Modular Design of Secure Group Messaging Protocols and the Security of MLS

Joël Alwen
Wickr
jalwen@wickr.com

Sandro Coretti
IOHK
sandro.coretti@iohk.io

Yevgeniy Dodis*
New York University
dodis@cs.nyu.edu

Yiannis Tselekounis†
University of Edinburgh
ytselekounis@ed.ac.uk

August 23, 2021

Abstract

The Messaging Layer Security (MLS) project is an IETF effort aiming to establish an industry-wide standard for *secure group messaging (SGM)*. Its development is supported by several major secure-messaging providers (with a combined user base in the billions) and a growing body of academic research.

MLS has evolved over many iterations to become a complex, non-trivial, yet relatively ad-hoc cryptographic protocol. In an effort to tame its complexity and build confidence in its security, past analyses of MLS have restricted themselves to sub-protocols of MLS—most prominently a type of sub-protocol embodying so-called *continuous group key agreement (CGKA)*. However, to date the task of proving or even defining the security of the full MLS protocol has been left open.

In this work, we fill in this missing piece. First, we formally capture the security of SGM protocols by defining a corresponding security game, which is parametrized by a safety predicate that characterizes the exact level of security achieved by a construction. Then, we cast MLS as an SGM protocol, showing how to modularly build it from the following three main components (and some additional standard cryptographic primitives) in a black-box fashion: (a) CGKA, (b) *forward-secure group AEAD (FS-GAEAD)*, which is a new primitive and roughly corresponds to an "epoch" of group messaging, and (c) a so-called *PRF-PRNG*, which is a two-input hash function that is a pseudorandom function (resp. generator with input) in its first (resp. second) input. Crucially, the security predicate for the SGM security of MLS can be expressed purely as a function of the security predicates of the underlying primitives, which allows to swap out any of the components and immediately obtain a security statement for the resulting SGM construction. Furthermore, we provide instantiations of all component primitives, in particular of CGKA with MLS's TreeKEM sub-protocol (which we prove adaptively secure) and of FS-GAEAD with a novel construction (which has already been adopted by MLS).

Along the way we introduce a collection of new techniques, primitives, and results with applications to other SGM protocols and beyond. For example, we extend the Generalized Selective Decryption proof technique (which is central in CGKA literature) and prove adaptive security for another (practical) *more secure* CGKA protocol called RTreeKEM (Alwen et al., CRYPTO '20). The modularity of our approach immediately yields a corollary characterizing the security of an SGM construction using RTreeKEM.

*Partially supported by gifts from VMware Labs and Google, and NSF grants 1619158, 1319051, 1314568.

†Partially supported by the "Quantum Computing for Modern Cryptography" project, funded by the UK Quantum Computing and Simulation Hub from the UKRI EPSRC grant EP/T001062/1.

Contents

1 Introduction

End-to-end encrypted asynchronous *secure group messaging (SGM)* is becoming one of the most widely used cryptographic applications with billions of daily users. To help solidify the foundations of this trend, the IETF is in the final stages of standardizing the *Messaging Layer Security* (MLS) SGM protocol [7]. The effort is being lead by industry (e.g. by Cloudflare, Cisco, Facebook, Google, Mozilla, Twitter, Wickr and Wire) with a lot of help from academia. Thus, MLS holds the potential to be widely deployed with in the coming years.

Due to the expected feature set and the multi-faceted security goals, MLS is quite complex. Furthermore, the description of MLS is quite monolithic. In an effort to come to terms with the complexity (both of the construction and desired security notions) all work formally analyzing MLS has restricted itself in scope. Besides simplifying the PKI [1, 3] the trend has been to focus only on a subset of the complete protocol. In particular, the majority of work has considered so-called *continuous group key agreement (CGKA)* [1, 3, 5]. First introduced by Cohn-Gordon et al. [12] (under the name "Group Ratcheting"), a CGKA protocol is often motivated by the folklore belief that CGKA encapsulates the essence of building SGM (not unlike how KEMs capture the essence of PKE). Indeed, this folklore has motivated a host of new CGKA protocols designed to improve on the CGKA protocol implicit to MLS. Some constructions aim for stronger security properties [3, 4], others for a more flexible communication model [21, 9], while the construction in [1] is optimized for certain common SGM settings.

CGKA abstracts the basic task of asynchronously maintaining secrets in a group with an evolving set of members. To this end, each time a member joins, leaves or *updates* their cryptographic state (in an effort to heal from past compromises and defend against future ones) a new *epoch* in the ongoing session is initiated. Each epoch is equipped with a symmetric group key that can be derived by all parties that are in the group during that epoch. The CGKA most often identified as underpinning MLS is a protocol called TreeKEM [24, 6]. Although, it has undergone at least 2 major iterations before reaching its current version [7] it remains, at most, a passively secure protocol, i.e., with no authenticity mechanisms included. Moreover, it provides weaker guarantees (for the group key) than is expected of the (encryption keys) in the full MLS protocol. Thus, there are clearly further significant security mechanisms at work in MLS beyond the underlying passively secure CGKA, i.e., the protocol requires additional components.

Indeed, [5] extends the abstraction boundary around TreeKEM including more of the MLS protocol to obtain an *actively* secure CGKA which they dub ITK (for ”Insider Secure TreeKEM”). Yet even ITK is only a CGKA and not an SGM protocol, and so here too there clearly remains a gulf between the full MLS protocol and what has actually been formally analyzed in [5].

Two exceptions to the focus on CGKA are: [14], which focuses exclusively on how to optimize the bandwidth required to updated cryptographic state concurrent sessions with overlapping membership sets, and [10], which considers the key derivation paths within MLS. While the paths extend beyond the CGKA (even ITK) into parts of MLS specific to messaging (e.g., key/nonce derivation for message encryption) the work leaves much open when it comes to a holistic understanding of MLS's security. For example, the work makes no statements at all about authenticity. So for example, it leaves out the relatively involved and interdependent mechanisms used by MLS to provide this property. Moreover, the work forgoes any formal modeling of network communication, leaving open the questions around how a network adversary can generally impact the security of a session by, say, arbitrarily manipulating packets in transit.

Our contributions

From a practice-oriented view point, the primary contribution in this work is a rigorous security proof showing that the basic instantiation of MLS is an SGM protocol. More generally, we validate the folklore claim that CGKA embodies the essence of SGM. In doing so, we make progress on several fronts in coming to terms with the complexity involved in describing MLS (and other SGM protocols) as well as defining (and proving) security for such protocols.

Black-Box Construction. In more detail, we give a black-box construction of an SGM protocol from a passively secure CGKA and several other primitives. Instantiated appropriately, the construction recovers the basic SGM protocol in MLS.[1] Besides passive CGKA, the construction also uses a new primitive called a *Forward Secure Group AEAD* (FS-GAEAD). Intuitively, it combines a forward secure key schedule with an *authenticated encryption with associated data* (AEAD) scheme, to allow a group of sender's sharing a single secret to send any number of AEAD encrypted messages with no further synchronization or communication such that any receiver can immediately decrypt any ciphertexts they receive regardless of the order in which they are delivered. We further abstract the key schedule of an FS-GAEAD with a new primitive called a *Forward Secure KDF* (FS-KDF) which captures a generic forward-secure symmetric key schedule supporting the derivation of keys in any order. Due to its improved efficiency profile, our construction of an FS-KDF has since replaced the original FS-KDF in the MLS standard.

History graphs. We introduce the first formal security notion for SGM.[2] Intuitively, it ensures correctness as well as privacy and authenticity in the form of *post-compromise forward secrecy*

[1]That is, the part of MLS that allows parties to manage group membership, update their cryptographic states at will and send/recieve encrypted and authenticated messages. We do not analyze more advanced features such as importing/exporting secrets, meta-data hiding and so called ”external commits” where an external party makes changes to the group state.

[2]We note that, concurrently to our work, [8] introduced a security notion for SGM defined using formal verification tools which we discuss in the Related Work section below.

(PCFS) [3]. That is, messages in a given epoch are both private and authenticated despite arbitrary state leakage of participants sufficiently in the past and future.

In effort to manage the complexity inherent in such a definition we developed the *history graph* (HG) paradigm for defining security of asynchronous group protocols like MLS and CGKA protocols. The paradigm allows for an intuitive (even visual) understanding of an otherwise complicated notion by representing the (security-relevant) semantics of an execution as an annotated graph. This allows cleanly separating functionality and communication model from the security details being captured. Those details are formalized as a predicate defined over HG and a particular challenge message (or group key for CGKA). The output of the predicate indicates if we can expect the challenge to be secure given the execution represented by the HG. This modular approach makes our SGM and CGKA definitions easy to adapt to, say, future versions of MLS. It also allows for more immediate comparison of the security enjoyed by different constructions of the same type.

PKI. Another contribution to the analysis of MLS is a more accurate modeling of the PKI used by MLS. With the notable exceptions of [14, 5]), the PKI in the works of [1, 3] the analysis of (earlier versions of) TreeKEM is based on simplified PKI models that encode strong (but implicit) assumptions. For example, their PKI precludes the adversary from registering ephemeral keys even for recently corrupted parties. Instantiating this faithfully seems to require strong authenticity checks by the PKI that may not rely on any state that can be leaked during a compromise. In contrast, the PKI in our work can be realized under assumptions more compatible with E2E security. Concretely, our PKI can be realized if parties use an out-of-band mechanism to authenticate each other's long-term keys while distributing ephemeral public keys via an *untrusted* key server. Reflecting the MLS standard, the out-of-band authentication mechanism is not made concrete but common instantiations include peer-to-peer key verification via 2D bracodes or (in the enterprise setting) certification authorities binding long term keys to a users identity.

Security Proofs. Armed with our HG based security notions for SGM and passively secure CGKA we prove security for the black-box SGM protocol there by making progress not just in validating the design of MLS but also the more general folklore motivating the study of CGKA protocols. At a high level the proof follows the outline of proof for the double-ratchet in [2]. However the similarities end there as we require new arguments and hybrids to instantiate this outline in light of the substantial differences between the two protocols and 1-on-1 vs. group settings they operate in.

To recover MLS security we also show that TreeKEM (as implicit to the current version in Draft 11 of MLS) [7] is indeed a passively secure CGKA according to our definition. In particular, in comparison to the TreeKEM analysis in [1], we more accurately model the PKI, capture transcript consistency and secure group management. Meanwhile, in comparison to the analysis in [3] we improve the PKI model and allow for a much more capable network adversary that can, e.g. deliver packets in different orders to different participants; a potentially unrealistic restriction for real world adversaries that controls, say, the network connection of a user (let alone the delivery server used by the session).

Beyond MLS and TreeKEM we also prove that RTreeKEM [3] is a passively secure CGKA. For this we extend the *generalized selective decryption (GSD)* technique to account for encrypting keys using the so-called *updatable public-key encryption (UPKE)* [3], used by RTreeKEM. GSD lies at the heart of almost all adaptive security proofs for CGKA protocols. To the best of our knowledge, it remains the only technique for proving adaptive security with meaningful security loss (e.g. quadratic in the group size). In contrast, the proof of the ART CGKA protocol in [12] suffers from an exponential loss in group size. Extending GSD to allow for protocols using UPKE is likely to have applications beyond analyzing RTreeKEM as UPKE is a very natural drop in replacement for standard PKE but with improved forward security [19]. Indeed, recent results in [16] construct post-quantum secure UPKE with an aim towards building a PQ variant of MLS.

Related work.

The history graph paradigm was conceived early on in this project. In an effort to keep pace with the rapid development of MLS the paradigm was shared with other researchers before this work was published. Consequently, the paradigm has been used in subsequent work appearing before this one (that cite this one as the source of the paradigm) [4, 5].[3] Symbolic representations of an execution, equipped with safety predicates, have been used in [13, 2]. The GSD proof technique, introduced by Panjwani [23] and generalized by Jafargholi et al. [18], was first applied to CGKA protocols in [1].

The most influential precursor to TreeKEM, the asynchronous ratchet tree (ART) protocol, was introduced by Cohn-Gordon *et al.* [12], focusing on adaptive security (informally sketched) for static groups. It inspired the initial version of TreeKEM [24] which was updated to provide secure group management in [6] and finally to its current form in [7]. Meanwhile, ART uses an older technique called "Tree-based DH groups" [25, 26, 28] which is also used by [20] to build key agreement. However, TreeKEM and ART differ significantly from [20], as discussed in [3]. TreeKEM is also related to schemes for (symmetric-key) broadcast encryption [17, 15] and multicast encryption [22, 28, 11]. Cremers *et al.* [14] note MLS/TreeKEM's disadvantages w.r.t. PCS for multiple groups, and Weider [27] suggests Causal TreeKEM, a variant that requires less ordering of protocol messages. Finally, in concurrent work appearing recently, [8] analyzed an older version of TreeKEM (from MLS Draft 7) using formal verification techniques.

2 Preliminaries

Notation. We use associative arrays which map arbitrary key strings to item strings. For array A we use "." to denote all entries. In particular, we (implicitly) declare variable A to be a (1-dimensional) array and set all of its entries empty string by writing $A[.] \leftarrow \varepsilon$. Similarly, we declare a new 2-dimensional array B with empty entries by $B[.,.] \leftarrow \varepsilon$. We use the shorthand $B[x,.] \leftarrow A$ to denote that $\forall y$ with $A[y] \neq \varepsilon$ we set $B[x,y] \leftarrow A[y]$. For subset $Y' \subseteq Y$ the term $Y' \subseteq A$ returns true iff A contains all elements in Y'; that is $\forall y \in Y' \; \exists x : A[x] = y$. For vector $\mathbf{x} = (x_1, \ldots, x_d)$ with all components $x_i \in X$ we write $A[\mathbf{x}]$ to denote the vector $(A[x_1], \ldots, A[x_d])$. We denote the empty list with $[.]$ and the length of a list L by $|L|$. The following special keywords are used to simplify the exposition of the security games: **req** is followed by a condition; if the condition is not satisfied, the oracle/procedure containing the keyword is exited and all actions by it are undone. **let** is followed by a variable and a condition. After evaluating the expression the variable is assigned with the value that satisfies the condition, if such value exists, and $\perp$ otherwise. **chk** is followed by a condition; if the condition is not satisfied, the oracle/procedure containing the keyword returns false, otherwise the next instruction is executed.

3 Secure Group Messaging

In this section we formally define *secure group messaging* (SGM). In more detail, in Section 3.1 we present a high-level overview of how SGM schemes work and what type of security one expects from

[3]While not ideal, we felt this approach was necessary given the strong incentive to make progress on MLS's analysis prior to it being to widely deployed.

them. Then, the formal syntax of SGM schemes is introduced in Section 3.2. Finally, and most crucially, Section 3.3 introduces the security game for SGM schemes.

3.1 Overview

Setting, PKI, Propose-and-commit. SGM schemes are run by parties wishing to engage in secure *group* communication, i.e., each transmitted message is sent to all group members. They are run over an asynchronous network, where, in the SGM context, "asynchronous" means that parties are not constantly online and, when they are, not necessarily at the same time. The communication between parties is assumed to be handled by an *untrusted* third party, the *delivery service* (DS), which is supposed to buffer protocol messages for offline parties until they come back online. In addition to the DS, there is a PKI (service) that parties use to exchange initial key material. The PKI is *untrusted* as well, but it is assumed that there are some kind of incorruptible long-term secrets that can be used to bind signature public keys to particular identities. These "bound" signature keys can then be used to authenticate the initial key material sent to the PKI.[4]

SGM schemes in this work adopt the so-called *propose-and-commit (P&C)* paradigm. That is, whenever a party wishes to change the group state (in particular, to add or remove group members or to update the internal key material), it issues a corresponding proposal. In arbitrary intervals, any party may subsequently collect a set of proposals and commit to them, which is when the group state is changed to reflect the proposals.

State. For security reasons, specifically, to limit the damage of state exposures, group members do not usually know all the secret values of the "global" group state. Each party only knows their own slice of the global state, where slices can intersect. In addition, each party stores some *personal* secret values only known to themselves.

Components of SGM schemes. The main components of an SGM scheme are the following:

- *Initialization:* An SGM's initialization algorithm sets up a party's state with basic values and parameters.
- *Interaction with the PKI:* An SGM scheme provides several algorithms to interact with the PKI; to create and register (with the PKI) as well as to remove signature public keys, to create and register initial key material, and to store signatures and initial key material of other parties.
- *Group creation:* A party may create a new group with only itself in it, using the group-creation algorithm.
- *Issue proposals:* A group member may issue three types of proposals: to *add* a new group member, *remove* a current group member, and *update* personal secrets. New proposals are delivered to all group members, and each party buffers them locally in order to be able to either create or process a *commit*.
- *Commit creation:* Occasionally, a party may pick an arbitrary (ordered) subset of the buffered proposals and create a commit. A commit produces a *commit message* that is to be sent to all group members for processing; if new group members are added as part of one of the committed proposals, the commit process also outputs a corresponding *welcome message* used by the new members to join. Each commit operation starts a new so-called *epoch*.

[4]One way to bind signature keys to identities would be to have them certified by a certificate authority.

- *Sending and receiving:* Finally, an SGM scheme provides algorithms for sending and receiving messages. The send algorithm takes a *plaintext* as well as *associated data (AD)* and outputs a *ciphertext* that encrypts the plaintext and authenticates both the plaintext and the AD. The receive algorithm takes as input a ciphertext and AD, and outputs a plaintext (or an error message if authentication fails).

In the remainder of this paper, proposals, commit messages, and welcome messages are referred to as *control messages*, and ciphertexts as *application messages*. Besides sending and receiving application messages, an SGM scheme provides algorithms for processing control messages.

Security and updates. SGM schemes are expected to protect the confidentiality and authenticity of sent messages. Furthermore, they must provide so-called *post-compromise forward secrecy (PCFS)*, which combines *forward-secrecy (FS)*, i.e., delivered messages remain secure even upon future state compromise, with *post-compromise security (PCS)*, i.e., the protocol recovers from state compromise as soon as all compromised group members *heal*.[5] There are two ways in which a party may heal: either by only updating their personal secrets or by updating their entire slice of the global state. The former is achieved via update proposals, whereas the latter occurs whenever a party creates a commit. It is important to note that both ways of updating key material constitute a heal, i.e., both ways must contribute to PCS. Generally, the difference between updates and commits is that commits help reduce the size of future ciphertexts while updates do not.

An additional property SGM schemes must possess is that the *absence of high-quality random values* only affects PCS, but none of the other properties of an SGM scheme. Finally, it is also required that SGM schemes defend against parties who purposely *do not delete old values* (i.e., expired key material); such parties must not be able to use said values to break the security of messages after they are removed from the group.

3.2 Syntax

This section introduces the formal syntax of an SGM scheme. Parties are identified by unique party IDs ID chosen from an arbitrary fixed set. In the following, s and s' denote the internal state of the SGM scheme before and after an operation, respectively. An SGM scheme SGM consists of 15 algorithms grouped into the categories above:

- For initialization:
  - $s \leftarrow \mathsf{Init}(\mathsf{ID})$: takes as input a party ID ID and generates the initial state.
- For interaction with the PKI:
  - $(s', \mathsf{spk}) \leftarrow \mathsf{Gen\text{-}SK}(s)$: generates new signature key pair and outputs the public key;
  - $s' \leftarrow \mathsf{Rem\text{-}SK}(s, \mathsf{spk})$: removes the key pair corresponding to spk from the state;
  - $s' \leftarrow \mathsf{Get\text{-}SK}(s, \mathsf{ID}', \mathsf{spk}')$: stores signature public key spk' for party ID';
  - $(s', \mathsf{kb}) \leftarrow \mathsf{Gen\text{-}KB}(s, \mathsf{spk})$: generates new key bundle (aka initial key material), signed with spk;
  - $(s', \mathsf{ok}) \leftarrow \mathsf{Get\text{-}KB}(s, \mathsf{ID}', \mathsf{kb}')$: stores key bundle kb' for party ID' — can reject kb' by outputting $\mathsf{ok} = \mathsf{false}$.

[5]Observe that protocols that only satisfy FS and PCS individually do not necessarily provide PCFS. Hence, PCFS is a strictly stronger property than FS and PCS together.

Key bundles have the format $\mathsf{kb} = (\mathsf{wpk}, \mathsf{spk}, \mathrm{sig})$, where sig is a signature of wpk under spk, and wpk is so-called *welcome key material*, which can be used to encrypt secret information for joining group members.

- For group creation:
  - $s' \leftarrow \mathsf{Create}(s, \mathsf{spk}, \mathsf{wpk})$: creates group with self, where the party uses key pair corresponding to spk to sign in this group, while control messages for which the recipient is the group creator, will be encrypted under keys in wpk.
- For proposals:
  - $(s', P) \leftarrow \mathsf{Add}(s, \mathsf{ID}')$: generates a proposal to add party ID';
  - $(s', P) \leftarrow \mathsf{Remove}(s, \mathsf{ID}')$: generates a proposal to remove party ID';
  - $(s', P) \leftarrow \mathsf{Update}(s, \mathsf{spk})$: generates a proposal for self to update the personal key material; the new signing verification key will be spk;
  - $(s', \mathsf{PI}) \leftarrow \mathsf{Proc\text{-}PM}(s, P)$: adds proposal P to the state and outputs information PI about P.
- For commits:
  - $(s', E, \mathbf{W}, T) \leftarrow \mathsf{Commit}(s, \mathbf{P})$: creates a commit corresponding to a vector $\mathbf{P}$ of proposals and outputs an epoch ID E, an array of welcome messages $\mathbf{W}$ (where $\mathbf{W}[\mathsf{ID}]$ is the welcome message for newly added party ID), and a commit message T (for existing group members);
  - $(s', \mathsf{GI}) \leftarrow \mathsf{Proc\text{-}CM}(s, T)$: used by existing group members to process a commit message T and reach a new epoch; outputs updated group information GI (where $\mathsf{GI} = \perp$ if T is considered invalid);
  - $(s', \mathsf{GI}) \leftarrow \mathsf{Proc\text{-}WM}(s, W)$: used by newly added group members to process welcome message W and join a group; outputs group information GI (where $\mathsf{GI} = \perp$ if W is considered invalid).
- For sending and receiving messages:
  - $(s', e) \leftarrow \mathsf{Send}(s, a, m)$: generates a ciphertext e encrypting plaintext m and authenticating AD a;
  - $(s', E, \mathsf{S}, i, m) \leftarrow \mathsf{Rcv}(s, a, e)$: decrypts ciphertext e to plaintext m and verifies AD a; also outputs a triple (E, S, i) consisting of epoch ID E, sender ID S, and message index i.

3.3 Security

Apart from satisfying *correctness*, i.e., messages sent by some party are received correctly by all other parties, the basic security expectation of SGM protocols is quite simple: they must provide confidentiality of messages, authenticity of messages and AD, as well as PCFS. Capturing this simple intuition, however, turns out to be quite involved and subtle, and, consequently, the SGM security game is somewhat complex.

The security game allows the adversary $\mathcal{A}$ to control the execution of and attack a single group.[6]

[6] All results in this work will carry over to the multi-group setting, but in order to tame the already high complexity, defining and proving security for multi-group settings is left to future work.

In particular, $\mathcal{A}$ controls who creates the group, who is added and removed, who updates, who commits, who sends, and who receives messages. The attacker is also allowed to leak the state of any party (whether currently part of the group or not) at any time. The security of messages is captured by considering a challenge oracle that allows $\mathcal{A}$ to have any particular group member send one of two messages. Which of the two messages is chosen by the game by sampling an internal random bit b, which must be guessed by $\mathcal{A}$ at the end of the game.

Given the adversarial power of leaking parties' states, the game must keep track of which messages can be expected to be secure given the attackers's knowledge. In order to avoid trivial attacks, $\mathcal{A}$ must be barred from (a) issuing challenges whenever it would know the key material used to encrypt the message and (b) leaking the state of a party that is yet to receive already issued challenges. This captures non-trivial *privacy* breaks. With respect to *authenticity*, $\mathcal{A}$ is assumed not to attempt to inject control messages whenever there is at least one group member in the process of healing. Without this restriction, the attacker could craft malicious control messages that destroy the group state. While this behavior could theoretically be defended against, it does appear that the necessary techniques would be quite expensive and the resulting schemes impractical. Note, however, that the attacker is allowed to forge ciphertexts at any time. The SGM scheme is expected to simply output the forged plaintext and AD and continue to function normally.

3.3.1 Bookkeeping.

The complexity of the SGM security definition stems from the bookkeeping required to determine which messages are safe to challenge and when the attacker is allowed to inject. The security game keeps track of all the relevant execution data with the help of a so-called *history graph.* The recorded data informs three *safety predicates* (each pertaining to privacy, authenticity, or both) used to determine whether a given execution was legal. These predicates are kept generic and considered parameters of the security definition.

History graphs. A history graph is a directed tree whose nodes correspond to the group state in the various epochs of an execution: The root of the tree is a special node v_{root} that corresponds to the state of not being part of a group. The children of the root node are v_{create}-nodes, which correspond to the creation of groups.[7] The remaining nodes all correspond to the group state after a particular commit operation. Two nodes v and v' are connected by an edge (v, v') if the commit operation leading to v' was created in v. Concretely, a history graph node consists of the following values: $v = (\mathsf{vid}, \mathsf{orig}, \mathsf{data}, \mathbf{pid})$, where vid is the node's (unique) ID, orig is the party that caused the node's creation, i.e., orig is either the group creator or the committer, data is additional data, and $\mathbf{pid}$ is the vector of (IDs of) proposals (see below) that were included in the commit. The history graph is accessed by the security-game oracles via the "HG object" HG, which provides information via several methods (explained later as they are needed). The (ID vid of the) node corresponding to a party ID's current state is stored by the array $\mathsf{V\text{-}Pt}[\mathsf{ID}]$.

Proposals. Similarly to HG and the history graph, the object Props keeps track of proposals. The information recorded about each proposal is a vector $p = (\mathsf{pid}, \mathsf{vid}, \mathsf{op}, \mathsf{orig}, \mathsf{data})$, where pid is the proposal's (unique) ID, vid is the (ID of) the HG node corresponding to the epoch in which the proposal was created, $\mathsf{op} \in \{\mathtt{add}, \mathtt{rem}, \mathtt{upd}\}$ is the type of proposal, orig is the party that issued the

[7] The security game allows multiple groups to be *created* (when multiple parties simultaneously create a group). However, in order to remain in the single-group setting, the attacker is required to pick a single one of these groups as the "canonical group." Parties may only join the canonical group.

proposal, and data is additional data. Similarly to the HG object, Props will also export several useful methods (also explained later) to the oracles of the security game.

PKI bookkeeping. Just as epochs and proposals are kept track of symbolically, so are welcome keys and signature keys. Specifically, each welcome key has a (unique) welcome-key ID wkid, and each signature key has a (unique) signature-key ID skid. The arrays $\mathsf{WK\text{-}ID}[\cdot]$ and $\mathsf{SK\text{-}ID}[\cdot]$ map wkid and skid values to the ID of the party that owns the corresponding secret keys; the arrays $\mathsf{WK\text{-}PK}[\cdot]$ and $\mathsf{SK\text{-}PK}[\cdot]$ map wkid and skid values to the corresponding public keys. Moreover, the array $\mathsf{WK\text{-}SK}[\cdot]$ remembers the binding of welcome keys to signature keys; these bindings are created by key bundles: when wpk with wkid is signed under spk with skid in a key bundle, the game sets $\mathsf{WK\text{-}SK}[\mathsf{wkid}] \leftarrow \mathsf{skid}$.

The array $\mathsf{CL\text{-}KB}[\cdot,\cdot]$ keeps track of the *contact list* for each party. Specifically, $\mathsf{CL\text{-}KB}[\mathsf{ID}, \mathsf{ID}']$ is a *queue* of pairs $(\mathsf{wkid}, \mathsf{skid})$ of (IDs of) initial key material that ID would use if it were to add ID' to the group.

Stored and leaked values. In order to stay on top of the information attacker $\mathcal{A}$ accumulates via state compromise, the security game maintains the following arrays/sets. *Stored values:* $\mathsf{V\text{-}St}[\mathsf{ID}]$ contains pairs $(\mathsf{vid}, \mathsf{flag})$ of (i) (IDs of) the epochs for which ID currently stores information and (ii) a flag recording whether said information can be used to infer information about subsequent states. This array is organized as a queue with maximum capacity r, where r is a parameter of the definition and of SGM protocols: it stands for the maximum number of "open" epochs a party keeps at any point in time.[8] $\mathsf{P\text{-}St}[\mathsf{ID}]$ contains the (IDs of) the proposals currently stored by ID. $\mathsf{WK\text{-}St}[\cdot]$ and $\mathsf{SK\text{-}St}[\cdot]$ contain the (IDs of) the welcome and signing keys, respectively, currently stored by ID. *Leaked values:* $\mathsf{V\text{-}Lk}$ is a set that contains triples $(\mathsf{vid}, \mathsf{ID})$. Each such tuple means that the attacker learned the state of ID in epoch vid, that for that state $\mathsf{flag} = \mathsf{true}$, which implies that leaked information can be used to infer information about subsequent epochs. $\mathsf{WK\text{-}Lk}$ and $\mathsf{SK\text{-}Lk}$ contain the (IDs of) the welcome and signing keys, respectively, for which the attacker has learned the secret keys. $\mathsf{AM\text{-}Lk}$ records application messages for which the attacker has learned the key material.

Deletions, and lack thereof. The SGM definition also requires that parties who fail to delete old values can not use them to their advantage after they are removed from the group. Correspondingly, the array Del keeps track of which parties *are* deleting values as they are supposed to (those with $\mathsf{Del}[\mathsf{ID}] = \mathsf{true}$) and which ones are not. Formally, a party with $\mathsf{Del}[\mathsf{ID}] = \mathsf{false}$ will simply move such values to a special "trash tape," instead of deleting them; the contents of the trash tape will be revealed to $\mathcal{A}$ upon state leakage. Consequently, there are several "trash arrays" that keep track of what is stored on the trash tape of each party: $\mathsf{V\text{-}Tr}$ (for undeleted information about epochs), $\mathsf{AM\text{-}Tr}$ (for undeleted information about application messages), $\mathsf{WK\text{-}Tr}$ (for undeleted information about welcome keys), and $\mathsf{SK\text{-}Tr}$ (for undeleted information about signature keys).

Challenges. The array $\mathsf{Chall}[\mathsf{vid}]$ stores, for each vid, pairs (S, i), indicating that the i^{th} message from S in epoch vid was a challenge message.

[8]The intuition behind the flag is that only the information stored about *newest* epoch should allow the attacker (upon state leakage) to compute key material for subsequent epochs (which is unavoidable). Information corresponding to older epochs, which are only kept open to receive delayed application messages, should not lend itself to compromise the security of subsequent epochs.

SGM Security Game: Initialization

```
// General
$b \leftarrow \{0,1\}$
$\mathsf{idCtr}{+}{+}$
$\forall \mathsf{ID} : s[\mathsf{ID}] \leftarrow \mathsf{Init}(\mathsf{ID})$

// Communication
$\mathsf{CM}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{WM}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{PM}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{AM}[\cdot,\cdot,\cdot,\cdot] \leftarrow \epsilon$

// History Graph
$\mathsf{vid}_{\mathsf{root}} \leftarrow \mathsf{HG.init}$
$\mathsf{V\text{-}Pt}[\cdot] \leftarrow \mathsf{vid}_{\mathsf{root}}$
$\mathsf{V\text{-}St}[\cdot] \leftarrow [(\mathsf{vid}_{\mathsf{root}}, \mathsf{false})]$
$\mathsf{V\text{-}Tr}[\cdot] \leftarrow \emptyset$
$\mathsf{V\text{-}Lk} \leftarrow \emptyset$
$\mathsf{P\text{-}St}[\cdot] \leftarrow \emptyset$
$\mathsf{Ep\text{-}ID}[\cdot] \leftarrow \epsilon$

// App. Messages
$\mathsf{AM\text{-}Tr}[\cdot] \leftarrow \emptyset$
$\mathsf{AM\text{-}Lk} \leftarrow \emptyset$

// Miscellaneous
$\mathsf{Chall}[\cdot] \leftarrow \emptyset$
$\mathsf{Del}[\cdot] \leftarrow \mathsf{true}$
$\mathsf{BR}[\cdot] \leftarrow \mathsf{false}$

// PKI
$\mathsf{CL\text{-}KB}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{WK\text{-}SK}[\cdot] \leftarrow \epsilon$
$\mathsf{WK\text{-}ID}[\cdot] \leftarrow \epsilon$
$\mathsf{WK\text{-}PK}[\cdot] \leftarrow \epsilon$
$\mathsf{WK\text{-}St}[\cdot] \leftarrow \emptyset$
$\mathsf{WK\text{-}Tr} \leftarrow \emptyset$
$\mathsf{WK\text{-}Lk} \leftarrow \emptyset$
$\mathsf{SK\text{-}ID}[\cdot] \leftarrow \epsilon$
$\mathsf{SK\text{-}PK}[\cdot] \leftarrow \epsilon$
$\mathsf{SK\text{-}St}[\cdot] \leftarrow \emptyset$
$\mathsf{SK\text{-}Tr} \leftarrow \emptyset$
$\mathsf{SK\text{-}Lk} \leftarrow \emptyset$
```

Figure 1: Initialization of the security game for secure group-messaging schemes.

Epoch IDs. The receiving algorithm Rcv must correctly output a "sequence number" for each message received. A natural way to identify application messages is by the triple of epoch ID, sender, and index (as above). However, the SGM scheme cannot be expected to output epoch identifiers vid used by the security game. Instead, the scheme gets to label the epochs itself whenever a commit is created by outputting an *epoch ID E*. Algorithm Rcv will use the same E to refer to messages sent in the corresponding epoch. The security game stores the E used by the SGM scheme in array $\mathsf{Ep\text{-}ID}[\mathsf{vid}]$.

Bad randomness. The game keeps track of which update proposals and commits were created with randomness known to the attacker with the help of the Boolean array $\mathsf{BR}[\cdot]$. This information must be recorded because such proposals/commits will not contribute to PCS.

In the oracles below that correspond to randomized SGM algorithms, the attacker gets to possibly supply the random coins r used by the affected party. Whenever $\mathcal{A}$ does not wish to specify said coins, it sets $r = \bot$, in which case uniformly random coins (unknown to $\mathcal{A}$) are used.

3.3.2 Initialization.

At the onset of the execution, the SGM security game initializes (cf. Figure 1) all of the bookkeeping variables listed above. Additionally, it randomly chooses the challenge bit b, initializes a counter idCtr that serves to provide IDs for epochs (vid), for proposals (pid), for welcome keys (wkid), as well as for signature keys (skid), and sets up communication arrays (explained where they are used) dedicated to control and application messages.

The initialization also initializes the state of all possible parties by running the Init algorithm and storing the result in the state array $s[\cdot]$.[9]

Finally, the first node of the history graph is created via a call to the HG.init method. This causes HG to create the root node $v_{\mathsf{root}} = (.\mathsf{vid} \leftarrow \mathsf{idCtr}{+}{+}, .\mathsf{orig} \leftarrow \bot, .\mathsf{data} \leftarrow \bot, .\mathbf{pid} \leftarrow \bot)$ and return $\mathsf{vid}_{\mathsf{root}} = v_{\mathsf{root}}.\mathsf{vid}$.

3.3.3 Oracles.

All adversary oracles described below proceed according to the same pattern: (a) verifying the validity of the oracle call, (b) retrieving values needed for (c), (c) running the corresponding SGM algorithm, and (d) updating the bookkeeping. Validity checks (a) are described informally in the text below; a formal description is provided in Figure 6. Note that, most of the time, (b) and (c) are

[9]Of course, this is really done on an on-demand basis.

SGM Security Game: PKI Oracles

```
// ID generates new sig. key
gen-new-SigK (ID)
    (s[ID], spk) ← Gen-SK(s[ID])
    skid ← idCtr++
    SK-ID[skid] ← ID
    SK-PK[skid] ← spk
    SK-St[ID] +← skid
    return (skid, spk)

// ID removes sig. key
rem-SigK (ID, skid)
    req SK-ID[skid] = ID
    req skid ∈ SK-St[ID]
    req skid ∉ HG.SKsUsed(ID)
    spk ← SK-PK[skid]
    s[ID] ← Rem-SK(s[ID], spk)
    if ¬Del[ID]
        SK-Tr[ID] +← skid
    SK-St[ID] −← skid
```

```
// ID stores sig. key of ID′
get-SK (ID, ID′, skid)
    req SK-ID[skid] = ID′
    spk ← SK-PK[skid]
    s[ID] ← Get-SK(s[ID], ID′, spk)
    SK-St[ID] +← skid

// ID generates new key bundle
gen-new-KB (ID, skid)
    req SK-ID[skid] = ID
    req skid ∈ SK-St[ID]
    spk ← SK-PK[skid]
    (s[ID], kb) ← Gen-KB(s[ID], spk)
    (wpk, ., .) ← kb
    wkid ← idCtr++
    WK-ID[wkid] ← ID
    WK-PK[wkid] ← wpk
    WK-St[ID] +← wkid
    WK-SK[wkid] +← skid
    return (wkid, kb)
```

```
// ID stores kb of ID′
get-KB (ID, ID′, kb)
    (wpk, spk, sig) ← kb
    let wkid : WK-PK[wkid] = wpk
    let skid : SK-PK[skid] = spk
    req SK-ID[skid] = ID′
        ∧ skid ∈ SK-St[ID]
    (s[ID], ok) ← Get-KB(s[ID], ID′, kb)
    if ok = true
        if skid ∉ SK-Lk
          ∧ WK-SK[wkid] ≠ skid
            win                          // Forgery
    if wkid = ⊥
        wkid ← idCtr++
        WK-PK[wkid] ← wpk
        WK-Lk +← wkid
    CL-KB[ID, ID′].enq((wkid, skid))
    WK-St[ID] +← wkid
    SK-St[ID] +← skid
```

Figure 2: PKI-related oracles of the security game for secure group-messaging schemes. The compatibility functions are described in the accompanying text; a formal description is provided in Figure 6.

straight-forward and are not mentioned in the descriptions. To improve readability, lines (c) are highlighted.

3.3.4 PKI Oracles.

The PKI oracles offer the following functionality to attacker $\mathcal{A}$:

- *New signature keys:* Have a party ID create a new signature key pair. This essentially boils down to ID running algorithm Gen-SK, which outputs a signature public key spk. The SGM game then generates an skid for spk and updates arrays SK-ID, SK-PK, and SK-St correspondingly.
- *Remove signature keys:* Have a party ID delete a signature public key spk, identified by the corresponding skid. In order for a call to this oracle to be legal, skid must correspond to an spk (a) currently stored by ID and (b) not used by ID in either the current epoch or any pending proposals or epochs.[10] The oracle simply runs Rem-SK on spk and subsequently updates arrays SK-ID, SK-PK, and SK-St to reflect the removal of spk. Additionally, if ID does not delete old values, i.e., if Del[ID] = false, skid is added to SK-Tr[ID].
- *Store new signature key:* This oracle lets the attacker instruct a party ID to store signature public key, identified by its skid, of a party ID′. The oracle ensures that SK-ID[skid] = ID′, i.e., that skid was really created by ID′. This models the fact that the bindings between signature keys and identities are incorruptible.
- *Generate new key bundle:* The attacker can make a party ID create a new key bundle and store it on the PKI server. To that end, $\mathcal{A}$ specifies the skid of the signature public key spk that is supposed to be used inside the key bundle. The call is only valid if the key pair corresponding to skid is currently stored by ID.

[10] Pending epochs are child epochs of V-Pt[ID]; they are explained in more detail later on.

The oracle runs $\mathsf{Gen\text{-}KB}$ on spk. The first component of the newly generated key bundle is a new welcome key wpk. Thus, the security game generates a new wkid associated with wpk and updates arrays $\mathsf{WK\text{-}ID}$, $\mathsf{WK\text{-}PK}$, and $\mathsf{WK\text{-}St}$ accordingly. The oracle also binds wkid to skid.

- *Store new key bundle:* This oracle lets the attacker instruct a party ID to store a key bundle $\mathsf{kb} = (\mathsf{wpk}, \mathsf{spk}, \mathrm{sig})$ belonging to another party ID'. Two conditions must be satisfied for the oracle call to be valid:

  1. The signature public key spk must belong to ID', i.e., $\mathsf{spk} = \mathsf{SK\text{-}PK}[\mathsf{skid}]$ for some skid with $\mathsf{SK\text{-}ID}[\mathsf{skid}] = \mathsf{ID}'$. This models the fact that the long-term secrets binding signature public keys to identities are incorruptible.
  2. The public key spk must be stored by ID.

  If the oracle call is valid, algorithm $\mathsf{Get\text{-}KB}$ is run on ID' and kb. If the secret key corresponding to spk has not been leaked, i.e., $\mathsf{skid} \notin \mathsf{SK\text{-}Lk}$, then $\mathsf{Get\text{-}KB}$ must only accept the key bundle if wpk exists and is bound to spk, i.e., $\mathsf{WK\text{-}SK}[\mathsf{wkid}] = \mathsf{skid}$. If $\mathsf{Get\text{-}KB}$ outputs ok, $\mathsf{skid} \notin \mathsf{SK\text{-}Lk}$, but $\mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$, then the adversary has attempted to create a forgery against the signing key with ID skid, and if it is successful, it wins the game. If wpk is adversarially generated (i.e., $\mathsf{wkid} = \perp$), the game generates a new wkid for wpk, updates $\mathsf{WK\text{-}PK}$ accordingly, and immediately marks wkid as leaked. Finally, ID's (symbolic) contact list is updated, by adding the pair $(\mathsf{wkid}, \mathsf{skid})$ to the end of queue $\mathsf{CL\text{-}KB}[\mathsf{ID}, \mathsf{ID}']$.

3.3.5 Main oracles.

The main oracles of the SGM game are split into four figures: oracles related to (i) group creation and proposals (Figure 3), (ii) commits (Figure 4), (iii) sending, receiving and corruptions (Figure 5). The validity of all oracle calls is checked by the corresponding compatibility functions (Figure 6).

Group creation. The attacker can instruct a party ID to create a new group by calling the group-creation oracle (Figure 3); such a call must also specify the skid of the signature key under which ID initially signs their messages, as well as the wkid of the welcome key material. A call to the group creation oracle is valid if (i) ID is not in a group yet, (ii) skid, wkid, are currently stored by ID, and (iii) wkid is signed under skid.

The bookkeeping is updated as follows: A call is made to the $\mathsf{HG.create}(\mathsf{ID}, \mathsf{skid})$ method. This causes HG to create a node $v = (.\mathsf{vid} \leftarrow \mathsf{idCtr}{+}{+}, .\mathsf{orig} \leftarrow \mathsf{ID}, .\mathsf{data} \leftarrow \mathsf{skid}, .\boldsymbol{\mathsf{pid}} \leftarrow \perp)$ as a child of v_{root} and return $\mathsf{vid} = v.\mathsf{vid}$. Then, the bad-randomness array is filled (depending on whether $\mathcal{A}$ specified coins r or not), ID's pointer is set to vid, and $\mathsf{V\text{-}St}[\mathsf{ID}]$ is set to a queue containing only $(\mathsf{vid}, \mathsf{true})$, where the second component records that from the information ID currently stores about vid, one can compute information about (future) child states of vid.

Creating proposals. The oracles **prop-$\{$add, rem, up$\}$-user** (Figure 3) allow the attacker to instruct a party ID to issue add/remove/update proposals. Calls to these proposal oracles are valid if ID is a group member and:

- *(add proposals)* the target ID' is *not* a group member already and ID has initial key material for ID' stored;
- *(remove proposals)* the target ID' *is* a group member;
- *(update proposals)* the skid with which ID is supposed to sign is currently stored by ID.

SGM Security Game: Group-Creation and Proposal Oracles

```
// ID creates group; signs with skid
create-group (ID, skid, wkid, $r$)
    req *compat-create(ID, skid, wkid)
    $s$[ID] ← Create($s$[ID], SK-PK[skid], WK-PK[wkid]; $r$)
    vid ← HG.create(ID, skid)
    BR[vid] ← $r \neq \perp$
    V-Pt[ID] ← vid
    V-St[ID] ← [(vid, true)]
    return vid

// ID proposes to add ID′
prop-add-user (ID, ID′)
    req *compat-prop(add, ID, ID′, $\perp$)
    ($s$[ID], $P$) ← Add($s$[ID], ID′)
    (wkid′, skid′) ← CL-KB[ID, ID′].deq
    pid ← Props.new(add, ID, (ID′, wkid′, skid′))
    PM[pid] ← $P$
    return (pid, $P$)

// ID proposes to remove ID′
prop-rem-user (ID, ID′)
    req *compat-prop(rem, ID, ID′, $\perp$)
    ($s$[ID], $P$) ← Remove($s$[ID], ID′)
    pid ← Props.new(rem, ID, ID′)
    PM[pid] ← $P$
    return (pid, $P$)

// ID proposes update
prop-up-user (ID, skid, $r$)
    req *compat-prop(upd, ID, $\perp$, skid)
    ($s$[ID], $P$) ← Update($s$[ID], SK-PK[skid]; $r$)
    pid ← Props.new(upd, ID, skid)
    PM[pid] ← $P$
    return (pid, $P$)

// Proposal pid delivered to ID
dlv-PM (ID, pid)
    req *compat-dlv-PM(ID, pid)
    ($s$[ID], PI) ← Proc-PM($s$[ID], PM[pid])
    if ¬Props.checkPI(pid, PI)
        win
    P-St[ID] +← pid

// Proposal $P'$ injected to ID
inj-PM (ID, $P'$)
    req *compat-inj-PM(ID, $P'$)
    ($s$[ID], PI) ← Proc-PM($s$[ID], $P'$)
    vid ← V-Pt[ID]
    $\mathsf{ID_O}$ ← PI.orig
    req ¬(*auth-compr(vid)
        ∧ *SK-compr(vid, $\mathsf{ID_O}$))
    if PI $\neq \perp$
        win
```

Figure 3: Group-creation and proposal-related oracles of the security game for secure group-messaging schemes. The compatibility functions are described in the accompanying text; a formal description is provided in Figure 6.

