not consider a modified predicate for FS. Addressing FS with the modified predicate will be a topic for future work. However, it is secure as TreeKEM with updatablePKE because of the property of modification.

6.5 Proof of Security for Our protocol

This section presents the security result for our protocol and provides a high-level intuition for the security proof.

Theorem 2 ***Non-adaptive security of Our protocol for PCS*** *Assume that*

- `prg` *is a* (t_{prg}, ε_{prg})*-secure pseudo-random generator,*
- `USKE` *is a* (t_{cpa}, $\varepsilon_{\text{LK-cpa}}$)*-LK-OT-CPA-secure updatable secret-key encryption with separate keys scheme,*

then, our protocol is a (t, c, n, $\mathbf{P}$, ε)*-secure protocol, for* $\mathbf{P}$ = **improved pcs predicate**, $\varepsilon = 2cn(\varepsilon_{\text{prg}} + \varepsilon_{\text{LK-cpa}})$, *and* $t \approx t_{\text{prg}} \approx t_{\text{LK-cpa}}$.

We prove this by hybrid games w.r.t. **improved pcs predicate** and by considering an attacker $\mathcal{A}$ that makes only a single challenge query. The final result is obtained by applying Lemma 1 **Single to multi** (Sect.2.6.).

Before the explanation of games we prove a lemma. Let τ^* be the ratchet tree at challenge epoch t^*.

Lemma 2 *Let* $\mathtt{sk} \in \tau^*$ *and* $\mathtt{sk}' \notin \tau^*$, *the attacker does not learn* `sk` *along with* $\mathtt{sk}'$.

Proof. We prove this by contradiction. First, let v is the node whose secret key is $\mathtt{sk} \in \tau^*$ and v' is the node whose secret key is $\mathtt{sk}' \notin \tau^*$. Assume that $\mathtt{sk} \in \tau^*$ and $\mathtt{sk} \notin \tau^*$, but the attacker learns `sk` along with $\mathtt{sk}'$. It implies that v' is in the subtree of v because of RT property and that the attacker would be able learn the update secret in another epoch using a path secret that is generated from PRG with an input identical to that of sk. Since

we check the predicate **P** at the end of the game and the value is true, there is either **send-update**(SU) or **remove-user**($*$, SU) before challenge epoch in that case. It contradicts that $\mathtt{sk}^* \in \tau^*$.

In the following we explain hybrid games. We focus on the case where the attacker obtains some node keys in τ^*. The adversary is not allowed to leak node keys and an extra key that enables the recovery of the update secret in the challenge epoch, because the predicate **P** is true. The proof proceeds in a series of hybrids that replace PRG outputs with random values and fake ciphertexts in a bottom-up fashion as in [5]. The difference is that we use the security of LK-OT-CPA-secure USKE and we do not replace PRG output which corresponds to the keys that the attacker learns. The other case (i.e., only the extra key in t^* is leaked or no keys leaked) are also proved as same but we replace all PRG outputs. Recall that when a user at a leaf of depth d performs an an update, it generate the following values for a uniformly random s_0:

$$s_0 \xrightarrow{\mathtt{prg}} (\mathtt{sk}_0, s_1) \xrightarrow{\mathtt{prg}} (\mathtt{sk}_1, s_2) \xrightarrow{\mathtt{prg}} ... \xrightarrow{\mathtt{prg}} (\mathtt{sk}_d, s_{d+1})$$

where $k = \mathtt{sk}_d$ is the new extra key and $I = s_{d+1}$ is the update secret. To use the CPA security of keys in τ^* we argue that the attacker obtains no information about update secrets. In this case, we assume the attacker obtains node keys or an extra key in each epoch. Therefore, the proof proceeds in a series of hybrids that fake ciphertexts and replace PRG outputs with random values in a bottom-up fashion. Using the example shown in Fig.3.1, the hybrids can be constructed as follows.