Note that the last bullet means that if a party wishes to change its active signature key, they must have generated one and registered it with the PKI before issuing the corresponding proposal.

Bookkeeping is updated by calling Props.new(op, ID, data), which records proposal data $p = (.\mathsf{pid} \leftarrow \mathsf{idCtr}{+}{+}, .\mathsf{vid} \leftarrow \mathsf{V\text{-}Pt[ID]}, .\mathsf{op} \leftarrow \mathsf{op}, .\mathsf{orig} \leftarrow \mathsf{ID}, .\mathsf{data} \leftarrow \mathsf{data})$, where $\mathsf{op} \in \{\mathtt{add}, \mathtt{rem}, \mathtt{upd}\}$ is the proposal type and where data stores *(add proposals)* data = (ID′, wkid′, skid′), where (wkid′, skid′) is the head of the contact list CL-KB[ID, ID′]; *(remove proposals)* data = ID′; *(update proposals)* data = skid. Furthermore, the proposal message P output by corresponding proposal algorithm (Add, Remove, or Update) is stored in the communication array PM for proposal messages.

Delivering and injecting proposals. There are two oracles for getting a proposal to a party ID (Figure 3): The first one, **dlv-PM**, is for honest proposal delivery. The attacker specifies a pid—which must belong to the epoch ID is currently in—and the corresponding proposal is fed to the SGM algorithm Proc-PM. Proc-PM is required to output proposal information PI, which is checked by function Props.checkPI. Function checkPI ensures that PI correctly identifies the *type*, the *originator*, as well as the *data* of the proposal.[11] The last action performed by the oracle is to update P-St[ID] to include pid, indicating that ID now stores the new proposal.

The second oracle, **inj-PM**, is used by $\mathcal{A}$ to inject proposals to a party ID. More precisely, $\mathcal{A}$ is allowed to submit proposals P' that (a) either belong to an epoch different from vid := V-Pt[ID] or (b) are completely made up (i.e., there exists no pid with PM[pid] = P'). A call to this second oracle is only allowed if the adversary is not currently able to forge messages for epoch vid. This is the case if either the key material used to authenticate in vid is compromised (***auth-compr**(vid)) or if the signing key used by the supposed originator $\mathsf{ID_O}$ of the proposal (as determined by the output of

[11]Of course, checkPI checks that PI contains the *actual* welcome and signature keys (and not wkid and skid).

$\mathsf{Proc\text{-}PM}$) is compromised ($\mathbf{*SK\text{-}compr}(\mathsf{vid}, \mathsf{ID}_\mathsf{O})$). The functions ***auth-compr** and ***SK-compr** are two of the *safety predicates* mentioned above.

Creating commits. The commit-related oracles (Figure 4) deal with commits and delivery of commit and welcome messages.

In order to have a party ID create a commit based on a set of proposals (with IDs) $\mathbf{pid}$, the attacker calls the **commit** oracle. The specified $\mathbf{pid}$ must be a subset of all proposals currently stored by ID, and all proposals must belong to ID's current epoch. In addition, a (rather permissive) sensibility check is run on the vector of proposal specified by $\mathbf{pid}$: the proposals are processed in the given order (changing the group roster accordingly), and for each proposal it is checked whether the originator would be in the group and, additionally, whether *(add proposal)* the target would be in the group; *(remove proposal)* the target would not be in the group.

Subsequently, the SGM algorithm Commit is run on the given set of proposals, which results in an epoch ID E, a vector of welcome messages $\mathbf{W}$, and a commit message T being output. Bookkeeping calls $\mathsf{HG.commit}(\mathsf{ID}, \mathbf{pid})$, which creates a new HG node $v = (.\mathsf{vid} \leftarrow \mathsf{idCtr}{+}{+}, .\mathsf{orig} \leftarrow \mathsf{ID}, .\mathsf{data} \leftarrow \perp, .\mathbf{pid} \leftarrow \mathbf{pid})$, as a child of ID's current epoch $\mathsf{V\text{-}Pt}[\mathsf{ID}]$ and returns $\mathsf{vid} = v.\mathsf{vid}$. Then, the bad-randomness array is filled (depending on whether $\mathcal{A}$ specified coins r or not), and E is stored in array $\mathsf{Ep\text{-}ID}$. Finally, the commit messages (for current group members) are stored in array CM, and the welcome messages (for new group members) in WM.

Delivering and injecting commit messages. As with proposals, there are two oracles that allow attacker $\mathcal{A}$ to get a commit message (CM) to a *current* group member—for honestly generated CMs and for adversarially generated CMs (Figure 4). The former oracle, **dlv-CM**, can be used by $\mathcal{A}$ to deliver to a party ID any CM T that corresponds to a child epoch vid of ID's current epoch $\mathsf{V\text{-}Pt}[\mathsf{ID}]$, provided all proposals that lead to vid have been delivered to ID. Oracle **dlv-CM** runs SGM algorithm $\mathsf{Proc\text{-}CM}$ on T, which results in group information GI being output. This information is checked by function $\mathsf{HG.checkGI}$. Function $\mathsf{checkGI}$ checks that GI correctly reports the new *group roster*, the *originator*, as well as the *parties added and removed.* Next, if ID has been removed as part of the commit (checked by function $\mathsf{HG.isRemoved}$), its pointer $\mathsf{V\text{-}Pt}[\mathsf{ID}]$ is set to $\mathsf{vid}_\mathsf{root}$; otherwise, $\mathsf{V\text{-}Pt}[\mathsf{ID}]$ is set to the new epoch vid. Furthermore, the index counter $i[\mathsf{ID}]$ is reset to 0, and $\mathsf{P\text{-}St}[\mathsf{ID}]$, the set of proposals stored, is set to the empty set. If ID deletes old values, i.e., if $\mathsf{Del}[\mathsf{ID}] = \mathsf{true}$, the game changes the flag in $(\cdot, \mathsf{flag}) = \mathsf{V\text{-}St}[\mathsf{ID}].\mathsf{last}$ to false since it is no longer the newest state. If ID does not delete old values, this change is not made, and the first element of $\mathsf{V\text{-}St}[\mathsf{ID}]$ is put into $\mathsf{V\text{-}Tr}[\mathsf{ID}]$ (because it is about to be removed from the queue). Finally, the new epoch vid is added to the end of queue $\mathsf{V\text{-}St}[\mathsf{ID}]$. Recall that this queue has a capacity of r, which means that the first element of the queue is removed. This captures that information about the oldest epoch in ID's state is now deleted by ID.

The second oracle, **inj-CM**, is used by $\mathcal{A}$ to inject CMs to a party ID. More precisely, $\mathcal{A}$ is allowed to submit any CM T' that (a) either belongs to an epoch different from any child epoch of $\mathsf{vid} := \mathsf{V\text{-}Pt}[\mathsf{ID}]$ or (b) is completely made up. A call to this second oracle is only allowed if the adversary is not currently able to forge messages for epoch vid. Similarly to proposals, whether this is the case is determined via the (generic) safety predicates ***auth-compr** and ***SK-compr**.

Delivering and injecting welcome messages. The oracles **dlv-WM** and **inj-WM** can be used by $\mathcal{A}$ to deliver and inject welcome messages, respectively. The work analogously to oracles **dlv-CM** and **inj-CM** above.

SGM Security Game: Commit-Related Oracles

```
// ID commits proposals pid
commit (ID, pid, $r$)
    req *compat-commit(ID, pid)
    $(s[\mathsf{ID}], E, \mathbf{W}, T) \leftarrow \mathsf{Commit}(s[\mathsf{ID}], \mathsf{PM}[\mathbf{pid}]; r)$
    vid ← HG.commit(ID, pid)
    BR[vid] ← $r \neq \bot$
    Ep-ID[vid] ← $E$
    for ID′ ∈ HG.roster(V-Pt[ID])
        CM[ID′, vid] ← $T$
    for ID′ ∈ Props.addedIDs(pid)
        WM[ID′, vid] ← $\mathbf{W}[\mathsf{ID}']$
    return vid

// Control msg. of vid delivered to ID
dlv-CM (ID, vid)
    req *compat-dlv-CM(ID, vid)
    $T$ ← CM[ID, vid]
    $(s[\mathsf{ID}], \mathsf{GI}) \leftarrow \mathsf{Proc\text{-}CM}(s[\mathsf{ID}], T)$
    if ¬HG.checkGI(vid, GI)
        win
    if HG.isRemoved(ID, vid)
        V-Pt[ID] ← $\mathsf{vid}_{\mathsf{root}}$
    else
        V-Pt[ID] ← vid
    $i$[ID] ← 0
    P-St[ID] ← ∅
    if Del[ID]
        V-St[ID].last.flag ← false
    else
        V-Tr[ID] +← V-St[ID].first
    V-St[ID].enq((V-Pt[ID], true))
```

```
// Control msg. $T'$ gets injected to ID
inj-CM (ID, $T'$)
    req *compat-inj-CM(ID, $T'$)
    $(s[\mathsf{ID}], \mathsf{GI}) \leftarrow \mathsf{Proc\text{-}CM}(s[\mathsf{ID}], T')$
    vid ← V-Pt[ID]
    $\mathsf{ID}_{\mathsf{O}}$ ← GI.orig
    req ¬(*auth-compr(vid)
        ∧ *SK-compr(vid, $\mathsf{ID}_{\mathsf{O}}$))
    if GI ≠ ⊥
        win

// Welcome msg. of vid delivered to ID
dlv-WM (ID, vid)
    req *compat-dlv-WM(ID, vid)
    $W$ ← WM[ID, vid]
    $(s[\mathsf{ID}], \mathsf{GI}) \leftarrow \mathsf{Proc\text{-}WM}(s[\mathsf{ID}], W)$
    if ¬HG.checkGI(vid, GI)
        win
    V-Pt[ID] ← vid
    V-St[ID] ← [vid]
    wkid ← HG.addedWK(ID, vid)
    if ¬Del[ID]
        WK-Tr[ID] +← wkid
    WK-St[ID] −← wkid

// Welcome msg. $W'$ injected to ID
inj-WM (ID, $W'$)
    req *compat-inj-WM(ID, $W'$)
    $(s[\mathsf{ID}], \mathsf{GI}) \leftarrow \mathsf{Proc\text{-}WM}(s[\mathsf{ID}], W')$
    $E$ ← GI.epID
    req ∃ vid : Ep-ID[vid] = $E$
    $\mathsf{ID}_{\mathsf{O}}$ ← GI.orig
    req ¬(*auth-compr(vid)
        ∧ *SK-compr(vid, $\mathsf{ID}_{\mathsf{O}}$))
    if GI ≠ ⊥
        win
```

Figure 4: Commit-related oracles of the security game for secure group-messaging schemes. The compatibility functions are described in the accompanying text; a formal description is provided in Figure 6.

Sending messages and challenges. Oracle **send** allows $\mathcal{A}$ to instruct any current group member S to send a message m and associated data (AD) a. The oracle runs algorithm Send on m and a, which creates a ciphertext e. The oracle increments S's message counter, and stores the triple (a, m, e) in AM.

The challenge oracle **chall** works quite similarly, except that $\mathcal{A}$ specifies two equal-length messages m_0 and m_1, and m_b is passed to Send (where b is the bit chosen during initialization). Furthermore, the pair $(\mathsf{S}, i[\mathsf{S}])$ is recorded in array Chall.

Delivering and injecting application messages. The attacker can get application messages to group members in two ways, by using either **dlv-AM** or **inj-AM**—for honestly and adversarially generated AMs, respectively. Oracle **dlv-AM** takes as input a tuple $(\mathsf{vid}, \mathsf{S}, i, \mathsf{R})$ identifying the epoch, the sender, the index, and the recipient of the AM to be delivered. Then, the corresponding AD and ciphertext are input to Rcv, which subsequently outputs $(E', \mathsf{S}', i', m')$. These values are checked, and if R misidentifies any of them, the attacker immediately wins the security game. Subsequently, $\mathsf{AM}[\mathsf{vid}, \mathsf{S}, i, \mathsf{R}]$ is set to `received`, indicating that R should no longer keep around any key material that can be used to decipher e. If R does not delete old values, then a corresponding entry is place in array $\mathsf{AM\text{-}Tr}[\mathsf{R}]$. Note that the oracle does not return any values since the attacker knows which values are output by Rcv.

SGM Security Game: Send/Receive, Corrupt/No-Del & Safety Oracles

```
// S sends message $m$ with AD $a$
send (S, $a$, $m$)
    req *compat-send(S)
    $(s[S], e) \leftarrow$ Send$(s[S], a, m)$
    $i[S]$++
    for R $\in$ HG.roster(V-Pt[S]) \ {S}
        AM[V-Pt[S], S, $i$[S], R] $\leftarrow (a, m, e)$
    return $e$

// Deliver $i^{th}$ msg of S in vid to R
dlv-AM (vid, S, $i$, R)
    req *compat-dlv-AM(vid, S, $i$, R)
    $(a, m, e) \leftarrow$ AM[S, vid, $i$, R]
    $(s[R], E', S', i', m') \leftarrow$ Rcv$(s[R], a, e)$
    if $(E', S', i', m') \neq$ (Ep-ID[vid], S, $i$, $m$)
        win
    if $\neg$Del[R]
        AM-Tr[R] $\leftarrow$ (vid, S, $i$)
    AM[vid, S, $i$, R] $\leftarrow$ received

// Attacker leaks state of ID
corr (ID)
    for (vid, flag) $\in$ V-St[ID] $\cup$ V-Tr[ID]
        if flag
            V-Lk +$\leftarrow$ (vid, ID)
        AM-Rcvd $\leftarrow$ {(S, $i$) | AM[vid, S, $i$, ID] = received}
        AM-Lk +$\leftarrow$ (vid, AM-Rcvd, AM-Tr[ID])
    $W \leftarrow$ WK-St[ID] $\cup$ WK-Tr[ID]
    WK-Lk +$\leftarrow$ {wkid $\in W$ | WK-ID[wkid] = ID}
    $S \leftarrow$ SK-St[ID] $\cup$ SK-Tr[ID]
    SK-Lk +$\leftarrow$ {skid $\in S$ | SK-ID[skid] = ID}
    HG.corrHanging(ID)
    return $s$[ID]

// ID is instructed to disable deletions
no-del (ID)
    disable deletions for ID
    Del[ID] $\leftarrow$ false

// Safety for privacy
*priv-safe
    return $\forall$vid, S, $i$ : ((S, $i$) $\in$ Chall[vid] $\implies$ *AM-sec(vid, S, $i$))

// S sends challenge with AD $a$
chall (S, $a$, $m_0$, $m_1$)
    req *compat-chall(S)
    $(s[S], e) \leftarrow$ Send$(s[S], a, m_b)$
    $i[S]$++
    for R $\in$ HG.roster(V-Pt[S]) \ {S}
        AM[V-Pt[S], S, $i$[S], R] $\leftarrow (a, m, e)$
    Chall[V-Pt[S]] +$\leftarrow$ (S, $i$[S])
    return $e$

// $a'$, $e'$ get injected to R
inj-AM ($a'$, $e'$, R)
    req *compat-inj-AM($a'$, $e'$, R)
    $(s[R], E', S', i', m') \leftarrow$ Rcv$(s[R], a', e')$
    if $m' \neq \perp$
        let vid' : Ep-ID[vid'] = $E'$
        if $\neg\Big(\neg$*AM-sec(vid', S', $i'$)
                $\wedge$ *SK-compr(vid', S')$\Big)$
            $\vee$ AM[vid', S', $i'$, R] = received
            win
        if $\neg$Del[vid]
            AM-Tr[R] $\leftarrow$ (vid', S', $i'$)
        AM[vid', S', $i'$, R] $\leftarrow$ received
    return $(E', S', i', m')$
```

Figure 5: Send/receive, corruption, no-deletion, and privacy-safety oracles of the security game for secure group-messaging schemes. The compatibility functions are described in the accompanying text; a formal description is provided in Figure 6.

Oracle **inj-AM** allows $\mathcal{A}$ to inject any AD/ciphertext pair (a', e') to a group member R as long as they do not correspond to an honestly generated AM. The game requires that if Rcv accepts e' and outputs (E', S', i', m'), it must be the case that (1) the key material for the i'^{th} message sent by S′ in epoch E' has not been compromised, and (2) no message has been output for these identifiers before (i.e., a successful replay attack is considered a break of authenticity). Whether (1) is the case is determined by the (generic) safety predicates ***AM-sec** and ***SK-compr**.

Corruption oracle. The corruption oracle **corr** allows the attacker to leak the state of any party ID. It does bookkeeping for the following:

- *Epochs:* All epochs ID currently stores information for are kept track of by V-St[ID]; recall that for (vid, flag) $\in$ V-St[ID] the variable flag records whether or not the information stored about vid allows to infer key material of child epochs. The security game copies the pairs in V-St[ID] with flag = true into the set V-Lk. Additionally, for all (vid, $\cdot$) $\in$ V-St[ID], records the current set of messages already received by ID in epoch vid. This helps keep track of which application messages are affected by this instance of state leakage. Finally, the values in V-Tr[ID] are also recorded.

- *Welcome keys:* By leaking ID's state, $\mathcal{A}$ learns all welcome secret keys currently stored by ID: either because ID is supposed to be storing them (recorded by WK-St[ID]) or because they are

SGM Security Game: Compatibility Helpers

```
*compat-create (ID, skid, wkid)
    chk V-Pt[ID] = $\mathsf{vid}_{\mathsf{root}}$
    chk SK-ID[skid] = ID
    chk WK-SK[wkid] = skid
    chk skid $\in$ SK-St[ID]
    chk wkid $\in$ WK-St[ID]
    return true

*compat-prop (op, ID, ID$'$, skid)
    vid $\leftarrow$ V-Pt[ID]
    chk vid $\neq \mathsf{vid}_{\mathsf{root}}$
    $G \leftarrow$ HG.roster(vid)
    select op
        case add do
            chk
              CL-KB[ID, ID$'$] $\neq \emptyset$
            chk ID$' \notin G$
        case rem do
            chk ID$' \in G$
        case upd do
            chk skid $\in$ SK-St[ID]
    return true

*compat-dlv-PM (ID, pid)
    chk Props[pid].vid = V-Pt[ID]
    return true

*compat-inj-PM (ID, $P'$)
    vid $\leftarrow$ V-Pt[ID]
    chk vid $\neq \mathsf{vid}_{\mathsf{root}}$
    chk $\nexists$ pid :
          Props[pid].vid = vid
          $\wedge$ $P'$ = PM[pid]
    return true
```

```
*compat-commit (ID, pid)
    vid $\leftarrow$ V-Pt[ID]
    chk vid $\neq \mathsf{vid}_{\mathsf{root}}$
    chk pid $\subseteq$ P-St[ID]
    G $\leftarrow$ HG.roster(vid)
    for pid $\in$ pid
        G $\leftarrow$ *app-prop(G, pid)
        chk G $\neq \bot$
    return true

*app-prop (G, pid)
    $p \leftarrow$ Props[pid]
    req $p$.orig $\in$ G
    select op
        case add do
            (ID$'$, $\cdot$, $\cdot$) $\leftarrow p$.data
            req ID$' \notin$ G
            G $+\!\leftarrow$ ID$'$
        case rem do
            ID$' \leftarrow p$.data
            req ID$' \in G$
            G $-\!\leftarrow$ ID$'$
    return G

*compat-dlv-CM (ID, vid)
    chk HG.isChild(V-Pt[ID], vid)
    chk HG[vid].pid $\subseteq$ P-St[ID]
    return true

*compat-inj-CM (ID, $T'$)
    vid $\leftarrow$ V-Pt[ID]
    chk vid $\neq \mathsf{vid}_{\mathsf{root}}$
    chk $\forall$vid$' \in$ HG.children(vid) :
          $T' \neq$ CM[vid$'$, ID]
    return true
```

```
*compat-dlv-WM (ID, vid)
    chk V-Pt[ID] = $\mathsf{vid}_{\mathsf{root}}$
    wkid $\leftarrow$ HG.addedWK(ID, vid)
    chk wkid $\neq \bot$
          $\wedge$ wkid $\in$ WK-St[ID]
    return true

*compat-inj-WM (ID, $W'$)
    vid $\leftarrow$ V-Pt[ID]
    chk vid = $\mathsf{vid}_{\mathsf{root}}$
    chk $\forall$vid$'$ :
          $W' \neq$ CM[vid$'$, ID]
    return true

*compat-send (S)
    chk V-Pt[ID] $\neq \mathsf{vid}_{\mathsf{root}}$
    return true

*compat-chall (S, $m_0$, $m_1$)
    chk V-Pt[ID] $\neq \mathsf{vid}_{\mathsf{root}}$
    chk $|m_0| = |m_1|$
    return true

*compat-dlv-AM (vid, S, $i$, R)
    chk vid $\in$ V-St[R]
    chk
      AM[vid, S, $i$, R] $\notin \{\epsilon, \mathtt{received}\}$
    return true

*compat-inj-AM ($a$, $e$, R)
    chk $\forall$S, vid, $i$, $m$ :
      AM[vid, S, $i$, R] $\neq (a, m, e)$
    return true
```

Figure 6: Compatibility oracles of the security game for secure group-messaging schemes.

in ID's trash (i.e., in WK-Tr[ID]). The corresponding wkids are added to the set WK-Lk.

- *Signature keys:* Handled analogously to welcome keys.
- *Application-message trash:* The attacker also learns all values in the array AM-Tr[ID].
- *Pending proposals and commits.* If at the time ID is corrupted, there are outstanding (i.e., uncommitted) update proposals (with ID) pid by ID or hanging (i.e, created but not processed yet) commits (with ID) vid by ID, the attacker learns the corresponding secrets. To that end HG.corrHanging(ID) sets BR[pid] $\leftarrow$ true resp. BR[vid] $\leftarrow$ true for all of them. This will reflect the fact that these update proposals / commits cannot be used for PCS.

Disabling deletions. When the attacker calls oracle **no-del** for a party ID, that party stops deleting old values and stores them on a special trash tape instead (which is leaked along with the rest of ID's state to the attacker upon state compromise).

3.3.6 Privacy-related safety.

At the end of the execution of the SGM security game, the procedure ***priv-safe** ensures that the attacker has only challenged messages that are considered secure by the (generic) safety predicate ***AM-sec**. If the condition is not satisfied, the attacker loses the game.

3.3.7 Advantage.

Let $\Pi = \{\textbf{*AM-sec}, \textbf{*auth-compr}, \textbf{*SK-compr}\}$ be the set of generic safety predicates used in the SGM definition. The attacker $\mathcal{A}$ is parameterized by it's running time, t, and the number of challenge queries, q, and referred to as (t, q)-attacker. The advantage of $\mathcal{A}$ against an SGM scheme Γ w.r.t. to predicates Π is denoted by $\mathrm{Adv}^{\Gamma}_{\mathsf{SGM},\Pi}(\mathcal{A})$.

Definition 1. *An SGM scheme* Γ *is* (t, q, ε)-secure w.r.t. predicates Π*, if for all* (t, q)*-attackers,*

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

4 Ingredients

This section introduces and explains the three main primitives used by the SGM construction in Section 5: continuous group key-agrement (CGKA), forward-secure group AEAD (FS-GEAEAD), and PRF-PRNGs. On a very high level, the PRF-PRNG is an entropy pool from which key material for FS-GAEAD is extracted and whose state is continually refreshed by values from CGKA.

4.1 Continuous Group Key Agreement

CGKA schemes are run by parties wishing to continuously agree on shared secret keys. As with SGM schemes (cf. Section 3.1), (a) CGKA schemes are run over an asynchronous network in which parties may be online/offline at arbitrary times; (b) the communication between parties is assumed to be handled by an untrusted delivery service; (c) there is a PKI used to exchange initial key material; (d) CGKA schemes follow the P&C paradigm; and (e) parties only know a subset of the global state.

4.1.1 Syntax.

The components of a CGKA scheme are (in the following, γ and γ' denote the internal state of the CGKA scheme before and after an operation, respectively):

- *Interaction with the PKI:* A CGKA scheme provides an algorithm $(\mathsf{ipk}, \mathsf{isk}) \leftarrow \mathsf{Gen\text{-}IK}$ to create so-called *initial key pairs*, where ipk can be registered with the PKI.
- *Group creation:* A party ID may create a new group with only itself in it, using the group-creation algorithm $\gamma' \leftarrow \mathsf{Create}(\mathsf{ID}, \mathsf{isk})$, where isk is the initial secret key used by ID.
- *Issue proposals:* A group member may issue three types of proposals P: *add* a new group member ID' with initial public key ipk' by calling $(\gamma', P) \leftarrow \mathsf{Add}(\gamma, \mathsf{ID}', \mathsf{ipk}')$, *remove* a current group member ID' by calling $(\gamma', P) \leftarrow \mathsf{Remove}(\gamma, \mathsf{ID}')$, and *update* personal secrets by calling $(\gamma', P) \leftarrow \mathsf{Update}(\gamma)$.
- *Commit creation:* Occasionally, a party may pick an arbitrary (ordered) subset of proposals $\mathbf{P}$ and execute a commit: $(\gamma', I, W_{\mathsf{pub}}, \mathbf{W}_{\mathsf{priv}}, T) \leftarrow \mathsf{Commit}(\gamma, \mathbf{P})$, which produces a *commit secret* I, a *commit message* (for existing group members), as well as *public and private welcome messages* (for joining group members) W_{pub} and $\mathbf{W}_{\mathsf{priv}}$.
- *Processing commits:* In order to process a commit message T corresponding to proposals $\mathbf{P}$, a party calls $(\gamma', \mathsf{GI}, I) \leftarrow \mathsf{Proc\text{-}Com}(\gamma, T, \mathbf{P})$, which will output the corresponding commit secret I as well as group information GI (where $\mathsf{GI} = \perp$ if T is considered invalid).

- *Joining:* In order to join a group, a party processes welcome messages W_{pub} and W_{priv} by running $(\gamma, \mathsf{GI}, I) \leftarrow \mathsf{Join}(\gamma, \mathsf{orig}, W_{\mathsf{pub}}, W_{\mathsf{priv}}, \mathsf{isk})$, where isk is the init secret key and orig is the ID of the party who created the commit that produced the welcome messages. The outputs are again the commit secret I and group information GI (where $\mathsf{GI} = \perp$ if W_{pub} or W_{priv} are considered invalid).

Observe that unlike SGM schemes, CGKA protocols do not have algorithms to process proposal messages P, and, consequently, these messages must be supplied as inputs to Commit and $\mathsf{Proc\text{-}Com}$. It is left to the higher-level protocol (SGM) to handle this. Similar considerations apply to the initial keys.

4.1.2 Security.

CGKA schemes are expected to guarantee the secrecy of the generated group keys. Furthermore, just like SGM schemes, they must provide *post-compromise forward secrecy (PCFS)*. Once more, update proposals and commits must both contribute to post-compromise security (PCS). CGKA schemes must also be able to deal with bad randomness (in that it only affects PCS) as well as parties who do not delete old values.

An important difference between CGKA and SGM schemes is that the former are designed in a fully authenticated setting. That is, the attacker in the CGKA security game is *not* permitted to *inject* control messages. This turns out to be sufficient as the "authentication layer" can be added by the protocol using the CGKA scheme.

Similarly to SGM, the CGKA security game allows adversary $\mathcal{A}$ to control the execution of and attack a single group. In particular, $\mathcal{A}$ controls who creates the group, who is added and removed, who updates, who commits, etc. The attacker is also allowed to leak the state of any party (whether currently part of the group or not) at any time. The privacy of keys is captured by considering a challenge that outputs either the actual key output by the CGKA scheme or a truly random one—which one is determined by an internal random bit b, which must be guessed by $\mathcal{A}$ at the end of the game.

The CGKA security game follows the history-graph paradigm to keep track of all the relevant execution data. The recorded data informs a *safety predicate* ***CGKA-priv** used at the end of the game to exclude trivial wins by $\mathcal{A}$. More details can be found in Appendix B.1.

4.1.3 (R)TreeKEM.

The (R)TreeKEM CGKA protocols are based on so-called (binary) *ratchet trees (RTs)*. In an RT, group members are arranged at the leaves, and all nodes have an associated public-key encryption (PKE) key pair, except for the root. The tree invariant is that each user knows all secret keys on their *direct path*, i.e., on the path from their leaf node to the root. In order to perform an update—the most crucial operation of a CGKA—and produce a new update secret I, a party first generates fresh key pairs on every node of their direct path. Then, for every node v' on its *co-path*—the sequence of siblings of nodes on the direct path—it encrypts specific information under the public key of v' that allows each party in the subtree of v' to learn all new secret keys from v's parent up to the root. In TreeKEM, standard (CPA-secure) PKE is used. As shown in previous work [3], this leads to subpar PCFS. RTreeKEM improves on this by using so-called *updatable PKE (UPKE)*, in which public and secret keys are "rolled forward" every time a message is encrypted and decrypted, respectively.

In order to establish *adaptive* security of the RTreeKEM protocol, this work applies the so-called Generalized-Selective-Decryption (GSD) paradigm to UPKE. Specifically, we prove GSD security of IND-CPA-secure UPKE in the random oracle model, by first defining a GSD game that models

UPKE based executions, in the presence of bad randomness and group splitting attacks, and then appropriately adapting the framework of [1]. The security of RTreeKEM itself is established by reduction to the GSD security of the underlying UPKE scheme. The safety predicate ***CGKA-priv** considers the commit secret of an epoch vid secure if (informally) the following conditions are satisfied:

- There is no "corrupted" ancestor epoch in the history graph, where an epoch can be corrupted if either a party gets added with leaked initial keys or if the state of a party in the epoch is leaked.
- No information about vid is known to the attacker as the result of a *splitting attack*. An example of a splitting attack is when the attacker creates a sibling epoch vid′ and corrupts a party that is a group member in both vid and vid′.
- Good randomness was used by the commiter creating vid.

More details can be found in Appendix D.1 (GSD security of UPKE) and Appendix D.2 (predicate ***CGKA-priv** and CGKA security of RTreeKEM).

Theorem 1. *Let* UPKE *be an* (t, ε)*-GSD secure UPKE scheme. Furthermore, let* ***CGKA-priv** *be the safety predicate above. Then, RTreeKEM is an* (t', ε)*-secure SGM scheme w.r.t.* ***CGKA-priv***, where* $t' \approx t$*.*

TreeKEM's security. TreeKEM is secure w.r.t. a weaker security predicate than RTreeKEM. Namely, it is required that, either there is no post-challenge compromise (the notion of PCS in [3]), or if there is, compromise happens after the party has already updated it's state (the FSU notion of [3]). Then, security proof for TreeKEM is similar to that of RTreeKEM and proceeds in two steps. First we consider a reduction from GSD with respect to standard public-key encryption, to the IND-CPA security of the underlying scheme, and then we reduce security of TreeKEM to GSD, deriving a theorem similar to the above.

4.2 Forward-Secure Group AEAD

FS-GAEAD schemes provide the convenient abstraction of an "epoch of group messaging." That is, they capture the sending and receiving of application messages within a single epoch of a full SGM scheme. In an execution of FS-GAEAD, all participating group members are initialized with the same random group key (i.e., that key is assumed to be generated and distributed among the group members by the higher-level protocol).

An FS-GAEAD scheme protects the authenticity and privacy of messages sent. Furthermore, it provides forward secrecy, i.e., the security of messages received will not be affected by state compromise. Note that FS-GAEAD is not required to provide any form of post-compromise security. This also allows to design completely deterministic schemes (apart from the initial key).

4.2.1 Syntax.

An FS-GAEAD scheme FS consists of three algorithms (in the following v and v' denote the state of an FS-GAEAD scheme before and after the execution of an operation, respectively):

- *Initialization:* To initialize an instance of FS-GAEAD, a party with ID ID runs $v \leftarrow \mathsf{Init}(k_e, n, \mathsf{ID})$, which takes as input a shared key k_e as well as the group size n and generates the initial state.

- *Send messages:* When a party wants to send a message m with associated data (AD) a (to the entire group) inside a FS-GAEAD instance, it runs $(v', e) \leftarrow \mathsf{Send}(v, a, m)$, which generates a ciphertext e encrypting and authenticating m and authenticating associated data a;
- *Receiving messages:* When a party receives a ciphertext e and AD a, it calls $(v', \mathsf{S}, i, m) \leftarrow \mathsf{Rcv}(v, a, e)$, which decrypts and verifies e to plaintext m and verifies a and also outputs a pair (S, i) consisting of sender ID S and message index i.

4.2.2 Security.

The security of FS-GAEAD is captured via a corresponding game, in which n parties share the same initial key. The attacker can have parties send and receive messages arbitrarily, ask for challenges, and leak the state of any party at any time. The game keeps track of the entire execution, but crucially of which messages have been received by which parties. When the state of some party ID is leaked, the set of messages received by ID is stored: these are the messages that must remain secure even given ID's leaked state. This information is used by a safety predicate ***FS-sec** to avoid trivial wins by $\mathcal{A}$, be it w.r.t. privacy or authenticity. More details can be found in Appendix E.1.

4.2.3 Construction.

We show how to build FS-GAEAD from a *forward-secure key-derivation function (FS-KDF)*. An FS-KDF keeps state s—initially set to a uniformly random string—and, upon request, derives keys corresponding to labels lab. After each such request, the FS-KDF also updates its own state. For each initial state, there a unique key corresponding to each label, irrespective of the order in which the labels were queried. The FS-KDF is forward-secret because even if its state is leaked, all keys output up to that point remain secure. An FS-KDF itself can be obtained via a tree construction based on a normal PRG.

Given an FS-KDF, the construction of FS-GAEAD is fairly straightforward: Messages and AD are encrypted/authenticated with a normal AEAD scheme, where the key for the i^{th} message sent be some party ID is derived using the label (ID, i). More information about the constructions can be found in Appendix F.

The safety predicate ***FS-sec** achieved by our construction consideres a message secure if no party's state is leaked before it receives the message.

Theorem 2. *Assuming $(t', \varepsilon_{\mathsf{ae}})$-secure authenticated encryption with associated data and $(t', \varepsilon_{\mathsf{kdf}})$-forward secure KDF, the scheme of Figure 27, is (t, q, ε')-FS-GAEAD secure w.r.t.* ***FS-sec***, where q is the number of challenge queries, $\varepsilon' = (q+1)t^2(\varepsilon_{\mathsf{kdf}} + \varepsilon_{\mathsf{ae}})$, and $t \approx t'$.*

4.3 PRF-PRNGs

A *PRF-PRNG* resembles both a pseudo-random function (PRF) and a pseudorandom number generator with input (PRNG)—hence the name. On the one hand, as a PRNG would, a PRF-PRNG (1) repeatedly accepts inputs I and context information C and uses them to refresh its state σ and (2) occasionally uses the state, provided it has sufficient entropy, to derive a pseudo-random pair of output R and new state; for the purposes of secure messaging, it suffices to combine properties (1) and (2) into a single procedure. On the other hand, a PRF-PRNG can be used as a PRF in the sense that if the state has high entropy, the answers to various pair (I, C) *on the same state* are indistinguishable from random and independent values.

4.3.1 Syntax.

A PRF-PNRG PP is an algorithm $(\sigma', R) \leftarrow \mathsf{PP}(\sigma, I, C)$: it takes the current state σ, absorbs input I along with context information C, and produces a new state σ' as well as an output string R.

4.3.2 Security.

The intuitive security properties for PRF-PRNGs mentioned at the beginning of this section must also hold in the presence of state compromise. In particular, a PRF-PRNG must satisfy PCFS (cf. Section 3.3) and be resilient to splitting-attacks. Therefore, the security game for PRF-PRNGs follows the same history-graph approach as the definitions of SGM and CKGA. However, since the game only consists of the state of the PRF-PRNG and there are no parties, it suffices to keep track of a much smaller amount information. Formal definitions for PRF-PRNGs are provided in Appendix G.1.

4.3.3 Construction.

A straight-forward construction of PRF-PRNGs is to model the algorithm PP as a random oracle. Such a construction achieves security w.r.t. safety predicate ***PP-secure**, which captures that in order for a value R in an epoch vid to be considered secure, vid

- must have an ancestor state vid' (possibly itself) that was reached via a random input I not known to $\mathcal{A}$, and
- there must have been no corruptions on the path from vid' to vid's ancestor.

More details about the construction can be found in Appendix G.2.

Theorem 3. *Let* ***PP-secure** *be the safety predicate above. Then,* PP *as a random oracle with outputs in* $\{0,1\}^\lambda$*, is an* (t, ε)*-secure PRF-PRNG w.r.t.* ***PP-secure***, where* $\varepsilon = \text{negl}(\lambda)$ *and* $t = \text{poly}(\lambda)$*.*

5 SGM Construction

Our construction is based on the following primitives: A CGKA scheme $\mathsf{K} = (\mathsf{K\text{-}Gen\text{-}IK}, \mathsf{K\text{-}Create}, \mathsf{K\text{-}Add}, \mathsf{K\text{-}Remove}, \mathsf{K\text{-}Update}, \mathsf{K\text{-}Commit}, \mathsf{K\text{-}Proc\text{-}Com}, \mathsf{K\text{-}Join})$, an FS-GAEAD scheme $\mathsf{F} = (\mathsf{F\text{-}Init}, \mathsf{F\text{-}Send}, \mathsf{F\text{-}Rcv})$, a CPA-secure public key encryption scheme $\mathsf{PKE} = (\mathsf{E\text{-}KeyGen}, \mathsf{E\text{-}Enc}, \mathsf{E\text{-}Dec})$ an existentially unforgeable signature scheme $\mathsf{S} = (\mathsf{S\text{-}KeyGen}, \mathsf{S\text{-}Sign}, \mathsf{S\text{-}Ver})$, a message authentication scheme $\mathsf{M} = (\mathsf{M\text{-}Tag}, \mathsf{M\text{-}Ver})$, a PRF-PRNG PP, a collision resistant hash function H. The initialization, PKI algorithms and helper functions of our construction are depicted on Figures 7,9, The comments on the figures explain the use of each variable and functionality. Below we describe the main SGM algorithms depicted on Figure 8, in which we use different colors to highlight the use of the underlying primitives: CGKA, FS-GAEAD, PKE, Signatures, MAC, PRF-PRNG, Hash.

Group creation. The group creation operation, Create, receives (possibly bad) randomness r, a signature verification key, spk, which is the key that will be used by the group members to verify messages sent by the group creator, as well as the welcome key material wpk. It adds the group creator's id to the roster ($\mathsf{s.G} \leftarrow [\mathsf{ME}]$), executes the CGKA group creation operation, absorbs the output I into the PRF-PRNG PP. PP outputs the new PRF-RPNG state $\mathsf{s}.\sigma$, the FS-GAEAD key k_e, a MAC key $\mathsf{s.k_m}$ (used to authenticate control messages), and the current epoch id, $\mathsf{s.C\text{-}epid}$.

SGM Construction: Initialization and PKI Algorithms

```
// Initialization
Init (ID)
    //  -- Global State --
    // Set Caller's ID
    ME ← ID
    // Stored Sig.  Keys
    SK-sk[.] ← $\varepsilon$
    // Stored Wel.  Keys
    WK-wk[.] ← $\varepsilon$
    // Contact List
    CL-KB[.] ← $\varepsilon$
    //  -- Group Specific State --
    // Buffered Props
    s.Props[.] ← $\varepsilon$
    // Roster
    s.G ← [.]
    // CGKA State
    s.$\gamma$ ← $\varepsilon$
    // PRF-PRG State
    s.$\sigma$ ← $\varepsilon$
    // FS-GAEAD States
    s.$v$[.] ← $\varepsilon$
    // Current Epoch ID
    s.C-epid ← $\varepsilon$
    // My Sig.  Key
    s.C-ssk ← $\varepsilon$
    // Verif.  Keys
    s.Ep-SPK[., .] ← $\varepsilon$
    // MAC Key
    s.$k_m$ ← $\varepsilon$
```

```
// Generate Signature Keys
Gen-SK
    (spk, ssk) ← S-KeyGen
    SK-sk[spk] ← ssk
    return spk

// Delete Signing Key
Rem-SK (spk)
    SK-sk[spk] ← $\varepsilon$

// Store Signing Key for Contact
Get-SK (ID, spk)
    CL-S[ID] +← spk

// Generate Key Bundle
Gen-KB (spk)
    (ipk, isk) ← K-Gen-IK
    (epk, esk) ← E-KeyGen
    (wpk, wsk) ← ((epk, ipk), (esk, isk))
    WK-wk[wpk] ← wsk
    ssk ← SK-sk[spk]
    sig ← S-Sign(ssk, wpk)
    return (wpk, spk, sig)

// Store Key Bundle of a Contact
Get-KB (ID$'$, kb)
    (wpk, spk, sig) ← kb
    req spk ∈ CL-S[ID$'$]
    req S-Ver(spk, wpk, sig)
    CL-KB[ID$'$].enq((wpk, spk))
```

Figure 7: The SGM Construction : Initialization and PKI Algorithms.

Next, the FS-GEAD init operation F-Init, is executed with inputs k_e, the group size, which is 1, and the id ME of the group creator.

Proposals. To add a party ID_a, Add recovers the key bundle kb$'$ for that ID from the contact list, and runs the CGKA add-proposal algorithm with keys from kb$'$, getting a CGKA proposal $\bar{\mathrm{P}}$, which it then uses to construct and authenticate (with MAC and signature) the SGM proposal message P'. Remove and Update are similar (where Update takes the new signing key of the updater as input). When processing any proposal, algorithm Proc-PM simply attempts to authenticate the proposal and stores it locally. Proc-PM returns proposal information: the operation op, the origin of the proposal, orig, and the data data, as computed by `*get-propInfo`(P') (cf. Figure 9).

Committing. To commit to a set $\mathbf{P}$ of proposals, Commit calls the CGKA commit operation to obtain CGKA welcome messages W_{pub} and **WPrv** as well as CGKA control message $\bar{\mathrm{T}}$ and update secret I. $\bar{\mathrm{T}}$ is used to construct and authenticate (with MAC and signature) the SGM control message T' (which includes a hash of the proposals). Then, Commit creates and authenticates (with MAC and signature) the SGM welcome messages for joining parties by adding an encryption of the PRF-PRNG state s.σ to the CGKA welcome messages. The MAC key used for authenticating control and welcome messages is derived by absorbing I into the PRF-PRNG (with context information that depends on T'). Note that s.σ is not updated yet, however.

Process commit/welcome messages. Algorithm Proc-CM first verifies the authentication information (MAC and signatures) of a commit message $\mathrm{T}' = (\texttt{"com"}, \mathsf{epid}, \mathsf{ID}, \mathbf{h}, \bar{\mathrm{T}})$. However, in order to obtain the MAC key, (1) the CGKA control message $\bar{\mathrm{T}}$ has to be processed by Proc-Com, which

(2) recovers the update secret I, which in turn (3) must be absorbed into the PRF-PRNG (this time updating its state). The last step also produces the shared key for the new FS-GAEAD session.

Algorithm $\mathsf{Proc\text{-}WM}$, used by joining parties to processes welcome messages W' proceeds similarly, except that the state of the PRF-PRNG must first be obtained by decrypting the corresponding ciphertext in W'.

Send message. To send associated data a and plaintext m, Send passes a and m to the current epoch's FS-GAEAD send operation, and, additionally, also signs the resulting ciphertext, a, and the identifier $\mathsf{s.C\text{-}epid}$ of the current epoch.

Receive message. The operation receives associated data a and ciphertext e, parses the ciphertext $(e', \mathrm{sig}) \leftarrow e$, $(\mathsf{epid}, a, \bar{e}) \leftarrow e'$, executes the receive FS-GAEAD operation, $(\mathsf{s}.v[\mathsf{epid}], \mathsf{ID}_s, i, m) \leftarrow \mathsf{F\text{-}Rcv}(\mathsf{s}.v[\mathsf{epid}], (\mathsf{epid}, a), \bar{e})$, checks whether $\mathsf{ID}_s \neq \perp$ and $\mathsf{S\text{-}Ver}(\mathsf{spk}, e', \mathrm{sig})$, and outputs $(\mathsf{epid}, \mathsf{ID}_s, i, m)$.

Helpers. `*added` receives a vector of proposals $\mathbf{P}$ and returns the ids and welcome public keys of newly added members. `*new-spks` receives an epoch id and vector of proposals, $(\mathsf{epid}, \mathbf{P})$, and returns the public verification keys for the ids affected by the proposals in $\mathbf{P}$. `*roster-pos` returns the position of ID in G, and `*get-propInfo` receives a proposal P' and returns the proposal information, namely the operation, op, the proposal origin, orig, and the data, data, where if $\mathsf{op} = \mathsf{add}$, data holds the id, ID_a, welcome key material, wpk, and the signature verification key spk, of the newly added member. If $\mathsf{op} = \mathtt{rem}$, data holds the id of the removed party ID_r, and if $\mathsf{op} = \mathsf{upd}$, data holds the updated verification key spk.

5.0.1 Security.

Main idea: The authenticity property of our construction relies on the EU-CMA security of signatures, the unforgeability of MACs, and the authenticity of the FS-GAEAD scheme. In particular, if the sender's signing key is secure, then an injection w.r.t. that key fails with overwhelming probability. If this is not the case, an honest commit operation provides post-compromise authenticity, by producing a secure CGKA update secret, I, which feeds the PRF-PRNG with good randomness, which in turn outputs secure MAC and FS-GAEAD keys, used for the authentication of future control (via MAC security) and application (via FS-GAEAD security) messages (this requires the adversary to remain passive for one epoch). PCFS with respect to privacy is similar and relies on the PCFS security CGKA and the FS of FS-GAEAD.