- $H_d^{\mathtt{c}}$: This game is of the original CGKA experiment.
- $H_d^{\mathtt{p}}$: When a user updates, the output of the first PRG is replaced with a uniformly random value except that the attacker knows the node key. That is, instead of computing $(\mathtt{sk}_0, s_1) \leftarrow \mathtt{prg}(s_0)$, $\mathtt{sk}_0$ and s_1 are simply chosen randomly. The rest of the update is computed normally. The adversary cannot computationally distinguish this game from the previous one, thanks to the security of PRG.
- $H_{d-1}^{\mathtt{c}}$: Instead of encrypting s_1 along with the secret key on the resolution of the co-path nodes and the extra key, the all-zero string is encrypted. The adversary cannot computationally distinguish this game from the previous one, thanks to the security of USKE.
- $H_{d-1}^{\mathtt{p}}$: This game is similar to $H_d^{\mathtt{p}}$. The output of the PRG computation at depth $d-1$ is replaced with uniformly random values except that

the attacker knows the node key. That is, instead of applying the PRG, the values ($\mathtt{sk}_1$,s_2) are chosen randomly. The adversary cannot computationally distinguish this game from the previous game, thanks to the security of PRG.

- $H^{\mathtt{c}}_{d-2}$: This game is similar to $H^{\mathtt{c}}_{d-1}$. Instead of encrypting s_2 along with the secret keys on the resolutions if the co-path nodes, the all-zero string is encrypted. The adversary cannot computationally distinguish this game from the previous game, thanks to the security of USKE.
- $H^{\mathtt{p}}_{d-2}$: Similarly to $H^{\mathtt{p}}_{d-1}$, the values ($\mathtt{sk}_2$, s_3) are chosen randomly.
- $H^{\mathtt{c}}_{d-3}$: The encryption of s_d is replaced by a dummy encryption.
- $H^{\mathtt{p}}_{d-3}$: Similarly to $H^{\mathtt{p}}_{d-1}$, the values (k, s_4) are chosen randomly.

Observe that in $H^{\mathtt{p}}_{d-3}$, there are some compromised node keys and an extra key, but the adversary is not provided with any information about I in the challenge epoch and its advantage in the final hybrids is 0. The difference between H^c_i and H^p_i is less than $2^i \cdot \varepsilon_{\mathrm{prg}}$ and also that between H^c_{i-1} and H^p_i is less than $2^i \cdot \varepsilon_{\mathrm{cpa}}$. Hence the advantage of the original game is less than ε.

Bibliography

[1] https://signal.org/docs/

[2] https://about.fb.com/news/2023/12/default-end-to-end-encryption-on-messenger/

[3] https://www.ietf.org/blog/support-for-mls-2023/

[4] R. Barnes, B. Beurdouche,J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert The messaging layer security (mls) protocol (2023)

[5] Joel Alwen, Sandro Coretti, Yevgeniy Dodis, Yiannis Tselekounis. Security Analysis and Improvements for the IETF MLS Standard for Group Messaging(2019)

[6] K. Bhargavan, R. Barnes, and E. Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. (2018)

[7] Joel Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein,Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, Michelle Yeo. Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement (2019)

[8] Joel Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual- Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable continuous group key agreement with fast healing (2022)

[9] Joel Alwen, Marta Mularczyk, Yiannis Tselekounis. Fork-Resilient Continuous Group Key Agreement (2023)

[10] Celine Chevalier, Guirec Lebrun, Ange Martinelli. Quarantined-TreeKEM: a Continuous Group Key Agreement for MLS, Secure In Presence of Inactive Users (2023)

[11] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the Price of Concurrency in Group Ratcheting Protocols (2020)

[12] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincint Rigimen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of LNCS, pages 159-188.Springer, Heidelberg, May 2019.

[13] B.Beurdouche, E.Rescorla, E.Omara, S.Inguva, A.Duric The Messaging Layer Security (MLS) Architecture (2024)