Simplified properties. Proving SGM security is facilitated by considering three simplified properties, namely *correctness*, *authenticity*, and *privacy*. We prove that these properties together imply full SGM security, as stated in Section H.1 and proved formally in Theorem 25. Besides modularity, simplified properties facilitate the transition from *selective* to *fully adaptive* security, as the individual "simplified" games, defined in Section H.1, consider selective adversaries that commit to the challenge (e.g., the challenge or the last healing, epoch, the message sender and index used for the challenge) at the beginning of the game. In the reduction from the simplified properties to full SGM security, the adversarial strategy is being guessed and the success probability is bounded by values that relate to the running time of the adversary. After proving the simplified properties theorem, one can prove *authenticity* and *privacy*, individually, against selective adversaries that commit to their strategy before the security game begins.

Safety predicates. Using safety predicates, we provide a generic theorem (cf. Theorem 4), that considers *any* CGKA, FS-GAEAD, and PRF-PRNG scheme. Those schemes come along with their security predicates, Π_{CGKA}, Π_{FS} and Π_{PP}, respectively, and we prove that as long as the attacker's actions are not violating those predicates, then the resulting SGM construction is secure w.r.t. the SGM predicate $\Pi_{\mathsf{SGM}} = \Pi_{\mathsf{SGM}}(\Pi_{\mathsf{CGKA}}, \Pi_{\mathsf{FS}}, \Pi_{\mathsf{PP}})$. Our SGM safety predicate Π_{SGM} (explained below) is depicted in Figure 10 and operates over history graph information, generated by the SGM security game. Here, we consider Π_{CGKA} = ***CGKA-priv**, Π_{FS} = ***FS-sec**, Π_{PP} = ***PP-secure** (cf. Figure 10).

Proof idea. The adversary breaks authenticity if it manages to make a non-trivial injection, which implies that either of the following holds: (1) injects a (proposal, welcome, commit or key bundle) message in epoch vid that is signed with a non-compromised signing key of the party ID (determined by ***SK-compr**($\mathsf{vid}, \mathsf{ID}$)), (2) injects a (proposal, welcome, commit) message when ***auth-compr**(vid) holds, which implies that ***PP-secure**(vid, ***Proj-PP**($\mathsf{SGM\text{-}Data}$)), i.e., PP is secure, and (3) injects an application message when the FS-GAEAD state is not trivially compromised (determined by ***FS-sec**). Clearly, security against (1) reduces to the EU-CMA security of S. For (2) the output of the PRF-PRNG, PP, in epoch vid is secure, therefore that MAC key (output by PP) is secure and we can rely on the security of PP and the unforgeability of M. This requires the following hybrids: (A) In the first hybrid, if the attacker finds a collision against H, the execution aborts (reduces to the collision resistance property of H). This hybrid is required for the protection of the PP state in the presence of group splitting attacks in which the adversary can split the group, corrupt in one branch to recover the PP state, and challenge on another branch in which the PP state is related to the corrupted one. Here, collision resistance ensures that different control messages have different hash values, therefore lead to independent PP states. (B) In the next hybrid, in the last healing epoch before the challenge, substitute the CGKA update secret (which feeds PP) with a uniformly random value (required for the reduction to the PRF-PRNG security). Note that, by the definitions of ***PP-secure**, ***Proj-PP**, a good healing epoch before the challenge epoch, exists, and this epoch satisfies ***CGKA-priv**. (C) In the next hybrid we use the CPA security of PKE. In particular, in the commit operation that creates the target epoch, encrypt the zero message instead of encrypting the PP state, as part of the welcome message (requires a reduction to the CPA security of PKE). (D) Next, substitute the output of the PRF-PRNG PP in a commit message for the target epoch, with a uniformly random value (reduces to PRF-PRNG security). Finally, since the output of PP is substituted by a uniformly random value, so does that MAC key, thus we have a reduction to the unfogeability of the MAC scheme M for case (2). Case (3) is similar, however in the last step we have a reduction to the FS-GAEAD authenticity property of F. For privacy we consider the same sequence of hybrids, however the final reduction is against the FS-GAEAD privacy property of F. For that reduction we use the fact that ***FS-sec** is satisfied. We show (cf. Section H):

SGM Construction: Main Algorithms

```
// Create a group
Create $(\mathsf{spk}, \mathsf{wpk}; r)$
    $\mathsf{s.G} \leftarrow [\mathsf{ME}]$
    $\mathsf{wsk} \leftarrow \mathsf{WK\text{-}wk}[\mathsf{wpk}]$
    $(\cdot, \mathsf{isk}) \leftarrow \mathsf{wsk}$
    $(\mathsf{s}.\gamma, I) \leftarrow \mathsf{K\text{-}Create}(\mathsf{ME}, \mathsf{isk}; r)$
    $(\mathsf{s}.\sigma, k_e, \mathsf{s.k_m}, \mathsf{s.C\text{-}epid}) \leftarrow \mathsf{PP}(0, I, 0)$
    $\mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, \mathsf{ME}] \leftarrow \mathsf{spk}$
    $\mathsf{s}.v[\mathsf{s.C\text{-}epid}] \leftarrow \mathsf{F\text{-}Init}(k_e, 1, \mathsf{ME})$

// Add proposal
Add $(\mathsf{ID}_a)$
    $\mathsf{kb}' \leftarrow \mathsf{CL\text{-}KB}[\mathsf{ID}_a]$
    $((\mathsf{epk}, \mathsf{ipk}), \mathsf{spk}) \leftarrow \mathsf{kb}'$
    $(\mathsf{s}.\gamma, \bar{\mathrm{P}}) \leftarrow \mathsf{K\text{-}Add}(\mathsf{s}.\gamma, \mathsf{ID}_a, \mathsf{ipk})$
    $\mathrm{P}' \leftarrow (\texttt{"add"}, \mathsf{s.C\text{-}epid}, \mathsf{ME}, (\mathsf{ID}_a, \mathsf{kb}'), \bar{\mathrm{P}})$
    $t \leftarrow \mathsf{M\text{-}Tag}(\mathsf{s.k_m}, \mathrm{P}')$
    $\mathrm{sig} \leftarrow \mathsf{S\text{-}Sign}(\mathsf{s.C\text{-}ssk}, (\mathrm{P}', t))$
    return $(\mathrm{P}', t, \mathrm{sig})$

// Remove proposal
Remove $(\mathsf{ID}_r)$
    $(\mathsf{s}.\gamma, \bar{\mathrm{P}}) \leftarrow \mathsf{K\text{-}Remove}(\mathsf{s}.\gamma, \mathsf{ID}_r)$
    $\mathrm{P}' \leftarrow (\texttt{"rem"}, \mathsf{s.C\text{-}epid}, \mathsf{ME}, \mathsf{ID}_r, \bar{\mathrm{P}})$
    $t \leftarrow \mathsf{M\text{-}Tag}(\mathsf{s.k_m}, \mathrm{P}')$
    $\mathrm{sig} \leftarrow \mathsf{S\text{-}Sign}(\mathsf{s.C\text{-}ssk}, (\mathrm{P}', t))$
    return $(\mathrm{P}', t, \mathrm{sig})$

// Update proposal
Update $(\mathsf{spk}; r)$
    req $\mathsf{SK\text{-}sk}[\mathsf{spk}] \neq \varepsilon$
    $(\mathsf{s}.\gamma, \bar{\mathrm{P}}) \leftarrow \mathsf{K\text{-}Update}(\mathsf{s}.\gamma; r)$
    $\mathrm{P}' \leftarrow (\texttt{"upd"}, \mathsf{s.C\text{-}epid}, \mathsf{ME}, \mathsf{spk}, \bar{\mathrm{P}})$
    $t \leftarrow \mathsf{M\text{-}Tag}(\mathsf{s.k_m}, \mathrm{P}')$
    $\mathrm{sig} \leftarrow \mathsf{S\text{-}Sign}(\mathsf{s.C\text{-}ssk}, (\mathrm{P}', t))$
    return $(\mathrm{P}', t, \mathrm{sig})$

// Commit
Commit $(\boldsymbol{P}; r)$
    req $\mathbf{P} \subseteq \mathsf{s.Props}$
    // Get CGKA Proposals
    $\bar{\mathbf{P}} \leftarrow \mathbf{P}.\bar{\mathrm{P}}$
    //  -- Prepare Commit Message --
    $(\mathsf{s}.\gamma, W_{\mathsf{pub}}, \mathbf{WPrv}, \bar{\mathrm{T}}, I) \leftarrow \mathsf{K\text{-}Commit}(\mathsf{s}.\gamma, \bar{\mathbf{P}}; r)$
    $\mathrm{T}' \leftarrow (\texttt{"com"}, \mathsf{s.C\text{-}epid}, \mathsf{ME}, \mathsf{H}(\mathbf{P}), \bar{\mathrm{T}})$
    $\mathsf{v} \leftarrow \mathsf{H}(\mathrm{T}')$
    // New MAC Key & Epoch ID
    $(., ., k_m, \mathsf{epid}) \leftarrow \mathsf{PP}(\mathsf{s}.\sigma, I, \mathsf{v})$
    $t \leftarrow \mathsf{M\text{-}Tag}(k_m, \mathrm{T}')$
    $\mathrm{sig} \leftarrow \mathsf{S\text{-}Sign}(\mathsf{s.C\text{-}ssk}, (\mathrm{T}', t))$
    // Commit Message
    $\mathrm{T} \leftarrow (\mathrm{T}', t, \mathrm{sig})$
    //  -- Prepare Welcome Messages --
    $(\mathbf{ID}, \mathbf{wpk}) \leftarrow$ *added$(\mathbf{P})$
    $\mathbf{spk} \leftarrow$ *new-spks$(\mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, \cdot], \mathbf{P})$
    for $i \in \mathbf{ID}$
        $e \leftarrow \mathsf{E\text{-}Enc}(\mathbf{wpk}[i].\mathsf{epk}, \mathsf{s}.\sigma)$
        $\mathrm{W}' \leftarrow (\texttt{"wel"}, \mathsf{ME}, \mathbf{ID}[i], \mathsf{v}, W_{\mathsf{pub}}, \ldots$
        $\ldots, \mathbf{WPrv}[i], e, \mathbf{wpk}[i], \mathbf{spk})$
        $t \leftarrow \mathsf{M\text{-}Tag}(k_m, (\mathrm{W}', \mathsf{epid}))$
        $\mathrm{sig} \leftarrow \mathsf{S\text{-}Sign}(\mathbf{spk}[\mathsf{ME}], (\mathrm{W}', t))$
        // Welcome Message
        $\mathbf{W}[i] \leftarrow (\mathrm{W}', t, \mathrm{sig})$
    return $(\mathsf{epid}, \mathbf{W}, \mathrm{T})$

// Process a proposal
Proc-PM $(P)$
    $(\mathrm{P}', t, \mathrm{sig}) \leftarrow \mathrm{P}$
    $(., \mathsf{epid}, \mathsf{ID}, ., .) \leftarrow \mathrm{P}'$
    req $\mathsf{epid} = \mathsf{s.C\text{-}epid}$
    $\mathsf{spk} \leftarrow \mathsf{s.Ep\text{-}SPK}[\mathsf{epid}, \mathsf{ID}]$
    req $\mathsf{M\text{-}Ver}(\mathsf{s.k_m}, \mathrm{P}', t)$
    req $\mathsf{S\text{-}Ver}(\mathsf{spk}, (\mathrm{P}', t), \mathrm{sig})$
    $\mathsf{s.Props} \mathrel{+}\!\!\leftarrow \mathrm{P}'$
    return *get-propInfo$(\mathrm{P}')$

// Process Commit Message
Proc-CM $(T)$
    $(\mathrm{T}', t, \mathrm{sig}) \leftarrow \mathrm{T}$
    $(., \mathsf{epid}, \mathsf{ID}, \mathbf{h}, \bar{\mathrm{T}}) \leftarrow \mathrm{T}'$
    // Matching Epochs?
    req $\mathsf{epid} = \mathsf{s.C\text{-}epid}$
    $\mathsf{spk} \leftarrow \mathsf{s.Ep\text{-}SPK}[\mathsf{epid}, \mathsf{ID}]$
    req $\mathsf{S\text{-}Ver}(\mathsf{spk}, (\mathrm{T}', t), \mathrm{sig})$
    $\mathbf{P} \leftarrow \mathsf{s.Props}[\mathbf{h}]$
    // Call CGKA
    $(\mathsf{s}.\gamma, \mathsf{GI}, I) \leftarrow \mathsf{K\text{-}Proc\text{-}Com}(\mathsf{s}.\gamma, \bar{\mathrm{T}})$
    $(\mathsf{s}.\sigma, k_e, \mathsf{s.k_m}, \mathsf{s.C\text{-}epid}) \leftarrow \mathsf{PP}(\mathsf{s}.\sigma, I, \mathsf{H}(\mathrm{T}'))$
    req $\mathsf{M\text{-}Ver}(\mathsf{s.k_m}, \mathrm{T}', t)$
    $\mathsf{s.G} \leftarrow \mathsf{GI.G}$
    $pos \leftarrow$ *roster-pos$(\mathsf{ME}, \mathsf{s.G})$
    // Start FS-GAEAD
    $\mathsf{s}.v[\mathsf{s.C\text{-}epid}] \leftarrow \mathsf{F\text{-}Init}(k_e, |\mathsf{s.G}|, pos)$
    $\mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, .] \leftarrow$ *new-spks$(\mathsf{s.Ep\text{-}SPK}[\mathsf{epid}, \cdot], \mathbf{P})$
    $\mathsf{spk} \leftarrow \mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, \mathsf{ME}]$
    $\mathsf{s.C\text{-}ssk} \leftarrow \mathsf{SK\text{-}sk}[\mathsf{spk}]$
    return $\mathsf{GI}$

// Join a Group
Proc-WM $(W)$
    $(\mathrm{W}', t, \mathrm{sig}) \leftarrow \mathrm{W}$
    $(., \mathsf{ID}_s, ., \mathsf{v}, W_{\mathsf{pub}}, W_{\mathsf{priv}}, e, \mathsf{wpk}, \mathbf{spk}) \leftarrow \mathrm{W}'$
    $\mathsf{spk} \leftarrow \mathbf{spk}[\mathsf{ID}_s]$
    req $\mathsf{spk} \in \mathsf{CL\text{-}S}[\mathsf{ID}_s] \;\wedge\; \mathsf{S\text{-}Ver}(\mathsf{spk}, \mathrm{W}', \mathrm{sig})$
    $(\mathsf{esk}, \mathsf{isk}) \leftarrow \mathsf{WK\text{-}wk}[\mathsf{wpk}]$
    $\mathsf{s}.\sigma \leftarrow \mathsf{E\text{-}Dec}(\mathsf{esk}, e)$
    $(\mathsf{s}.\gamma, \mathsf{GI}, I) \leftarrow \mathsf{K\text{-}Join}(\mathsf{ME}, \mathsf{ID}_s, W_{\mathsf{pub}}, W_{\mathsf{priv}}, \mathsf{isk})$
    $(\mathsf{s}.\sigma, k_e, \mathsf{s.k_m}, \mathsf{s.C\text{-}epid}) \leftarrow \mathsf{PP}(\mathsf{s}.\sigma, I, \mathsf{v})$
    req $\mathsf{M\text{-}Ver}(\mathsf{s.k_m}, (\mathrm{W}', \mathsf{s.C\text{-}epid}), t)$
    $\mathsf{s.G} \leftarrow \mathsf{GI.G}$
    $pos \leftarrow$ *roster-pos$(\mathsf{ME}, \mathsf{s.G})$
    $\mathsf{s}.v[\mathsf{s.C\text{-}epid}] \leftarrow \mathsf{F\text{-}Init}(k_e, |\mathsf{s.G}|, pos)$
    $\mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, .] \leftarrow \mathbf{spk}$
    $\mathsf{spk} \leftarrow \mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, \mathsf{ME}]$
    $\mathsf{s.C\text{-}ssk} \leftarrow \mathsf{SK\text{-}sk}[\mathsf{spk}]$
    $\mathsf{s.Props} \leftarrow \varepsilon$

// Send A Message
Send $(a, m)$
    $E \leftarrow \mathsf{s.C\text{-}epid}$
    $(\mathsf{s}.v[E], \bar{e}) \leftarrow \mathsf{F\text{-}Send}(\mathsf{s}.v[E], (E, a), m)$
    $e' \leftarrow (E, a, \bar{e})$
    $\mathrm{sig} \leftarrow \mathsf{S\text{-}Sign}(\mathsf{s.C\text{-}ssk}, e')$
    return $(e', \mathrm{sig})$

// Receive A Message
Rcv $(a, e)$
    $(e', \mathrm{sig}) \leftarrow e$
    $(\mathsf{epid}, a, \bar{e}) \leftarrow e'$
    $(\mathsf{s}.v[\mathsf{epid}], \mathsf{ID}_s, i, m) \leftarrow \mathsf{F\text{-}Rcv}(\mathsf{s}.v[\mathsf{epid}], (\mathsf{epid}, a), \bar{e})$
    req $\mathsf{ID}_s \neq \perp$
    $\mathsf{spk} \leftarrow \mathsf{s.Ep\text{-}SPK}[\mathsf{epid}, \mathsf{ID}_s]$
    req $\mathsf{S\text{-}Ver}(\mathsf{spk}, e', \mathrm{sig})$
    return $(\mathsf{epid}, \mathsf{ID}_s, i, m)$
```

Figure 8: The SGM Construction: main algorithms.

SGM Construction: Helper Functions

```
// Returns ids, wpks, of new parties
*added ($\boldsymbol{P}$)
    $\mathbf{ID}[\cdot] \leftarrow \varepsilon$
    $\mathbf{wpk}[\cdot] \leftarrow \varepsilon$
    $i \leftarrow 1$ for $P \in \boldsymbol{P}$
        if $P = (\text{"add"}, \cdot, \cdot, (\mathsf{ID}_a, \mathsf{kb}'), \cdot)$
            $(\mathsf{wpk}, \cdot) \leftarrow \mathsf{kb}'$
            $\mathbf{ID}[i] \leftarrow \mathsf{ID}_a$
            $\mathbf{wpk}[i] \leftarrow \mathsf{wpk}$
            $i$++
    return $(\mathbf{ID}, \mathbf{wpk})$

// Returns spks after applying props
*new-spks (epid, $\boldsymbol{P}$)
    $\mathbf{spk} \leftarrow$ s.Ep-SPK[epid]
    for $P \in \boldsymbol{P}$
        if $P = (\text{"add"}, \cdot, \cdot, (\mathsf{ID}_a, \mathsf{kb}'), \cdot)$
            $(\cdot, \mathsf{spk}) \leftarrow \mathsf{kb}'$
            $\mathbf{spk}[\mathsf{ID}_a] \leftarrow \mathsf{spk}$
        if $P = (\text{"rem"}, \cdot, \cdot, \mathsf{ID}_r, \cdot)$
            $\mathbf{spk}[\mathsf{ID}_r] \leftarrow \varepsilon$
        if $P = (\text{"upd"}, \cdot, \mathsf{ID}_u, \mathsf{spk}, \cdot)$
            $\mathbf{spk}[\mathsf{ID}_u] \leftarrow \mathsf{spk}$
    return $\mathbf{spk}$

// Returns position of ID in roster
*roster-pos (ID, G)
    $[\mathsf{ID}_1, \ldots, \mathsf{ID}_n] \leftarrow \mathsf{G}$
    for $i \in [1, n]$
        if $\mathsf{ID}_i = \mathsf{ID}$
            return $i$

// Returns Proposal Info
*get-propInfo ($P'$)
    if $P' = (\text{"add"}, \cdot, \mathsf{ID}_s, (\mathsf{ID}_a, \mathsf{kb}'), \bar{P})$
        op = add
        orig = $\mathsf{ID}_s$
        $(\mathsf{wpk}, \mathsf{spk}) \leftarrow \mathsf{kb}'$; data = $(\mathsf{ID}_a, \mathsf{wpk}, \mathsf{spk})$
    if $P' = (\text{"rem"}, \cdot, \mathsf{ID}_s, \mathsf{ID}_r, \bar{P})$
        op = rem
        orig = $\mathsf{ID}_s$
        data = $\mathsf{ID}_r$
    if $P' = (\text{"upd"}, \cdot, \mathsf{ID}_s, \mathsf{spk}, \bar{P})$
        op = upd
        orig = $\mathsf{ID}_s$
        data = spk
    return (op, orig, data)
```

Figure 9: The SGM Construction : Helper Algorithm.

Safety Helpers for SGM Construction

```
// Determines if authenticity
// of epoch vid compromised
*auth-compr (vid)
    return $\neg$*PP-secure(vid, *Proj-PP(SGM-Data))

// Determines if ID's signature
// key is compromised in epoch vid
*SK-compr (vid, ID)
    chk HG.getSKIDs(vid, ID) $\cap$ SK-Lk = $\emptyset$

// Privacy Predicate
*AM-sec (vid, S, $i$)
    if *PP-secure(vid, *Proj-PP(SGM-Data))
        return *FS-sec((S, $i$), *Proj-FS(SGM-Data, vid))
    return false

// Project History Graph to PRF-PRNG Game
*Proj-PP (SGM-Data = ($\mathcal{V}$, $\mathcal{P}$, V-Lk, AM-Lk, BR, WK-Lk))
    for (vid, orig, data, pid) $\in \mathcal{V}$
        $\mathcal{V}'$ +← vid
    for vid $\in \mathcal{V}'$
        BI[vid] ← $\neg$*CGKA-priv(vid', *Proj-CGKA(SGM-Data))
    return ($\mathcal{V}'$, V-Lk, BI)

// Project History Graph to CGKA Game
*Proj-CGKA (SGM-Data = ($\mathcal{V}$, $\mathcal{P}$, V-Lk, AM-Lk, BR, WK-Lk))
    for (vid, orig, data, pid) $\in \mathcal{V}$
        $\mathcal{V}'$ +← (vid, orig, pid)
    for $p$ = (pid, vid, orig, op, data) $\in \mathcal{V}$
        if op = add
            let data = (ID', wkid', skid')
            data' ← (ID', wkid')
            $p'$ ← (pid, vid, op, orig, data')
        if op = rem
            $p' \leftarrow p$
        if op = upd
            $p' \leftarrow \perp$
        $\mathcal{P}'$ +← $p'$
    return ($\mathcal{V}'$, $\mathcal{P}'$, V-Lk, BR, WK-Lk)

// Project History Graph to FS-GAEAD Game
*Proj-FS (SGM-Data = ($\mathcal{V}$, $\mathcal{P}$, V-Lk, AM-Lk, BR, WK-Lk), vid)
    for (vid, ID, AM-Rcvd[ID], AM-Tr[ID]) $\in$ AM-Lk
        AM-Lk' +← (ID, AM-Rcvd[ID], AM-Tr[ID])
    return AM-Lk'
```

Figure 10: Safety oracles and safety predicate of the security game for SGM schemes.

Theorem 4. *Let* $\mathsf{SGM} = \mathsf{SGM}(\mathsf{K}, \mathsf{F}, \mathsf{PKE}, \mathsf{S}, \mathsf{M}, \mathsf{PP}, \mathsf{H})$ *be the SGM scheme presented above and let* $\Pi_{\mathsf{SGM}} = \Pi_{\mathsf{SGM}}(\Pi_{\mathsf{CGKA}}, \Pi_{\mathsf{FS}}, \Pi_{\mathsf{PP}})$ *be predicates such that: (1)* K *is an* $(t, q, \varepsilon_{\mathsf{CGKA}})$*-secure CGKA scheme with respect to predicate* Π_{CGKA}*, (2)* F *is an* $(t, \varepsilon_{\mathsf{FS}})$*-secure FS-GAEAD scheme with respect to predicate* Π_{FS}*, (3)* PKE *is an* $(t, \varepsilon_{\mathsf{PKE}})$*-CPA secure public-key encryption scheme, (4)* S *is an* $(t, \varepsilon_{\mathsf{sig}})$*-existentially unforgeable signature scheme, (5)* M *is an* $(t, \varepsilon_{\mathsf{mac}})$*-secure message authentication code, (6)* PP *be an* $(t, \varepsilon_{\mathsf{PP}})$*-secure PRF-PRNG with respect to predicate* Π_{PP}*, (7)* H *is an* $(t, \varepsilon_{\mathsf{crh}})$*-collision resistant hash. Then,* SGM *is a* $(t', \varepsilon_{\mathsf{SGM}})$*-secure SGM scheme with respect to predicate* Π_{SGM}*, where* $t' \approx t$ *and* $\varepsilon_{\mathsf{SGM}} \ = \ 2qt^4 \cdot (\varepsilon_{\mathsf{sig}} + \varepsilon_{\mathsf{CR}} + \varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{PKE}} + \varepsilon_{\mathsf{PP}} + \varepsilon_{\mathsf{mac}} + \varepsilon_{\mathsf{FS}})$.

References

[1] J. Alwen, M. Capretto, M. Cueto, C. Kamath, K. Klein, G. Pascual-Perez, K. Pietrzak, and M. Walter. Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol. *IACR Cryptol. ePrint Arch.*, 2019:1489, 2019.

[2] J. Alwen, S. Coretti, and Y. Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.

[3] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, Aug. 2020.

[4] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk. Continuous group key agreement with active security. In R. Pass and K. Pietrzak, editors, *TCC 2020*, 2020.

[5] J. Alwen, D. Jost, and M. Mularczyk. On the insider security of MLS. *IACR Cryptol. ePrint Arch.*, 2020:1327, 2020.

[6] R. Barnes. Subject: [MLS] Remove without double-join (in TreeKEM), 2018. `https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik`.

[7] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-11, Internet Engineering Task Force, Dec. 2020. Work in Progress.

[8] K. Bhargavan, B. Beurdouche, and P. Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, Dec. 2019.

[9] A. Bienstock, Y. Dodis, and P. Rösler. On the price of concurrency in group ratcheting protocols. In R. Pass and K. Pietrzak, editors, *TCC*, 2020.

[10] C. Brzuska, E. Cornelissen, and K. Kohbrok. Cryptographic security of the mls rfc, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021.

[11] R. Canetti, J. A. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM'99*, pages 708–716, New York, NY, USA, Mar. 21–25, 1999.

[12] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, Oct. 2018.

[13] K. Cohn-Gordon, C. J. F. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017*, pages 451–466, 2017.

[14] C. Cremers, B. Hale, and K. Kohbrok. Revisiting post-compromise security guarantees in group messaging. *IACR Cryptol. ePrint Arch.*, 2019:477, 2019.

[15] Y. Dodis and N. Fazio. Public key broadcast encryption for stateless receivers. In J. Feigenbaum, editor, *Digital Rights Management*, pages 61–80, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[16] E. Eaton, D. Jao, , and C. Komlo. Towards post-quantum updatable public-key encryption via supersingular isogenies. Cryptology ePrint Archive, Report 2020/1593, 2020. `https://eprint.iacr.org/2020/1593`.

[17] A. Fiat and M. Naor. Broadcast encryption. In D. R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 480–491. Springer, Heidelberg, Aug. 1994.

[18] Z. Jafargholi, C. Kamath, K. Klein, I. Komargodski, K. Pietrzak, and D. Wichs. Be adaptive, avoid overcommitting. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 133–163. Springer, Heidelberg, Aug. 2017.

[19] D. Jost, U. Maurer, and M. Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Heidelberg, May 2019.

[20] Y. Kim, A. Perrig, and G. Tsudik. Group key agreement efficient in communication. *IEEE Trans. Computers*, 53(7):905–921, 2004.

[21] Matthew A. Weidner. Group Messaging for Secure Asynchronous Collaboration. Master's thesis, University of Cambridge, June 2019.

[22] S. Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of ACM SIGCOMM*, pages 277–288, Cannes, France, Sept. 14–18, 1997.

[23] S. Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In S. P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 21–40. Springer, Heidelberg, Feb. 2007.

[24] E. Rescorla. Subject: [MLS] TreeKEM: An alternative to ART. MLS Mailing List, 2018. `https://mailarchive.ietf.org/arch/msg/mls/WRdXVr8iUwibaQu0tH6sDnqU1no.`

[25] D. G. Steer, L. Strawczynski, W. Diffie, and M. J. Wiener. A secure audio teleconference system. In S. Goldwasser, editor, *CRYPTO '88*, 1988.

[26] D. Wallner, E. Hardner, and R. Agee. Key management for multicast: Issues and architectures. IETF RFC2676, 1999. `https://tools.ietf.org/html/rfc2627`.

[27] M. Weidner. Group messaging for secure asynchronous collaboration. MPhil Dissertation, 2019. `https://mattweidner.com/acs-dissertation.pdf`.

[28] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, Feb. 2000.

A Preliminaries

A.1 Updatable Public Key Encryption

In this section we revisit the definition of Updatable Public Key Encryption (UPKE), as defined by [3].

Definition 2. *An* updatable public-key encryption *(UPKE) scheme is a triple of algorithms* $\mathsf{UPKE} = (\mathsf{UKGen}, \mathsf{UEnc}, \mathsf{UDec})$ *with the following syntax:*

- Key generation: UKGen *receives a uniformly random key* sk_0 *and outputs a fresh initial public key* $\mathsf{pk}_0 \leftarrow \mathsf{UKGen}(\mathsf{sk}_0)$.
- Encryption: UEnc *receives a public key* pk *and a message* m *and produces a ciphertext* e *and a new public key* pk'.
- Decryption: UDec *receives a secret key* sk *and a ciphertext* e *and outputs a message* m *and a new secret key* sk'.

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

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

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

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

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

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

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

B Continuous Group Key Agreement

B.1 Security

CGKA schemes must satisfy *correctness*, i.e., all group members output the same keys in every epoch. Furthermore, the keys must be private, and the CGKA scheme must satisfy post-compromise forward secrecy (PCFS).

Similarly to the SGM game, the CGKA game allows adversary $\mathcal{A}$ to control the execution of and attack a single group. In particular, $\mathcal{A}$ controls who creates the group, who is added and removed, who updates, who commits, etc. The attacker is also allowed to leak the state of any party (whether currently part of the group or not) at any time. The privacy of keys is captured by considering a challenge that outputs either the actual key output by the CGKA scheme or a truly random one—which one is determined by an internal random bit b, which must be guessed by $\mathcal{A}$ at the end of the game.

B.1.1 Bookkeeping.

Similarly to SGM schemes, the CGKA security game keeps track of all the relevant execution data with the help of a so-called *history graph*, and the recorded data informs a *safety predicate* (pertaining to privacy) evaluated at the end of the game. The following paragraphs outline the differences in bookkeeping between the CGKA game and the SGM game.

History graphs. A history graph is again a directed tree whose nodes correspond to the group state in the various epochs of an execution. As with the SGM game, there are three types of nodes: v_{root}, v_{create} nodes, and commit nodes v. A node consists of the following values:

$$v = (\mathsf{vid}, \mathsf{orig}, \mathbf{pid}) ,$$

where

- vid is the node's (unique) ID,
- orig is the party that caused the node's creation, i.e., orig is either the group creator or the committer,
- $\mathbf{pid}$ is the vector of (IDs of) proposals (see below) that were included in the commit.

The history graph is accessed by the security-game oracles via the "HG object" HG, which provides information via several methods (explained later as they are needed). The (ID of the) node corresponding to a party's current state is stored by the array $\mathsf{V\text{-}Pt}[\mathsf{ID}]$.

Proposals. Similarly to HG and the history graph, the object Props keeps track of proposals. The information recorded about each proposal is a vector

$$p = (\mathsf{pid}, \mathsf{vid}, \mathsf{op}, \mathsf{orig}, \mathsf{data}) ,$$

where

- pid is the proposal's (unique) ID,
- vid is the (ID of) the HG node corresponding to the epoch in which the proposal was created,
- $\mathsf{op} \in \{\mathtt{add}, \mathtt{rem}, \mathtt{upd}\}$ is the type of proposal,
- orig is the party that issued the proposal, and
- data is additional data.

Similarly to the HG object, Props will also export several useful methods (also explained later) to the oracles of the security game.

PKI bookkeeping. The PKI is significantly simpler in the CGKA game: it only handles so-called *init keys (IKs)*, each of which has an ID ikid. Arrays are maintained to map ikids to the corresponding public init key ($\mathsf{IK\text{-}PK}$) and secret init key ($\mathsf{IK\text{-}SK}$). Moreover, $\mathsf{IK\text{-}St}$ records the IKs stored by each party, and $\mathsf{IK\text{-}Tr}$ and $\mathsf{IK\text{-}Lk}$ keep track of "trash" IKs (old secret init keys not deleted) and leaked IKs.

CGKA Security Game: Initialization

```
// General
$b \leftarrow \{0,1\}$
$\mathsf{idCtr}{+}{+}$
$\forall \mathsf{ID} : \gamma[\mathsf{ID}] \leftarrow \mathsf{Init}(\mathsf{ID})$

// Communication
$\mathsf{CM}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{PM}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{WM}_{\mathsf{pub}}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{WM}_{\mathsf{priv}}[\cdot,\cdot] \leftarrow \epsilon$
$\mathsf{WM\text{-}Lk} \leftarrow \emptyset$

// History Graph
$\mathsf{vid}_{\mathsf{root}} \leftarrow \mathsf{HG.init}$
$\mathsf{V\text{-}Pt}[\cdot] \leftarrow \mathsf{vid}_{\mathsf{root}}$
$\mathsf{V\text{-}Tr}[\cdot] \leftarrow \emptyset$
$\mathsf{V\text{-}Lk} \leftarrow \emptyset$
$\mathsf{P\text{-}St}[\cdot] \leftarrow \emptyset$

// Miscellaneous
$\mathsf{Key}[\cdot] \leftarrow \emptyset$
$\mathsf{Reveal}[\cdot] \leftarrow \emptyset$
$\mathsf{Chall}[\cdot] \leftarrow \emptyset$
$\mathsf{Del}[\cdot] \leftarrow \mathsf{true}$
$\mathsf{BR}[\cdot] \leftarrow \mathsf{false}$

// PKI
$\mathsf{IK\text{-}PK}[\cdot] \leftarrow \epsilon$
$\mathsf{IK\text{-}SK}[\cdot] \leftarrow \epsilon$
$\mathsf{IK\text{-}St}[\cdot] \leftarrow \emptyset$
$\mathsf{IK\text{-}Tr} \leftarrow \emptyset$
$\mathsf{IK\text{-}Lk} \leftarrow \emptyset$
```

Figure 11: Initialization of the security game for CGKA schemes.

Keys and Challenges. The array Key[vid] stores, for each vid the corresponding key, and (the Boolean arrays) Reveal[vid] and Chall[vid] store whether the key was revealed and challenged, respectively.

B.1.2 Initialization.

The initialization of the CGKA game proceeds along very similar lines to that of the SGM game. It is depicted in Figure 11.

B.1.3 Oracles.

All adversary oracles described below proceed according to the same pattern: (a) verifying the validity of the oracle call, (b) retrieving values needed for (c), (c) running the corresponding SGM algorithm, and (d) updating the bookkeeping. Validity checks (a) are described informally in the text below; a formal description is provided in Figure 15. Note that, most of the time, (b) and (c) are straight-forward and are not mentioned in the descriptions. To improve readability, lines (c) are highlighted.

B.1.4 PKI.

The spirit of the PKI (Figure 12) in the CGKA game differs somewhat from the SGM case. Init keys (IKs) are either honestly generated by a party, via oracle **gen-new-ik**, or registered by the attacker $\mathcal{A}$, via oracle **reg-ik**. Note that the CGKA game will allow the attacker to pick which IK a new group member is added with; hence, there is no association between IKs and identities.

Oracle **gen-new-ik**(ID) runs the IK-generation algorithm Gen-IK and stores the resulting key pair with a new ikid. The oracle also records in IK-St that ID now stores the IK with ID ikid. Oracle **reg-ik**(ipk) allows $\mathcal{A}$ to register an IK public key ipk with the game (and get an ikid for it).

B.1.5 Main oracles.

The main oracles of the CGKA game are split into two figures: oracles related to (i) group creation, proposals, and commits (Figure 13), and (ii) message processing, corruption, and challenges (Figure 14). The validity of all oracle calls is checked by the corresponding compatibility functions (Figure 6).

CGKA Security Game: PKI Oracles

```
// Instruct ID to generate a new initial key
gen-new-ik (ID)
    $(\gamma[\mathsf{ID}], (\mathsf{ipk}, \mathsf{isk})) \leftarrow \mathsf{Gen\text{-}IK}(\gamma[\mathsf{ID}])$
    $\mathsf{ikid} \leftarrow \mathsf{idCtr}{+}{+}$
    $\mathsf{IK\text{-}PK}[\mathsf{ikid}] \leftarrow \mathsf{ipk}$
    $\mathsf{IK\text{-}SK}[\mathsf{ikid}] \leftarrow \mathsf{isk}$
    $\mathsf{IK\text{-}St}[\mathsf{ID}] \mathrel{+}\!\leftarrow \mathsf{ikid}$
    return $(\mathsf{ikid}, \mathsf{ipk})$

// Allows attacker to register IKs with game
reg-ik (ipk)
    req $\nexists\, \mathsf{ikid} : \mathsf{IK\text{-}PK}[\mathsf{ikid}] = \mathsf{ipk}$
    $\mathsf{ikid} \leftarrow \mathsf{idCtr}{+}{+}$
    $\mathsf{IK\text{-}Lk} \mathrel{+}\!\leftarrow \mathsf{ikid}$
    $\mathsf{IK\text{-}PK}[\mathsf{ikid}] \leftarrow \mathsf{ipk}$
    return ikid
```

Figure 12: PKI-related oracles of the security game for continuous group key agreement schemes.

Group creation. The attacker can instruct a party ID to create a new group by calling the group-creation oracle (Figure 3), which works analogously to the corresponding oracle in the SGM game. The bookkeeping is updated as follows: A call is made to the $\mathsf{HG.create}(\mathsf{ID}, \mathsf{skid})$ method. This causes HG to create a node

$$v = \begin{pmatrix} .\mathsf{vid} & \leftarrow & \mathsf{idCtr}{+}{+} \\ .\mathsf{orig} & \leftarrow & \mathsf{ID} \\ \boldsymbol{.\mathsf{pid}} & \leftarrow & \bot \end{pmatrix}$$

as a child of v_{root} and return $\mathsf{vid} = v.\mathsf{vid}$. Note that the CGKA group-creation oracle also stores the key I output by Create in the array Key.

Proposal oracles. The oracles **prop-{add, rem, up}-user** (Figure 13) allow the attacker to instruct a party ID to issue add/remove/update proposals. These oracles work much like their counterparts in the SGM game. Bookkeeping is updated by calling $\mathsf{Props.new}(\mathsf{op}, \mathsf{ID}, \mathsf{data})$, which records proposal data

$$p = \begin{pmatrix} .\mathsf{pid} & \leftarrow & \mathsf{idCtr}{+}{+} \\ .\mathsf{vid} & \leftarrow & \mathsf{V\text{-}Pt}[\mathsf{ID}] \\ .\mathsf{op} & \leftarrow & \mathsf{op} \\ .\mathsf{orig} & \leftarrow & \mathsf{ID} \\ .\mathsf{data} & \leftarrow & \mathsf{data} \end{pmatrix},$$

where $\mathsf{op} \in \{\mathtt{add}, \mathtt{rem}, \mathtt{upd}\}$ is the proposal type and where data stores

- *(add proposals)* $\mathsf{data} = (\mathsf{ID}', \mathsf{ikid}')$, where ikid' is the (ID of the) init key ID' is to be added with;
- *(remove proposals)* $\mathsf{data} = \mathsf{ID}'$;
- *(update proposals)* $\mathsf{data} = \bot$.

Observe that one crucial difference between the CGKA and SGM games is that the CGKA game does not explicitly model the delivery of proposals. This is due to the fact that a CGKA is assumed to be run over authenticated channels, and hence there is no way for $\mathcal{A}$ to inject malicious proposals, and, consequently, the proper delivery of proposals can be outsourced to the higher-level protocol.

Creating commits. The commit oracle (Figure 13) allows $\mathcal{A}$ to instruct a party ID to execute a commit operations. It works analogously to its SGM-game counterpart. Bookkeeping calls

CGKA Security Game: Create group, Proposal and Commit Oracles

```
// ID creates new group
create-group (ID, ikid, $r$)
    req *compat-create(ID, ikid)
    isk ← IK-SK[ikid]
    $(\gamma[\mathsf{ID}], I) \leftarrow \mathsf{Create}(\gamma[\mathsf{ID}], \mathsf{isk}; r)$
    vid ← HG.create(ID, $r$)
    Key[vid] ← $I$
    BR[vid] ← $(r \neq \perp)$
    V-Pt[ID] ← vid
    return vid

// ID proposes to add ID′
prop-add-user (ID, ID′, ikid′)
    req *compat-prop(add, ID, ID′, ikid′)
    ipk′ ← IK-PK[ikid′]
    $(\gamma[\mathsf{ID}], P) \leftarrow \mathsf{Add}(\gamma[\mathsf{ID}], \mathsf{ID}', \mathsf{ipk}')$
    pid ← Props.new(add, ID, (ID′, ikid′))
    PM[pid] ← $P$
    return (pid, $P$)

// ID proposes to remove ID′
prop-rem-user (ID, ID′)
    req *compat-prop(rem, ID, ID′, $\perp$)
    $(\gamma[\mathsf{ID}], P) \leftarrow \mathsf{Remove}(\gamma[\mathsf{ID}], \mathsf{ID}')$
    pid ← Props.new(rem, ID, ID′)
    PM[pid] ← $P$
    return (pid, $P$)

// ID proposes to update
prop-up-user (ID, $r$)
    req *compat-prop(upd, ID, $\perp$, $\perp$)
    $(\gamma[\mathsf{ID}], P) \leftarrow \mathsf{Update}(\gamma[\mathsf{ID}]; r)$
    pid ← Props.new(upd, ID, $\perp$)
    BR[vid] ← $(r \neq \perp)$
    PM[pid] ← $P$
    return (pid, $P$)

// ID commits proposals pid
commit (ID, pid, $r$)
    req *compat-commit(ID, pid)
    vid ← HG.commit(ID, pid)
    $(\gamma[\mathsf{ID}], I, W_{\mathsf{pub}}, \mathbf{W}_{\mathsf{priv}}, T) \leftarrow \mathsf{Commit}(\gamma[\mathsf{ID}], \mathsf{PM}[\mathbf{pid}]; r)$
    Key[vid] ← $I$
    BR[vid] ← $(r \neq \perp)$
    for ID′ ∈ HG.roster(V-Pt[ID])
        CM[vid, ID′] ← $T$
    $(\mathsf{ID}_1, \ldots, \mathsf{ID}_m) \leftarrow$ Props.addedIDs(pid)
    for $i = 1, \ldots, m$
        $\mathsf{WM}_{\mathsf{pub}}[\mathsf{vid}, \mathsf{ID}_i] \leftarrow W_{\mathsf{pub}}$
        $\mathsf{WM}_{\mathsf{priv}}[\mathsf{vid}, \mathsf{ID}_i] \leftarrow \mathbf{W}_{\mathsf{priv}}[\mathsf{ID}_i]$
    return (vid, $T$, $W_{\mathsf{pub}}$, $\mathbf{W}_{\mathsf{priv}}$)
```

Figure 13: Oracles for group creation, add, remove, update proposals and commits of the security game for continuous group key agreement schemes. The compatibility functions are described in the accompanying text; a formal description is provided in Figure 15.

$\mathsf{HG.commit}(\mathsf{ID}, \mathbf{pid})$, which creates a new HG node

$$v = \begin{pmatrix} .\mathsf{vid} & \leftarrow & \mathsf{idCtr}{+}{+} \\ .\mathsf{orig} & \leftarrow & \mathsf{ID} \\ .\mathbf{pid} & \leftarrow & \mathbf{pid} \end{pmatrix}$$

as a child of ID's current epoch V-Pt[ID] and returns $\mathsf{vid} = v.\mathsf{vid}$. Note that the key output by Commit is stored in array Key. Furthermore, observe that the algorithm outputs (one) public and several private welcome messages (one for each newly added party). The private welcome messages are not returned to $\mathcal{A}$ and are delivered securely to their intended recipient—the idea being that these messages are encrypted by the higher-level application.

Process control messages. The oracles **process** and **dlv-WM** (Figure 14) allow the attacker to deliver commit messages (to existing group members) resp. welcome messages (to new group members). The oracles works much like their SGM counterparts, except that all the information required by Proc-Com resp. Join is supplied by the game. The oracles also check that the key I output by Proc-Com resp. Join and matches the one stored in array Key during the corresponding call to **commit**.

Key-reveal and challenge oracles. For each epoch, the attacker $\mathcal{A}$ gets to either see the actual key output by the protocol or a challenge by calling **reveal** or **chall** (Figure 14), respectively. Oracle **chall** outputs either the real key or a completely random one, depending on the secret internal bit b chosen at the onset of the game.

CGKA Security Game: Message processing and corruption/leakage attacks oracles

```
// Process commit msg for ID and epoch vid
process (vid, ID)
    req *compat-process(vid, ID)
    $T \leftarrow \mathsf{CM}[\mathsf{vid}, \mathsf{ID}]$
    $\mathbf{P} \leftarrow \mathsf{PM}[\mathsf{HG}[\mathsf{vid}].\mathsf{pid}]$
    $(\gamma[\mathsf{ID}], \mathsf{GI}, I) \leftarrow \mathsf{Proc\text{-}Com}(\gamma[\mathsf{ID}], T, \mathbf{P})$
    if $\neg\mathsf{HG.checkGI}(\mathsf{vid}, \mathsf{GI}) \vee \mathsf{Key}[\mathsf{vid}] \neq I$
        win
    if $\neg\mathsf{Del}[\mathsf{ID}]$
        $\mathsf{V\text{-}Tr}[\mathsf{ID}] \leftarrow \mathsf{V\text{-}Pt}[\mathsf{ID}]$
    if $\mathsf{HG.isRemoved}(\mathsf{vid}, \mathsf{ID})$
        $\mathsf{V\text{-}Pt}[\mathsf{ID}] \leftarrow \mathsf{vid}_{\mathsf{root}}$
    else
        $\mathsf{V\text{-}Pt}[\mathsf{ID}] \leftarrow \mathsf{vid}$
    $i[\mathsf{ID}] \leftarrow 0$
    $\mathsf{P\text{-}St}[\mathsf{ID}] \leftarrow \emptyset$
    return GI

// Reveal the update secret of epoch vid
reveal (vid)
    req $\mathsf{Key}[\mathsf{vid}] \neq \epsilon$
    req $\neg(\mathsf{Reveal}[\mathsf{vid}] \vee \mathsf{Chall}[\mathsf{vid}])$
    $\mathsf{Reveal}[\mathsf{vid}] \leftarrow \mathsf{true}$
    return $\mathsf{Key}[\mathsf{vid}]$

// Challenge the update secret of epoch vid
chall (vid)
    req $\mathsf{Key}[\mathsf{vid}] \neq \epsilon$
    req $\neg(\mathsf{Reveal}[\mathsf{vid}] \vee \mathsf{Chall}[\mathsf{vid}])$
    $I_0 \leftarrow \mathsf{Key}[\mathsf{vid}]$
    $I_1 \leftarrow \mathcal{I}$
    $\mathsf{Chall}[\mathsf{vid}] \leftarrow \mathsf{true}$
    return $I_b$

// Welcome msg. of epoch vid delivered to ID
dlv-WM (vid, ID)
    req *compat-dlv-WM(vid, ID)
    $W_{\mathsf{pub}} \leftarrow \mathsf{WM\text{-}pub}[\mathsf{vid}, \mathsf{ID}]$
    $W_{\mathsf{priv}} \leftarrow \mathsf{WM\text{-}priv}[\mathsf{vid}, \mathsf{ID}]$
    $\mathsf{ikid} \leftarrow \mathsf{HG.addedIK}(\mathsf{vid}, \mathsf{ID})$
    $\mathsf{isk} \leftarrow \mathsf{IK\text{-}SK}[\mathsf{ikid}]$
    let $v$ s.t. $v.\mathsf{vid} = \mathsf{vid}$
    $\mathsf{orig} \leftarrow v.\mathsf{orig} = \mathsf{vid}$
    $(\gamma[\mathsf{ID}], \mathsf{GI}, I) \leftarrow \mathsf{Join}(\mathsf{ID}, \mathsf{orig}, W_{\mathsf{pub}}, W_{\mathsf{priv}}, \mathsf{isk})$
    if $\neg\mathsf{HG.checkGI}(\mathsf{vid}, \mathsf{GI}) \vee \mathsf{Key}[\mathsf{vid}] \neq I$
        win
    if $\neg\mathsf{Del}[\mathsf{ID}]$
        $\mathsf{IK\text{-}Tr}[\mathsf{ID}] \mathrel{+\!\!\leftarrow} \mathsf{ikid}$
    $\mathsf{IK\text{-}St}[\mathsf{ID}] \mathrel{-\!\!\leftarrow} \mathsf{ikid}$
    $\mathsf{V\text{-}Pt}[\mathsf{ID}] \leftarrow \mathsf{vid}$
    return GI

// Corrupt ID
corr (ID)
    $\mathsf{V\text{-}Lk} \mathrel{+\!\!\leftarrow} \{(\mathsf{vid}, \mathsf{ID}) \mid \mathsf{vid} \in \{\mathsf{V\text{-}Pt}[\mathsf{ID}]\} \cup \mathsf{V\text{-}Tr}[\mathsf{ID}]\}$
    $\mathsf{IK\text{-}Lk} \mathrel{+\!\!\leftarrow} \mathsf{IK\text{-}St}[\mathsf{ID}] \cup \mathsf{IK\text{-}Tr}[\mathsf{ID}]$
    $\mathsf{HG.corrHanging}(\mathsf{ID})$
    return $\gamma[\mathsf{ID}]$

// Enable no-delete for ID
no-del (ID)
    disable deletions for ID
    $\mathsf{Del}[\mathsf{ID}] \leftarrow \mathsf{false}$

// Safety for privacy
*priv-safe
    return $\forall \mathsf{vid} : \mathsf{Chall}[\mathsf{vid}] \implies$ *CGKA-priv(vid)
```

Figure 14: Oracles for message processing, corruption, and challenges of the security game for continuous group key agreement schemes. The compatibility functions are described in the accompanying text; a formal description is provided in Figure 15.

Corruption oracles. The oracles related to corruption in the CGKA game (Figure 14) are:

- **corr**(ID): leaks the state of ID to $\mathcal{A}$;
- **no-del**(ID): instructs ID to stop deleting old values.

B.1.6 Safety.

At the end of the execution of the CGKA security game, the procedure ***priv-safe** ensures that the attacker has only challenged in epochs that are considered secure by the (generic) safety predicate ***CGKA-priv**. If the condition is not satisfied, the attacker loses the game.

B.1.7 Advantage.

Let $\Pi =$ ***CGKA-priv** be the generic safety predicate used in the CGKA definition. The attacker $\mathcal{A}$ is parameterized by it's running time, t, and the number of challenge queries, q, and referred to as (t, q)-attacker. The advantage of $\mathcal{A}$ against a CGKA scheme K w.r.t. to predicate Π is denoted by $\mathrm{Adv}^{\mathsf{K}}_{\mathsf{CGKA},\Pi}(\mathcal{A})$.

Definition 4. *A CGKA scheme* K *is* (t, q, ε)-secure w.r.t. predicate Π, *if for all* (t, q)*-attackers,*

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

CGKA Security Game: Compatibility Helpers

```
*compat-create (ID, ikid)
    chk ikid ∈ IK-St[ID]
    return V-Pt[ID] = vid_root

*compat-prop (op, ID, ID', ikid)
    vid ← V-Pt[ID]
    chk vid ≠ vid_root
    G ← HG.roster(vid)
    select op
        case add do
            chk ∃ipk' : IK-PK[ikid'] = ipk'
            chk ID' ∉ G
        case rem do
            chk ID' ∈ G
    return true

*compat-commit (ID, pid)
    vid ← V-Pt[ID]
    chk vid ≠ vid_root
    G ← HG.roster(vid)
    for pid ∈ pid
        p ← Props[pid]
        chk p.vid = vid
        G ← *app-prop(G, p)
        chk G ≠ ⊥
    chk ID ∉ G
    return true

*compat-process (vid, ID)
    chk HG.isChild(V-Pt[ID], vid)
    return true

*app-prop (G, p)
    req p.orig ∈ G
    select p.op
        case add do
            (ID', ·) ← p.data
            req ID' ∉ G
            G +← ID'
        case rem do
            ID' ← p.data
            req ID' ∈ G
            G −← ID'
    return G

*compat-dlv-WM (vid, ID)
    chk V-Pt[ID] = vid_root
    ikid ← HG.addedIK(vid, ID)
    chk ikid ≠ ⊥
        ∧ ikid ∈ IK-St[ID]
    return true
```

Figure 15: Compatibility oracles of the security game for continuous group key agreement schemes.

C (R)TreeKEM

C.1 Overview

The (R)TreeKEM CGKA protocol is based on so-called (binary) *ratchet trees (RTs).* In a (R)TreeKEM RT, group members are arranged at the leaves, and all nodes have an associated public-key encryption (PKE) key-pair, except for the root. The tree invariant is that each user knows all secret keys on their *direct path*, i.e., on the path from their leaf node to the root. In order to perform an update—the most crucial operation of a CGKA—and produce a new update secret I, a party first generates fresh key pairs on every node of their direct path. Then, for every node v' on its *co-path*—the sequence of siblings of nodes on the direct path—it encrypts specific information under the public key of v' that allows each party in the subtree of v' to learn all new secret keys from v's parent up to the root.

Before presenting the formal description of (R)TreeKEM in Sections C.3,C.4, basic concepts around ratchet trees are explored in Section C.2.

C.2 Ratchet Trees

C.2.1 Basics

The following are some basic concepts around TreeKEMs ratchet trees.

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

Definition 5 (Left-Balanced Binary Tree)**.** *For $n \in \mathbb{N}$, the* left-balanced binary tree (LBBT) on n nodes LBBT_n *is the binary tree constructed as follows: The tree* LBBT_1 *is a single node. Let*

$x = \mathsf{mp2}(n)$.[12] *Then, the root of* LBBT_n *has the full subtree* FT_x *as the left subtree and* LBBT_{n-x} *as the right subtree.*

Observe that LBBT_n has exactly n leaves and that every internal node has two children. In an RT, nodes are *labeled* as follows: *Root:* The root is labeled by an *update secret* I. *Internal nodes:* Internal nodes are labeled by a *key pair*, $(\mathsf{pk}, \mathsf{sk})$, for the PKE scheme, PKE. *Leaf nodes:* Leaf nodes are labeled like internal nodes, except that they additionally have an *owner* ID.

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

Paths and blanking. As hinted at the beginning of this section, it will be useful to consider the following types of paths: the *direct path* $\mathsf{dPath}(\tau, \mathsf{ID})$, which is the path from the leaf node labeled by ID to the root; the *co-path* $\mathsf{coPath}(\tau, \mathsf{ID})$, which is the sequence of siblings of nodes on the direct path $\mathsf{dPath}(\tau, \mathsf{ID})$. Furthermore, given an ID, ID, and an RT, τ, the function $\tau' \leftarrow \mathtt{Blank}(\tau, \mathsf{ID})$ blanks all nodes on $\mathsf{dPath}(\tau, \mathsf{ID})$.

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

Definition 6 (Resolution). *Let τ be a tree with node set V. The* resolution $\mathsf{Res}(v) \subseteq V$ *of a node $v \in V$ is defined recursively as follows: If v is* not blank, *then* $\mathsf{Res}(v) = \{v\}$, *else if v is a* blank leaf, *then* $\mathsf{Res}(v) = \emptyset$, *otherwise,* $\mathsf{Res}(v) := \cup_{v' \in C(v)} \mathsf{Res}(v')$, *where $C(v)$ are the children of v.*

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

Definition 7 (Representative). *Consider a tree τ and two leaf nodes ℓ and ℓ'.*

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

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

C.2.2 Simple RT operations

The following paragraphs describe how RTs are initialized as well as how they grow and shrink. The proofs of Lemmas 5 and 6 below can be found in the full version [3].

RT Initialization. Given an ID ID and secret key sk, the initialization operation creates a tree with a single node, as depicted in Figure 16.

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

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

Lemma 5. *If $\tau = \mathsf{LBBT}_n$, then $\tau' = \mathsf{LBBT}_{n+1}$.*

Removing an ID. The procedure $\tau' \leftarrow \texttt{RemID}(\tau, \mathsf{ID})$ blanks the leaf labeled with ID and truncates the tree such that the rightmost non-blank leaf is the last node of the tree. Specifically, the following recursive procedure $\texttt{Trunc}(v)$ is called on the rightmost leaf v of τ, resulting in a new tree $\tau' \leftarrow \texttt{Trunc}(\tau)$:[13] 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 $\texttt{Trunc}(v')$. If v is non-blank and the root, execute $\texttt{Trunc}(v'')$ on the rightmost leaf node in the tree. Otherwise, do nothing.

Lemma 6. *If $\tau = \mathsf{LBBT}_n$, then $\tau' = \mathsf{LBBT}_y$ for some $0 < y \leq n$. Furthermore, unless $y = 1$, the rightmost leaf of τ' is non-blank.*

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

C.3 The (R)TreeKEM protocol

This section presents the RTreeKEM protocol in detail by describing all the algorithms involved in the scheme, which is depicted in Figure 17. Then, in Section C.4 we discuss how TreeKEM diverges from RTreeKEM.

RTreeKEM makes (black-box) use of the following cryptographic primitives:

- a hash function H mapping elements from $\{0,1\}^*$ to $\{0,1\}^\lambda$.
- a UPKE scheme $\mathsf{UPKE} = (\mathsf{UKGen}, \mathsf{UEnc}, \mathsf{UDec})$.

Before presenting the protocols, we define additional methods and algorithms over ratchet trees.

Ratchet tree methods.

- $\tau[v]$: the node v in τ (for brevity we sometimes use v and the tree is implict).
- $\tau[\mathsf{ID}]$: the leaf node with id ID.
- $\tau[v].\mathsf{pk}$: the public key at node v.
- $\tau[v].\mathsf{sk}$: the secret key at node v.
- $\tau.I$: the root key of the ratchet tree.
- $\mathsf{LCA}(\tau, v, v')$: the least common ancestor of v, v' in τ.
- $\mathsf{LCA}(\tau, \mathsf{ID}, \mathsf{ID}')$: the least common ancestor of the nodes with ids ID, ID', in τ.

[13] Overloading function $\texttt{Trunc}$ for convenience here.

- $\tau.\mathsf{GI}$: the group information. $\tau.\mathsf{GI} = (\mathsf{roster}, \mathsf{orig}, \mathbf{added}, \mathbf{removed})$, where
  - roster: vector of current group-member IDs (computed on-the-fly).
  - orig: originator of τ (via either Create or Commit).
  - **added**: contains a dictionary s.t. $\mathbf{added}[\mathsf{ID_a}] = \mathsf{ID_s}$ for adder $\mathsf{ID_s}$, addee $\mathsf{ID_a}$.
  - **removed**: contains a dictionary s.t. $\mathbf{removed}[\mathsf{ID_r}] = \mathsf{ID_s}$ for remover $\mathsf{ID_s}$, removee $\mathsf{ID_r}$.
- $\mathtt{DEPTH}(\tau, \mathsf{ID})$: the depth of the leaf node with id ID in τ.
- $\mathtt{FBL}(\tau)$: returns the first blank leaf of τ.

We now describe the ratchet tree algorithms depicted in Figure 16

Ratchet Tree Algorithms

```
// Ratchet tree initialization
Init (ID, isk)
    ipk ← UKGen(isk)
    $\tau \leftarrow \mathsf{LBBT}_1$
    let $v$ be the only node in $\tau$
    $\tau[v]$.ID ← ID
    $\tau[v]$.pk ← ipk
    $\tau[v]$.sk ← isk
    $\tau$.GI.orig ← ID
    $\tau.\mathsf{GI}.\{\mathbf{added}, \mathbf{removed}\} \leftarrow \emptyset$
    return $\tau$
```

```
// Id removal
RemID ($\tau$, ID)
    $\tau$[ID].label ← $\perp$
    $\tau$ ← TRUNC($\tau$)
    return $\tau$

// Node blanking
Blank ($\tau$, ID)
    for $v \in \mathsf{dPath}(\tau, \mathsf{ID})$
        $\tau[v]$.label ← $\perp$
    return $\tau$

// Leaf node update
UPID ($\tau$, ID, ipk)
    $\tau$[ID].pk ← ipk
    return $\tau$
```

```
// Addition of new leaf id
AddID ($\tau$, ID, ipk)
    $v$ ← FBL($\tau$)
    if $v = \perp$
        $(\tau, v)$ ← AddLeaf($\tau$)
    $\tau[v]$.label ← (ID, ipk, $\perp$)
    for $u \in \mathsf{dPath}(\tau, v)$
        if $\tau[u].\mathsf{label} \neq \perp$
            $\tau[u]$.unmerged +← $v$
    return $\tau$
```

Figure 16: Basic ratchet tree algorithms. Other operations are defined in Section C.2.

Ratchet tree algorithms.

- $\mathtt{Init}(\mathsf{ID}, \mathsf{isk})$: creates a ratchet tree with labels
  - $\tau.\mathsf{GI.orig} = \mathsf{ID}$,
  - $\tau.\{\mathbf{added}, \mathbf{removed}\} = \emptyset$.

  and single node v with labels $\tau[v].\mathsf{label}$, consisting of
  - $\tau[v].\mathsf{ID} = \mathsf{ID}$ (only for leaves),
  - $\tau[v].\mathsf{pk} = \mathsf{UKGen}(\mathsf{isk})$,
  - $\tau[v].\mathsf{sk} = \mathsf{isk}$.
- $\mathtt{RemID}(\tau, \mathsf{ID})$: erases the label for the leaf node with id ID and truncates the ratchet tree (if needed).
- $\mathtt{Blank}(\tau, \mathsf{ID})$: blanks all nodes in the direct path of the leaf with id ID.
- $\mathtt{UPID}(\tau, \mathsf{ID}, \mathsf{ipk})$: updates the public key of the leaf node with id ID to ipk.
- $\mathtt{AddID}(\tau, \mathsf{ID}, \mathsf{ipk})$: adds a new id ID with public key ipk to τ, blanks all nodes in the direct path of the new node v and adds v to their set of unmerged leaves.

We now describe the core of the RTreeKEM protocol, depicted in Figure 17.

RTreeKEM

```
// State initialization
RTK-init (ID)
    ME ← ID
    $\tau \leftarrow \bot$
    ctr ← 0
    $\tau'[\cdot]$, conf$[\cdot]$, UPK$[\cdot] \leftarrow \bot$

// Init key generation
RTK-Gen-IK
    isk ← UKS
    ipk ← UKGen(isk)
    return (ipk, isk)

// Group creation
RTK-Create (ID, isk; $r$)
    RTK-init(ID)
    if $r = \bot$
        $I \leftarrow \{0,1\}^\lambda$
    else
        $I \leftarrow r$
    $\tau \leftarrow$ Init(ME, isk)
    return ($\tau$.GI, $I$)
```

```
// Add proposal
RTK-Add (ID′, ipk′)
    $P \leftarrow$ (add, ME, ID′, ipk′)
    return $P$

// Remove proposal
RTK-Rem (ID′)
    $P \leftarrow$ (rem, ME, ID′)
    return $P$

// Update proposal
RTK-Update ($r$)
    if $r = \bot$
        sk ← UKS
    else
        sk ← $r$
    pk ← UKGen(sk)
    UPK[pk] ← sk
    $P \leftarrow$ (upd, ME, pk)
    return $P$

// Group join
RTK-Join (ID, orig, $W_{\text{pub}}$, $W_{\text{priv}}$, isk)
    RTK-init(ID)
    $\tau \leftarrow W_{\text{pub}}$
    ($\tau$[ME].sk, $s$) ← UDec(isk, $W_{\text{priv}}$)
    $(\tau, \cdot, \cdot) \leftarrow$
      *refresh-path($\tau$, $s$, LCA(ME, orig))
    return ($\tau$.GI, $\tau.I$)
```

```
// Commit operation
RTK-Commit ($\mathbf{P}; r$)
    $\tau' \leftarrow \tau$
    $\tau' \leftarrow$ *apply-props($\tau', \mathbf{P}$)
    $(\tau', T, \mathbf{W}_{\text{priv}}) \leftarrow$ *refresh-gen($\tau', r$)
    $\tau'$.GI.orig ← ME
    $W_{\text{pub}} \leftarrow$ Pub($\tau'$)
    ctr++
    $\tau'$[ctr] ← $\tau'$
    conf[ctr] ← $T$
    $I \leftarrow \tau'.I$
    $\tau'.I \leftarrow \bot$
    return ($I, W_{\text{pub}}, \mathbf{W}_{\text{priv}}, T$)

// Process commit message
RTK-Proc-Com ($T, \mathbf{P}$)
    if $\exists j$ : conf$[j] = T$
        $\tau \leftarrow \tau'[T]$
    else
        $\tau \leftarrow$ *apply-props($\tau, \mathbf{P}$)
        $\tau \leftarrow$ *refresh-proc($\tau, T$)
    (ID, $\cdot, \cdot$) ← $T$
    $\tau$.GI.orig ← ID
    $I \leftarrow \tau.I$
    $\tau.I \leftarrow \bot$
    ctr ← 0
    $\tau'[\cdot]$, conf$[\cdot]$, UPK$[\cdot] \leftarrow \varepsilon$
    return ($\tau$.GI, $I$)
```

```
// Apply proposals
*apply-props ($\tau, \mathbf{P}$)
    added$[\cdot]$, removed$[\cdot] \leftarrow \varepsilon$
    for $P \in \mathbf{P}$
        if $P$ = (add, ID$_s$, ID$_a$, ipk)
            $\tau \leftarrow$ AddID($\tau$, ID$_a$, ipk)
            added[ID$_a$] ← ID$_s$
            removed[ID$_a$] ← $\varepsilon$
        if $P$ = (rem, ID$_s$, ID$_r$)
            $\tau \leftarrow$ Blank($\tau$, ID$_r$)
            $\tau \leftarrow$ RemID($\tau$, ID$_r$)
            removed[ID$_r$] ← ID$_s$
            added[ID$_r$] ← $\varepsilon$
        if $P$ = (upd, $\cdot$, ID, pk, $\bot$)
            $\tau \leftarrow$ UPID($\tau$, ID, pk)
            $\tau \leftarrow$ Blank($\tau$, ID)
            if ID = ME
                $\tau$[ID].sk ← UPK[pk]
    $\tau$.GI.added ← added
    $\tau$.GI.removed ← removed
    return $\tau$
```

```
// Refresh message generation
*refresh-gen ($\tau, r$)
    if $r = \bot$
        $s \leftarrow \{0,1\}^\lambda$
    else
        $s \leftarrow r$
    $v \leftarrow \tau$[ID]
    $\mathbf{W}_{\text{priv}}[\cdot], \mathbf{c}[\cdot] \leftarrow \varepsilon$
    ($\tau$, $\mathbf{s}$, PK) ← *refresh-path($\tau, s, v$)
    $(v'_0, \ldots, v'_{\mathsf{d}-1}) \leftarrow$ coPath($\tau, v$)
    for $i = 1, \ldots, \mathsf{d}$
        for $w \in$ Res($v'_{i-1}$)
            if $\tau[w]$.ID $\in \tau$.GI.added
                ($\tau[w]$.pk, $\mathbf{W}_{\text{priv}}[\tau[w]$.ID]) ←
                  UEnc($\tau[w]$.pk, $\mathbf{s}[i]$)
            else
                ($\tau[w]$.pk, $\mathbf{c}[w]$) ← UEnc($\tau[w]$.pk, $\mathbf{s}[i]$)
    $T \leftarrow$ (ID, PK, $\mathbf{c}$)
    return ($\tau, T, \mathbf{W}_{\text{priv}}$)

// Refresh path secrets
*refresh-path ($\tau, s_0, v$)
    d ← DEPTH($\tau, v$)
    $\mathbf{s}[\cdot]$, PK$[\cdot] \leftarrow \varepsilon$
    $\mathbf{s}[0] \leftarrow s_0$
    for $i = 1, \ldots, \mathsf{d} - 1$
        sk ← H($\mathbf{s}[i]$, kgen)
        $\mathbf{s}[i+1]$ ← H($\mathbf{s}[i]$, path)
        PK$[i]$ ← UKGen(sk)
        $\tau[v]$.label ← (PK$[i]$, sk)
        $\tau[v]$.unmerged ← $\emptyset$
        $v \leftarrow \tau[v]$.parent
        $\tau[v].I \leftarrow \mathbf{s}$[d]
    return ($\tau$, $\mathbf{s}$, PK)
```

```
// Process control message
*refresh-proc ($\tau, T$)
    (ID, PK, $\mathbf{c}$) ← $T$
    $v \leftarrow \tau$[ME]
    $v' \leftarrow \tau$[ID]
    $w \leftarrow$ Rep($\tau, v, v'$)
    ($\tau[w]$.sk, $s_0$) ← UDec($\tau[w]$.sk, $\mathbf{c}[v]$)
    $(z_0, \ldots, z_{\mathsf{d}}) \leftarrow$ dPath($\tau, v'$)
    for $i = 0, \ldots, \mathsf{d} - 1$
        $\tau[z_i]$.pk ← PK$[i]$
    $(\tau, \cdot, \cdot) \leftarrow$ *refresh-path($\tau, s_0$, LCA($v, v'$))
    return $\tau$
```

Figure 17: The RTreeKEM protocol operations.

C.3.1 Initialization.

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

C.3.2 Generation of init keys.

$\mathsf{RTK\text{-}Gen\text{-}IK}$ simply generates a new UPKE key pair.

C.3.3 Group creation.

The group creation operation $\mathsf{RTK\text{-}Create}$ receives an id ID, a secret init key isk and randomness r, initializes the RTreeKEM state for ID by executing $\mathsf{RTK\text{-}init}(\mathsf{ID})$, creates a ratchet tree with id ID and secret key isk by executing $\mathtt{Init}(\mathsf{ME}, \mathsf{isk})$ and sets the CGKA update secret to r, if $r \neq \perp$, otherwise, to a uniformly random value. It returns the group information and the update secret.

C.3.4 Proposals.

- $\mathsf{RTK\text{-}Add}(\mathsf{ID}', \mathsf{ipk}')$: creates an add proposal for adding to the CGKA group the id ID' with init public key ipk'.
- $\mathsf{RTK\text{-}Rem}(\mathsf{ID}')$: creates a remove proposal for removing from the CGKA group the id ID'.
- $\mathsf{RTK\text{-}Update}(r)$: creates an update proposal by generating a new UPKE key pair. If $r = \perp$ the key generation process uses fresh randomness, otherwise it sets $\mathsf{sk} = r$. The newly generated UPKE pair is stored to UPK and will only be used if there is a commit operation that includes the specific proposal.

Observe that proposals store the id ME of the issuer.

C.3.5 Group join.

The group join operation receives the id ID of the new member, the sender of the commit message that included the proposal for adding ID to the group, orig, the public version of the ratchet tree W_{pub}, the private welcome information W_{priv} and the joiner's secret init key isk and

- Initializes the RTreeKEM state by executing $\mathsf{RTK\text{-}init}(\mathsf{ID})$.
- Sets the ratchet tree to W_{pub}.
- Decrypts W_{priv} with isk, stores the new UPKE secret key sk to the leaf $\tau[\mathsf{ME}]$ and using the recovered path secret s, executes $\mathsf{*refresh\text{-}path}$ which generates path secrets and UPKE keys along the direct path of $\mathsf{LCA}(\mathsf{ME}, \mathsf{orig})$ (see below).

C.3.6 Commits.

The $\mathsf{RTK\text{-}Commit}$ operation receives a vector of proposals $\mathbf{P}$ and randomness r, applies $\mathbf{P}$ to the current ratchet tree by executing $\mathsf{*apply\text{-}props}$ and generates path secretes along the direct path of the issuer by executing $\mathsf{*refresh\text{-}gen}$ (see below). The issuer switches to the new ratchet tree τ' only after receiving a confirmation for the control message T. The operation outputs the new update secret I, T, the public copy of the new ratchet tree τ' and the welcome private information $\mathbf{W}_{\mathsf{priv}}$. A detailed description of those variables is provided below.

C.3.7 Processing commits.

The $\mathsf{RTK\text{-}Proc\text{-}Com}$ operation is processing commit messages. In particular, it receives a control message T and a vector of proposals $\mathbf{P}$ and

- If T awaits for confirmation (checked by $\exists j : \mathsf{conf}[j] = T$) then the party is the issuer of the commit message thus it has already computed the new ratchet tree τ'.
- Otherwise, it applies the vector of proposals to it's current ratchet tree by executing *apply-props and then processes the control message T by executing *refresh-proc (see below).

Subsequently, the operation sets the originator of the commit message and the new CGKA update secret and outputs that secret as well as the group information.

We now describe the helper functions of Figure 17.

C.3.8 Applying proposals.

The operation *apply-props receives a ratchet tree τ and a vector of proposals, applies the proposals to τ and outputs the new ratchet tree. In particular,

- For an add proposal $(\mathtt{add}, \mathsf{ID_s}, \mathsf{ID_a}, \mathsf{ipk})$ the new id $\mathsf{ID_a}$ with init public key ipk is added to the ratchet tree by executing $\mathtt{AddID}(\tau, \mathsf{ID_a}, \mathsf{ipk})$. Note that $\mathtt{AddID}$ includes the node of $\mathsf{ID_a}$ to the set of the unmerged nodes along the direct path of $\mathsf{ID_a}$. Future protocol messages will be also encrypted under the public key of $\mathsf{ID_a}$, until the node of $\mathsf{ID_a}$ is removed from the set of unmerged leaves.
- For a remove proposal $(\mathtt{rem}, \mathsf{ID_s}, \mathsf{ID_r})$, the leaf node of $\mathsf{ID_r}$ is removed from the ratchet tree by executing $\mathtt{RemID}(\tau, \mathsf{ID_r})$ and all nodes in the direct path of $\mathsf{ID_r}$ are blanked. This ensures that future protocol messages won't be encrypted under secret keys known by $\mathsf{ID_r}$.
- For an update proposal $(\mathsf{upd}, \cdot, \mathsf{ID}, \mathsf{pk}, \bot)$, the UPKE public key of ID is updated by executing $\mathtt{UPID}(\tau, \mathsf{ID}, \mathsf{pk})$ and all nodes in the direct path of ID are blanked. If the operation is executed by the proposal issuer, it also updates the UPKE secret key by setting $\tau[\mathsf{ID}].\mathsf{sk} \leftarrow \mathsf{UPK}[\mathsf{pk}]$.

The operation also updates the group information, GI, by computing the sets of newly added members and removed ones, as well as the ids that issued the proposals.

C.3.9 Refresh path secrets.

The operation *refresh-path receives a ratchet tree τ, a seed s_0 and node v and generates secrets and UPKE keys along the direct path of v. In particular, for $i \in [d-1]$, where d is the depth of v in τ, the operation computes the UPKE key $\mathsf{sk} \leftarrow \mathsf{H}(\mathbf{s}[i], \mathsf{kgen})$, where $\mathbf{s}[0] \leftarrow s_0$, and the path secret of the father $\mathbf{s}[i+1] \leftarrow \mathsf{H}(\mathbf{s}[i], \mathsf{path})$. In addition, for all nodes in the direct path of v the set of unmerged leaves is set to $\emptyset$. The operation returns the new ratchet tree, the path secrets $\mathbf{s}$ and the newly generated public UPKE keys PK.

C.3.10 Generation of protocol messages for refreshed secrets.

The operation *refresh-gen receives a ratchet tree τ and randomness r (if $r \neq \bot$ then the operation uses adversarially controlled randomness). It generates path secrets and UPKE keys by executing *refresh-path(τ, s, v) and then encrypts the path secrets $\mathbf{s}$ under the public keys of the co-path nodes. In particular, 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 $\mathbf{s}[i]$, and every node $w \in \mathsf{Res}(v'_{i-1})$, if w corresponds to a node of a newly added user, then construct welcome ciphertext for that user, $\mathbf{W}_{\mathsf{priv}}[\tau[w].\mathsf{ID}]$, by encrypting the path secret $\mathbf{s}[i]$ directly under the user's UPKE public key. The operation also outputs an new public key for that user's node $\tau[w]$. If w does not correspond to a newly added user, then a single ciphertext is generated and stored in $\mathbf{c}[w]$ (this ciphertext will be possibly decrypted by multiple nodes). Ciphertexts for existing users are stored in $\mathbf{c}$, newly generated public keys are stored in PK, and T stores $(\mathsf{ID}, T, \mathbf{c})$, where ID is the id of the party that issue the commit operation. *refresh-gen outputs the new ratchet tree, the control message T and the private welcome message $\mathbf{W}_{\mathsf{priv}}$ that will be processed by newly added members (each member processes the part of $\mathbf{W}_{\mathsf{priv}}$ that is encrypted under it's own key).

C.3.11 Processing control messages.

The operation *refresh-proc receives a ratchet tree τ and control message T, parses T as $(\mathsf{ID}, \mathsf{PK}, \mathbf{c})$, and recovers the ratchet tree nodes of the current user, $v \leftarrow \tau[\mathsf{ME}]$, and the one that generated the control message, $v' \leftarrow \tau[\mathsf{ID}]$. Then computes the representative $w \leftarrow \mathsf{Rep}(\tau, v, v')$, and uses it's private key to decrypt the corresponding ciphertext, $(\tau[w].\mathsf{sk}, s_0) \leftarrow \mathsf{UDec}(\tau[w].\mathsf{sk}, \mathbf{c}[v])$. Finally, the operation updates the ratchet tree of the caller by overriding the public UPKE keys on the direct path of v and then producing a new ratchet tree by executing $(\tau, \cdot, \cdot) \leftarrow$ *refresh-path$(\tau, s_0, \mathsf{LCA}(v, v'))$. The last operation generates the path secrets and UPKE keys along the direct path of $\mathsf{LCA}(v, v')$.

C.4 TreeKEM

The TreeKEM protocol is similar to RTreeKEM, with the following difference: the protocol uses a regular public-key encryption scheme (E-KeyGen, E-Enc, E-Dec) (not an updatable one). Therefore, the only differences appear in the protocol operations that involve encryption/decryption. For completeness, we define those operations in Figure 18

TreeKEM

```
// Init key generation
TK-Gen-IK
    isk ← KS
    ipk ← E-KeyGen(isk)
    return (ipk, isk)
```

```
// Update proposal
TK-Update ($r$)
    if $r = \bot$
        sk ← KS
    else
        sk ← $r$
    pk ← E-KeyGen(sk)
    UPK[pk] ← sk
    $P$ ← (upd, ME, pk)
    return $P$
```

```
// Group join
TK-Join (ID, orig, $W_{\mathsf{pub}}$, $W_{\mathsf{priv}}$, esk)
    // Same as RTK-init
    TK-init(ID)
    $\tau \leftarrow W_{\mathsf{pub}}$
    $s$ ← E-Dec(esk, $W_{\mathsf{priv}}$)
    $(\tau, \cdot, \cdot)$ ← *refresh-path($\tau$, $s$, LCA(ME, orig))
    return ($\tau$.GI, $\tau.I$)
```

```
// Refresh path secrets
*refresh-path ($\tau$, $s_0$, $v$)
    d ← DEPTH($\tau$, $v$)
    $\mathbf{s}[\cdot]$, PK[·] ← $\varepsilon$
    $\mathbf{s}[0] \leftarrow s_0$
    for $i = 1, \ldots, \mathsf{d} - 1$
        sk ← H($\mathbf{s}[i]$, kgen)
        $\mathbf{s}[i+1]$ ← H($\mathbf{s}[i]$, path)
        PK[$i$] ← E-KeyGen(sk)
        $\tau[v]$.label ← (PK[$i$], sk)
        $\tau[v]$.unmerged ← $\emptyset$
        $v \leftarrow \tau[v]$.parent
        $\tau[v].I \leftarrow \mathbf{s}[\mathsf{d}]$
    return ($\tau$, $\mathbf{s}$, PK)
```

```
// Refresh message generation
*refresh-gen ($\tau$, $r$)
    if $r = \bot$
        $s \leftarrow \{0,1\}^\lambda$
    else
        $s \leftarrow r$
    $v \leftarrow \tau$[ID]
    $\mathbf{W}_{\mathsf{priv}}[\cdot]$, $\mathbf{c}[\cdot]$ ← $\varepsilon$
    ($\tau$, $\mathbf{s}$, PK) ← *refresh-path($\tau$, $s$, $v$)
    $(v'_0, \ldots, v'_{\mathsf{d}-1})$ ← coPath($\tau$, $v$)
    for $i = 1, \ldots, \mathsf{d}$
        for $w \in$ Res($v'_{i-1}$)
            if $\tau[w]$.ID ∈ $\tau$.GI.added
                $\mathbf{W}_{\mathsf{priv}}[\tau[w]$.ID] ←
                  E-Enc($\tau[w]$.epk, $\mathbf{s}[i]$)
            else
                $\mathbf{c}[w]$ ← E-Enc($\tau[w]$.pk, $\mathbf{s}[i]$)
    $T$ ← (ID, PK, $\mathbf{c}$)
    return ($\tau$, $T$, $\mathbf{W}_{\mathsf{priv}}$)
```

```
// Process control message
*refresh-proc ($\tau$, $T$)
    (ID, PK, $\mathbf{c}$) ← $T$
    $v \leftarrow \tau$[ME]
    $v' \leftarrow \tau$[ID]
    $w$ ← Rep($\tau$, $v$, $v'$)
    $s_0$ ← E-Dec($\tau[w]$.sk, $\mathbf{c}[v]$)
    $(z_0, \ldots, z_{\mathsf{d}})$ ← dPath($\tau$, $v'$)
    for $i = 0, \ldots, \mathsf{d} - 1$
        $\tau[z_i]$.pk ← PK[$i$]
    $(\tau, \cdot, \cdot)$ ← *refresh-path($\tau$, $s_0$, LCA($v$, $v'$))
    return $\tau$
```

Figure 18: The TreeKEM protocol operations that differ from RTreeKEM.

D Security of (R)TreeKEM

In this section we prove security or (R)TreeKEM against adaptive adversaries, with respect to specific safety predicates that instantiate the generic predicate of the CGKA security definition.

D.1 GSD for UPKE

In the current section we present the Generalized Selective Decryption (GSD) security notion for public-key encryption, modified to capture executions of the RTreeKEM protocol, in the presence of bad randomness and group splitting attacks. In particular, we allow the adversary to compute hashes of node secrets, capturing the path secret generation process of RTreeKEM (and TreeKEM), via hashing, while the underlying encryption scheme is updatable, thus, after encrypting under any public-key, it's value is updated as dictated by the UPKE scheme. We prove GSD security assuming IND-CPA UPKE and the random oracle model, by appropriately adapting the proof of [1].

GSD security is formally presented by the game defined in Figure 19, which is parameterized by the security parameter λ, the number of nodes N, a hash function H, the UPKE scheme $(\mathsf{UKGen}, \mathsf{UEnc}, \mathsf{UDec})$ and the adversary $\mathcal{A}$. We now describe the game.

Main execution. The main execution of the GSD game, **main**, initializes the following variables

- A graph (V, E) with N nodes and an empty set of edges.
- The set $\mathcal{C}$ of corrupted nodes stores node and GSD key id pairs, (u, gid), where gid is unique id for every UPKE key pair. gid is initialized to 0.
- $\mathsf{BR\text{-}E}[\cdot]$: dictionary s.t. $\mathsf{gid} \in \mathsf{BR\text{-}E}[u]$, if the UPKE key pair with id gid has been generated via encryption that uses bad (adversarially chosen) randomness.
- $\mathsf{BR\text{-}G}[\cdot]$: dictionary s.t. $\mathsf{BR\text{-}G}[u] = \mathsf{true}$ if the keys of u have been generated with bad randomness.
- $\mathsf{ctx}[\cdot, \cdot]$: dictionary s.t. $\mathsf{ctx}[u, \mathsf{gid}'] = (e, \mathsf{gid})$, if e is generated under the public key of node u with id gid' and results in a UPKE pair with id gid.
- $\mathsf{pk}[\cdot, \cdot]$: dictionary that stores UPKE public-keys.
- $\mathsf{sk}[\cdot, \cdot]$: dictionary that stores UPKE secret-keys.
- $\mathsf{s}[\cdot]$: dictionary that stores node secrets.
- $\mathsf{NS}[\cdot]$: dictionary that stores whether a node in the GSD graph is (not) a sink.
- $\mathcal{T}[\cdot]$: for a node u, $\mathcal{T}[u]$ is a tree in which each node is labeled by a key id, gid. An edge $(\mathsf{gid}, \mathsf{gid}')$, represents the fact that the key pair with id gid' has been generated by encrypting under the public-key with id gid. The root of $\mathcal{T}[u]$ is the id gid output by **key-gen** over u. Finally, $\mathcal{T}[u].\mathsf{isChild}(\mathsf{gid}, \mathsf{gid}') = \mathsf{true}$, if gid' is a child of gid in $\mathcal{T}[u]$, and $\mathcal{T}[u].\mathsf{isAnc}(\mathsf{gid}, \mathsf{gid}') = \mathsf{true}$, if gid is an ancestor of gid' in $\mathcal{T}[u]$.

After initializing the above variables, the main execution samples a uniformly random bit b and seed s', and executes the adversary with access to the remaining oracles (explained below). If the adversary produces a graph that complies with the restrictions of the GSD game, i.e., the challenge node is a sink and there is no trivial win (checked by ***gsd-comp**), the game returns $b = b'$, otherwise, returns false. Below we present the oracles of the GSD game.

The encryption oracle. The encryption oracle receives a pair of nodes, u, v, randomness r and key id gid', encrypts the node secret of v under the public-key of u with id gid'. This operation introduces the new edge $(u, v, (\mathsf{e}, \mathsf{gid}'))$ as well as a new child gid of gid' in $\mathcal{T}[u]$. If $r \neq \perp$, then r is used as the randomness for the encryption. Note that, u is not required to be part of the input to the encryption oracle, as the key id gid' uniquely defines the public key that will be used in the encryption operation, but we include it for clarity.

The decryption oracle. On input (u, gid'), the oracle decrypts the next ciphertext for the secret key of u with id gid'. Note that the oracle is not outputting any plaintexts; it only makes the secret key to evolve so that it remains synchronized with the corresponding public key. The reason for not outputting any plaintexts is that, in the proof, access to plaintexts will be required only w.r.t. compromised keys.

The hash computation oracle. On input (u, v) the hash oracle produces the node secret of v by hashing the secret of node u. In addition, a "hash edge", (u, v, h), is added to the set of edges. If the node secret of u is generated with bad randomness, then the node secret of v is marked as bad too.

The key-generation oracle. The oracle receives $(u, (r, \mathsf{pk}'), \mathsf{p})$. If $r \neq \perp$ then the node secret and keys of u are generated with bad randomness. If $\mathsf{pk}' \neq \perp$, the adversary sets the public key of node u to be equal to pk' (this is required whenever the adversary sets it's own public-key for a specific GSD node which relates to the **reg-ik** oracle of the CGKA game). If $\mathsf{p} = \mathsf{true}$, then **key-gen** outputs the key-pair, in which case u is not a sink.

Node compromise predicate. This predicate checks whether the adversary has trivially compromised security of a node u, which happens in the following cases: (1) the secrets of u have been generated with bad randomness, (2) $(v, \mathsf{gid}'') \in \mathcal{C}$, and there is a path from the compromised node v with the first edge being $(v, \cdot, (\mathsf{e}, \mathsf{gid}')$ and gid'' is not an ancestor of gid' in $\mathcal{T}[v]$, which implies that the adversary can compute the secret key with id gid', and (3), the key with id gid'' has been generated via an update over the key with id gid', with bad randomness, and then gid'' is compromised, which implies that the adversary can compute the secret key with id gid'.

The oracles **reveal**, **chall**, **corr**, are straightforward.

GSD Security Game

```
// Main execution
main (·)
    // Initialization
    $(V, E) \leftarrow ([N], \emptyset)$
    $\mathcal{C} \leftarrow \emptyset$
    $\mathsf{BR\text{-}E}[\cdot], \mathsf{BR\text{-}G}[\cdot], \mathsf{ctx}[\cdot,\cdot] \leftarrow \varepsilon$
    $\mathsf{pk}[\cdot,\cdot,\cdot] \leftarrow \varepsilon$
    $\mathsf{sk}[\cdot,\cdot,\cdot] \leftarrow \varepsilon$
    $\mathbf{s}[\cdot], \mathsf{NS}[\cdot] \leftarrow \varepsilon$
    $\mathcal{T}[\cdot] \leftarrow \varepsilon$
    $\mathsf{gid} \leftarrow 0$
    $b \leftarrow \{0,1\}$
    $s' \leftarrow \{0,1\}^\lambda$
    $\mathsf{chl} \leftarrow \mathsf{false}$
    $\mathcal{O} \leftarrow (\mathbf{enc}, \mathbf{dec}, \mathbf{hash}, \ldots$
      $\ldots, \mathbf{chall}, \mathbf{corr}, \mathbf{key\text{-}gen})$
    // Execution w.r.t. $\mathcal{A}$
    $b' \leftarrow \mathcal{A}^{\mathcal{O}}$
    // Game output
    if
        1. $(V, E)$ acyclic
        2. $\neg\mathsf{NS}[u^*]$ and
        3. $\neg$*gsd-comp$(u^*)$
        return $b = b'$
    else
        return false

// Corruption oracle
corr $(u, \mathsf{gid}')$
    req $\mathsf{sk}[u, \mathsf{gid}'] \neq \varepsilon$,
    $\mathcal{C} \mathrel{+}\!\!\leftarrow (u, \mathsf{gid}')$
    $(\mathsf{sk}_u, \cdot) \leftarrow \mathsf{sk}[u, \mathsf{gid}']$
    return $\mathsf{sk}_u$
```

```
// Encryption oracle
enc $(u, v, r, \mathsf{gid}')$
    req $\mathsf{pk}[u, \mathsf{gid}'], \mathbf{s}[v] \neq \varepsilon$
    $\mathsf{pk}_u \leftarrow \mathsf{pk}[u, \mathsf{gid}']$
    $E \mathrel{+}\!\!\leftarrow (u, v, (\mathsf{e}, \mathsf{gid}'))$
    $\mathsf{gid}{+}{+}$
    $\mathcal{T}[u].\mathsf{addChild}(\mathsf{gid}', \mathsf{gid})$
    if $r = \bot$
        $r \leftarrow \{0,1\}^\lambda$
    else
        $\mathsf{BR\text{-}E}[u] \mathrel{+}\!\!\leftarrow \mathsf{gid}$
    $(\mathsf{pk}_u, e) \leftarrow \mathsf{UEnc}(\mathsf{pk}_u, \mathbf{s}[v]; r)$
    $\mathsf{ctx}[u, \mathsf{gid}'] \leftarrow (e, \mathsf{gid})$
    $\mathsf{pk}[u, \mathsf{gid}] \leftarrow \mathsf{pk}_u$
    $\mathsf{NS}[u] \leftarrow \mathsf{true}$
    return $(\mathsf{pk}_u, e, \mathsf{gid})$

// Decryption oracle
dec $(u, \mathsf{gid}')$
    req $\mathsf{ctx}[u, \mathsf{gid}']$
    $\mathsf{sk}[u, \mathsf{gid}'] \neq \varepsilon$
    $(e, \mathsf{gid}'') \leftarrow \mathsf{ctx}[u, \mathsf{gid}']$
    $\mathsf{sk}_u \leftarrow \mathsf{sk}[u, \mathsf{gid}']$
    $(\mathsf{sk}_u, \cdot) \leftarrow \mathsf{UDec}(\mathsf{sk}_u, e)$
    $\mathsf{sk}[u, \mathsf{gid}''] \leftarrow \mathsf{sk}_u$

// Hashing oracle
hash $(u, v)$
    req $\mathbf{s}[u] \neq \varepsilon \wedge \mathbf{s}[v] = \varepsilon \wedge (u, *, \mathsf{h}) \notin E$
    $\mathbf{s}[v] \leftarrow \mathsf{H}(\mathbf{s}[u], \mathsf{path})$
    $E \mathrel{+}\!\!\leftarrow (u, v, \mathsf{h})$
    $\mathsf{BR\text{-}G}[v] \leftarrow (\mathsf{BR\text{-}G}[u] = \mathsf{true})$
    $\mathsf{NS}[u] \leftarrow \mathsf{true}$
```

```
// Challenge oracle
chall $(u)$
    req $\neg\mathsf{chl} \wedge \mathbf{s}[u] \neq \varepsilon$
    $u^* \leftarrow u$
    $\mathsf{chl} \leftarrow \mathsf{true}$
    if $b = 0$
        return $\mathbf{s}[u]$
    else
        return $s'$

// Key/node secret gen
key-gen $(u, (r, \mathsf{pk}'), \mathsf{p})$
    if $\mathbf{s}[u] = \varepsilon$
        if $r \neq \bot$
            $\mathbf{s}[u] \leftarrow r$
            $\mathsf{BR\text{-}G}[u] \leftarrow \mathsf{true}$
        else
            $\mathbf{s}[u] \leftarrow \{0,1\}^\lambda$
    if $\mathsf{pk}[u, \cdot] = \varepsilon$
        if $\mathsf{pk}' \neq \bot$
            $\mathsf{pk}_u \leftarrow \mathsf{pk}'$
            $\mathsf{sk}_u \leftarrow \varepsilon$
            $\mathcal{C} \mathrel{+}\!\!\leftarrow (u, \mathsf{gid})$
        else
            $\mathsf{sk}_u \leftarrow \mathsf{H}(\mathbf{s}[u], \mathsf{kgen})$
            $\mathsf{pk}_u \leftarrow \mathsf{UKGen}(\mathsf{sk})$
        $\mathsf{sk}[u, \mathsf{gid}] \leftarrow \mathsf{sk}_u$
        $\mathsf{pk}[u, \mathsf{gid}] \leftarrow \mathsf{pk}_u$
        $\mathcal{T}[u].\mathsf{root} \leftarrow \mathsf{gid}$
        $\mathsf{gid}{+}{+}$
    if $\mathsf{p}$
        $\mathsf{NS}[u] \leftarrow \mathsf{true}$
        return $(\mathsf{pk}, \mathsf{gid})$

// Node secret reveal oracle
reveal $(u)$
    req $u \neq u^* \wedge \mathbf{s}[u] \neq \varepsilon$
    $\mathsf{BR\text{-}G}[u] \leftarrow \mathsf{true}$
    return $\mathbf{s}[u]$
```

```
// Node compromise safety check
*gsd-comp $(u)$
    if $\mathsf{BR\text{-}G}[u] = \mathsf{true}$
        return true
    for $(v, \mathsf{gid}')$, s.t. a $v$-$u$ path exists,
      with first edge $(v, \cdot, (\mathsf{e}, \mathsf{gid}'))$
        if $\exists \mathsf{gid}''$ s.t.
            1. $(v, \mathsf{gid}'') \in \mathcal{C}$
            2. $\neg\mathcal{T}[v].\mathsf{isAnc}(\mathsf{gid}', \mathsf{gid}'')$
        or
            1. $\mathsf{gid}'' \in \mathsf{BR\text{-}E}[v]$
            2. $(v, \mathsf{gid}'') \in \mathcal{C}$
            3. $\mathcal{T}[v].\mathsf{isChild}(\mathsf{gid}', \mathsf{gid}'')$
            return true
    return false
```

Figure 19: The GSD game with respect to UPKE schemes, parameterized by the number of nodes N, a hash function H and security parameter λ.

Let $\mathsf{GSD}_{\mathcal{A},\mathsf{UPKE}}$ be the output of the GSD game with respect to $\mathcal{A}$ and scheme UPKE (here N and λ are implicit). The advantage of $\mathcal{A}$ in this game is defined as

$$\mathrm{Adv}_{\mathsf{gsd}}^{\mathsf{UPKE}}(\mathcal{A}) := 2\left|\Pr\left[\mathsf{GSD}_{\mathcal{A},\mathsf{UPKE}}(\lambda) = \mathsf{true}\right] - 1\right|.$$

Definition 8 (GSD security)**.** *A scheme* UPKE *is* (t, ε)*-GSD secure if for all* t*-attackers* $\mathcal{A}$*,*

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

We now prove the following theorem.

Theorem 7. *Assuming* UPKE *is an* ε*-IND-CPA secure updatable public-key encryption scheme and* H *is modeled as a random oracle with domain* $\{0,1\}^\lambda$*, then* UPKE *is* ε'*-GSD secure, where* $\varepsilon' = 2N^2 \cdot \varepsilon + \frac{mN}{2^\lambda}$*,* N *is the number of nodes in the GSD game and* m *is the number of queries to the random oracle.*

Proof. Similar to [1] we prove GSD security using a sequence of hybrids between the real game GSD_0, where a challenge query for the node v is answered with the real node secret s_v and the game GSD_1, where it is answered with an independent uniformly random seed. We define the sequence of hybrids as follows:

- $H_0 := \mathsf{GSD}_0$, i.e., the real GSD game.
- Let v be the challenge node and s' be a uniformly random and independent seed. For $1 \leq i \leq \mathsf{indeg}(v)$, the hybrid H_i is similar to H_{i-1}, except that the i^{th} query to the encryption oracle that encrypts the node secret of v returns the encryption of s'.

Observe that the game $\mathsf{GSD}_{\mathsf{indeg}(v)}$ matches GSD_1. Therefore, for any GSD adversary $\mathcal{A}$ with advantage ε, the advantage of $\mathcal{A}$ in distinguishing hybrids H_{i-1} and H_i is upper bounded by ε. Since two subsequent hybrid games differ in exactly one encryption edge, we will use this distinguishing advantage to solve an IND-CPA challenge for UPKE. To simulate game H_i, the reduction guesses the challenge node v as well as the source node u of the i^{th} encryption incident on v. We denote those guesses by v^* and u^*, respectively. Assuming $\neg$***gsd-comp**(v), the simulation, is only possible if $\mathcal{A}$ does not query its oracle on any of the node secrets that belongs to a parent node of of the challenge node $v = v^*$, otherwise it can trivially distinguish. This is captured by the event $\mathcal{E}$, adapted from [1]:

> Event $\mathcal{E}$: $\mathcal{A}$ queries the random oracle on a value that contains a node secret s_u for a non-challenge vertex u for which ***gsd-comp** is false (at the time of the RO query).

We follow the proof strategy of [1] and we prove our result, initially for adversaries that trigger $\mathcal{E}$ with small (negligible) probability (cf. Lemma 8), and then for adversaries that trigger $\mathcal{E}$ with large probability (cf. Lemma 9).

Lemma 8. *Let* GSD' *be a game which is defined similar to* GSD*, but the challenger aborts once event* $\mathcal{E}$ *(defined above) is triggered. Let* $\mathcal{A}$ *be an adversary against* GSD' *over* N *nodes, with advantage* ε*. Then, in ROM, there exists an adversary* $\mathcal{A}'$ *against the IND-CPA security of the underlying UPKE scheme with advantage greater than* $\frac{\varepsilon}{N^2}$*.*

Proof. Given an adversary $\mathcal{A}$ against GSD' we define an adversary $\mathcal{A}'$ against the IND-CPA security of UPKE, which, after receiving the challenge public key pk^*, executes the following steps:

- (**Initialization**):
  1. Execute the initialization steps of GSD' (cf. Figure 19).
  2. Sample $s, s' \leftarrow \{0,1\}^\lambda$, $v^* \leftarrow [N]$ and set $\mathbf{s}[v^*] \leftarrow s$.
  3. Sample $u^* \leftarrow [N]$.
  4. Set V, U, $\mathbf{ptx}[\cdot] \leftarrow \varepsilon$.

- (**Query simulation**): Run $\mathcal{A}$ and reply to it's queries as follows:
  - **enc**(u, v, r, gid'):
    - If $u = u^*$, $v \neq v^*$:
      1. Execute the code defined by the oracle.
      2. Set $\mathbf{ptx}\left[\mathsf{gid}'\right] \leftarrow (r, \mathbf{s}[v])$.
    - Else, if $(u, v) = (u^*, v^*)$:
      1. For $\mathsf{gid}'' \in \mathsf{dPath}(\mathcal{T}[u^*], \mathsf{gid}')\backslash\{\mathsf{gid}'\}$, send the encryption query $\mathbf{ptx}[\mathsf{gid}'']$ to GSD'.
      2. Send the challenge (s, s') to the GSD' challenger and receive $(\mathsf{pk}^*, \mathsf{sk}^*, e^*)$.
      3. Compute $\mathsf{gid}^* \leftarrow \mathsf{gid}{+}{+}$ and set $\mathsf{pk}[u^*, \mathsf{gid}^*] \leftarrow \mathsf{pk}^*$, $\mathsf{sk}[u^*, \mathsf{gid}^*] \leftarrow \mathsf{sk}^*$.
      4. Execute the remaining code (after the encryption operation) of the encryption oracle of GSD' and then set $\mathbf{s}[v^*] \leftarrow s'$, $\mathsf{U} \leftarrow \mathtt{true}$ and send the output of the simulated oracle query, $(\mathsf{pk}^*, e^*, \mathsf{gid}^*)$, to $\mathcal{A}$.
    - Else, execute the query normally and return the reply to $\mathcal{A}$.
  - **dec**(u, gid'):
    - If $u \neq u^* \vee (u = u^* \wedge \mathcal{T}[u].\mathsf{isChild}(\mathsf{gid}^*, \mathsf{gid}'))$, execute the query as in GSD'.
    - Otherwise, do nothing.
  - **hash**(u, v):
    - If $v = v^*$, then if $\mathsf{BR\text{-}G}[u]$, abort and output 1, else, execute as in GSD' but omit the hashing and edge creation operations.
    - Else, execute the query as in GSD' and return the output to $\mathcal{A}$.
  - **chall**(u): Execute the code of **chall**, and if $u = v^*$, set $\mathsf{V} \leftarrow \mathtt{true}$. Send $\mathbf{s}[v^*]$ to $\mathcal{A}$.
  - **corr**(u, gid'):
    - If $u = u^* \wedge \neg\mathcal{T}[u^*].\mathsf{isChild}(\mathsf{gid}^*, \mathsf{gid}')$, abort and output 1.
    - Otherwise, execute the query as in GSD'.
  - **key-gen**$(u, (r, \mathsf{pk}'), \mathsf{p})$:
    - If $u = u^*$, then if $r \neq \perp$ or $\mathsf{pk}' \neq \perp$, abort and output 1, otherwise, execute the query as in GSD' but set $\mathsf{pk}[u^*, \mathsf{gid}] \leftarrow \mathsf{pk}^*$.
    - Otherwise, execute the query as in GSD' (respecting all values set during initialization).
  - **reveal**(u):
    - If $u = u^*$, abort and output 1.
    - Otherwise, execute the code of the oracle and return the output to $\mathcal{A}$.
  - $\mathsf{H}(s)$: query its own random oracle with s and return the output to $\mathcal{A}$.

  Let b' the bit output by $\mathcal{A}$ after finishing it's queries to the GSD' oracles.
- (**Output**):
  - If $\mathcal{E} \vee \neg\mathsf{chl} \vee$ ***gsd-comp**$(v^*) \vee \neg\mathsf{V} \vee \neg\mathsf{U} \vee \neg\mathsf{NS}[v^*]$, output 1.
  - Else, output b'.

Description of $\mathcal{A}'$. During initialization $\mathcal{A}'$ samples challenge messages (node secrets), s, s', without sending them to the challenger yet as in the UPKE setting the challenge needs to be sent when the execution reaches the right version of the UPKE key-pair. Then, $\mathcal{A}'$ makes a guess on the challenge node by sampling v^* and sets it's seed value to be equal to s. Observe that this value will be used for pre-challenge queries; for post challenge queries the node secret of v^* is set to s'. Then, it guesses under which node's public key the node secret of v^* will be encrypted; this node is denoted by u^* and it's public key is later set to the challenge public key pk^* received by the challenger of the IND-CPA game against UPKE. The events/variables V, U, check whether the aforementioned guesses with respect to u^* and v^* are correct, and $\mathbf{ptx}$ stores pre-challenge messages that will be sent to the IND-CPA challenger.

In the "query simulation" phase $\mathcal{A}'$ simulates the responses to the queries that $\mathcal{A}$ makes against the GSD' oracles. For $\mathbf{enc}(u^*, v, r, \mathsf{gid}')$, $v \neq v^*$, $\mathcal{A}'$ executes the code of the encryption oracle, while also storing $(r, \mathsf{s}[v])$; $\mathcal{A}'$ stores that pair so that it sends it to the IND-CPA game as a pre-challenge message only when $\mathcal{A}$ challenges v^*. In this way, $\mathsf{pk}[u^*]$ will be fully synchronized with the corresponding public key of the IND-CPA game, while after the challenge query, the key sk^* output by the IND-CPA game will be synchronized with the corresponding secret key. For $\mathbf{enc}(u^*, v^*, r, \mathsf{gid}')$, $\mathcal{A}'$ first recovers the key ids gid'' in the direct path of gid' and the randomness/message pairs that were used in those encryptions, and sends those pairs, stored in $\mathbf{ptx}[\cdot]$, to the challenger. It then executes the challenge phase of the IND-CPA game, i.e., it sends s, s' to the UPKE challenger and receives $(\mathsf{pk}^*, \mathsf{sk}^*, e^*)$, and sets $\mathsf{s}[v^*]$ to s'. Also sets U to true which means that an (u^*, v^*) encryption edge has been created and u^* was guessed correctly. Other encryption queries are executed as $\mathbf{enc}$ dictates. Note that, pre-challenge encryptions of the node secret of v^* are answered with s while post-challenge ones with s'.

For $\mathbf{dec}(u, \mathsf{gid}')$, if $u \neq u^*$ or $u = u^*$ and decryption is for a key id gid', which is a child of the key id gid^* of $(\mathsf{pk}^*, \mathsf{sk}^*)$, then decryption is executed normally. In the former case, $\mathcal{A}'$ has sampled on it's own u's node secret and keys, while in the latter, $\mathcal{A}'$ learned the corresponding keys from the IND-CPA challenger, since this is a post-challenge decryption query related to a key-pair derived from sk^*, pk^*. For all other queries, $\mathcal{A}'$ performs no action, as they are related to keys in $\mathcal{T}[u^*]$, what will be never accessed by $\mathcal{A}$ (this is enforced by ***gsd-comp**).

For $\mathbf{hash}(u, v^*)$, $\mathcal{A}'$ simply omits the hashing and edge creation operations, in case u has been produced with good randomness, otherwise it aborts. For $v \neq v^*$ the query is executed normally. Note that, for u^*, $\mathcal{A}'$'s computation is with respect to $\mathsf{s}[u^*]$ and doesn't ever modify the initial key of u^* which is set to the challenge initial UPKE key pk^*. $\mathbf{chall}(u)$ is executed as in GSD' but also sets V to true if $u = v^*$, i.e., if $\mathcal{A}$ guesses correctly the challenged node. Corruption queries are identical to GSD', except when $\mathcal{A}$ corrupts a key-pair related to node u^*, which is not derived by the challenge pair $(\mathsf{pk}^*, \mathsf{sk}^*)$ (this is enforced by ***gsd-comp**), $\mathbf{key\text{-}gen}(u, (r, \mathsf{pk}'), \mathsf{p})$ queries execute as in GSD', unless $u = u^*$ and $r \neq \perp$ or $\mathsf{pk}' \neq \perp$, in which case $\mathcal{A}'$ aborts as the adversary fully defines the seed and/or keys for u^*. Assuming u^* correctly guesses the source node for the i^{th} incoming encryption edge to v^*, this is not a valid query. Finally, $\mathbf{reveal}(u)$ queries are executed normally, unless $u = u^*$, in which case v^* is not a valid challenge and $\mathcal{A}'$ aborts, and RO queries are forwarded to $\mathcal{A}$'s own RO.

Regarding the output of $\mathcal{A}'$, if the (bad) event $\mathcal{E}$ happens, or $\mathcal{A}$ does not challenge any node, or the challenged node is compromised according to the ***gsd-comp** predicate, or v^*, u^* are incorrect guesses for the challenge and source node, or v^* is not a sink node (as required by GSD'), $\mathcal{A}'$ outputs 1, otherwise, it outputs the bit output by $\mathcal{A}$.

Correctness of the simulation. Assuming that (u^*, v^*) is a correct guess of the "challenge edge", it's not hard to see that responses to queries unrelated to u^*, v^*, are distributed identically to GSD', since $\mathcal{A}'$ simulates perfectly the node secrets for those nodes, as well as, the corresponding keys. Since $\mathcal{E}$ is not triggered, we know that $\mathbf{s}[u^*]$ is not queried to the random oracle. Note that, in the presence of such a query, $\mathcal{A}'$ would have to come up with an RO query whose reply is consistent with the injected key-pair $(\mathsf{pk}^*, \mathsf{sk}^*)$, otherwise $\mathcal{A}$ would trivially distinguish the simulated execution from the real one; conditioned on $\neg\mathcal{E}$, and the fact that $\neg$***gsd-comp**(v^*) holds, the aforementioned attack cannot be mounted. Similarly, if v^* is created via a hash query, $\mathcal{A}'$ can simply ignore this edge, and choose a uniformly random $\mathbf{s}[v^*]$, as the inconsistency of this edge can only be verified by triggering $\mathcal{E}$. Therefore, assuming that (u^*, v^*) is a correct guess of the "challenge edge" and $\mathcal{E}$ is not triggered we have that all encryption queries with $u = u^*$ use the correctly updated version of the initial UPKE pk^* (i.e., $\mathcal{A}'$ fully simulates the key evolvement process of the UPKE challenge public key of the real GSD') since the public key can be updated in the absence of the secret key. Evolvement of the secret key is done via decryption, however, $\mathcal{A}'$ doesn't know how to perform pre-challenge decryptions related to u^* as it does not have access to the corresponding secret key (for all other nodes it decrypts normally). However, ***gsd-comp** allows only post-challenge compromise of u^*, and only with respect to keys that have been generated by updating $(\mathsf{pk}^*, \mathsf{sk}^*)$. Since $\mathcal{A}'$ learns $(\mathsf{pk}^*, \mathsf{sk}^*)$ after the challenge query, key evolvement of subsequent keys, as well as, decryption queries related to u^* are simulated correctly. Pre-challenge queries, encrypt s, while post-challenge ones encrypt s', thus if $\mathbf{enc}(u^*, v^*, r, \mathsf{gid}')$ is the i^{th} encryption request for an edge incident to v^*, then if $b^* = 0$, $\mathcal{A}'$ simulates H'_{i-1}[14] (i.e., the first i encryption requests related to edges incident to v^* encrypt s, while the remaining encrypt s'), while if $b^* = 1$, $\mathcal{A}'$ simulates H'_i (i.e., the first i encryption requests related to edges incident to v^* encrypt s, while the remaining encrypt s'). The two cases differ by only a single encryption edge, and distinguishing between H'_{i-1} and H'_i reduces to the IND-CPA security of UPKE, conditioned on $\neg\mathcal{E}$ and the fact that u^*, v^* are correct guesses for the "challenge edge". Since the use of the UPKE scheme does require any additional guessing by $\mathcal{A}'$ the probabilistic analysis and bound is along the lines of [1] and only depends on whether (u^*, v^*) are correct guesses of the challenge edge, which happens with probability $1/N^2$. □ □

The above lemma proves the needed when $\mathcal{E}$ is not triggered (or is triggered with very small probability). For adversaries that trigger $\mathcal{E}$ with larger probability, and following the proof strategy of [1], we first prove that any GSD adversary triggering event $\mathcal{E}$ can be reduced to an adversary against a partially selective version of GSD where the challenger aborts whenever $\mathcal{E}$ happens. In particular, let GSD'' be defined like GSD, with two differences: (1) the game aborts if $\mathcal{E}$ happens (in which case the adversary outputs 1) and (2) the adversary has to commit to the challenge node at the beginning of the game, i.e., GSD'' is partially selective. We prove the following lemma.

Lemma 9. *Assuming an adversary $\mathcal{A}$ that triggers $\mathcal{E}$ with probability ε in GSD with N nodes, there exists an adversary $\mathcal{A}''$ against GSD'' over $N+1$ nodes, with advantage $\frac{\varepsilon}{N} - \frac{m}{2^\lambda}$.*

Proof. The main idea behind the proof is the following: $\mathcal{A}''$ plays against GSD'' and simulates GSD for $\mathcal{A}$ until the latter triggers $\mathcal{E}$. Then $\mathcal{A}''$ uses $\mathcal{A}$'s query that triggered $\mathcal{E}$ to break security in GSD''. $\mathcal{A}''$ executes the following steps.

- (**Initialization**):
  - Sample $v^* \leftarrow [N]$, set $\mathsf{chl} \leftarrow \mathsf{false}$.
  - Query GSD'' with $\mathbf{key\text{-}gen}(N+1, (\perp, \perp), \mathsf{true})$ and let $(\mathsf{pk}_{N+1}, \mathsf{gid}_{N+1})$ be the reply.
- (**Query simulation**): Run $\mathcal{A}$ and reply to it's queries as follows:

[14] By H'_i we refer to a modified version of H_i (as defined above) in which the execution aborts if $\mathcal{E}$ happens.

– **enc**(u, v, r, gid'):

  * If $u = v^*$, query the encryption oracle of GSD'' with $(N+1, v, r, \mathsf{gid}')$.
  * Else, forward the query to the GSD'' challenger.

  Return the response to $\mathcal{A}$.

– **dec**(u, gid'):

  * If $u = v^*$, query the decryption oracle of GSD'' with $(N+1, \mathsf{gid}')$.
  * Else, forward the query to GSD''.

– **hash**(u, v):

  * If $u = v^*$, query the hash oracle of GSD'' with $(N+1, v)$.
  * Else, forward the query to the GSD'' challenger.
  * If $\neg\mathsf{chl} \wedge v = v^*$, send a challenge query to the GSD'' challenger for v^*, and let s^* be the reply.

– **chall**(u):

  * if $u \neq v^*$, send a **corr**(u) query to GSD'' and let s_u be the reply. Sample $c \leftarrow \{0,1\}$, set $s_0 \leftarrow s_u$, $s_1 \leftarrow \{0,1\}^\lambda$ and return s_c to $\mathcal{A}$.
  * Otherwise, abort and return 1.

– **corr**(u, gid'): forward the query to the GSD'' challenger. If $\mathcal{A}$ issues a corruption or encryption query such that ***gsd-comp**(v^*) is satisfied, abort and output 1.

– **key-gen**$(u, (r, \mathsf{pk}'), \mathsf{p})$:

  * If $u = v^*$,
    · if $r \neq \perp$ or $\mathsf{pk}' \neq \perp$, abort and output 1.
    · if p, send $(\mathsf{pk}_{N+1}, \mathsf{gid}_{N+1})$ to $\mathcal{A}$.
    · if $\neg\mathsf{chl}$, query GSD'' with **key-gen**$(v^*, (\perp, \perp), \mathsf{false})$, and query GSD'' with **chall**(v^*). Let s^* be the reply.
  * Otherwise, forward the query to GSD''.

– $\mathsf{H}(s)$:

  * If $s \neq s^*$, forward the query to the GSD'' challenger and forward the reply to $\mathcal{A}$.
  * Else, abort and output 0.

• (**Output**): Output 1 (executed only if no output has been produced yet).

$\mathcal{A}''$ initially guesses the node v^* that triggers $\mathcal{E}$, i.e., the node for which $\mathcal{A}$ makes an RO query with it's secret. Furthermore, it executes **key-gen** for the extra node $N+1$. $\mathcal{A}''$ challenges v^* as soon as its secret is defined (via a **hash** or a **key-gen** query). In order for v^* to be a valid challenge for GSD'' it must be a sink, thus $\mathcal{A}''$ substitutes encryption and hash queries with source v^* using the special node $N+1$. Assuming v^* is a correct guess, if s^* is the real node secret, then at some point during the execution it is queried by $\mathcal{A}$ and $\mathcal{A}''$ aborts and outputs 0. Otherwise, s^* is an independent value that with overwhelming probability is not queried by $\mathcal{A}$, and $\mathcal{A}''$ outputs 1. The way hash edges are treated is similar to encryption edges, i.e, for hash edges with source node v^*, node $N+1$ is used instead. However, since $\neg$***gsd-comp**(v^*) is satisfied, (1) s_{v^*} is generated with good randomness, and (2) there is no corrupted UPKE key for which there is a path from it to the challenge node in the key graph, thus the only way for $\mathcal{A}$ to detect the inconsistency is by querying the RO on the seed of either v^* or $N+1$. Also, ***gsd-comp** should be equal to false for $N+1$, since it is a source node in GSD'' and does not appear in GSD. Therefore, inconsistencies can be detected only by triggering $\mathcal{E}$. Clearly, the use of UPKE does not require additional guesses by $\mathcal{A}''$, thus the probabilistic argument is along the lines of [1]: if the challenge bit is 0, then $\mathcal{A}''$ outputs 0 with probability at least ε/N, otherwise, s^* is a uniformly random value and the probability that $\mathcal{A}$ queries RO with s^* (in which case $\mathcal{A}$ outputs 0) is upper bounded by $m/2^\lambda$. □ □

In order to finalize the proof and prove the needed for any $\mathcal{A}$ that triggers $\mathcal{E}$, first apply Lemma 9, to construct an adversary against GSD'', and then having a similar proof as in Lemma 9, to get a reduction to IND-CPA security of UPKE. The only difference in the proof is that the GSD game is less adaptive, i.e., the adversary commits to the challenge node at the beginning of the game, which implies that the reduction does not have to guess the challenge node, therefore, we only lose an additional factor $1/N$ due to guessing u^*. Therefore, the overall advantage is at least $\frac{\varepsilon}{N^2} - \frac{m}{N2^\lambda}$.
□ □

D.2 RTreeKEM security proof

In this section we prove security of RTreeKEM via a reduction to the GSD security of the underlying UPKE scheme. First we have to instantiate the safety predicate ***CGKA-priv** that is satisfied by RTreeKEM; the predicate is formally presented on Figure 20. In a high level, the key of epoch vid is considered insecure, if

- the state of some party ID' of a strict predecessor epoch vid' is leaked *and* there was *no update or commit operation with good randomness by* ID' *on the path from* vid' *to* vid (which is determined by $\mathsf{HG.path}$) or
- the state of some party ID' of an epoch vid' that lies in another branch, is leaked *and* there was *no update or commit operation with good randomness by* ID' *before* vid' (which is determined by $\mathsf{HG.path}$), or
- some party ID' was added to a predecessor vid' (potentially vid itself) with leaked initial keys, or
- the key of epoch vid has been generated with bad randomness.

Concretely, we prove the following theorem.

Theorem 1. *Let* UPKE *be an* (t,ε)*-GSD secure UPKE scheme. Furthermore, let* ***CGKA-priv** *be the safety predicate above. Then, RTreeKEM is an* (t',ε)*-secure SGM scheme w.r.t.* ****CGKA-priv****, where* $t' \approx t$*.*

Proof. Initially, we prove privacy of RTreeKEM assuming correctness, i.e., that the **win** condition inside the **process** and **dlv-WM** oracles of the CGKA game (cf. Figure 14), is not triggered, and then we prove that RTreeKEM satisfies correctness.

For the first part, given an adversary $\mathcal{A}$ that breaks the CGKA security of RTreeKEM with advantage ε we construct an adversary $\mathcal{A}'$ that breaks GSD security of UPKE with the same advantage. $\mathcal{A}'$ plays against the GSD game and simulates for $\mathcal{A}$ the CGKA game. $\mathcal{A}'$ is defined as follows.[15]

- (**Initialization**):

[15]GSD (resp. ratchet tree) nodes will be denoted using capital (resp. minuscule) letters.

CGKA Safety Predicate for RTreeKEM

```
*CGKA-priv (vid)
    if $\exists \mathsf{ID}', \mathsf{vid}' \neq \mathsf{vid}_{\mathsf{root}}$ s.t.
        1. $\mathsf{vid}' \neq \mathsf{vid} \wedge (\mathsf{vid}', \mathsf{ID}') \in \mathsf{V\text{-}Lk}$
           $\vee\ \mathsf{HG.addedIK}(\mathsf{vid}', \mathsf{ID}') \in \mathsf{IK\text{-}Lk}$
        2. $\mathsf{HG.path}(\mathsf{vid}', \mathsf{vid}, \mathsf{ID}')$
      or
        1. $\mathsf{vid}'' := \mathsf{LCA}(\mathsf{vid}, \mathsf{vid}') \notin \{\mathsf{vid}, \mathsf{vid}'\}$
        2. $(\mathsf{vid}', \mathsf{ID}') \in \mathsf{V\text{-}Lk}$
        3. $\mathsf{HG.path}(\mathsf{vid}'', \mathsf{vid}, \mathsf{ID}')$
        4. $\mathsf{HG.path}(\mathsf{vid}'', \mathsf{vid}', \mathsf{ID}')$
      return false
    if BR[vid] : return false
    return true

HG.path (vid', vid, ID)
    if $\exists$ direct path vid' to vid
      and
    $\nexists\ \mathsf{vid}'' \in (\mathsf{vid}', \mathsf{vid}]$ s.t.
        1. ID' removed in vid'' or
        2. ID' updates in vid'' with good randomness or
        3. ID' commits in vid'' with good randomness
        return true
return false
```

Figure 20: RTreeKEM's safety predicate.

- Execute the initialization phase of the CGKA security game (cf. Figure 11).
- Set $\mathcal{N}, \mathcal{C} \leftarrow \emptyset$ and $\mathsf{SK}[\cdot, \cdot] \leftarrow \varepsilon$.

- (**Query simulation**): Run $\mathcal{A}$ and reply to it's CGKA queries as follows:
  - **gen-new-ik**(ID):
    1. Compute $U \leftarrow \mathsf{gsd\text{-}node}$.
    2. Send the query **key-gen**$(U, (\perp, \perp), \mathsf{true})$ to the GSD challenger and let $(\mathsf{ipk}_U, \mathsf{gid})$ be the reply.
    3. Simulate **gen-new-ik**(ID) using ipk_U as the public key of ID, omitting $\mathsf{Gen\text{-}IK}$ and any instruction that is related to isk. Furthermore, set $\mathsf{IK\text{-}PK}[\mathsf{ikid}] \leftarrow (\mathsf{ipk}_U, U, \mathsf{gid})$. Return the output to $\mathcal{A}$.
  - **reg-ik**(ipk):
    1. Compute $U \leftarrow \mathsf{gsd\text{-}node}$.
    2. Send the query **key-gen**$(U, (\perp, \mathsf{ipk}), \mathsf{true})$ to the GSD challenger and let $(\mathsf{ipk}_U, \mathsf{gid})$ be the reply.
    3. Execute **reg-ik**(ipk) and set $\mathsf{IK\text{-}PK}[\mathsf{ikid}] \leftarrow (\mathsf{ipk}_U, U, \mathsf{gid})$. Return the output to $\mathcal{A}$.
  - **create-group**$(\mathsf{ID}, \mathsf{ikid}, r)$:
    1. Compute $U \leftarrow \mathsf{gsd\text{-}node}$.
    2. Send the query **key-gen**$(U, (r, \perp), \mathsf{false})$ to the GSD challenger.
    3. Execute the oracle instructions omitting $\mathsf{isk} \leftarrow \mathsf{IK\text{-}SK}[\mathsf{ikid}]$, $\mathsf{Key}[\mathsf{vid}] \leftarrow I$. Set $\mathsf{PC}[\mathsf{vid}] \leftarrow U$.
    4. Compute $(\mathsf{ipk}, U, \mathsf{gid}) \leftarrow \mathsf{IK\text{-}PK}[\mathsf{ikid}]$ and in $\mathsf{RTK\text{-}Create}(\mathsf{ID}, \cdot, r)$, execute a modified $\tau \leftarrow \mathtt{Init}(\mathsf{ID}, \cdot)$, such that $\tau[\mathsf{ID}].\mathsf{pk} = \mathsf{ipk}$ and $\tau[\mathsf{ID}].\mathsf{sk} = \varepsilon$.
    5. Set $\tau[\mathsf{ID}].\mathsf{gsd\text{-}inf} \leftarrow (U, \mathsf{gid})$.
  - **prop-add-user**$(\mathsf{ID}, \mathsf{ID}', \mathsf{ikid}')$: execute the code of the CGKA oracle with respect to

$\mathsf{RTK\text{-}Add}$.

– **prop-rem-user**$(\mathsf{ID}, \mathsf{ID}')$: execute the code of the CGKA oracle with respect to $\mathsf{RTK\text{-}Rem}$.

– **prop-up-user**(ID, r):

  1. Set $U \leftarrow \mathsf{gsd\text{-}node}$.
  2. Query the GSD challenger with **key-gen**$(U, (r, \perp), \mathsf{true})$ and let $(\mathsf{pk}_U, \mathsf{gid})$ be the reply.
  3. Set $\tau[\mathsf{ID}].\mathsf{gsd\text{-}inf} \leftarrow (\mathsf{pk}_U, \mathsf{gid})$.
  4. Execute all the instructions of the **prop-up-user** oracle with respect to $\mathsf{RTK\text{-}Update}$, omitting all operations related to sk.
  5. Return $(\mathsf{upd}, \mathsf{ID}, \mathsf{pk}_U)$ to $\mathcal{A}$.

– **commit**$(\mathsf{ID}, \mathbf{pid}, r)$: execute the instructions of the **commit** oracle w.r.t. a modified $\mathsf{RTK\text{-}Commit}(\mathbf{P}, r)$, that

  1. In $\mathsf{*apply\text{-}props}(\tau, \mathbf{P})$, omit the operation $\tau[\mathsf{ID}].\mathsf{sk} \leftarrow \mathsf{UPK}[\mathsf{pk}]$, in case $\mathsf{ID} = \mathsf{ME}$.
  2. Execute $\mathsf{*refresh\text{-}gen}'(\tau, r)$ and $\mathsf{*refresh\text{-}path}'(\tau, \tau[\mathsf{ID}], r)$ as they are depicted in Figure 21.
  3. Return the output of the oracle to $\mathcal{A}$, after striping W_{pub} from $\mathsf{gsd\text{-}inf}$.

– **process**$(\mathsf{vid}, \mathsf{ID})$: execute the code of the **process** oracle (besides the "If" statement that leads to **win**) w.r.t. a modified $\mathsf{RTK\text{-}Proc\text{-}Com}(T, \mathbf{P})$, that

  1. In $\mathsf{*apply\text{-}props}(\tau, \mathbf{P})$, omit the operation $\tau[\mathsf{ID}].\mathsf{sk} \leftarrow \mathsf{UPK}[\mathsf{pk}]$, in case $\mathsf{ID} = \mathsf{ME}$.
  2. Execute $\mathsf{*refresh\text{-}proc}(\tau, T)$ as it is depicted in Figure 21.

– **reveal**(vid): if $\mathsf{PC}[\mathsf{vid}] \neq \epsilon \wedge \neg\mathsf{Chall}[\mathsf{vid}]$,

  1. Set $\mathsf{Reveal}[\mathsf{vid}] \leftarrow \mathsf{true}$, $V \leftarrow \mathsf{PC}[\mathsf{vid}]$.
  2. Send a **reveal**(V) query to the GSD challenger and forward the reply, s, to $\mathcal{A}$.

– **chall**(vid): if $\mathsf{PC}[\mathsf{vid}] \neq \epsilon \wedge \neg\mathsf{Reveal}[\mathsf{vid}]$,

  1. Set $\mathsf{Chall}[\mathsf{vid}] \leftarrow \mathsf{true}$, $V \leftarrow \mathsf{PC}[\mathsf{vid}]$.
  2. Send a challenge query for V to the GSD challenger and forward the reply, s, to $\mathcal{A}$.

– **dlv-WM**$(\mathsf{vid}, \mathsf{ID})$: execute the code of the **dlv-WM** oracle (besides the "If" statement that leads to **win**) w.r.t. a modified $\mathsf{Join}(\mathsf{ID}, \mathsf{orig}, W_{\mathsf{pub}}, W_{\mathsf{priv}}, \mathsf{ikid})$, that

  1. Executes $\mathsf{RTK\text{-}init}(\mathsf{ID})$, $\tau \leftarrow W_{\mathsf{pub}}$, $(\mathsf{ipk}, V, \mathsf{gid}) \leftarrow \mathsf{IK\text{-}PK}[\mathsf{ikid}]$, **dec**$(V, \mathsf{gid})$.
  2. If $\mathsf{ikid} \in \mathsf{IK\text{-}Lk}$,
     (a) $\tau[\mathsf{ID}].\mathsf{sk} \leftarrow \mathsf{SK}[V, \mathsf{gid}]$.
     (b) $\tau[\mathsf{ID}].\mathsf{gsd\text{-}inf} \leftarrow (V, \mathsf{gid})$.
     (c) $(\tau[\mathsf{ID}].\mathsf{sk}, s) \leftarrow \mathsf{UDec}(\tau[\mathsf{ID}].\mathsf{sk}, W_{\mathsf{priv}})$.
     (d) $(\tau, \cdot, \cdot) \leftarrow \mathsf{*refresh\text{-}path}(\tau, s, \mathsf{LCA}(\mathsf{ME}, \mathsf{orig}))$.

– **corr**(ID): Set $\mathcal{G} \leftarrow \emptyset$.

  1. For vid s.t. $(\mathsf{vid}, \mathsf{ID}) \in \mathsf{V\text{-}Lk}$,
     (a) Set $\tau' \leftarrow \mathsf{WM}_{\mathsf{pub}}[\mathsf{vid}, \cdot]$.

   (b) For $v \in \mathsf{dPath}(\tau', \tau'[\mathsf{ID}])$, if $\tau'[v].\mathsf{pk} \neq \varepsilon$, $\mathcal{G} \mathrel{+\!\!\leftarrow} \tau'[v].\mathsf{gsd\text{-}inf}$.

2. For $\mathsf{ikid} \in \mathsf{IK\text{-}Lk}$, $(\cdot, V, \mathsf{gid}) \leftarrow \mathsf{IK\text{-}PK}[\mathsf{ikid}]$, $\mathcal{G} \mathrel{+\!\!\leftarrow} (V, \mathsf{gid})$.
3. For $(V, \mathsf{gid}) \in \mathcal{G}$,
   (a) If $(V, \mathsf{gid}) \notin \mathcal{C}$, set $\mathcal{C} \mathrel{+\!\!\leftarrow} (V, \mathsf{gid})$, query the GSD challenger with **corr**(V, gid), and let sk_v be the reply. Set $\mathsf{SK}[V, \mathsf{gid}] \leftarrow \mathsf{sk}_v$.
   (b) For $\tau \in \gamma[\mathsf{ID}]$ and for $v \in \mathsf{dPath}(\tau, \tau[\mathsf{ID}])$, if $\tau[v].\mathsf{gsd\text{-}inf} = (V, \mathsf{gid})$, set $\tau[v].\mathsf{sk} \leftarrow \mathsf{SK}[V, \mathsf{gid}]$.
4. Send $\gamma[\mathsf{ID}]$ to $\mathcal{A}$ (striped from $\mathsf{gsd\text{-}inf}$).

– **no-del**(ID): set $\mathsf{Del}[\mathsf{ID}] \leftarrow \mathsf{false}$.

RTreeKEM to GSD reduction part

```
// Sample an unused GSD node
gsd-node
    $U \leftarrow \min_x \{x \in [N] \mid x \notin \mathcal{N}\}$
    $\mathcal{N} \mathrel{+\!\!\leftarrow} U$
    return $U$

// Modified *refresh-gen
*refresh-gen' $(\tau, r)$
    $v \leftarrow \tau[\mathsf{ID}]$
    $\mathsf{d} \leftarrow \mathtt{DEPTH}(\tau, v)$
    $\mathbf{W}_{\mathsf{priv}}[\cdot], \mathbf{c}[\cdot] \leftarrow \varepsilon$
    $(\tau, \mathbf{s}, \mathsf{PK}) \leftarrow$ *refresh-path$(\tau, r, v)$
    $(v'_0, \ldots, v'_{\mathsf{d}-1}) \leftarrow \mathsf{coPath}(\tau, v)$
    for $i = 1, \ldots, \mathsf{d}$
        $v \leftarrow v.\mathsf{parent}$
        $(V, \cdot) \leftarrow \tau[v].\mathsf{gsd\text{-}inf}$
        for $w \in \mathsf{Res}(v'_{i-1})$
            $(W, \mathsf{gid}) \leftarrow \tau[w].\mathsf{gsd\text{-}inf}$
            if $\tau[w].\mathsf{ID} \in \tau.\mathsf{GI}.\mathbf{added}$
                // GSD oracle call
                $(\tau[w].\mathsf{pk}, \mathbf{W}_{\mathsf{priv}}[\tau[w].\mathsf{ID}], \mathsf{gid}') \leftarrow \mathbf{enc}(W, V, r, \mathsf{gid})$
            else
                // GSD oracle call
                $(\tau[w].\mathsf{pk}, \mathbf{c}[w], \mathsf{gid}') \leftarrow \mathbf{enc}(W, V, r, \mathsf{gid})$
            $\tau[w].\mathsf{gsd\text{-}inf} \leftarrow (W, \mathsf{gid}')$
    $T \leftarrow (\mathsf{ID}, \mathsf{PK}, \mathbf{c})$
    return $(\tau, T, \mathbf{W}_{\mathsf{priv}})$

// Modified *refresh-path
*refresh-path' $(\tau, r, v)$
    $\mathsf{ID} \leftarrow \tau[v].\mathsf{ID}$
    $\mathsf{d} \leftarrow \mathtt{DEPTH}(\tau, v)$
    $\mathbf{s}[\cdot], \mathsf{PK}[\cdot], \mathsf{SK}[\cdot] \leftarrow \varepsilon$
    $V \leftarrow$ gsd-node
    for $i = 0, \ldots, \mathsf{d} - 1$
        // GSD oracle call
        $(\mathsf{PK}[i], \mathsf{gid}) \leftarrow \mathbf{key\text{-}gen}(V, r, \mathsf{true})$
        $\tau[v].\mathsf{gsd\text{-}inf} \leftarrow (V, \mathsf{gid})$
        if $r \neq \bot$
            $\mathbf{s}[i] \leftarrow r$
            $\mathsf{SK}[i] \leftarrow \mathsf{UKGen}(\mathsf{H}(r, \mathsf{kgen}))$
            $r \leftarrow \mathsf{H}(r, \mathsf{path})$
        $V' \leftarrow$ gsd-node
        // GSD oracle call
        $\mathbf{hash}(V, V')$
        $\tau[v].\mathsf{label} \leftarrow (\mathsf{PK}[i], \mathsf{SK}[i])$
        if $i > 0 : \tau[v].\mathsf{unmerged} \leftarrow \emptyset$
        $v \leftarrow \tau[v].\mathsf{parent}$
        $V \leftarrow V'$
    if $r \neq \bot : \mathbf{s}[\mathsf{d}] \leftarrow r; \tau[v].I \leftarrow \mathbf{s}[\mathsf{d}]$
    $\mathsf{PC}[\mathsf{vid}] \leftarrow V$
    return $(\tau, \mathbf{s}, \mathsf{PK})$

// Modified *refresh-proc
*refresh-proc' $(\tau, T)$
    $(\mathsf{ID}, \mathsf{PK}, \mathbf{c}) \leftarrow T$
    $v \leftarrow \tau[\mathsf{ME}]$
    $v' \leftarrow \tau[\mathsf{ID}]$
    $w \leftarrow \mathsf{Rep}(\tau, v, v')$
    $(V, \mathsf{gid}) \leftarrow \tau[w].\mathsf{gsd\text{-}inf}$
    // GSD oracle call
    $\mathbf{dec}(W, \mathsf{gid})$
    if $\tau[w].\mathsf{sk} \neq \varepsilon$
        $(\tau[w].\mathsf{sk}, s_0) \leftarrow \mathsf{UDec}(\tau[w].\mathsf{sk}, \mathbf{c}[v])$
        $(\tau, \cdot, \cdot) \leftarrow$ *refresh-path$(\tau, s_0, \mathsf{LCA}(v, v'))$
    $(z_0, \ldots, z_{\mathsf{d}}) \leftarrow \mathsf{dPath}(\tau, v')$
    for $i = 0, \ldots, \mathsf{d} - 1$
        $\tau[z_i].\mathsf{pk} \leftarrow \mathsf{PK}[i]$
    return $\tau$
```

Figure 21: Part of the reduction for the proof of Theorem 1.

Correctness of $\mathcal{A}'$'s simulation. During initialization, $\mathcal{A}'$ executes the initialization phase of the CGKA security game and in addition initializes $\mathcal{N}, \mathcal{C}$, which denote the already assigned and corrupted, respectively, GSD nodes, and SK which will store the compromised UPKE secrete keys.

For key generation queries, **gen-new-ik**(ID), $\mathcal{A}'$ first allocates a new GSD node $U \leftarrow \mathsf{gsd\text{-}node}$ and then sends a key generation query **key-gen**$(U, (\bot, \bot), \mathsf{true})$, that uses honest randomness (since

$r = \perp$) and publishes the generated public key (since $\mathsf{p} = \mathsf{true}$). Using the public key returned by the **key-gen** oracle, ipk_U, $\mathcal{A}'$ executes the code of **gen-new-ik**(ID) and (1) omits the any operation that uses isk, (2) sets $\mathsf{IK\text{-}PK}[\mathsf{ikid}] \leftarrow (\mathsf{ipk}_U, U, \mathsf{gid})$, i.e., $\mathsf{IK\text{-}PK}[\mathsf{ikid}]$ stores the GSD node and gid of the newly generated init key-pair (this information will be used later to annotate the ratchet tree with the init key of ID). After executing the above steps, a new key pair for ID has been generated via the corresponding GSD node. $\mathcal{A}'$ does not get access to the isk_U but this does not affect the simulation as $\mathcal{A}$ is currently not having access to it. Correctness for **reg-ik** is trivial.

For **create-group**$(\mathsf{ID}, \mathsf{ikid}, r)$, $\mathcal{A}'$ first allocates a new GSD node $U \leftarrow \mathsf{gsd\text{-}node}$, that relates to the update secret I generated by the KwCreate operation. Then, it sends a key generation query to the GSD challenger with **key-gen**$(U, (r, \perp), \mathsf{false})$, i.e., it uses the randomness r provided by $\mathcal{A}$ and also sets $\mathsf{p} = \mathsf{false}$, since U relates to the update secret of the newly created group and no keys are published for it (it is a sink). Then, $\mathcal{A}$ execute the oracle instructions omitting $\mathsf{isk} \leftarrow \mathsf{IK\text{-}SK}[\mathsf{ikid}]$ and $\mathsf{Key}[\mathsf{vid}] \leftarrow I$, and marks U to be a potential challenge node (update secret) for epoch vid. It then creates a ratchet tree with a single node, and sets only the public key of that node to be the public key of the group creator ID. In particular, it executes, $(\mathsf{ipk}, U, \mathsf{gid}) \leftarrow \mathsf{IK\text{-}PK}[\mathsf{ikid}]$ and executes a modified $\tau \leftarrow \mathtt{Init}(\mathsf{ID}, \cdot)$, such that $\tau[\mathsf{ID}].\mathsf{pk} = \mathsf{ipk}$ and $\tau[\mathsf{ID}].\mathsf{sk} = \varepsilon$. Finally, it annotates the ratchet tree with the corresponding GSD node U, and GSD key id gid, by setting $\tau[\mathsf{ID}].\mathsf{gsd\text{-}inf} \leftarrow (U, \mathsf{gid})$. Clearly, the above steps create a single node ratchet tree with the public-key of ID, and relates the update secret to the GSD node U; if $r = \perp$ (resp. $r \neq \perp$), the update secret I is unknown (resp. known) to $\mathcal{A}$.

For **prop-add-user**$(\mathsf{ID}, \mathsf{ID}', \mathsf{ikid}')$, **prop-rem-user**$(\mathsf{ID}, \mathsf{ID}')$, $\mathcal{A}'$ simply executes the code of the CGKA oracles with respect to $\mathsf{RTK\text{-}Add}$ and $\mathsf{RTK\text{-}Rem}$, respectively, as all operations depend on public keys or history graph information, or just perform compatibility checks.

For **prop-up-user**(ID, r), $\mathcal{A}'$ first allocates a new GSD node $U \leftarrow \mathsf{gsd\text{-}node}$ that relates to the key-pair generated for ID's update operation. Then, it sends a key generation query, **key-gen**$(U, (r, \perp), \mathsf{true})$ to the GSD challenger, it receives the reply $(\mathsf{pk}_U, \mathsf{gid})$ and annotates the ratchet tree node of ID with GSD information by executing $\tau[\mathsf{ID}].\mathsf{gsd\text{-}inf} \leftarrow (\mathsf{pk}_U, \mathsf{gid})$. Finally, it executes all the instructions of the **prop-up-user** oracle with respect to $\mathsf{RTK\text{-}Update}$, omitting all operations related to sk. After executing the above steps, a new GSD node has been created which keys correspond the ID's new key-pair. Observe that, if $r = \perp$, then secret key remains private, while if $r \neq \perp$, the newly generated secret key is known to $\mathcal{A}$. However, at this point of the execution there is no need for $\mathcal{A}'$ to simulate sk_U as $\mathcal{A}$ can only get access to ID's state via corruption.

For **commit**$(\mathsf{ID}, \mathbf{pid}, r)$, $\mathcal{A}'$ executes the instructions of the **commit** oracle w.r.t. a modified $\mathsf{RTK\text{-}Commit}(\mathbf{P}, r)$, that (1) in $*\mathsf{apply\text{-}props}(\tau, \mathbf{P})$, omits the operation $\tau[\mathsf{ID}].\mathsf{sk} \leftarrow \mathsf{UPK}[\mathsf{pk}]$, in case $\mathsf{ID} = \mathsf{ME}$, i.e., does not set the secret key of UPKE (this will only be set if ID is corrupted, see below) and (2) executes $*\mathsf{refresh\text{-}gen}'(\tau, r)$ and $*\mathsf{refresh\text{-}path}'(\tau, \tau[\mathsf{ID}], r)$, as they are depicted in Figure 21. In particular, the modified "refresh path" operation, $*\mathsf{refresh\text{-}path}'$, refreshes the path secrets along the direct path of ID, by making calls the GSD oracles. Namely, for each node v in the direct path of ID, $*\mathsf{refresh\text{-}path}'$ allocates a new GSD node V, generates keys for V by calling GSD **key-gen**$(V, (r, \perp), \mathsf{true})$ (possibly using adversarially chosen randomness r), and computes the path secret of the parent node of v, by allocating a new GSD node V', and sampling it's path secret by making a call to the hash oracle, namely by calling **hash**(V, V'). Observe that if $r \neq \perp$, the ratchet tree output by $*\mathsf{refresh\text{-}path}'$ contains the newly generated secret keys in the direct path of ID. Furthermore, each node in the direct path is annotated with the $\mathsf{gsd\text{-}inf}$ variable that relates the node to the appropriate GSD node V and key id gid (this is needed for executing future operations). Finally, it sets the potential challenge for epoch vid to be the last node generated by the refresh path operation. Therefore, $*\mathsf{refresh\text{-}path}'$ simulates the original $*\mathsf{refresh\text{-}path}$, and manages, via calls to the GSD oracles, to generate path secrets and keys without touching the node's secrets and private keys (unless $r \neq \perp$). The information output by $^*\mathsf{refresh\text{-}path}'$, is used by $^*\mathsf{refresh\text{-}gen}'$, that simulates the encryption instruction made by $^*\mathsf{refresh\text{-}path}$, by making calls to the **enc** GSD oracles using the GSD information $\mathsf{gsd\text{-}inf}$ set by $^*\mathsf{refresh\text{-}path}'$. Therefore, $\mathcal{A}'$ correctly simulates the output of the **commit** oracle; it perfectly simulates W_{pub} and all ciphertexts in T and $\mathbf{W}_{\mathsf{priv}}$. Finally we note that, if $r \neq \perp$, $\mathcal{A}'$ does not compute the correct I, however, this does not affect the simulation, as $\mathcal{A}$ gets access to it only via reveal and challenge queries (see below).

For **process**$(\mathsf{vid}, \mathsf{ID})$, $\mathcal{A}'$ executes the code of the **process** oracle, besides the "If" statement that leads to **win** (since we have assumed correctness of RTreeKEM) w.r.t. a modified $\mathsf{RTK\text{-}Proc\text{-}Com}(T, \mathbf{P})$, that (1) in $^*\mathsf{apply\text{-}props}(\tau, \mathbf{P})$, omits the operation $\tau[\mathsf{ID}].\mathsf{sk} \leftarrow \mathsf{UPK}[\mathsf{pk}]$, in case $\mathsf{ID} = \mathsf{ME}$ (as we discussed above this does not affect the simulation since $\mathcal{A}$ only gets access to sk via a corruption query), and (2) executes a $^*\mathsf{refresh\text{-}proc}'(\tau, T)$ as it is depicted in Figure 21. Concretely, for a node v such that $v = \tau[\mathsf{ID}]$, $^*\mathsf{refresh\text{-}proc}'$ decrypts the ciphertext $\mathbf{c}[v]$, only if the secret key of the representative, w, has been already set in the ratchet tree of ID (which only happens after corrupting ID). If this is the case, $^*\mathsf{refresh\text{-}proc}'$ decrypts $\mathbf{c}[v]$, recovers the path secret and executes $^*\mathsf{refresh\text{-}path}$. Furthermore, it always sends a decryption query, $\mathbf{dec}(w, \mathsf{gid})$, to the GSD challenger so that the secret key of node w (denoted as W in the GSD game) is updated accordingly.

For **reveal**(vid), $\mathcal{A}'$ checks whether there is an update secret to epoch vid by checking whether $\mathsf{PC}[\mathsf{vid}] \neq \epsilon$, and if this is the case, and $\neg\mathsf{Chall}[\mathsf{vid}]$ is satisfied (as required by the CGKA definition), it computes $V \leftarrow \mathsf{PC}[\mathsf{vid}]$, sends a reveal query to GSD for V, and forwards the reply to $\mathcal{A}$. Therefore, simulation is perfect. The idea behind **chall** is similar, having a challenge query in place of the reveal query.

For **dlv-WM** $(\mathsf{vid}, \mathsf{ID})$, $\mathcal{A}'$ executes the code of the **dlv-WM** oracle (besides the "If" statement that leads to **win** since we have assumed correctness of RTreeKEM) w.r.t. a modified $\mathsf{Join}(\mathsf{ID}, \mathsf{orig}, W_{\mathsf{pub}}, W_{\mathsf{priv}}, \mathsf{ikid})$, that first relates the ipk of ID with the corresponding GSD node (by executing $(\mathsf{ipk}, V, \mathsf{gid}) \leftarrow \mathsf{IK\text{-}PK}[\mathsf{ikid}]$) and also sending a decryption query $\mathbf{dec}(V, \mathsf{gid})$ to the GSD challenger (so that isk is updated accordingly). In addition, if ipk is a leaked key, $\mathcal{A}'$ recovers the corresponding leaked isk by computing, $\tau[\mathsf{ID}].\mathsf{sk} \leftarrow \mathsf{SK}[V, \mathsf{gid}]$, $\tau[\mathsf{ID}].\mathsf{gsd\text{-}inf} \leftarrow (V, \mathsf{gid})$, and finally performs the normal decryption and refresh path operations, as defined in the code of Join. Again, the simulation is perfect.

For **corr** (ID), $\mathcal{A}'$ computes the epochs vid for which the state of ID is in $\mathsf{V\text{-}Lk}$, and for those epochs, checks the published ratchet trees and recovers the GSD information $\mathsf{gsd\text{-}inf}$ along the direct path of ID in all such trees and includes them in $\mathcal{G}$(this is in step 1). $\mathcal{A}'$ acts similarly in step 2 for leaked init keys, i.e., ikids in $\mathsf{IK\text{-}Lk}$. Then, in step 3, $(V, \mathsf{gid}) \in \mathcal{G}$, if $\mathcal{A}'$ sens a corruption query $\mathbf{corr}(V, \mathsf{gid})$ to the GSD challenger (if $(V, \mathsf{gid}) \notin \mathcal{C}$) and stores the reply. Then, for all ratchet trees τ in the state of ID $\gamma[\mathsf{ID}]$, and all nodes v in the direct path of ID in τ, if the GSD information $\mathsf{gsd\text{-}inf}$ is related to a corrupted node, $\mathcal{A}'$ sets it's secret key (this is step 3). Thus, all secret keys in the state of ID are being set and finally, the state $\gamma[\mathsf{ID}]$ (striped from $\mathsf{gsd\text{-}inf}$) is sent to $\mathcal{A}$. The secret keys that were set above will be used by all subsequent decryption operations for ID and the simulation is perfect. The case of **no-del**(ID) is trivial.

Clearly, if the bit b sampled by the GSD game challenger is equal to 0 (resp. 1), $\mathcal{A}'$ simulates the CGKA game with $b = 0$ (resp. $b = 1$). Therefore, if $\mathcal{A}$ breaks CGKA security with advantage ε, $\mathcal{A}'$ breaks GSD security with the same advantage. It only remains to prove that, if $\mathcal{A}$ is an admissible adversary against CGKA then $\mathcal{A}'$ is an admissible adversary against GSD security.

Claim 10. *If the CGKA's safety predicate* ***priv-safe** *(cf. Figure 14) w.r.t.* ***CGKA-priv** *(cf. Figure 20) is satisfied, then the GSD safety predicate (cf. Figure 19) is also satisfied.*

By definition, CGKA's safety predicate ***priv-safe** w.r.t. ***CGKA-priv** is satisfied if $\forall \mathsf{vid}$: $\mathsf{Chall}[\mathsf{vid}] \implies$ ***CGKA-priv**(vid). If there is no challenge in CGKA, the claim holds trivially as there is no challenge in the GSD game. Therefore, we assume a single challenge vid for which ***CGKA-priv**(vid) is satisfied; we refer to the node that keeps the update secret of vid in the GSD game by U^*. ***CGKA-priv**(vid) implies $\neg\mathsf{BR}[\mathsf{vid}]$, which in turn implies $\neg\mathsf{BR\text{-}G}[U^*]$, i.e., if the update secret of epoch vid is generated with good randomness, then the node secret of the challenge node of the GSD game is generated with good randomness. Furthermore, $\neg\mathsf{BR}[\mathsf{vid}]$ implies that all ciphertexts generated by the commit message for vid, are generated with good randomness. More formally, for all nodes v in the resolution of the co-path nodes of ID when it generates the commit for vid, let $(V, \mathsf{gid}') \leftarrow \tau[v].\mathsf{gsd\text{-}inf}$. Due to the commit operation for all such nodes V there exists a path from V to U^* in the GSD graph, however, since the commit for vid uses good randomness there should be no gid'' such that $\mathsf{gid}'' \in \mathsf{BR\text{-}E}[V]$ and $\mathcal{T}[V].\mathsf{isChild}(\mathsf{gid}', \mathsf{gid}'')$, therefore, the "**or**" clause of the ***gsd-comp**(U^*) is false.

Now by the CGKA safety predicate we have that there is no $\mathsf{ID}', \mathsf{vid}' \neq \mathsf{vid}_{\mathsf{root}}$, such that $\mathsf{vid}' \neq \mathsf{vid} \wedge (\mathsf{vid}', \mathsf{ID}') \in \mathsf{V\text{-}Lk}$ or $\mathsf{HG.addedIK}(\mathsf{vid}', \mathsf{ID}') \in \mathsf{IK\text{-}Lk}$, and $\mathsf{HG.path}(\mathsf{vid}', \mathsf{vid}, \mathsf{ID}')$, i.e., all corrupted users in epochs before vid, or users added with leaked init keys, have either been removed, or they have performed an update or commit operation. Furthermore, the same should condition should hold in all other branches of the history graph in case $\mathcal{A}$ performs splitting attacks (this is being enforced by the "**or**" clause of the CGKA safety predicate). We thus conclude that for all (V, gid'), s.t. a V-U^* path exists in the GSD graph, with first edge $(V, \cdot, (\mathsf{e}, \mathsf{gid}'))$, either $(V, \mathsf{gid}'') \notin \mathcal{C}$ or $\mathcal{T}[V].\mathsf{isAnc}(\mathsf{gid}', \mathsf{gid}'')$.

RTreeKEM correctness. Correctness of RTreeKEM is proven in [3], thus we we informally argue about it using the history graph terminology. Correctness is violated if for some vid, $\neg\mathsf{HG.checkGI}(\mathsf{vid}, \mathsf{GI})$ or $\mathsf{Key}[\mathsf{vid}] \neq I$, during the execution of **process** or **dlv-WM**. However, this is impossible. For any vid, the commit message for vid includes the proposals for newly added and removed users, as well as user updates. The same proposals that are added by the committer in the commit message are processed by the receivers of that message. Given the correctness of the compatibility check functions (cf. Figure 15) that are executed prior to each proposal operation and enforce consistency with respect to the current group state, and correctness of the *apply-props operation (cf. Figure 17) that updates the group information GI with newly added, removed parties, and updates, we conclude that group members that process commit messages for the same vid's compute the correct group state, thus $\neg\mathsf{HG.checkGI}(\mathsf{vid}, \mathsf{GI})$ is never triggered. Similarly, $\mathsf{Key}[\mathsf{vid}] \neq I$ is never triggered assuming the correctness of the underlying UPKE encryption scheme, as (part of) the computation that is being executed by the committer, is identical to the computation performed by the receiver of a commit message (cf. Figure 17). The argument for **dlv-WM** is identical. □ □

D.3 TreeKEM security

As we have already mentioned, TreeKEM is secure w.r.t. a weaker security predicate than RTreeKEM. Namely, we require, that either there is no post-challenge compromise (the notion of PCS in [3]), or if there is, compromise happens after the party has already updated it's state (the FSU notion of [3]).

The security proof for TreeKEM is similar to that of RTreeKEM and proceeds in two steps. First we consider a reduction from GSD with respect to standard public-key encryption, to the IND-CPA security of the underlying scheme. The GSD game is similar to the one for UPKE, but it's safety predicate ***gsd-comp** on node u is independent of gid, and in addition to bad randomness, it checks whether there exists an v-u path in the GSD graph for which v belongs to the set of compromised nodes. The result is formalized in the following theorem.

CGKA Safety Predicate for TreeKEM

```
// determines whether an epoch is compromised
*CGKA-priv (vid)
    if $\exists \mathsf{ID}', \mathsf{vid}' \neq \mathsf{vid}_{\mathsf{root}}$ s.t.
        1.  $\mathsf{vid}' \neq \mathsf{vid} \wedge (\mathsf{vid}', \mathsf{ID}') \in \mathsf{V\text{-}Lk}$
            $\vee\ \mathsf{HG.addedIK}(\mathsf{vid}', \mathsf{ID}') \in \mathsf{IK\text{-}Lk}$
        2.  $\mathsf{HG.path}(\mathsf{vid}', \mathsf{vid}, \mathsf{ID}')$
    or
        1.  $\mathsf{vid}'' := \mathsf{LCA}(\mathsf{vid}, \mathsf{vid}') \notin \{\mathsf{vid}, \mathsf{vid}'\}$
        2.  $(\mathsf{vid}', \mathsf{ID}') \in \mathsf{V\text{-}Lk}$
        3.  $\mathsf{HG.path}(\mathsf{vid}'', \mathsf{vid}, \mathsf{ID}')$
        4.  $\mathsf{HG.path}(\mathsf{vid}'', \mathsf{vid}', \mathsf{ID}')$
    or
    // Check for post-challenge corruptions
        1.  $\mathsf{vid}' \neq \mathsf{vid} \wedge (\mathsf{vid}', \mathsf{ID}') \in \mathsf{V\text{-}Lk}$
        2.  $\mathsf{HG.path}(\mathsf{vid}, \mathsf{vid}', \mathsf{ID}')$
        return false
    if $\mathsf{BR}[\mathsf{vid}]$ : return false
    return true

// Safety predicate
safe
    if $\forall \mathsf{vid}\ \neg\mathsf{Chall}[\mathsf{vid}] \vee$ *CGKA-priv(vid)
        return true
    return false
```

Figure 22: TreeKEM's safety predicate.

Theorem 11. *Assuming* PKE *is an ε-IND-CPA secure public-key encryption scheme and* H *is modeled as a random oracle with domain $\{0,1\}^\lambda$, then* PKE *is ε'-GSD secure, where $\varepsilon' = 2N^2 \cdot \varepsilon + \frac{mN}{2^\lambda}$, N is the number of nodes in the GSD game and m is the number of queries to the random oracle.*

The proof of the above theorem is similar to the one of Theorem 7.

Security of TreeKEM is reduced to the GSD security of the underlying PKE scheme. The safety predicate needs to also perform the FSU type of checks mentioned above (cf. Figure 22). The result is formalized in the following theorem.

Theorem 12. *Assuming* PKE *is an (t, ε)-GSD secure public-key encryption scheme, then TreeKEM is an (t', ε)-secure CGKA protocol, where $t' \approx t$.*

The proof of the theorem is similar to that of Theorem 1.

E Forward-Secure Group AEAD

E.1 Security

E.1.1 Main oracles.

The oracles of the FS-GAEAD game are depicted in Figure 23.

Initialization. At the onset of the FS-GAEAD security game, a random bit b and a uniformly random key k_e are chosen, and the initialization algorithm is run for all group members in G (which is specified by an argument to the initialization procedure). The security game also initializes a message counter $i[\mathsf{ID}]$ for each party. Additionally, the game maintains the following variables, which work analogously to their counterparts in the SGM game.

- a set Chall of pairs (S, i) recording that the ith message sent by S was a challenge,

FS-GAEAD Security Game: Main Oracles

```
// Initialize group
init ($G$)
    $b \leftarrow \{0,1\}$
    $k_e \leftarrow \mathcal{K}$
    for ID $\in G$
        $v$[ID] $\leftarrow$ Init($k_e$, $|G|$, ID)
        $i$[ID] $\leftarrow 0$
    Chall $\leftarrow \emptyset$
    AM-Tr[$\cdot$] $\leftarrow \emptyset$
    AM-Lk $\leftarrow \emptyset$
    AM[$\cdot,\cdot,\cdot$] $\leftarrow \epsilon$
    Del[$\cdot$] $\leftarrow \epsilon$

// S sends $m$ with AD $a$
send (S, $a$, $m$)
    ($v$[S], $e$) $\leftarrow$ Send($v$[S], $a$, $m$)
    $i$[S]++
    for R $\in G \setminus \{$S$\}$
        AM[S, $i$[S], R] $\leftarrow (a, m, e)$
    return $e$

// Corruption of ID
corr (ID)
    AM-Rcvd $\leftarrow$ {(S, $i$) | AM[S, $i$, ID] =
      received}
    AM-Lk +$\leftarrow$ (ID, AM-Rcvd, AM-Tr[ID])
    return $v$[ID]

// Message delivery
dlv-AM (S, $i$, R)
    req AM[S, $i$, R] $\neq$ {$\epsilon$, received}
    $(a, m, e) \leftarrow$ AM[S, $i$, R]
    ($v$[R], S$'$, $i'$, $m'$) $\leftarrow$ Rcv($v$[R], $a$, $e$)
    if (S$'$, $i'$, $m'$) $\neq$ (S, $i$, $m$)
        win
    if $\neg$Del[R]
        AM-Tr[R] +$\leftarrow$ (S, $i$)
    AM[S, $i$, R] $\leftarrow$ received

// stop deletions for ID
no-del (ID)
    Del[ID] $\leftarrow$ false

// Message injection
inj-AM ($a'$, $e'$, R)
    req $\forall$S, $i$, $i$ : AM[S, $i$, R] $\neq$ ($a'$, $i$, $e'$)
    ($v$[S], S$'$, $i'$, $m'$) $\leftarrow$ Rcv($v$[R], $a$, $e$)
    if $m' \neq \perp$
        if (*AM-sec(S$'$, $i'$)) $\vee$
          AM[S$'$, $i'$, R] = received
            win
    AM[S$'$, $i'$, R] $\leftarrow$ received
    return (S$'$, $i'$, $m'$)

// S challs $m_0$, $m_1$
chall (S, $a$, $m_0$, $m_1$)
    req $|m_0| = |m_1|$
    ($v$[S], $e$) $\leftarrow$ Send($v$[S], $a$, $m_b$)
    $i$[S]++
    for R $\in G \setminus \{$S$\}$
        AM[S, $i$[S], R] $\leftarrow (a, m, e)$
    Chall +$\leftarrow$ (S, $i$)
    return $e$

// Safety predicate
*priv-safe
    if $\forall$(S, $i$) : (S, $i$) $\in$ Chall $\Longrightarrow$
      *FS-sec(S, $i$)
        return true
    return false
```

Figure 23: The main oracles of the FS-GAEAD security game.

- a trash array AM-Tr[ID] of pairs (S, i) keeping track of non-deleted key material by ID,
- a set AM-Lk of elements (ID, AM-Rcvd, AM-Tr[ID]), where AM-Tr[ID] keeps track of the messages whose key material is leaked via corruption of ID, and AM-Rcvd are the messages that have been already received at the time of corruption.
- a Boolean array Del[ID] keeping track of which parties are deleting old values, and
- an array AM[S, i, R] of triples (a, m, e) consisting of associated data (AD), plaintext, and ciphertext.

The recorded data informs a safety predicate ***priv-safe** evaluated at the end of the game to determine whether a given execution was legal.

Sending messages and challenges. The oracle **send**(S, a, m) allows the attacker to have party S send AD a and message m (to all other group members). The oracle runs algorithm Send, which produces a ciphertext e. The triple (a, m, e) is recorded in array AM. Oracle **chall** works similarly, except that it takes two (equal-length) messages as input and passes one of them to Send; which one is chosen is determined by the secret random bit b chosen initially. Furthermore, the pair (S, i[S]) is recorded as being a challenge.

Delivering and injecting ciphertexts. Oracle **dlv-AM**(S, i, R) allows $\mathcal{A}$ to have the ciphertext corresponding to the i^{th} message sent by S delivered to R. The ciphertext and the corresponding AD are fed to algorithm Rcv, which must correctly decrypt the ciphertext and identify sender S and message number i. The pair (S, i) is set as "received" by R by setting AM[S, i, R] $\leftarrow$ `received` which indicates that the corresponding key material should now have been deleted by R; in case R does not delete old values (i.e., Del[R] = false), the pair is added to AM-Tr.

Oracle **inj-AM** can be used by $\mathcal{A}$ to inject any non-honestly generated AD/ciphertext pair. Algorithm Rcv must reject all such pairs unless they are compromised, which is the case if $\mathcal{A}$ has learned the corresponding key material via state compromise. Whether or not this is the case is determined by the safety helper function ***FS-sec**, which is discussed below. Irrespective of whether a compromise has occurred, Rcv is required to detect and prevent replays.

Corruption. By calling oracle **corr**(ID) the attacker can learn the current state of party ID. The game adds the triple $(\mathsf{ID}, \mathsf{AM\text{-}Rcvd}, \mathsf{AM\text{-}Tr}[\mathsf{ID}])$ to the set $\mathsf{AM\text{-}Lk}$ to indicate that ID's state is now compromised, but the messages recorded in $\mathsf{AM\text{-}Rcvd}$ are supposed to remain secure (because the corresponding key material must have been deleted); the game also adds potential trash, $\mathsf{AM\text{-}Tr}[\mathsf{ID}]$, stored by ID to the same set.

E.1.2 Privacy-related safety.

At the end of the execution of the FS-GAEAD security game, the procedure ***priv-safe** ensures that the attacker has only challenged messages that are considered secure by the (generic) safety predicate ***FS-sec**. If the condition is not satisfied, the attacker loses the game.

E.1.3 Advantage.

Let $\Pi =$ ***FS-sec** be the generic safety predicate used in the FS-GAEAD definition. The attacker $\mathcal{A}$ is parameterized by it's running time t and the number of challenge queries q, referred to as (t, q)-attacker. The advantage of $\mathcal{A}$ against an FS-GAEAD scheme F w.r.t. to predicate Π is denoted by $\mathrm{Adv}^{\mathsf{FS}}_{\mathrm{fs},\Pi}(\mathcal{A})$.

Definition 9. *An FS-GAEAD scheme* F *is* (t, q, ε)-secure w.r.t. predicate Π*, if for all* (t, q)*-attackers,*

$$\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FS\text{-}GAEAD},\Pi}(\mathcal{A}) \ \leq \ \varepsilon \ .$$

E.2 Simplified Properties

Proving FS-GAEAD security is facilitated by considering three simplified properties, namely *correctness*, *authenticity*, and *privacy*. These properties together imply full FS-GAEAD security, as stated at the end of this section. All three of them are obtained by modifying the original game (cf. 23).

Correctness. For the correctness game $\mathsf{FSAE\text{-}Corr}$, there is no challenge oracle **chall** and we introduce a *reduced* injection oracle **reduced-inject**, which is presented in Figure 24.

The advantage of $\mathcal{A}$ in the correctness game with respect to a scheme F and predicate Π, is denoted by $\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Corr},\Pi}(\mathcal{A})$.

Definition 10. *An encryption scheme* F *is* (t, ε)-correct *w.r.t predicates* Π*, if for all t-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Corr},\Pi}(\mathcal{A}) \ \leq \ \varepsilon \ .$$

Authenticity. In the authenticity variant of the FS-GAEAD game, there are is no challenge oracle **chall**, and no **win** instruction inside the delivery oracle **dlv-AM**. Moreover, the adversary needs to call **init** with two additional arguments, namely, the id of the sender S and the message index i that he will use for the injection attack. If during the execution, the adversary uses different sender and message index, the adversary loses the game immediately.
The advantage of $\mathcal{A}$ in the authenticity game w.r.t. scheme F and predicate Π, is denoted by $\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Auth},\Pi}(\mathcal{A})$.

FS-GAEAD Correctness Game: The Reduced Inject Oracle

```
// Reduced message injection
reduced-inject ($a$, S, $i$[S], $e$, R)
    req ($a$, $e$) $\neq$ AM[S, $i$[S], R]
    if *FS-sec(S, $i$[S])
        ($v$[S], S$'$, $i'$, $m'$) $\leftarrow$ Rcv($v$[R], $a$, $e$)
        AM[S$'$, $i'$, R] $\leftarrow$ received
        if $\neg$Del[R]
            AM-Tr[R] $+\!\leftarrow$ (S$'$, $i'$)
    return (S, $i$[S], $m$)
```

Figure 24: The oracle for injecting application messages for the correctness and privacy properties.

Definition 11. *A encryption scheme* F *is* (t, ε)-authentic *w.r.t. predicates* Π*, if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Auth},\Pi}(\mathcal{A}) \;\leq\; \varepsilon\ .$$

Privacy. The privacy variant of the FS-GAEAD game only provides the reduced inject oracle of Figure 24, and there is no **win** instruction inside the delivery oracle **dlv-AM**. Moreover, the adversary needs to call **init** with two additional arguments, namely, the id of the sender S and the message index i of ID in the challenge query. If during the execution, the adversary challenges a different sender and message index, it loses the game immediately.
The advantage of $\mathcal{A}$ in the privacy game w.r.t. scheme F and predicate Π, is denoted by $\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Priv},\Pi}(\mathcal{A})$.

Definition 12. *An encryption scheme* F *is* (t, ε)-private *w.r.t. predicates* Π*, if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Priv},\Pi}(\mathcal{A}) \;\leq\; \varepsilon\ .$$

Combining the properties. The following theorem relates the simplified properties to the full SGM security definition.

Theorem 13. *Assume an encryption scheme* F *is*

- $(t', \varepsilon_{\mathsf{corr}})$*-correct,*
- $(t', \varepsilon_{\mathsf{auth}})$*-authentic, and*
- $(t', \varepsilon_{\mathsf{priv}})$*-private,*

w.r.t. predicate Π*. Then, it is also* (t, q, ε)*-secure w.r.t.* Π*, where* $\varepsilon \;\leq\; \varepsilon_{\mathsf{corr}} + t^2 \cdot (\varepsilon_{\mathsf{auth}} + q \cdot \varepsilon_{\mathsf{priv}})$ *and* $t \approx t'$*.*

Proof. The proof is via a series of hybrid experiments, where in the last experiment the adversarial advantage is 0. The distances between successive hybrids are bounded via reductions to correctness, authenticity, and privacy.

Let $\mathcal{A}$ be any (t, q)-fs-gaead adversary. We define the hybrids as follows.

Hybrid H_0**.** Hybrid H_0 is the original FS-GAEAD game with respect to F. We denote the advantage of $\mathcal{A}$ in H_0 by $\mathrm{Adv}^{\mathsf{F}}_{H_0,\Pi}(\mathcal{A})$.

E.3 Dealing with Correctness

Hybrid H_1. Hybrid H_1 works as H_0, except that the **win** condition inside the oracle **dlv-AM** is removed. We prove the following lemma.

Lemma 14. *There exists a t'-attacker $\mathcal{B}_1$ with $t' \approx t$ such that,*

$$\mathrm{Adv}^{\mathsf{F}}_{H_0,\Pi}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\mathsf{F}}_{H_1,\Pi}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Corr},\Pi}(\mathcal{B}_1)\ .$$

Proof. For H_0 we consider the event $\mathcal{E}$ that the **win** condition inside **dlv-AM** is provoked; observe that, since the game ends upon **win**, if $\mathcal{E}$ occurs, it occurs before condition **win** is triggered inside the inject oracle. It suffices to upper bound the probability of $\mathcal{E}$ occurring in H_0. This is achieved via a reduction: Consider an attacker $\mathcal{B}_1$ that plays against the correctness game with respect to F, executes $\mathcal{A}$ and attempts to simulate H_0 to $\mathcal{A}$. I.e., $\mathcal{B}_1$ has access to all FS-GAEAD oracles but **chall** and also receives access to the reduced injection oracle **reduced-inject** (instead of **inj-AM**). On the other hand, H_0 provides access to all FS-GAEAD oracles.

The simulation proceeds as follows (the oracles $\mathcal{B}_1$ interacts with are referred to as $\mathcal{B}_1$'s *own* oracles):

- Initially, choose a bit $b \in \{0,1\}$ randomly, set $\mathsf{Chall} \leftarrow \emptyset$ and for all $\mathsf{ID} \in G$ set $i[\mathsf{ID}] \leftarrow 0$. Run $\mathcal{A}$.
- Simulate the FS-GAEAD game oracles for $\mathcal{A}$ as follows:
  - For calls to **init**, **send**, **corr**, **no-del**, simply forward to own and return the outputs to $\mathcal{A}$. For every call to **send** with sender S set $i[\mathsf{S}]$++.
  - **chall** $(\mathsf{S}, a, m_0, m_1)$: If $|m_0| = |m_1|$ send (S, a, m_b) to **send**, return the output to $\mathcal{A}$, set $i[\mathsf{ID}]$++ and $\mathsf{Chall} \mathrel{+}\!\!\leftarrow (\mathsf{S}, i[\mathsf{S}])$. Otherwise, do nothing.
  - **dlv-AM** $(\mathsf{S}, i, \mathsf{R})$: execute own oracle on the same input, and if $(\mathsf{S}, i) \in \mathsf{Chall}$, return $\perp$ to $\mathcal{A}$ and set $\mathsf{Chall} \mathrel{-}\!\!\leftarrow (\mathsf{S}, i)$, otherwise pass the response to $\mathcal{A}$.
  - **inj-AM**: forward to **reduced-inject**.

We now show that $\mathcal{B}_1$ correctly simulates H_0. For challenge queries, if $|m_0| = |m_1|$, then the output of the **send** oracle is identical to that of **chall** and by setting $\mathsf{Chall} \mathrel{+}\!\!\leftarrow (\mathsf{S}, i[\mathsf{S}])$, $\mathcal{B}_1$ fully simulates the computation of **chall** (here note that $\mathcal{B}_1$ correctly simulates the index i). For **dlv-AM**, an extra check is required, namely $\mathcal{B}_1$ checks if $(\mathsf{S}, i) \in \mathsf{Chall}$, in which case it returns $\perp$ to $\mathcal{A}$, as it would happen in H_0.

For injection queries, recall that whenever $\mathcal{E}$ occurs, a **win** condition inside **dlv-AM** is provoked, thus no **win** is triggered inside the injection oracle. Therefore, the reduced inject oracle **reduced-inject** of the correctness game game is identical to **inj-AM**, as the message returned by both oracles is generated using keys from compromised epochs. All the remaining queries are handled by $\mathcal{B}_1$ as in H_0. Hence, the probability that $\mathcal{B}_1$ wins against the correctness game is equal to the probability that $\mathcal{A}$ provokes $\mathcal{E}$ in H_0, and the proof is concluded. □ □

E.4 Getting Rid of Injections

Hybrid H_2. Hyrbid H_2 works as H_1, but with the *reduced* inject oracle **reduced-inject**, as defined in Figure 24. In the following, denote the advantage of $\mathcal{A}$ in H_2 by $\mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{A})$.

Lemma 15. *There exists a t'-attacker $\mathcal{B}_2$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\mathsf{F}}_{H_1,\Pi}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{A}) + t^2 \cdot \mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Auth},\Pi}(\mathcal{B}_2)\ .$$

Proof. In order to distinguish H_1 and H_2, $\mathcal{A}$ must provoke the **win** condition inside the inject oracle since H_1 and H_2 behave identically otherwise. Note that, if **win** is not triggered, the reduced inject oracle, **reduced-inject**, is identical to **inj-AM**, i.e., either $m' = \perp$ or the adversary injects a ciphertext that has been computed with compromised keys. Therefore, it suffices to upper bound the probability of this event $\mathcal{E}$ in H_1, which is achieved via a reduction: Consider an attacker $\mathcal{B}_2$ against authenticity that uses $\mathcal{A}$ and attempts to simulate H_1 to $\mathcal{A}$.

The simulation proceeds as follows (the oracles $\mathcal{B}_2$ interacts with are referred to as $\mathcal{B}_2$'s *own* oracles):

- Initially, choose uniformly random $\mathsf{S} \in G$, index $i \in [t]$ and bit $b \in \{0, 1\}$, set $\mathsf{Chall} \leftarrow \emptyset$ and for all $\mathsf{ID} \in G$ set $i[\mathsf{ID}] \leftarrow 0$. Run $\mathcal{A}$.
- Simulate the FS-GAEAD game oracles for $\mathcal{A}$ as follows:
  - For calls to **send**, **corr**, **no-del**, simply forward to own and return the output to $\mathcal{A}$. For every call to **send** with sender S set $i[\mathsf{S}]{+}{+}$.
  - **init**: execute it's own **init** with input (S, i).
  - **chall** $(\mathsf{S}, a, m_0, m_1)$: If $|m_0| = |m_1|$ send (S, a, m_b) to **send**, return the output to $\mathcal{A}$, set $i[\mathsf{ID}]{+}{+}$ and $\mathsf{Chall} \mathrel{+}\!\!\leftarrow (\mathsf{S}, i[\mathsf{S}])$. Otherwise, do nothing.
  - **dlv-AM** $(\mathsf{S}, i, \mathsf{R})$: execute own oracle on the same input, and if $(\mathsf{S}, i) \in \mathsf{Chall}$, return $\perp$ to $\mathcal{A}$ and set $\mathsf{Chall} \mathrel{-}\!\!\leftarrow (\mathsf{S}, i)$, otherwise pass the response to $\mathcal{A}$.
  - **inj-AM**: forward to own **inj-AM** oracle.

Observe that for both the authenticity game and H_1, the **win** conditions has been removed from the delivery oracle, while in the former there is no challenge oracle. The arguments for proving that challenge queries are simulated correctly, is identical to that of Lemma 15. For queries to the remaining oracles $\mathcal{B}_2$ simply forwards them to it's own oracles. Hence, the probability that $\mathcal{B}_2$ triggers **win** in the authenticity game is at least $\frac{1}{t^2} \cdot \mathsf{Pr}[\mathcal{E}]$, i.e., the probability that $\mathcal{A}$ provokes $\mathcal{E}$ in H_1 times the probability of correctly guessing the id of the sender and the index of the message used for the injection (the number of all id/index pairs is upper bounded by t^2, where t is $\mathcal{A}$'s running time). □ □

E.5 Privacy

Recall that, in H_2 there is only a *reduced* inject oracle **reduced-inject**. Also, there is no **win** condition inside the oracle **dlv-AM**. Thus, H_2 constitutes an adaptive, multi-challenge version, of the privacy game. We prove the following lemma.

Lemma 16. *There exists a $(t', 1)$-attacker $\mathcal{B}$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{A}) \;\leq\; q \cdot \mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{B}) \ .$$

Proof. We define the following sequence of hybrids.

Hybrids H_i^{p}. These hybrids, for $i = 1, \ldots, q$, work as H_2, but up to the i-th challenge query the game encrypts m_0 (i.e., we assume $b = 0$), while for the remaining challenge queries it encrypts m_1. Let $\mathcal{E}$ be the event that $\mathcal{A}$ wins the privacy game against F, and let let $\mathcal{A}(H_i^{\mathsf{p}})$ be the output of $\mathcal{A}$

when playing against H_i^{P}. Using the hybrid definitions and the facts that (i) H_c^{P} matches H_2 and the privacy game, for $b = 0$, and (ii) H_0^{P} matches H_2 and the privacy game, for $b = 1$, we compute,

$$\mathsf{Pr}[\mathcal{E}] = \frac{1}{2}\left(\mathsf{Pr}[\mathcal{A}(H_c^{\mathsf{P}}) = 0] + \mathsf{Pr}\left[\mathcal{A}(H_0^{\mathsf{P}}) = 1\right]\right).$$

Given $\mathcal{A}$ we define a singe challenge adversary $\mathcal{B}$ as follows:

- Choose uniformly random $i \leftarrow [q]$, set $\mathsf{Chall} \leftarrow \emptyset$, $j \leftarrow 0$, and for all $\mathsf{ID} \in G$ set $i[\mathsf{ID}] \leftarrow 0$. Run $\mathcal{A}$.
- Forward $\mathcal{A}$'s messages to own oracles and return the outputs to $\mathcal{A}$, except for the calls to **send**, **chall** and **dlv-AM**, which are handled as follows:
  - **send**(S, a, m): set $i[\mathsf{S}]{++}$, forward to own and return the output of the oracle to $\mathcal{A}$.
  - **chall**$(\mathsf{S}, a, m_0, m_1)$: If $|m_0| = |m_1|$, set $j{++}$, $i[\mathsf{S}]{++}$, $\mathsf{Chall} \mathrel{+}\!\leftarrow (\mathsf{S}, i[\mathsf{S}])$, and
    - If $j = i$, forward to own and return the reply to $\mathcal{A}$.
    - Otherwise,
      1. If $j < i$, set $m \leftarrow m_0$.
      2. If $j > i$, set $m \leftarrow m_1$.
      3. Send (S, a, m) to own **send** and return the output to $\mathcal{A}$.
  - **dlv-AM** $(\mathsf{S}, i, \mathsf{R})$: forward to own oracle, and if $(\mathsf{S}, i) \in \mathsf{Chall}$, return $\perp$ to $\mathcal{A}$, otherwise forward the response to $\mathcal{A}$.

Finally, $\mathcal{B}$ outputs the same guess of bit b as $\mathcal{A}$.

Note that in the above execution, the i^{th} ciphertext given to $\mathcal{A}$ is the challenge ciphertext of $\mathcal{B}$ in the single challenge game. Intuitively, $\mathcal{B}$ is guessing an index $i \in [q]$ for which $\mathcal{A}$ distinguishes between H_i^{P} and H_{i-1}^{P}, which implies it can distinguish between an encryption of m_0 and m_1 (the messages used in the i^{th} call to the challenge oracle), which is the only difference between H_i^{P} and H_{i-1}^{P}. The probabilistic analysis is similar to the one given in the proof of Lemma 28, and therefore we omit it. □ □

Finally, we prove the following lemma

Lemma 17. *There exists a $(t', 1)$-attacker $\mathcal{B}'$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{B}) \;\leq\; t^2 \cdot \mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Priv},\Pi}(\mathcal{B}')\ .$$

Proof. Recall that in privacy game the adversary decides the challenge (S, i) during game initialization. Thus, for any adaptive adversary $\mathcal{B}$, there exists a non-adaptive adversary $\mathcal{B}'$, which simply guesses (S, i) (with probability $\frac{1}{t^2}$) and then executes $\mathcal{B}$. The advantage for this adversary is at least $\frac{1}{t^2} \cdot \mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{B})$, and the proof of the lemma is concluded. □ □

From the above lemmas we derive that

$$\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FS\text{-}GAEAD},\Pi}(\mathcal{A}) \leq \varepsilon_{\mathsf{corr}} + t^2 \cdot (\varepsilon_{\mathsf{auth}} + q \cdot \varepsilon_{\mathsf{priv}})\ .$$

□ □

F Forward-Secure Group AEAD Construction

In what follows (cf. Section F.4) we present an FS-GAEAD construction based on key derivation functions (cf. Section F.1) and authenticated encryption with associated data (cf. Section F.3)

F.1 Key Derivation Functions

F.1.1 Syntax

A key derivation function KDF with respect to set of labels $\mathsf{L} \subseteq \{0,1\}^*$, consists of two algorithms:

- $s \leftarrow \mathsf{KInit}(s^*, n)$: a probabilistic algorithm that receives a uniformly random seed $s^* \in \{0,1\}^\lambda$ and parameter n, and outputs the initial state $s \in \{0,1\}^*$.
- $(s', k) \leftarrow \mathsf{Derive}(s, \mathsf{lab})$: a deterministic algorithm that receives a label $\mathsf{lab} \in \mathsf{L}$ and state s and outputs an new state s' and key $k \in \{0,1\}^\lambda \cup \{\perp\}$. If $\mathsf{lab} \notin \mathsf{L}$ or lab has been queried before, $k = \perp$.

F.1.2 Correctness & security

Before defining correctness we state what it means for a sequence of calls to the KDF to be valid.

Valid sequence of calls. For any s, a sequence of calls $Q_1, \ldots, Q_q$ to Derive, with $Q_i = (s_i, \mathsf{lab}_i)$ is called valid w.r.t. s if:

1. *Calls are consecutive*: $s_1 = s$ and for all $i > 1$, $\mathsf{Derive}(Q_{i-1}) = (s_i, k_i)$.
2. *Labels are distinct*: if $i \neq j$, then $\mathsf{lab}_i \neq \mathsf{lab}_j$.

Now we define correctness.

Definition 13 (KDF correctness)**.** *An* KDF *with label set* L *is correct, if for all* $s \in \{0,1\}^\lambda$ *and all valid w.r.t.* s *call sequences* $\mathbb{Q}_1$ *and* $\mathbb{Q}_2$*, it holds that for any* $Q_i \in \mathbb{Q}_1$*,* $Q_j \in \mathbb{Q}_2$*, having the same label it holds that* $(s', k) \leftarrow \mathsf{Derive}(Q_i)$ *and* $(s, k) \leftarrow \mathsf{Derive}(Q_j)$*, i.e., the KDF outputs the same* k*.*

The security definition for KDFs follows.

Definition 14 (Forward secure KDF)**.** *A KDF* $\mathsf{KDF} = (\mathsf{KInit}, \mathsf{Derive})$ *is a* (t, ε)*-forward secure KDF (FS-KDF) with respect to a label set* L *if for any* t*-attacker* $\mathcal{A}$ *and* $n \leq t$*,* $\mathcal{A}$ *can't win the following game with probability greater than* $1/2 + \varepsilon$*:*

1. *Sample* $s^* \leftarrow \{0,1\}^\lambda$*,* $s \leftarrow \mathsf{KInit}(s^*, n)$*.*
2. $\mathcal{A}$ *chooses* $\mathsf{lab}^* \in \mathsf{L}$*.*
3. *Compute* $(s, k) \leftarrow \mathsf{Derive}(s, \mathsf{lab}^*)$*.*
4. $b \leftarrow \{0,1\}$*.*
5. *If* $b = 1$ *then* $k \leftarrow \{0,1\}^\lambda$*.*
6. $\mathcal{A}$ *is provided with* s *and* k*, and outputs a bit* $b' \in \{0,1\}$*.*

$\mathcal{A}$ *wins if* $b' = b$*.*

Below we define an additional required property that is satisfied by our FS-KDF construction.

Definition 15 (Independence w.r.t. the order of evaluations.)**.** *Let* $\mathbf{L} = \{\mathsf{lab}_1, \ldots, \mathsf{lab}_q\}$ *be any sequence of labels and* $\mathbf{L}'$ *be any permutation of* $\mathbf{L}$*. For any initial state* s_1*, set* $s'_1 \leftarrow s_1$ *and for* $i \in [q]$*, compute* $(s_{i+1}, \cdot) \leftarrow \mathsf{Derive}(s_i, \mathsf{lab}_i)$*,* $(s'_{i+1}, \cdot) \leftarrow \mathsf{Derive}(s'_i, \mathsf{lab}'_i)$*. Then, it is required that* $s_q = s'_q$*.*

F.2 FS-KDF construction & security proof

In the current section we present our FS-KDF construction which is based on a length tripling PRG,

$$\mathsf{prg} : \{0,1\}^\lambda \to \left(\{0,1\}^\lambda\right)^3 \; .$$

We first present the main data structures and methods used by our construction.

Labeled trees. Let T be any labeled tree. We consider the following methods.

- $\mathsf{T.L}$: the set of all labels in T.
- $\mathsf{T.root}$: the root node
- Each node v in T has a secret, $v.\mathsf{sec} \in \left\{\{0,1\}^\lambda, \perp_1, \perp_2\right\}$, where
  - $\perp_1$: the secret has already been deleted.
  - $\perp_2$: the secret has not been defined yet.

  All secrets are initialized to $\perp_2$.
- $v.\mathsf{label}$: the unique label of a leaf node v, where $v.\mathsf{label} \in \mathsf{T.L}$.
- $\mathsf{T.leaf}(\mathsf{lab})$: the leaf node v of T such that $v.\mathsf{label} = \mathsf{lab}$.
- $\mathsf{T.ldef}(\mathsf{lab})$: returns a vector consisting of the nodes $(v_1, \ldots, v_z)$, such that (1) $v_1.\mathsf{sec} \in \{0,1\}^\lambda$, (2) for $i = 2, \ldots, z$, $v_i.\mathsf{sec} = \perp_2$, and (3) $v_z.\mathsf{label} = \mathsf{lab}$.

We consider three types of trees, *namely left-balanced binary trees*, *vine tress*, and *application secret trees.*

Left-balanced binary trees. An LBBT with n leafs is referenced by lbbt_n and we have

- $v.\mathsf{lc}$: the left child of v.
- $v.\mathsf{rc}$: the right child of v.

Vine Trees. A Vine Tree vt_i is a labeled tree for the i^{th} symmetric ratchet, $i > 0$.

- $\mathsf{vt}_i.v_j$: internal node for positive integer $j > 0$.
- node v_j has 3 children: key_j, nonce_j, v_{j+1}, where
  - key_j: a leaf node with label (key, i, j).
  - nonce_j: a leaf node with label (nonce, i, j).
  - v_{j+1}: the next internal node.

FS-KDF construction

```
// FS-KDF initialization
KInit ($s^*$, $n$)
    $s \leftarrow$ new $\mathsf{AS}_n$
    $s$.root.sec $\leftarrow s^*$
    return $s$

// The key derivation function
Derive ($s$, (type, $i$, $j$))
    $k \leftarrow s$.leaf(type, $i$, $j$).sec
    if $k = \perp_1$ return $\perp_1$
    if $k \neq \perp_2$
        $s$.leaf(type, $i$, $j$).sec $\leftarrow \perp_1$
        return $k$
    if $s$.vt$_i$.root.sec $= \perp_2$
        $(v_1, \ldots, v_z) \leftarrow s$.lbbt.ldef($i$)
        for $l = 1, \ldots, z$
            ($v_l$.lc.sec, $v_l$.rc.sec, $\cdot$) $\leftarrow$ prg($v_l$.sec)
            $v_l$.sec $\leftarrow \perp_1$
    $(v_1, \ldots, v_z) \leftarrow s$.vt$_i$.ldef($j$)
    for $l = 1, \ldots, z$
        (vt$_i$.key$_l$.sec, vt$_i$.nonce$_l$.sec, vt$_i$.$v_{l+1}$.sec) $\leftarrow$ prg($v_l$.sec)
        $v_l$.sec $\leftarrow \perp_1$
    $k \leftarrow s$.leaf(type, $i$, $j$).sec
    $s$.leaf(type, $i$, $j$).sec $\leftarrow \perp_1$
    return $k$
```

Figure 25: Our FS-KDF construction based on PRGs.

Application secret trees. The application secret tree for n parties, AS_n, is a labeled tree consisting of

- A left balanced binary tree lbbt_n rooted at $\mathsf{AS}_n.\mathsf{root}$.
- Leaf $i \in [n]$ of lbbt_n is the root of vt_i.
- $\mathsf{AS}_n.\mathsf{L} = \{(\mathsf{key}, i, j), (\mathsf{nonce}, i, j) : i \in [n], j > 0\}$.

Our construction is depicted in figure 25.

Before proving security of our construction we argue about it's properties. First of all, or construction works over the set of labels $\mathsf{AS}_n.\mathsf{L}$ and clearly if a label lab has been queried before, the output of the KDF is $\perp_1$, i.e., the requested secret is deleted right after being queried (this is required by the KDF syntax, cf. Section F.1.1). Also, the correctness (cf. Definition 13) and independence w.r.t. to the order of evaluations (cf. Definition 15), properties, are straightforward.

Below we prove security of our construction.

Theorem 18. *Assuming* prg *is an* $\varepsilon_{\mathsf{prg}}$*-secure PRG, the construction of Figure 25 is an* $(t, 2(\log t + t)\varepsilon_{\mathsf{prg}})$*-secure FS-KDF.*

Proof. Our proof is via a sequence of hybrids: H_0 is the original FS-KDF security game (cf. Definition 14) and for $i > 0$, H_i substitutes the output of the first i PRG calls with uniformly random values and uses the real PRG output for the rest. A single call for the label (type, i, j) to Derive requires (at most) $\log n + j$ PRG calls; at most $\log n$ calls for computing the secret of the root of vt_i, plus j for computing the secret of the node type_j. Clearly, we have $\log n + j \leq \log t + t$. Let H_l be the last hybrid, $l \leq \log t + t$, and let W_i be the probability that $\mathcal{A}$ wins in H_i. We prove the following claim.

Claim 19. *For all* $i \geq 0$*,* $W_i - W_{i+1} \leq 2\varepsilon_{\mathsf{prg}}$*.*

Proof. We define an adversary $\mathcal{A}'$ as follows: $\mathcal{A}'$ plays against the PRG security game, which samples a uniformly random bit b and seed s, and if $b = 1$ it returns $\mathsf{prg}(s)$ to $\mathcal{A}'$, otherwise returns a uniformly random string. $\mathcal{A}'$ uses the PRG challenge to answer the i^{th} PRG call in H_{i-1} and outputs 1 if $\mathcal{A}$ wins, otherwise outputs 0. Clearly, if $b = 1$ then $\mathcal{A}'$ simulates H_i, while if $b = 1$ it simulates

H_{i+1}. Let W be the probability that $\mathcal{A}'$ wins in the PRG security game. We compute,

$$\begin{aligned}\frac{1}{2}+\varepsilon_{\mathsf{prg}} \geq W &= \Pr[W \mid b=1]\cdot\Pr[b=1]+\Pr[W \mid b=0]\cdot\Pr[b=0] \\ &= \frac{1}{2}\cdot(W_i+(1-W_{i+1})) \\ &= \frac{1}{2}+\frac{W_i-W_{i+1}}{2}\end{aligned}$$

and the proof of the claim is complete. □ □

In the last hybrid, H_l, the view of $\mathcal{A}$ is independent of the challenge bit, therefore $W_l = 1/2$ and from above claim we derive that $W_0 \leq 2l\varepsilon_{\mathsf{prg}} \leq 2(\log t + t)\varepsilon_{\mathsf{prg}}$. □ □

F.3 Authenticated Encryption with Associated Data

Syntax. An AEAD scheme AEAD consists of three algorithms:

- $k \leftarrow \mathsf{KGen}(1^\lambda)$: receives security parameter and outputs a key k.
- $e \leftarrow \mathsf{AEnc}(k,n,m,a)$: receives key k, nonce n, message m and associated data a, and outputs a ciphertext e.
- $m \leftarrow \mathsf{ADec}(k,n,e,a)$: receives key k, nonce n, ciphertext e and associated data a, and outputs a message m or $\perp$.

AEAD correctness. An AEAD scheme $\mathsf{AEAD} = (\mathsf{KGen},\mathsf{AEnc},\mathsf{ADec})$ is correct if for every nonce n, message m and any associated data a,

$$\Pr[\mathsf{ADec}(k,n,\mathsf{AEnc}(k,n,m,a),a)=m]=1,$$

where $k \leftarrow \mathsf{KGen}(1^\lambda)$.

Our setting requires one-time security, i.e., only one ciphertext per secret key is generated. The security games defined below capture this notion.

AEAD Security. An AEAD scheme AEAD is (t,ε)-secure if:

- *CPA security*: For every t-attacker $\mathcal{A}$,

$$\Pr\left[\, b=b' \;\middle|\; \begin{array}{c} k \leftarrow \mathsf{KGen}(1^\lambda); (n,a,m_0,m_1) \leftarrow \mathcal{A}(1^\lambda); \\ b \leftarrow \{0,1\};\; e \leftarrow \mathsf{AEnc}(k,n,m_b,a);\; b' \leftarrow \mathcal{A}(e) \end{array}\right] \leq \varepsilon.$$

  The advantage of $\mathcal{A}$ in winning the above game is denoted by $\mathrm{Adv}^{\mathsf{AEAD}}_{\mathsf{AE\text{-}Priv}}$.

- *Authenticity*: For every t-attacker $\mathcal{A}$,

$$\Pr\left[(\tilde{n},\tilde{e},\tilde{a}) \neq (n,e,a) \wedge \mathsf{ADec}(k,\tilde{n},\tilde{e},\tilde{a}) \neq \perp \;\middle|\; \begin{array}{c} k \leftarrow \mathsf{KGen}(1^\lambda); (n,m,a) \leftarrow \mathcal{A}(1^\lambda); \\ e \leftarrow \mathsf{AEnc}(k,n,m,a); (\tilde{n},\tilde{e},\tilde{a}) \leftarrow \mathcal{A}(e) \end{array}\right] \leq \varepsilon.$$

  The advantage of $\mathcal{A}$ in winning the above game is denoted by $\mathrm{Adv}^{\mathsf{AEAD}}_{\mathsf{AE\text{-}Auth}}$.

FS-GAEAD Security Game: Safety Helpers and Safety Predicate

```
// determines whether a message is not compromised
*FS-sec (S, i)
    return $\forall(\mathsf{R}, \mathsf{Tr}, \mathsf{Rv}) : (\mathsf{R}, \mathsf{Tr}, \mathsf{Rv}) \notin \mathsf{AM\text{-}Lk} \lor ((\mathsf{S}, i) \in \mathsf{Rv} \land (\mathsf{S}, i) \notin \mathsf{Tr})$
```

Figure 26: The safety predicate for our FS-GAEAD construction.

FS-GAEAD Construction

```
// Initialization
Init $(k, n, \mathsf{ID})$
    $s \leftarrow \mathsf{KInit}(k, n)$
    $\mathsf{pos} \leftarrow 1$
    return $(s, \mathsf{pos}, \mathsf{ID})$

// Send message
Send $(v, a, m)$
    $(s, \mathsf{pos}, \mathsf{ID}) \leftarrow v$
    $(s, n) \leftarrow \mathsf{Derive}(s, (\mathsf{nonce}, \mathsf{ID}, \mathsf{pos}))$
    $(s, k) \leftarrow \mathsf{Derive}(s, (\mathsf{key}, \mathsf{ID}, \mathsf{pos}))$
    $\mathsf{pos}{+}{+}$
    $e' \leftarrow \mathsf{AEnc}(k, n, m, (\mathsf{ID}, \mathsf{pos}, a))$
    $e \leftarrow (\mathsf{ID}, \mathsf{pos}, e')$
    return $((s, \mathsf{pos}, \mathsf{ID}), e)$

// Receive message
Rcv $(v, a, e)$
    $(s, \mathsf{pos}, \mathsf{ID}) \leftarrow v$
    $(\mathsf{S}, i, e') \leftarrow e$
    $(s, n) \leftarrow \mathsf{Derive}(s, (\mathsf{nonce}, \mathsf{S}, i))$
    $(s, k) \leftarrow \mathsf{Derive}(s, (\mathsf{key}, \mathsf{S}, i))$
    req $n, k \neq \perp$
    $m \leftarrow \mathsf{ADec}(k, n, e', (\mathsf{S}, i, a))$
    req $m \neq \perp$
    return $((s, \mathsf{pos}, \mathsf{ID}), \mathsf{S}, i, m)$
```

Figure 27: The FS-GAEAD construction.

F.4 FS-GAEAD construction & security

Our FS-GAEAD construction is depicted in Figure 27.

For our construction, the safety predicate ***FS-sec** of the FS-GAEAD security game of Figure 27 is instantiated by ***FS-sec** of Figure 26. The safety predicate ***FS-sec** ensures that no compromised message has been challenged. A message, identified by (S, i) is considered compromised if there exist $(\mathsf{R}, \mathsf{Rv}, \mathsf{Tr}) \in \mathsf{AM\text{-}Lk}$, such that (S, i) is compromised

- *(via state leakage)* $(\mathsf{S}, i) \notin \mathsf{Rv}$, i.e., at the time of compromise R didn't receive (S, i), or
- *(via trash)* $(\mathsf{S}, i) \in \mathsf{Tr}$, i.e., at the time of compromise (S, i) was in the trash of R, which implies that R didn't delete the corresponding key material.

In what follows we prove security of our construction.

Theorem 20. *Assuming $(t', \varepsilon_{\mathsf{ae}})$-secure AEAD and $(t', \varepsilon_{\mathsf{kdf}})$-forward secure KDF, the scheme of Figure 27, denoted as F, is*

- $(t, 0)$*-correct,*
- $(t, \varepsilon_{\mathsf{kdf}} + \varepsilon_{\mathsf{ae}})$*-authentic,*
- $(t, \varepsilon_{\mathsf{kdf}} + \varepsilon_{\mathsf{ae}})$*-private,*

w.r.t. the predicate ***FS-sec** *of Figure 26, with $t \approx t'$.*

Proof.

Correctness. First observe that the sequence of calls made to Derive is valid with respect to the output s of KInit (cf. Section F.1):

1. *Calls are consecutive*: each call to Derive uses the previously generated KDF state s, where the first state is output by KInit.

2. *Labels are distinct*: each call to the Send oracle increases the counter pos by one thus no consecutive calls to Send use the same label. A similar argument holds for Rcv since the value of the index i used by the calls to Derive matches the counter value pos used by the sender.

By the above and the correctness property of the KDF (cf. Section F.1) we derive that all calls to Derive that share the same label produce the same keys and nonces, which in our construction implies that the keys, k, and nonces n, computed by the Send oracle are equal to those computed by the Rcv oracle. I.e., senders and receivers are fully synchronized.

Let $\mathcal{A}$ be a t-attacker that triggers the **win** condition inside the **dlv-AM** oracle of the FS-GAEAD game of Figure 23. This implies that for some S, R, on input $(\mathsf{S}, i, \mathsf{R})$, the **dlv-AM** oracle computes $(v[\mathsf{R}], \mathsf{S}', i', m') \leftarrow \mathsf{Rcv}(v[\mathsf{R}], a, e)$ and $(\mathsf{S}', i', m') \neq (\mathsf{S}, i, m)$, where m is the i^{th} message sent by S. However, S', i' are part of the input ciphertext e to Rcv and defined by the sender (recall that there are no injections, i.e., the game delivers only honestly generated ciphertexts stored in AM). Therefore $(\mathsf{S}', i', m') \neq (\mathsf{S}, i, m)$ implies that $m \neq m'$. Thus, for the i^{th} message of sent by S, S computes $e \leftarrow (\mathsf{S}, i, \mathsf{AEnc}(k, n, m, (\mathsf{S}, i, a)))$, while the recipient computes $m' \leftarrow \mathsf{ADec}(k, n, e, (\mathsf{S}, i, a))$, where $m' \neq m$, violating the correctness property of the encryption scheme (in our reasoning we assume the same k and n, for both AEnc and ADec since we have already argued above that the outputs of Derive are synchronized). We have reached a contradiction, thus no adversary can trigger the **win** condition inside the **dlv-AM** oracle and the scheme is correct.

Authenticity. The **win** condition inside the **inj-AM** oracle is triggered when the following condition is satisfied

$$\textbf{*FS-sec}(\mathsf{S}, i) \vee \mathsf{AM}[\mathsf{S}, i[\mathsf{S}], \mathsf{R}] = \mathtt{received}.$$

Let $\mathcal{E}_1$ (resp. $\mathcal{E}_2$) be the event in which the left part (resp. right part) of the above disjunction is satisfied (for the first time) via a call to **inj-AM** for which $m' \neq \bot$.

First we bound the probability that $\mathcal{E}_2$ happens. Assuming it does, then the adversary triggers the **win** condition inside **inj-AM** by injecting a, e, to R as the i^{th} message of S which *has already been received* (due to $\mathcal{E}_2$). Therefore, Derive will be called twice on the same label $(\mathsf{key}, \mathsf{S}, i)$ which by the KDF definition will output $k = \bot$ (in this setting execution is aborted and all changes made by the call to **inj-AM** are reverted). Thus conditioned on $\mathcal{E}_2$ the **win** condition is triggered with zero probability.

For $\mathcal{E}_1$, ***FS-sec**(S, i) implies that

$$\forall (\mathsf{R}, \mathsf{Tr}, \mathsf{Rv}) : (\mathsf{R}, \mathsf{Tr}, \mathsf{Rv}) \notin \mathsf{AM\text{-}Lk} \vee ((\mathsf{S}, i) \in \mathsf{Rv} \wedge (\mathsf{S}, i) \notin \mathsf{Tr}),$$

i.e., for all receivers R, at the injection time, either the state of the receiver has not been been compromised, or if it does, at the time of compromise the i^{th} message has been already delivered to R and the corresponding key is not included in R's trash. For the latter case the argumentation is identical to the above: delivering the i^{th} message of S to R twice results in calling Derive twice with the same label, in which case Derive outputs $\bot$. It only remains to bound the probability of a successful injection against non-compromised parties, for messages that have not been delivered yet (other parties may have been compromised still at the time of compromise the i^{th} message of S has been already delivered). This relies on KDF security and the authenticity property of the encryption scheme.

Let S^*, i^*, be the ID of the sender and the message index, respectively, for which $\mathcal{A}$ will try to inject. Let H_0 be identical to the authenticity game with one difference: the $i^{*\text{th}}$ message sent by S^* is not encrypted using the key output by Derive but using a uniformly random key (the decryption operation uses that key also). In the following, denote the advantage of $\mathcal{A}$ in H_0 w.r.t. $\Pi =$ ***FS-sec**, by $\mathrm{Adv}^{\mathsf{F}}_{H_0,\Pi}(\mathcal{A})$. We define an adversary $\mathcal{B}_0$ that plays against the FS-KDF security game and simulates for $\mathcal{A}$ the authenticity game w.r.t. F.

Lemma 21. *There exists a t'-attacker $\mathcal{B}_0$, with $t' \approx t$ such that*

$$\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Auth},\Pi}(\mathcal{A}) \ \leq \ \mathrm{Adv}^{\mathsf{F}}_{H_0,\Pi}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{KDF}}_{\mathsf{FS\text{-}KDF}}(\mathcal{B}_0) \ .$$

Proof. We define how $\mathcal{B}_0$ simulates the replies of the queries made by $\mathcal{A}$ to the FS-GAEAD authenticity game, while playing against the FS-KDF security game. $\mathcal{B}_0$ is defined below.

- **(Initialization)**:
  1. Run $\mathcal{A}$ and receive (S^*, i^*).
  2. Set the challenge $\mathsf{lab}^* \leftarrow (\mathsf{key}, \mathsf{S}^*, i^*)$, send lab^* to the FS-KDF challenger, and let s^*, k^*, be the reply.
  3. Compute $(s^*, n^*) \leftarrow \mathsf{Derive}(s^*, (\mathsf{nonce}, \mathsf{S}^*, i^*))$.
- **(Query simulation)**:
  - **init**(G): execute all steps of **init**(G), besides Init. Instead, for $\mathsf{ID} \in G$, set $v[\mathsf{ID}] \leftarrow (s^*, 1, \mathsf{ID})$.
  - **send**(S, a, m):
    * If $(\mathsf{S}, \mathsf{pos}) \neq (\mathsf{S}^*, i^*)$, execute the query normally w.r.t. to the defined FS-GAEAD states.
    * Otherwise, in Send, use k^*, n^*, as the key and nonce, respectively, for AEnc (i.e., omit the computation of Derive).
  - **corr**(ID): Compute $\mathsf{AM\text{-}Rcvd}$ and $\mathsf{AM\text{-}Lk}$ as in **corr** oracle.
    * If $(\mathsf{S}^*, i^*) \notin \mathsf{AM\text{-}Rcvd}$ or $(\mathsf{S}^*, i^*) \in \mathsf{AM\text{-}Tr}[\mathsf{ID}]$, abort and output 1.
    * Otherwise, send $v[\mathsf{ID}]$ to $\mathcal{A}$.
  - **dlv-AM**$(\mathsf{S}, i, \mathsf{R})$:
    * If $(\mathsf{S}, i) \neq (\mathsf{S}^*, i^*)$, execute the query normally w.r.t. to the defined FS-GAEAD states.
    * Otherwise, in Rcv, use k^*, n^*, as the key and nonce, respectively, for ADec (i.e., omit the computation of Derive).
  - **no-del**$[\mathsf{ID}]$: set $\mathsf{Del}[\mathsf{ID}] \leftarrow \mathsf{false}$.
  - **inj-AM**$(a, \mathsf{S}, i, e, \mathsf{R})$:
    * If $(\mathsf{S}, i) \neq (\mathsf{S}^*, i^*)$, execute the query normally w.r.t. to the defined FS-GAEAD states.
    * Otherwise, in Rcv, use k^*, n^*, as the key and nonce, respectively, for ADec (i.e., omit the computation of Derive).

We show that the above simulation is correct. Observe that the **init** operation is indistinguishable to the one computed in the authenticity game, however, the key, k^*, and nonce, n^*, for (S^*, i^*), have been already computed via a challenge query to the FS-KDF challenger, and the KDF states of all ids have been initialized with s^* (the FS-KDF state after computing over (S^*, i^*)). The simulation is correct since our safety predicate guarantees that no party is compromised before receiving the $i^{*\text{th}}$ message by S^*, and for all parties compromised after receiving that message, **no-del** is disabled before receiving that message. For **send**, $\mathcal{B}_0$ executes normally the code of the oracle based on the aforementioned states, unless the oracle call is for the $i^{*\text{th}}$ message of S^*, in which case $\mathcal{B}_0$ uses key k^*, and nonce n^*, and again the simulation is correct. For **corr**, $\mathcal{B}_0$, simply executes the code of the oracle, and aborts in case $\mathcal{A}$ violates the safety predicate. Also, the state leaked to the adversary is consistent with the one computed in the authenticity game, and this is due to the independence w.r.t. the order of evaluations (IOE) property of the KDF function (cf. Definition 15), which ensures that the FS-KDF state is independent of the order of operations. Observe that, if a corruption happens, the corrupted id should already have received (S^*, i^*), which means that the corresponding labels should have been queried in both games. However, the order of queries might be different as (S^*, i^*) is always $\mathcal{B}_0$'s first query to FS-KDF, still, IOE ensures that the leaked state is not affected by that order. **dlv-AM** execution is identical to the original, unless the query is for (S^*, i^*) in which case $\mathcal{B}_0$ uses k^*, n^*, which guarantee correctness of the simulation. **inj-AM** is similar, while **no-del** is trivially correct.

Now observe that, if $b = 0$, $\mathcal{B}_0$ correctly simulates the FS-GAEAD authenticity game (since k^* is output by Derive), while if $b = 1$ it simulates H_0 (since k^* is uniformly random), thus the distinguishing advantage cannot be more than $\varepsilon_{\mathsf{kdf}}$ and the proof is concluded. □ □

Now we prove that the advantage of $\mathcal{A}$ in H_0 is upper bounded by the advantage of breaking the authenticity property of the underlying AEAD scheme.

Lemma 22. *There exists a t'-attacker $\mathcal{B}_1$ with $t' \approx t$ such that*

$$\mathrm{Adv}^{\mathsf{F}}_{H_0,\Pi}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\mathsf{AEAD}}_{\mathsf{AE\text{-}Auth}}(\mathcal{B}_1)\ .$$

Proof. We define an attacker $\mathcal{B}_1$ that simulates the replies of the queries made by $\mathcal{A}$ in H_0 while playing against the AEAD authenticity game. $\mathcal{B}_1$ executes $\mathcal{A}$, receives (S^*, i^*) and replies to $\mathcal{A}$'s queries as defined below.

- **init**(G, S^*, i^*): execute all steps defined in the authenticity game.
- **send**(S, a, m):
  - If $(\mathsf{S}, \mathsf{pos}) = (\mathsf{S}^*, i^*)$: let n be the nonce computed by executing **send**. Send the challenge $(n, m, (\mathsf{S}, \mathsf{pos}, a))$ to the AEAD authenticity game challenger, receive the ciphertext e' and set $e \leftarrow (\mathsf{S}, \mathsf{pos}, e')$, $m^* \leftarrow m$. Use e in the rest of the **send** execution and return e to $\mathcal{A}$.
  - Otherwise: execute **send**(S, a, m) normally.
- **corr**(ID): return $v[\mathsf{ID}]$ to $\mathcal{A}$.
- **dlv-AM**$(\mathsf{S}, i, \mathsf{R})$:
  - If $(\mathsf{S}, i) = (\mathsf{S}^*, i^*)$: execute **dlv-AM** but in the Rcv execution omit the decryption operation and set $m \leftarrow m^*$.
  - Otherwise: execute **dlv-AM**$(\mathsf{S}, i, \mathsf{R})$ normally.
- **inj-AM**$(a, \mathsf{S}, i, e, \mathsf{R})$:
  - If $(\mathsf{S}, i) = (\mathsf{S}^*, i^*)$: send (n, e, a) to the AEAD authenticity challenger and halt.
  - Otherwise: execute **inj-AM**$(a, \mathsf{S}, i, e, \mathsf{R})$ normally.

Observe that $\mathcal{B}_1$ correctly simulates H_0, i.e., for the $i^{*\text{th}}$ message by S^* it uses a uniformly random key k, which is the key used by the challenger of the AEAD authenticity game, and sends that message as the challenge plaintext for the authenticity game. All other messages are processed normally. Queries **corr**(ID) are processed by returning the state of ID. Observe here, that, since there is no corruption before receiving the $i^{*\text{th}}$ message by S^*, or **no-del** before receiving the message and corruption after that moment, the adversary will never request access to F key k. Furthermore, injections are processed normally, however when injecting the $i^{*\text{th}}$ message of S^*, $\mathcal{B}_1$ sends the nonce n, the mauled ciphertext e and associated data a to the challenger of the AEAD authenticity game. Assuming (n, e, a) is a valid injection, we clearly have that $\mathcal{B}_0$ breaks the authenticity property of the underlying AEAD scheme, reaching a contradiction. Therefore, the winning probability of $\mathcal{B}_1$ in H_0 is bounded by the probability of breaking the authenticity property of the underlying AEAD scheme. □ □

Privacy. Let S^*, i^*, be the ID of the sender and the message index w.r.t. which the challenge query happens. Our strategy is similar to the one used for proving authenticity, i.e., for the challenge message we substitute the underlying encryption key with a uniformly random key.

Let H_1 be identical to the privacy game with one difference: the $i^{*\text{th}}$ message sent by S^* via the call to the **chall** oracle is not encrypted using the key output by Derive but using a uniformly random key (the decryption operation uses that key also). In the following, denote the advantage of $\mathcal{A}$ in H_1 w.r.t. $\Pi =$ ***FS-sec**, by $\mathrm{Adv}^{\mathsf{F}}_{H_1,\Pi}(\mathcal{A})$. We define an adversary $\mathcal{B}_2$ that plays against the FS-KDF security game and simulates for $\mathcal{A}$ the privacy game w.r.t. F.

Lemma 23. *There exists a t'-attacker $\mathcal{B}_2$ with $t' \approx t$ such that*

$$\mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Priv},\Pi}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\mathsf{F}}_{H_1,\Pi}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{KDF}}_{\mathsf{FS\text{-}KDF}}(\mathcal{B}_2) \;.$$

Proof. We define how $\mathcal{B}_2$ simulates the replies of the queries made by $\mathcal{A}$ to the FS-GAEAD privacy game, while playing against the FS-KDF security game. $\mathcal{B}_2$ replies to $\mathcal{A}$'s queries as defined below.

- **(Initialization)**:
  1. Run $\mathcal{A}$ and receive (S^*, i^*).
  2. Set the challenge $\mathsf{lab}^* \leftarrow (\mathsf{key}, \mathsf{S}^*, i^*)$, send lab^* to the FS-KDF challenger, and let s^*, k^*, be the reply.
  3. Compute $(s^*, n^*) \leftarrow \mathsf{Derive}(s^*, (\mathsf{nonce}, \mathsf{S}^*, i^*))$.
- **(Query simulation)**:
  - **init**(G): execute all steps of **init**(G), besides Init. Instead, for $\mathsf{ID} \in G$, set $v[\mathsf{ID}] \leftarrow (s^*, 1, \mathsf{ID})$.
  - **send**(S, a, m):
    - ∗ If $(\mathsf{S}, i[\mathsf{S}]+1) \neq (\mathsf{S}^*, i^*)$, execute the query normally w.r.t. to the defined FS-GAEAD states.
    - ∗ Otherwise, abort and output 1.
  - **corr**(ID): Compute $\mathsf{AM\text{-}Rcvd}$ and $\mathsf{AM\text{-}Lk}$ as in **corr** oracle.
    - ∗ If $(\mathsf{S}^*, i^*) \notin \mathsf{AM\text{-}Rcvd}$ or $(\mathsf{S}^*, i^*) \in \mathsf{AM\text{-}Tr}[\mathsf{ID}]$, abort and output 1.
    - ∗ Otherwise, send $v[\mathsf{ID}]$ to $\mathcal{A}$.

- **dlv-AM**$(\mathsf{S}, i, \mathsf{R})$:
  - If $(\mathsf{S}, i) \neq (\mathsf{S}^*, i^*)$, execute the query normally w.r.t. to the defined FS-GAEAD states.
  - Otherwise, in Rcv, use k^*, n^*, as the key and nonce, respectively, for ADec (i.e., omit the computation of Derive).
- **no-del**$[\mathsf{ID}]$: set $\mathsf{Del}[\mathsf{ID}] \leftarrow \mathsf{false}$.
- **chall**$(\mathsf{S}, a, m_0, m_1)$:
  - If $(\mathsf{S}, i[\mathsf{S}] + 1) \neq (\mathsf{S}^*, i^*)$, abort and output 1.
  - Otherwise, execute the query normally but in Rcv, use k^*, n^*, as the key and nonce, respectively, for ADec (i.e., omit the computation of Derive).
- **reduced-inject**$(a, \mathsf{S}, i, e, \mathsf{R})$: similar to deliver.

The arguments behind the correctness of the above simulation are similar to those in the proof of Lemma 21. Observe that, if $b = 0$, $\mathcal{B}_2$ correctly simulates the FS-GAEAD privacy game (since k^* is output by Derive), while if $b = 1$ it simulates H_1 (since k^* is uniformly random), thus the distinguishing advantage cannot be more than $\varepsilon_{\mathsf{kdf}}$ and the proof is concluded. □ □

Now we prove that the advantage of $\mathcal{A}$ in H_1 is upper bounded by the advantage of breaking the privacy property of the underlying AEAD scheme.

Lemma 24. *There exists a t'-attacker $\mathcal{B}_3$ with $t' \approx t$ such that*

$$\mathrm{Adv}^{\mathsf{F}}_{H_1,\Pi}(\mathcal{A}) \ \leq \ \mathrm{Adv}^{\mathsf{AEAD}}_{\mathsf{AE\text{-}Priv}}(\mathcal{B}_3) \ .$$

Proof. We define an attacker $\mathcal{B}_3$ that simulates the replies of the queries made by $\mathcal{A}$ to H_1 while playing against the AEAD privacy game. $\mathcal{B}_3$ executes $\mathcal{A}$, receives (S^*, i^*) and replies to $\mathcal{A}$'s queries as defined below.

- **init**(G, S^*, i^*): execute all steps defined in the privacy game.
- **send**(S, a, m):
  - If $(\mathsf{S}, i[\mathsf{S}] + 1) \neq (\mathsf{S}^*, i^*)$, execute the query normally w.r.t. to the defined FS-GAEAD states.
  - Otherwise, abort and output 1.
- **corr**(ID): Compute $\mathsf{AM\text{-}Rcvd}$ and $\mathsf{AM\text{-}Lk}$ as in **corr** oracle.
  - If $(\mathsf{S}^*, i^*) \notin \mathsf{AM\text{-}Rcvd}$ or $(\mathsf{S}^*, i^*) \in \mathsf{AM\text{-}Tr}[\mathsf{ID}]$, abort and output 1.
  - Otherwise, send $v[\mathsf{ID}]$ to $\mathcal{A}$.
- **dlv-AM**$(\mathsf{S}, i, \mathsf{R})$: execute the query normally while if $(\mathsf{S}, i) = (\mathsf{S}^*, i^*)$, omit ADec.
- **no-del**$[\mathsf{ID}]$: set $\mathsf{Del}[\mathsf{ID}] \leftarrow \mathsf{false}$.
- **chall**$(\mathsf{S}, a, m_0, m_1)$:
  - If $(\mathsf{S}, i[\mathsf{S}] + 1) \neq (\mathsf{S}^*, i^*)$, abort and output 1.

Security Game for PRF-PRNG

```
init
    $b \leftarrow_{\mathsf{R}} \{0,1\}$
    $\mathsf{vid}_{\mathsf{root}} \leftarrow \mathsf{HG.init}$
    $\sigma[\mathsf{vid}_{\mathsf{root}}] \leftarrow 0$
    $\mathsf{V\text{-}Lk} \leftarrow \emptyset$
    $R[\cdot] \leftarrow \emptyset$
    $\mathsf{Reveal}[\cdot] \leftarrow \emptyset$
    $\mathsf{Chall}[\cdot] \leftarrow \emptyset$
    $\mathsf{BI}[\cdot] \leftarrow \emptyset$

corr (vid)
    $\mathsf{V\text{-}Lk} \mathrel{+}\leftarrow \mathsf{vid}$
    return $\sigma[\mathsf{vid}]$

process (vid, $I, C$)
    req $\nexists$ child of vid for $(I, C)$
    $\mathsf{vid}' \leftarrow \mathsf{HG.create}(\mathsf{vid}, I)$
    $\mathsf{BI}[\mathsf{vid}'] \leftarrow (I \neq \bot)$
    $(\sigma[\mathsf{vid}'], R[\mathsf{vid}']) \leftarrow \mathsf{PP}(\sigma[\mathsf{vid}], I, C)$
    return vid'

reveal (vid)
    req $R[\mathsf{vid}] \neq \varepsilon$
    req $\neg(\mathsf{Reveal}[\mathsf{vid}] \vee \mathsf{Chall}[\mathsf{vid}])$
    $\mathsf{Reveal}[\mathsf{vid}] \leftarrow \mathsf{true}$
    return $R[\mathsf{vid}]$

chall (vid)
    req $R[\mathsf{vid}] \neq \varepsilon$
    req $\neg(\mathsf{Reveal}[\mathsf{vid}] \vee \mathsf{Chall}[\mathsf{vid}])$
    $R_0 \leftarrow R[\mathsf{vid}]$
    $R_1 \leftarrow \mathcal{R}$
    $\mathsf{Chall}[\mathsf{vid}] \leftarrow \mathsf{true}$
    return $R_b$

safe
    return $\forall \mathsf{vid} : \mathsf{Chall}[\mathsf{vid}] \Longrightarrow$
      *PP-secure(vid)
```

Figure 28: PRF-PRNG security game.

- Otherwise, execute the code of the oracle normally, but instead of computing AEnc, send the challenge query (n, a, m_0, m_1) to the CPA challenger and forward the reply to $\mathcal{A}$.

- **reduced-inject**$(a, \mathsf{S}, i, e, \mathsf{R})$: similar to deliver.

Observe that $\mathcal{B}_3$ correctly simulates H_1, i.e., for the $i^{*\text{th}}$ message sent by S^* it uses a uniformly random key k, which is the key used by the AEAD privacy game, and sends the challenge messages as challenge plaintexts for the privacy game. Moreover, since there is no corruption before receiving the $i^{*\text{th}}$ message by S^*, and for any party for which **no-del** has been enabled before processing the $i^{*\text{th}}$ message by S^*, no corruption can happen even after processing that message, the adversary will never request access to F's key k. **send** is executed normally, unless $\mathcal{A}$ requests transmission of the challenge sender/index pair, **corr** is executed normally, unless $\mathcal{A}$ violates the safety predicate, **dlv-AM** is executed normally, unless $\mathcal{A}$ requests the deliver of the $i^{*\text{th}}$ message by S^*, in which case $\mathcal{B}_3$ executes all oracle operations besides decryption (since it does not have access to the secret of AEAD), and **reduced-inject** is similar to **dlv-AM**. Thus, the simulation is correct. If in the AEAD privacy game $b = 0$ then $\mathcal{B}_3$ simulates H_1 with $b = 0$. On the other hand, if in the AEAD privacy game $b = 1$, $\mathcal{B}_3$ simulates H_1 with $b = 1$. Therefore, the advantage of $\mathcal{A}$ in guessing b in H_1 is bounded by the advantage of guessing b in the AEAD privacy game, which is at most $\varepsilon_{\mathsf{ae}}$. The proof is concluded. □ □

□ □

Putting things together. By the above proof and Theorem 13, we derive that our FS-GAEAD construction is (t, q, ε')-FS-GAEAD secure, with $\varepsilon' = (q+1)t^2(\varepsilon_{\mathsf{kdf}} + \varepsilon_{\mathsf{ae}})$ and $t \approx t'$.

G PRF-PRNGs

G.1 Security

A PRF-PRNG must satisfy PCFS (cf. Section 3.3) and be resilient to splitting-attacks. Therefore, the security game for PRF-PRNGs (cf. Figure 28) follows the same history-graph approach as the definitions of SGM and CKGA. However, since the game only consists of the state of the PRF-PRNG and there are no parties, it suffices to keep track of a much smaller amount information:

- Nodes of the history graph only consist of the vid.

- For every node vid,
  - the value $\sigma[\mathsf{vid}]$ stores the corresponding state of the PRF-PRNG,
  - the value $R[\mathsf{vid}]$ stores the corresponding output of the PRF-PRNG, and
  - the value $\mathsf{BI}[\mathsf{vid}]$ is a flag indicating whether the input I absorbed to reach the state $\sigma[\mathsf{vid}]$ is known to the attacker.
- The set $\mathsf{V\text{-}Lk}$ records the vids for which the PRF-PRNG state is leaked to the attacker.

The root node $\mathsf{vid}_{\mathsf{root}}$ of the history graph corresponds to the initial state of the PRF-PRNG, which is assumed to be the all-zero string. The attacker $\mathcal{A}$ has the following capabilities:

- He may create a new child state of any node vid by calling oracle **process** and specifying an input/context pair (I, C); of course, only one such call per triple (vid, I, C) is allowed. If the call is made with $I \neq \perp$, the game samples I it randomly. The value $\mathsf{BI}[\mathsf{vid}]$ is set accordingly.
- He may reveal or challenge outputs $R[\mathsf{vid}]$ corresponding to arbitrary nodes $\mathsf{vid} \neq \mathsf{vid}_{\mathsf{root}}$ by calling the corresponding oracles **reveal** and **chall**, respectively. The flags $\mathsf{Reveal}[\mathsf{vid}]$ resp. store $\mathsf{Chall}[\mathsf{vid}]$ whether a reveal resp. a challenge has been requested for vid.
- Finally, $\mathcal{A}$ can also leak the state $\sigma[\mathsf{vid}]$ for any epoch $\mathsf{vid} \neq \mathsf{vid}_{\mathsf{root}}$ using oracle **corr**.

As per usual, at the end of the game, the oracle **safe** ensures that $\mathcal{A}$ does not win the game trivially; **safe** uses a generic safety predicate ***PP-secure**.

G.1.1 Advantage.

Let $\Pi =$ ***PP-secure** be the generic safety predicate used in the PRF-PRNG definition. The attacker $\mathcal{A}$ is parameterized by it's running time t, referred to as t-attacker. The advantage of $\mathcal{A}$ against a PP scheme $\mathsf{PRF\text{-}PRNG}$ w.r.t. to predicate Π is denoted by $\mathrm{Adv}^{\mathsf{PP}}_{\mathsf{PRF\text{-}PRNG},\Pi}(\mathcal{A})$.

Definition 16. *A PRF-PRNG scheme* PP *is* (t, ε)-secure w.r.t. predicate Π*, if for all t-attackers,*

$$\mathrm{Adv}^{\mathsf{PP}}_{\mathsf{PRF\text{-}PRNG},\Pi}(\mathcal{A}) \;\leq\; \varepsilon \;.$$

G.2 Construction

A straight-forward construction of PRF-PRNGs is to model the algorithm PP as a random oracle. Such a construction achieves security w.r.t. safety predicate ***PP-secure**, depicted in Figure 29. It captures that in order for a value $R[\mathsf{vid}]$ to be considered secure, vid

- must have an ancestor state vid' (possibly itself) that was reached via a random input I not known to $\mathcal{A}$, and
- there must have been no corruptions on the path from vid' to vid's ancestor.

Theorem 3. *Let* ***PP-secure** *be the safety predicate above. Then,* PP *as a random oracle with outputs in* $\{0,1\}^{\lambda}$*, is an* (t, ε)*-secure PRF-PRNG w.r.t.* ***PP-secure***, where* $\varepsilon = \mathrm{negl}(\lambda)$ *and* $t = \mathrm{poly}(\lambda)$*.*

sketch. Let $\mathcal{A}$ be an attacker against the PRF-PRNG game that challenges the node vid. By the safety predicate of Figure 29, we require that $\exists\ \mathsf{vid}' \succeq \mathsf{vid}$ s.t. $\neg\mathsf{BI}[\mathsf{vid}']$ and $\nexists\ \mathsf{vid}'' \in [\mathsf{vid}', \mathsf{vid})$: $(\mathsf{vid}'', \cdot) \in \mathsf{V\text{-}Lk}$, i.e., there should be an ancestor node vid' of vid, such that the input I to PP is

Safety Predicate for PRF-PRNG Construction

```
// Is PRF-PRNG state secret?
*PP-secure (vid)
    if ∃ vid′ ⪰ vid s.t.
        1. ¬BI[vid′]
        2. ∄ vid″ ∈ [vid′, vid) : (vid″, ·) ∈ V-Lk
        return true
    return false
```

Figure 29: The safety predicate determines whether a PRF-PRNG output R belonging to a particular vid is considered secure.

uniformly random, and there is no state compromise in [vid′, vid). Then, assuming $\mathcal{A}$ is not querying PP with I, σ[vid′], R[vid′], are uniformly random, thus the next evaluation on vid′, independently of I and C, will also output uniformly random values, and the same will hold for all nodes in [vid′, vid). Note that, if the adversary creates a fork for some vid″ ∈ [vid′, vid), for instance by querying the process oracle with $(I, C) \neq (I', C')$, then since σ[vid″] is uniformly random (by the above arguments), the updated PRF-PRNG states are uniformly random and independent, thus any PRF-PRNG state compromise on the fork branch will not affect security on [vid′, vid). The above hold if the adversary does not query the random oracle with I, σ[vid′], and any σ along the path from vid′ to vid, considered as a bad event $\mathcal{E}$. Then, one could split the security proof w.r.t. adversaries that don't trigger, or trigger, $\mathcal{E}$, which is along the lines of the proof of Theorem 7. □ □

H SGM Security

H.1 Simplified Properties

Proving SGM security is facilitated by considering three simplified properties, namely *correctness*, *authenticity*, and *privacy*. These properties together imply full SGM security, as stated at the end of this section. All three of them are obtained by modifying the original SGM game.

In the following, a (t, q)-*attacker* is an attacker $\mathcal{A}$ that runs in time at most t and makes at most q challenge queries.

Correctness. For the correctness game SGM-Corr, there is no challenge oracle **chall**, no inject oracles for control messages **inj-CM**, welcome messages **inj-WM** and proposals **inj-PM**, while we introduce a *reduced* injection oracle **red-inj-AM**, which is presented in Figure 30.

The advantage of $\mathcal{A}$ in the correctness game with respect to a scheme Γ, is denoted by $\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Corr},\Pi}(\mathcal{A})$.

Definition 17. *A group messaging scheme* Γ *is* (t, ε)-correct *w.r.t. predicates* Π*, if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Corr},\Pi}(\mathcal{A}) \ \leq \ \varepsilon \ .$$

Authenticity. In the authenticity variant of the SGM game, there are is no challenge oracle **chall**, and no **win** instruction inside the delivery oracles **dlv-PM**, **dlv-CM**, **dlv-WM**, **dlv-AM**. Moreover, the adversary needs to initialize the game with the following arguments

- comp-sig ∈ {true, false}: commits to whether at the time of injection the target signature keys will compromised.

SGM Correctness Game: The Inject Oracle

```
**red-inj-AM** $(a', e', \mathsf{R})$
    **req *compat-inj-AM**$(a', e', \mathsf{R})$
    $(E', \mathsf{S}', i') \leftarrow$ ***get-epoch-sender-index**$(e)$
    $\mathsf{vid}' \leftarrow \mathsf{epID}[E']$
    $m' \leftarrow \perp$
    **if *FS-sec**$(\mathsf{vid}', \mathsf{S}', i')$
        $(s[\mathsf{R}], E', \mathsf{S}', i', m') \leftarrow \mathsf{Rcv}(s[\mathsf{R}], a', e')$
        $\mathsf{AM}[\mathsf{vid}', \mathsf{S}', i', \mathsf{R}] \leftarrow \texttt{received}$
        **if** $\mathsf{Del}[\mathsf{R}]$
            $\mathsf{AM\text{-}Rcvd}[\mathsf{vid}] \mathrel{+\!\!\leftarrow} (\mathsf{S}', i', \mathsf{R})$
    **return** $(E', \mathsf{S}', i', m')$
```

Figure 30: The oracle for injecting application messages for the correctness and privacy properties.

- $\mathsf{inj\text{-}type} \in \{\mathsf{gkb}, \mathsf{pm}, \mathsf{cm}, \mathsf{wm}, \mathsf{am}\}$: the injection type can be one of the following
    - gkb: injection against **get-KB**; requires the adversary to set the id of the target signature key, skid^*.
    - pm: injection against **inj-PM**; requires the adversary to set the target epoch, ep^*, and the epoch, ep_h^*, of the last honest commit (healing), before ep^*.
    - cm: injection against **inj-CM**; requires ep^*, ep_h^*, as defined above.
    - wm: injection against **inj-WM**; requires ep^*, ep_h^*, as defined above.
    - am: injection against **inj-AM**; requires ep^*, ep_h^*, as defined above, as well as the target sender ID^* and message index i^*.

If during the execution, the adversary mounts an injection attack that is not consistent with the above values, then loses the game immediately.
The advantage of $\mathcal{A}$ in the authenticity game is denoted by $\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth},\Pi}(\mathcal{A})$.

Definition 18. *A group messaging scheme* Γ *is* (t, ε)-authentic *w.r.t. predicates* Π*, if for all* t*-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth},\Pi}(\mathcal{A}) \ \leq \ \varepsilon \ .$$

Privacy. The privacy variant of the SGM game only provides the reduced inject oracle of Figure 30, and there is no **win** instruction inside the delivery oracles **dlv-PM**, **dlv-CM**, **dlv-AM** and if $\mathsf{skid} \notin \mathsf{SK\text{-}Lk} \wedge \mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$ is satisfied inside **get-KB**, the adversary loses the game. Moreover, the adversary needs to initialize the game with arguments ep^*, ep_h^*, ID^*, i^*, such that, the adversary issues a single challenge query in epoch ep^*, for the $i^{*\text{th}}$ message sent by ID^*. As above, ep_h^* is the epoch before ep^* of the last honest commit. If during the execution, the adversary issues a challenge query that is not consistent with the above values, then loses the game immediately.
The advantage of $\mathcal{A}$ in the privacy game is denoted by $\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Priv},\Pi}(\mathcal{A})$.

Definition 19. *A group messaging scheme* Γ *is* $(t, 1, \varepsilon)$-private *w.r.t. predicates* Π*, if for all* $(t, 1)$*-attackers* $\mathcal{A}$*,*

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Priv},\Pi}(\mathcal{A}) \ \leq \ \varepsilon \ .$$

Combining the properties. The following theorem relates the simplified properties to the full SGM security definition.

Theorem 25. *Assume a group messaging scheme* Γ *is*

- $(t', \varepsilon_{\mathsf{corr}})$*-correct,*
- $(t', \varepsilon_{\mathsf{auth}})$*-authentic, and*
- $(t', 1, \varepsilon_{\mathsf{priv}})$*-private,*

with respect to predicate Π*, then, it is also* (t, q, ε)*-secure GM w.r.t. predicate* Π*, where,*

$$\varepsilon \;\leq\; \varepsilon_{\mathsf{corr}} + t^4 \cdot (10 \cdot \varepsilon_{\mathsf{auth}} + q \cdot \varepsilon_{\mathsf{priv}})$$

and $t \approx t'$*.*

Proof. Let $\mathcal{A}$ be any (t, q) adversary. The proof is via a series of hybrid experiments and the distances between successive hybrids are bounded via reductions to correctness, authenticity, and privacy.

Hybrid H_0. Hybrid H_0 is the original SGM game with respect to Γ. We denote the advantage of $\mathcal{A}$ in H_0 by $\mathrm{Adv}^{\Gamma}_{H_0,\Pi}(\mathcal{A})$.

H.2 Dealing with Correctness

Hybrid H_1. Hybrid H_1 works as H_0, except that the **win** condition inside the oracles **dlv-PM**, **dlv-CM** **dlv-WM**, **dlv-AM**, is removed.

Lemma 26. *There exists a* t'*-attacker* $\mathcal{B}_1$*, with* $t' \approx t$*, such that,*

$$\mathrm{Adv}^{\Gamma}_{H_0,\Pi}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\Gamma}_{H_1,\Pi}(\mathcal{A}) + \mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Corr},\Pi}(\mathcal{B}_1)\;.$$

Proof. For both H_0 and H_1, consider the event $\mathcal{E}$ that the **win** condition inside one of the **dlv-PM**, **dlv-CM**, **dlv-WM** and **dlv-AM** oracles is provoked. Observe that, since the game ends upon **win**, if $\mathcal{E}$ occurs, it occurs before condition **win** in one of the remaining oracles is triggered. It suffices to upper bound the probability of $\mathcal{E}$ occurring in, say, H_0. This is achieved via a reduction: consider an attacker $\mathcal{B}_1$ that plays against the correctness game with respect to Γ, executes $\mathcal{A}$ and attempts to simulate H_0 to $\mathcal{A}$. I.e., $\mathcal{B}_1$ has access to all SGM oracles but **chall**, **inj-CM**, **inj-PM**, **inj-WM**. It also receives access to the reduced injection oracle **red-inj-AM** (instead of **inj-AM**). On the other hand, in H_0, $\mathcal{A}$ has access to all SGM oracles.

The simulation proceeds as follows (the oracles $\mathcal{B}_1$ interacts with are referred to as $\mathcal{B}_1$'s *own* oracles):

- Initially, choose a bit $b \in \{0, 1\}$ randomly and initialize empty Chall.
- Initialize a local copy of the history graph variables (cf. Figure 1) and run $\mathcal{A}$.
- Calls to the PKI oracles made by $\mathcal{A}$ are simply forwarded to it's own PKI oracles.
- Simulate the remaining SGM game oracles for $\mathcal{A}$ as follows:
  - For calls to **commit**, **send**, **create-group**, **prop-add-user**, **prop-rem-user**, **prop-up-user**, **corr**, **no-del**, **dlv-PM**, **dlv-CM**, **dlv-WM**, simply forward to own and return the outputs to $\mathcal{A}$.
  - **chall** $(\mathsf{S}, a, m_0, m_1)$: If $\mathsf{V\text{-}Pt[S]} \neq \mathsf{vid_{root}}$ and $|m_0| = |m_1|$, execute **send** (S, a, m_b) to retrieve e, set $\mathsf{Chall[V\text{-}Pt[S]]} \mathrel{+}\!\!\leftarrow (\mathsf{S}, i[\mathsf{S}])$, and pass e to $\mathcal{A}$.

- **dlv-AM** $(\mathsf{vid}, \mathsf{S}, i, \mathsf{R})$: execute own oracle on the same input, and if $(\mathsf{S}, i) \in \mathsf{Chall}[\mathsf{vid}]$, return $\perp$ to $\mathcal{A}$ and set $\mathsf{Chall}[\mathsf{vid}] \mathrel{-\!\!\leftarrow} (\mathsf{S}, i)$, otherwise pass the response m to $\mathcal{A}$.
- **inj-AM**: forward to own **red-inj-AM**.
- **inj-CM**, **inj-PM**, **inj-WM**: no action.

After each operation, update the local copy of the history graph accordingly.

We now show that $\mathcal{B}_1$ correctly simulates H_0. For challenge queries, if $\mathsf{V\text{-}Pt}[\mathsf{S}] \neq \mathsf{vid}_{\mathsf{root}}$, then the output of the **send** oracle is identical to that of **chall** and by setting $\mathsf{Chall}[\mathsf{V\text{-}Pt}[\mathsf{S}]] \mathrel{+\!\!\leftarrow} (\mathsf{S}, i[\mathsf{S}])$, $\mathcal{B}_1$ fully simulates the computation of **chall**. Clearly, $\mathcal{B}_1$ is consistently updating it's local copy of the history graph as it only depends on public information, thus the value of $\mathsf{V\text{-}Pt}[\mathsf{S}]$ matches that of the actual game. For **dlv-AM**, an extra check is required, namely $\mathcal{B}_1$ checks if $(\mathsf{S}, i) \in \mathsf{Chall}[\mathsf{vid}]$, in which case it returns $\perp$ to $\mathcal{A}$, as it would happen in H_0.

For injection queries, recall that whenever $\mathcal{E}$ occurs, a **win** condition inside one of the **dlv-PM**, **dlv-CM**, **dlv-WM** and **dlv-AM**, is provoked, thus no **win** is triggered inside the injection oracles. Therefore, the reduced inject oracle **red-inj-AM** of the correctness game game is identical to **inj-AM**, as the message returned by both oracles is generated using keys from compromised epochs. Similarly, the **win** condition inside **inj-CM**, **inj-PM**, **inj-WM** is not being triggered, thus users' states remain unchanged and $\mathcal{B}_1$ correctly takes no action for such queries. All the remaining queries are handled by $\mathcal{B}_1$ as in H_0. Hence, the probability that $\mathcal{B}_1$ wins the correctness game is equal to the probability that $\mathcal{A}$ provokes $\mathcal{E}$ in H_0. □ □

H.3 Getting Rid of Injections

Hybrid H_2. Hyrbid H_2 works as H_1, but without the **inj-CM**, **inj-PM**, **inj-WM**, oracles and with the *reduced* inject oracle **red-inj-AM**, as defined in Section H.1. In the following, denote the advantage of $\mathcal{A}$ in H_2 by $\mathrm{Adv}^{\Gamma}_{H_1,\Pi}(\mathcal{A})$.

Lemma 27. *There exists a t-attacker $\mathcal{B}_2$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_1,\Pi}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\Gamma}_{H_2,\Pi}(\mathcal{A}) + 10t^4 \cdot \mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth},\Pi}(\mathcal{B}_2) \;.$$

Proof. In order to distinguish H_1 and H_2, $\mathcal{A}$ must provoke the **win** condition in the inject oracles since H_1 and H_2 behave identically otherwise. Note that if **win** is not triggered, the reduced inject oracle **red-inj-AM** is identical to **inj-AM**, i.e., either $m' = \perp$ or the adversary injects a ciphertext that has been computed with compromised keys. Therefore, it suffices to upper bound the probability of this event $\mathcal{E}$ in, say, H_1, which is achieved via a reduction: Consider an attacker $\mathcal{B}_2$ against authenticity that uses $\mathcal{A}$ and attempts to simulate H_1 to $\mathcal{A}$.

The simulation proceeds as follows (the oracles $\mathcal{B}_2$ interacts with are referred to as $\mathcal{B}_2$'s *own* oracles):

- Sample $b \leftarrow \{0, 1\}$ and initialize empty Chall.
- Initialize a local copy of the history graph variables (cf. Figure 1).
- Sample uniformly random $\mathsf{inj\text{-}type} \leftarrow \{\mathsf{gkb}, \mathsf{pm}, \mathsf{wm}, \mathsf{cm}, \mathsf{am}\}$, $\mathsf{comp\text{-}sig} \in \{\mathsf{true}, \mathsf{false}\}$.
- Sample uniformly random values ep^*, $\mathsf{ep}^*_{\mathsf{h}}$, skid^*, ID^*, i^*,[16] and initialize the SGM authenticity game parameterized by those values and $\mathsf{inj\text{-}type}$, $\mathsf{comp\text{-}sig}$. Run $\mathcal{A}$.

[16]The space size for those variables is bounded by the running time of the adversary.

- Calls to the PKI oracles made by $\mathcal{A}$, are simply forwarded to it's own PKI oracles. Update the local copy of the history graph accordingly.
- Simulate the remaining SGM game oracles for $\mathcal{A}$ as follows:
  - For calls to **commit**, **send**, **create-group**, **prop-add-user**, **prop-rem-user**, **prop-up-user**, **corr**, **no-del**, **dlv-PM**, **dlv-CM**, **dlv-WM**, simply forward to own and return the outputs to $\mathcal{A}$.
  - **chall** $(\mathsf{S}, a, m_0, m_1)$: If $\mathsf{V\text{-}Pt[S]} \neq \mathsf{vid_{root}}$ and $|m_0| = |m_1|$, execute **send** (S, a, m_b) to retrieve e, set $\mathsf{Chall[V\text{-}Pt[S]]} \mathrel{+}\!\!\leftarrow (\mathsf{S}, i[\mathsf{S}])$, and pass e to $\mathcal{A}$.
  - **dlv-AM** $(\mathsf{vid}, \mathsf{S}, i, \mathsf{R})$: execute own oracle on the same input, and if $(\mathsf{S}, i) \in \mathsf{Chall[vid]}$, return $\perp$ to $\mathcal{A}$ and set $\mathsf{Chall[vid]} \mathrel{-}\!\!\leftarrow (\mathsf{S}, i)$, otherwise pass the response m to $\mathcal{A}$.
  - **inj-CM**, **inj-PM**, **inj-WM**, **inj-AM**: forward to own.

After each operation, update the local copy of the history graph accordingly.

Observe that for both authenticity game and H_1, the **win** conditions has been removed from the delivery oracles, while in the former there is no challenge oracle. The arguments for proving that challenge queries are simulated correctly, is identical to that of Lemma 27. For queries to the remaining oracles $\mathcal{B}_2$ simply forwards them to it's own oracles. Hence, the probability that $\mathcal{B}_2$ wins the authenticity game is equal to the probability that $\mathcal{A}$ provokes $\mathcal{E}$ in H_1, times the probability that $\mathcal{B}_2$ makes correct guesses for the parameters of the authenticity game, which is at least $1/10t^4$: observe that not all types of injections require correct guessing of all parameters. The biggest number of parameters that one needs to correctly guess is at most four, as for instance in **inj-AM** when $\mathsf{comp\text{-}sig} = \mathsf{true}$, which requires correct guessing of ID^*, i^*, ep^*, $\mathsf{ep}^*_{\mathsf{h}}$, therefore the probability of guessing correctly the type of injection $\mathsf{inj\text{-}type}$ and whether is an attack against signatures, $\mathsf{comp\text{-}sig}$, as well as the correct parameters for that type is at least $\frac{1}{5 \cdot 2 \cdot t^4}$. □ □

H.4 Privacy

Recall that, in H_2 there is are no **inj-CM**, **inj-PM**, **inj-WM**, oracles and there is a *reduced* inject oracle **red-inj-AM**. Also, there is no **win** condition inside the oracles **dlv-PM**, **dlv-CM** **dlv-WM**, **dlv-AM**. Thus, H_2 constitutes an adaptive, multi-challenge version, of the privacy game. We prove the following lemma.

Lemma 28. *There exists a $(t', 1)$-attacker $\mathcal{B}$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Priv},\Pi}(\mathcal{A}) \;\leq\; q \cdot \mathrm{Adv}^{\Gamma}_{H_2,\Pi}(\mathcal{B}) \;.$$

Proof. We define the following sequence of hybrids.

Hybrids H^{p}_i. These hybrids, for $i = 1, \ldots, q$, work as H_2, but up to the i-th challenge query the game encrypts m_0 (i.e., we assume $b = 0$), while for the remaining challenge queries it encrypts m_1. Let $\mathcal{E}$ be the event that $\mathcal{A}$ wins the privacy game against Γ, and let let $\mathcal{A}(H^{\mathsf{p}}_i)$ be the output of $\mathcal{A}$ when playing against H^{p}_i. Using the hybrid definitions and the facts that (i) H^{p}_q matches H_2 for $b = 0$, and (ii) H^{p}_0 matches H_2 for $b = 1$, we compute,

$$\mathsf{Pr}\left[\mathcal{E}\right] = \frac{1}{2}\left(\mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_q) = 0\right] + \mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_0) = 1\right]\right).$$

Given $\mathcal{A}$ we define a singe challenge adversary $\mathcal{B}$ as follows:

- Initialize a local copy of the history graph variables (cf. Figure 1) and run $\mathcal{A}$.
- Choose uniformly random $i \leftarrow [q]$, set $\mathsf{Chall} \leftarrow \emptyset$, $j \leftarrow 0$, and initialize $i[\cdot] \leftarrow 0$.
- Forwards all communication between $\mathcal{A}$ and the H_i^{p} oracles and update the history graph information, except for the calls to **send**, **chall** and **dlv-AM**, which are handled as follows:
  - **send**(S, a, m): set $i[\mathsf{S}]{+}{+}$, forward to own and return the output of the oracle to $\mathcal{A}$.
  - **chall**$(\mathsf{S}, a, m_0, m_1)$: If ***compat-chall**$(\mathsf{S})$ and $|m_0| = |m_1|$, set $i[\mathsf{S}]{+}{+}$, $j{+}{+}$, $\mathsf{Chall} \mathrel{+}\!\!\leftarrow (\mathsf{S}, i[\mathsf{S}])$, $\mathcal{C} \mathrel{+}\!\!\leftarrow (\mathsf{V\text{-}Pt}[\mathsf{S}], \mathsf{S}, i[\mathsf{S}], a)$ and
    - If $j = i$, forward to own and return the reply to $\mathcal{A}$.
    - Otherwise,
      1. If $j < i$, set $m \leftarrow m_0$.
      2. If $j > i$, set $m \leftarrow m_1$.
      3. Send (S, a, m) to own **send** and return the output to $\mathcal{A}$.
  - **dlv-AM**$(\mathsf{vid}, \mathsf{S}, i, a)$: if $(\mathsf{vid}, \mathsf{S}, i, a) \in \mathcal{C}$ return $\perp$. Otherwise, forward to own and return the result to $\mathcal{A}$.

  After each query update the local copy of the history graph accordingly.
- Output the bit b that is output by $\mathcal{A}$.

Note that in the above execution, the i^{th} ciphertext given to $\mathcal{A}$ is the challenge ciphertext of $\mathcal{B}$ in the single challenge game. Intuitively, $\mathcal{B}$ is guessing an index $i \in [q]$ for which $\mathcal{A}$ distinguishes between H_i^{p} and H_{i-1}^{p}, which implies it can distinguish between an encryption of m_0 and m_1 (the messages used in the i^{th} call to the challenge oracle), which is the only difference between H_i^{p} and H_{i-1}^{p}.

Let $\mathcal{O}_0$ (resp. $\mathcal{O}_1$) be the event in which $\mathcal{B}$ outputs 0 (resp. 1) in the above execution. We compute

$$\mathsf{Pr}\left[\mathcal{O}_0 \mid b = 0\right] = \sum_{k=1}^{q} \mathsf{Pr}\left[\mathcal{O}_0 \mid b = 0 \wedge i = k\right] \cdot \mathsf{Pr}\left[i = k\right] = \sum_{k=1}^{q} \frac{1}{q} \cdot \mathsf{Pr}\left[\mathcal{A}(H_k^{\mathsf{p}}) = 0\right].$$

Symmetrically,

$$\begin{aligned}\mathsf{Pr}\left[\mathcal{O}_1 \mid b = 1\right] &= \sum_{k=1}^{q} \mathsf{Pr}\left[\mathcal{O}_1 \mid b = 1 \wedge i = k\right] \cdot \mathsf{Pr}\left[i = k\right] \\ &= \sum_{k=1}^{q} \frac{1}{q} \cdot \mathsf{Pr}\left[\mathcal{A}(H_{k-1}^{\mathsf{p}}) = 1\right] = \sum_{k=0}^{q-1} \frac{1}{q} \cdot \mathsf{Pr}\left[\mathcal{A}(H_k^{\mathsf{p}}) = 1\right].\end{aligned}$$

Let $\mathcal{E}'$ be the event in which $\mathcal{B}$ wins in H_2 using a single challenge. Then, we compute

$$\begin{aligned}
\frac{1}{2} + \mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{B}) &\geq \mathsf{Pr}\left[\mathcal{E}'\right] = \frac{1}{2}\left(\mathsf{Pr}\left[\mathcal{O}_0 \mid b=0\right] + \mathsf{Pr}\left[\mathcal{O}_1 \mid b=1\right]\right) \\
&= \frac{1}{2}\left(\sum_{k=1}^{q}\frac{1}{q}\cdot \mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_k)=0\right] + \sum_{k=0}^{q-1}\frac{1}{q}\cdot \mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_k)=1\right]\right) \\
&= \frac{1}{2q}\cdot\sum_{k=1}^{q-1}\left(\mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_k)=0\right] + \mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_k)=1\right]\right) \\
&+ \frac{1}{2q}\cdot\left(\mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_q)=0\right] + \mathsf{Pr}\left[\mathcal{A}(H^{\mathsf{p}}_0)=1\right]\right) \\
&= \frac{q-1}{2q} + \frac{1}{q}\cdot \mathsf{Pr}\left[\mathcal{E}\right]
\end{aligned}$$

From the above we derive that

$$\mathsf{Pr}\left[\mathcal{E}\right] \leq q\cdot \mathrm{Adv}^{\Gamma}_{H_2,\Pi}(\mathcal{B}) + \frac{1}{2}\ ,$$

and the proof is concluded. □ □

Finally, we prove the following lemma

Lemma 29. *There exists a t'-attacker $\mathcal{B}'$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_2,\Pi}(\mathcal{B}) \ \leq\ t^4\cdot \mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Priv},\Pi}(\mathcal{B}')\ .$$

Proof. Recall that in the privacy game the adversary commits to $(\mathsf{S}^*, i^*, \mathsf{ep}^*, \mathsf{ep}^*_{\mathsf{h}})$ during game initialization. Thus for any adaptive adversary $\mathcal{B}$, there exists a non-adaptive adversary $\mathcal{B}'$, which simply guesses $(\mathsf{S}^*, i^*, \mathsf{ep}^*, \mathsf{ep}^*_{\mathsf{h}})$ (with probability at least $\frac{1}{t^4}$) and then executes $\mathcal{B}$. The advantage of $\mathcal{B}'$ is at least $\frac{1}{t^4}\cdot \mathrm{Adv}^{\mathsf{F}}_{H_2,\Pi}(\mathcal{B})$, and the proof of the lemma is concluded. □ □

From the above lemmas we derive that

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM}}(\mathcal{A}) \leq \varepsilon_{\mathsf{corr}} + 5t^4\cdot\varepsilon_{\mathsf{auth}} + q\cdot t^4\cdot\varepsilon_{\mathsf{priv}}\ .$$

□ □

H.5 Security of SGM construction

Theorem 25 reduces SGM security to three simpler properties. As above, let Γ denote the SGM construction depicted in Figures 7, 8 and 9. In what follows we show that Γ satisfies the three properties.

H.6 Proof of Authenticity

In this section we prove the following Theorem.

Theorem 30. *The scheme Γ is (t', ε')-authentic w.r.t. Π_{SGM}, for $\varepsilon' = (\varepsilon_{\mathsf{sig}} + \varepsilon_{\mathsf{CR}} + \varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{PKE}} + \varepsilon_{\mathsf{PP}} + \varepsilon_{\mathsf{mac}} + \varepsilon_{\mathsf{FS}})$, and $t' \approx t$.*

Proof. Authenticity of our scheme is proven via reductions to the properties of the underlying schemes, depending on the injection type committed by the adversary when the game begins. In particular, we consider the following cases:

1. $\mathsf{comp\text{-}sig} = \mathsf{false}$: requires a reduction to the security of the underlying EU-CMA signature scheme independently of $\mathsf{inj\text{-}type}$. $\mathcal{A}$ is required to set skid^*.
2. $\mathsf{comp\text{-}sig} = \mathsf{true}$:
   (a) $\mathsf{inj\text{-}type} \in \{\mathsf{pm}, \mathsf{cm}, \mathsf{wm}\}$: requires reductions to the collision resistance of H, and CGKA, PKE, PRF-PRNG, and MAC security. $\mathcal{A}$ is required to set ep^*, $\mathsf{ep}^*_{\mathsf{h}}$.
   (b) $\mathsf{inj\text{-}type} = \mathsf{am}$: requires reductions to the collision resistance of H , and CGKA, PKE, PRF-PRNG, and FS-GAEAD security. $\mathcal{A}$ is required to set ep^*, $\mathsf{ep}^*_{\mathsf{h}}$, ID^*, i^*.
   (c) $\mathsf{inj\text{-}type} = \mathsf{gkb}$: the adversary loses the game.

One could consider a single reduction covering the above cases, however, for the ease of exposition, we split our reduction into parts that handle similar cases. First we introduce some variables that will be used by our reductions:

- $\mathsf{ep}_{\mathsf{cur}}$: refers to the current epoch of a party at the time, or right before, the party executes an SGM protocol operation.
- $\mathsf{ep}_{\mathsf{new}}$: refers to the new epoch that will be created after a party executes an SGM protocol operation.

We now prove each case separately.

H.6.1 Case 1: $\mathsf{comp\text{-}sig} = \mathsf{false}$.

Let $\mathcal{A}$ be any t-adversary against authenticity of Γ. We prove the following lemma.

Lemma 31. *There exists a t'-attacker $\mathcal{B}_1$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth},\Pi_{\mathsf{SGM}}}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\mathsf{S}}_{\mathsf{EU\text{-}CMA}}(\mathcal{B}_1) \ .$$

Proof.
$\mathcal{B}_1$ simulates the SGM authenticity game for $\mathcal{A}$ while playing against the EU-CMA game w.r.t. S. $\mathcal{B}_1$ receives the verification key, spk^*, of S and executes the following steps.

- Run $\mathcal{A}$, receive $\mathsf{inj\text{-}type}$, skid^*, and initialize the SGM authenticity game with those parameters.
- For calls to **rem-SigK**, **get-SK**, **create-group**, **dlv-PM**, **dlv-CM**, **dlv-WM**, **dlv-AM**, **no-del**, execute the code of the oracles on the inputs and return the outputs to $\mathcal{A}$. Other calls are handled below.
- **gen-new-SigK**(ID): compute $\mathsf{skid} \leftarrow \mathsf{idCtr}{+}{+}$ and simulate the rest of **gen-new-SigK** as follows
  - If $\mathsf{skid} \neq \mathsf{skid}^*$, execute the remaining code of **gen-new-SigK** normally,
  - Otherwise, omit $(s[\mathsf{ID}], \mathsf{spk}) \leftarrow \mathsf{Gen\text{-}SK}(s[\mathsf{ID}])$, include spk^* in $s[\mathsf{ID}]$, set $\mathsf{SK\text{-}PK}[\mathsf{skid}^*] \leftarrow \mathsf{spk}^*$, execute the remaining operations normally and return the output to $\mathcal{A}$.
- **gen-new-KB**($\mathsf{ID}, \mathsf{skid}$):
  - If $\mathsf{skid} \neq \mathsf{skid}^*$, execute the code of the oracle normally.

  - Otherwise, execute a modified $\mathsf{Gen\text{-}KB}(s[\mathsf{ID}], \mathsf{spk})$, that omits all operations related to ssk, send wpk (generated inside $\mathsf{Gen\text{-}KB}$) to the singing oracle of the EU-CMA game, receive the signature sig and executes the remaining code w.r.t. sig.

- **prop-add-user**$(\mathsf{ID}, \mathsf{ID}')$: execute the code of the oracle normally, however, in $\mathsf{Add}(s[\mathsf{ID}], \mathsf{ID}')$, if $\mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, \mathsf{ME}] = \mathsf{spk}^*$, send a signing query (P', t) (as they are computed by Add) to the EU-CMA challenger, receive sig, and use it in the rest of the computation. If $\mathsf{s.Ep\text{-}SPK}[\mathsf{s.C\text{-}epid}, \mathsf{ME}] \neq \mathsf{spk}^*$, execute the query normally.

- **prop-rem-user**$(\mathsf{ID}, \mathsf{ID}')$: similar to **prop-add-user** w.r.t. Remove.

- **prop-up-user**$(\mathsf{ID}, \mathsf{skid}, r)$: similar to **prop-add-user** w.r.t. Update.

- **commit**$(\mathsf{ID}, \textbf{pid}, r)$: similar to **prop-add-user** w.r.t. Commit (the message to the signing oracle is (T', t)).

- **send**(S, a, m): similar to **prop-add-user** w.r.t. Send (the message to the signing oracle is e').

- **get-KB**$(\mathsf{ID}, \mathsf{ID}', \mathsf{kb})$: compute $(\mathsf{wpk}, \mathsf{spk}, \mathrm{sig}) \leftarrow \mathsf{kb}$ and if $\mathsf{spk} \neq \mathsf{spk}^*$, execute the query normally. Otherwise,

  - If $\mathsf{inj\text{-}type} = \mathsf{gkb}$,

    * If $\mathsf{skid} \notin \mathsf{SK\text{-}Lk} \wedge \mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$ is satisfied, send the forgery $(\mathsf{wpk}, \mathrm{sig})$ to the challenger of the EU-CMA game and halt.
    * Otherwise, abort ($\mathcal{A}$ loses the game).

  - If $\mathsf{inj\text{-}type} \neq \mathsf{gkb}$ and **win** is triggered, abort ($\mathcal{A}$ loses), otherwise execute the rest of **get-KB**.

- **inj-PM**(ID, P'):

  - If $\mathsf{inj\text{-}type} \neq \mathsf{pm}$, abort ($\mathcal{A}$ loses).
  - Otherwise, if $\mathsf{SK\text{-}ID}[\mathsf{skid}^*] \neq \mathsf{ID}$, abort ($\mathcal{A}$ loses).
  - Otherwise, if $\neg$***compat-inj-PM**(ID, P'), abort ($\mathcal{A}$ loses).
  - Otherwise, parse $(\mathrm{P}', t, \mathrm{sig}) \leftarrow P'$ and output $((\mathrm{P}', t), \mathrm{sig})$ as a forgery to EU-CMA game.

- **inj-CM**(ID, T'):

  - If $\mathsf{inj\text{-}type} \neq \mathsf{cm}$, abort ($\mathcal{A}$ loses).
  - Otherwise, if $\mathsf{SK\text{-}ID}[\mathsf{skid}^*] \neq \mathsf{ID}$, abort ($\mathcal{A}$ loses).
  - Otherwise, if $\neg$***compat-inj-CM**(ID, T'), abort ($\mathcal{A}$ loses).
  - Otherwise, parse $(\mathrm{T}', t, \mathrm{sig}) \leftarrow T'$ and output $((\mathrm{T}', t), \mathrm{sig})$ as a forgery to EU-CMA game.

- **inj-WM**(ID, W'):

  - If $\mathsf{inj\text{-}type} \neq \mathsf{wm}$, abort ($\mathcal{A}$ loses).
  - Otherwise, if $\mathsf{SK\text{-}ID}[\mathsf{skid}^*] \neq \mathsf{ID}$, abort ($\mathcal{A}$ loses).
  - Otherwise, if $\neg$***compat-inj-WM**(ID, W'), abort ($\mathcal{A}$ loses).

    - Otherwise, parse $(\mathrm{W}', t, \mathrm{sig}) \leftarrow W'$ and output $((\mathrm{W}', t), \mathrm{sig})$ as a forgery to EU-CMA game.

- **inj-AM**(a', e', R):
    - If $\mathsf{inj\text{-}type} \neq \mathsf{am}$, abort ($\mathcal{A}$ loses).
    - Otherwise, if $\mathsf{SK\text{-}ID}[\mathsf{skid}^*] \neq \mathsf{ID}$, abort ($\mathcal{A}$ loses).
    - Otherwise, if $\neg$***compat-inj-AM**(a', e', R), abort ($\mathcal{A}$ loses).
    - Otherwise, parse $(e, \mathrm{sig}) \leftarrow e'$ and output (e, sig) as a forgery to EU-CMA game.

- **corr**(ID): execute the oracle code normally and if $\mathsf{skid}^* \in \mathsf{SK\text{-}Lk}$, abort ($\mathcal{A}$ losses), otherwise return the output to $\mathcal{A}$.

We now argue about correctness of the above simulation. Clearly, calls to **rem-SigK**, **get-SK**, **create-group**, **dlv-PM**, **dlv-CM**, **dlv-WM**, **dlv-AM**, **no-del**, are completely independent of any private signature key and their execution depends only on public values, thus simulation is straightforward. **gen-new-SigK** executes normally if the newly generated key has id different than skid^*, but for skid^* injects the public key spk^* of the EC-CMA game, and omits any operation that is related to ssk^*. **gen-new-KB**, if $\mathsf{skid} \neq \mathsf{skid}^*$, executes normally, otherwise, sends a signing request to the EU-CMA game. **prop-add-user**, **prop-rem-user**, **prop-up-user**, are similar: if the proposals is signed with a key different than ssk^*, they execute normally, otherwise a signing request is sent to the EU-CMA game. **commit**, **send**, are similar. **get-KB** checks the verification key of the input signature, and if it corresponds to spk^*, checks whether the adversary has committed to a gkb type of injection, and if $\mathsf{skid} \notin \mathsf{SK\text{-}Lk} \wedge \mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$, outputs the forgery to the EU-CMA game. If the verification key is not spk^*, it executes normally. **inj-PM**, **inj-CM**, **inj-WM**, **inj-AM**, work similar. Finally, **corr** executes normally, and only aborts if the adversary compromises the secret key of skid^*. However, such an adversary would violate the condition $\mathsf{comp\text{-}sig} = \mathsf{false}$ and ***SK-compr**$(\mathsf{ep}_{\mathsf{h}}^*, \mathsf{ID})$, and would lose the game immediately. $\square$ $\square$

We conclude that for an $\mathcal{A}$ with $\mathsf{comp\text{-}sig} = \mathsf{false}$,

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth},\Pi_{\mathsf{SGM}}}(\mathcal{A}) \leq \varepsilon_{\mathsf{sig}} \ .$$

H.6.2 Case 2: $\mathsf{comp\text{-}sig} = \mathsf{true}$.

We consider a sequence of hybrids which are common for both Cases 2a,2b. For Case 2c, there is nothing to prove.

H_0 executes as the original game, but in addition, keeps a list of hash/pre-image pairs $(\mathsf{v}, \mathrm{T}')$, computed by the commit operation, and if it finds T', T'', $\mathrm{T}' \neq \mathrm{T}''$, such that $\mathsf{H}(\mathrm{T}') = \mathsf{H}(\mathrm{T}'')$, execution aborts. Clearly, indistinguishability between the H_0 and the original game, follows by the collision resistance property of H. More formally, we have the following lemma.

Lemma 32. *There exists a t'-attacker $\mathcal{B}_0$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth},\Pi_{\mathsf{SGM}}}(\mathcal{A}) \ \leq \ \mathrm{Adv}^{\Gamma}_{H_0,\Pi_{\mathsf{SGM}}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{H}}_{\mathsf{CR}}(\mathcal{B}_0) \ .$$

The proof is by a straightforward reduction to the collision resistance property of H: the only way to distinguish between the original authenticity game and H_0 is by finding a collision against H.

Therefore, $\mathcal{B}_0$ simulates the SGM authenticity game w.r.t. to Γ until $\mathcal{A}$ finds a collision, and outputs that collision against H.

H_1, in epoch $\mathsf{ep}^*_{\mathsf{h}}$, substitutes the CGKA update secret, I, computed by a commit (or group create) operation of the CGKA scheme, with a uniformly random value. We formally prove the following.

Lemma 33. *There exists a t'-attacker $\mathcal{B}_1$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_0,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\Gamma}_{H_1,\Pi_{\mathsf{SGM}}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{K}}_{\mathsf{CGKA},\Pi_{\mathsf{CGKA}}}(\mathcal{B}_1) \ .$$

Proof. $\mathcal{B}_1$ plays against the CGKA security game w.r.t. K and simulates for $\mathcal{A}$, H_0 or H_1, depending on the secret bit sampled by the CGKA game.

- Run $\mathcal{A}$, receive $\mathsf{inj\text{-}type}$, ep^*, $\mathsf{ep}^*_{\mathsf{h}}$, initialize the SGM authenticity game with those parameters and set $\mathsf{WK\text{-}IK}[\cdot] \leftarrow \varepsilon$.
- For calls to **gen-new-SigK**, **rem-SigK**, **get-SK**, **dlv-PM**, **send**, **dlv-AM**, execute the code of the oracle normally and return the output to $\mathcal{A}$.
- **get-KB**$(\mathsf{ID}, \mathsf{ID}', \mathsf{kb})$: execute normally and if $\mathsf{skid} \notin \mathsf{SK\text{-}Lk} \wedge \mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$, abort ($\mathcal{A}$ loses).
- **gen-new-KB**$(\mathsf{ID}, \mathsf{skid})$: execute the code of the oracle with the following differences: in $\mathsf{Gen\text{-}KB}$ instead of executing $\mathsf{K\text{-}Gen\text{-}IK}$, make a query to the CGKA oracle **gen-new-ik** with input ID, receive ikid, ipk, and execute the remaining code of $\mathsf{Gen\text{-}KB}$ without setting wsk, and receive $(\mathsf{wkid}, \mathsf{kb})$. Set $\mathsf{WK\text{-}IK}[\mathsf{wkid}] \leftarrow \mathsf{ikid}$.
- **create-group**$(\mathsf{ID}, \mathsf{skid}, \mathsf{wkid}, r)$:
  - If $\mathsf{ep}_{\mathsf{new}} \neq \mathsf{ep}^*_{\mathsf{h}}$, send the query **create-group**$(\mathsf{ID}, \mathsf{WK\text{-}IK}[\mathsf{wkid}], r)$ to the CGKA game challenger, and if **create-group** returns vid, send a **reveal** query for vid (the epoch created by the group creation operation) to the CGKA challenger and receive I. Then, execute $\mathsf{Create}(s[\mathsf{ID}], \mathsf{SK\text{-}PK}[\mathsf{skid}], \mathsf{WK\text{-}PK}[\mathsf{wkid}]; r)$, however, omit the execution of $\mathsf{K\text{-}Create}$, and use I as being the output of $\mathsf{K\text{-}Create}$ for the rest of the computation.
  - If $\mathsf{ep}_{\mathsf{new}} = \mathsf{ep}^*_{\mathsf{h}}$, execute as above, however instead of a reveal query, send a challenge query **chall**(vid), for vid that corresponds to epoch $\mathsf{ep}^*_{\mathsf{h}}$, and use the reply I^*, to simulate the rest of the computation.
- **prop-add-user**$(\mathsf{ID}, \mathsf{ID}')$: execute the code of **prop-add-user**, however instead of executing $\mathsf{Add}(s[\mathsf{ID}], \mathsf{ID}')$, send the query **prop-add-user**$(\mathsf{ID}, \mathsf{ID}', \mathsf{ikid}')$ to the CGKA game, where ikid' is the id of ID''s ipk used by the add operation. If the query is successful, **prop-add-user** returns (pid, P). Use P to simulate the rest of the execution of **prop-add-user**.
- **prop-rem-user**$(\mathsf{ID}, \mathsf{ID}')$: Similar to **prop-add-user** but substitute the call to Remove with a CGKA oracle query to **prop-rem-user**.
- **prop-up-user**$(\mathsf{ID}, \mathsf{skid}, r)$: Similar to **prop-add-user** but substitute the call to Update with a CGKA oracle query to **prop-up-user**.
- **commit**$(\mathsf{ID}, \mathbf{pid}, r)$:

  - If $\mathsf{ep}_{\mathsf{new}} \neq \mathsf{ep}^*_{\mathsf{h}}$, send the query $\mathbf{commit}(\mathsf{ID}, \mathbf{pid}, r)$ to CGKA game challenger, and let W_{pub}, $\mathbf{W}_{\mathsf{priv}}$, T, be the reply. Send a **reveal** query for vid (the epoch created by the commit operation) to the CGKA challenger and receive I. Then, execute **commit**, however omit the execution of $\mathsf{K\text{-}Commit}$, and use W_{pub}, $\mathbf{W}_{\mathsf{priv}}$, T, I, as being the output of $\mathsf{K\text{-}Commit}$ for the rest of the computation.
  - If $\mathsf{ep}_{\mathsf{new}} = \mathsf{ep}^*_{\mathsf{h}}$, execute as above, however instead of a reveal query, send a challenge query, $\mathbf{chall}(\mathsf{vid})$, to the CGKA game challenger, for vid that corresponds to epoch $\mathsf{ep}^*_{\mathsf{h}}$, and use the reply I^*, together with W_{pub}, $\mathbf{W}_{\mathsf{priv}}$, T (as computed above), to simulate the rest of the commit operation.

- $\mathbf{dlv\text{-}CM}(\mathsf{ID}, \mathsf{vid})$:
  - Query the oracle **process** of the CGKA game with input $(\mathsf{vid}, \mathsf{ID})$, and let GI be the reply (if the call fails, abort).
  - Execute the code of **dlv-CM**, and when computing $\mathsf{Proc\text{-}CM}(s[\mathsf{ID}], T)$, omit the execution of $\mathsf{K\text{-}Proc\text{-}Com}$, and use GI and the corresponding CGKA key I used in the **commit** that created vid (see above), to simulate the rest of the execution.
- $\mathbf{dlv\text{-}WM}(\mathsf{ID}, \mathsf{vid})$: query the oracle **dlv-WM** of the CGKA game with input $(\mathsf{vid}, \mathsf{ID})$, and let GI be the reply (if the call fails, abort). Execute the code of the SGM **dlv-WM**, and when computing $\mathsf{Proc\text{-}WM}(s[\mathsf{ID}], W)$, omit the execution of $\mathsf{K\text{-}Join}$, and use GI and the corresponding CGKA key I used in the **commit** that created vid (see above), to simulate the rest of the execution.
- $\mathbf{inj\text{-}PM}(\mathsf{ID}, P')$:
  - If $\mathsf{inj\text{-}type} \neq \mathsf{pm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, execute the code of the oracle normally and halt.
- $\mathbf{inj\text{-}CM}(\mathsf{ID}, T')$:
  - If $\mathsf{inj\text{-}type} \neq \mathsf{cm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, send a $\mathbf{corr}(\mathsf{ID})$ query to CGKA game, and using ID's state execute the code of **inj-CM** and halt.
- $\mathbf{inj\text{-}WM}(\mathsf{ID}, W')$:
  - If $\mathsf{inj\text{-}type} \neq \mathsf{wm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, send a $\mathbf{corr}(\mathsf{ID})$ query to CGKA game, and using ID's state execute the code of **inj-WM** and halt.
- $\mathbf{inj\text{-}AM}(a', e', \mathsf{R})$:
  - If $\mathsf{inj\text{-}type} \neq \mathsf{am}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).

  – Otherwise, execute the code of **inj-AM** normally.

- **corr**(ID): query **corr** of the CGKA game with ID, incorporate the returned CGKA state in the SGM state of ID, execute the code of **corr**(ID) in the SGM game and send the output to $\mathcal{A}$. If $\mathcal{A}$ violates the safety predicate w.r.t. ep^*_h, abort ($\mathcal{A}$ loses).
- **no-del**(ID): execute SGM's **no-del** and query the CGKA game with **no-del**(ID).

$\mathcal{B}_1$ outputs the bit that is output by $\mathcal{A}$.

We argue that the above simulation is correct. Clearly, calls to **gen-new-SigK**, **rem-SigK**, **get-SK**, **dlv-PM**, **send**, **dlv-AM**, are independent of any CGKA key material, thus execution proceeds without modifications. In **get-KB**, simply check if the adversary tries to inject, in which case immediately loses the game as the $\mathcal{B}_1$ only deals with insecure signature keys ($\mathsf{comp\text{-}sig} = \mathsf{true}$). **gen-new-KB** substitute the $\mathsf{K\text{-}Gen\text{-}IK}$ operation with a call to the **gen-new-ik** oracle, and avoids operations related to wsk. In this way, $\mathcal{B}_1$ manages to properly initialize the appropriate CGKA key material without touching the CGKA state. **create-group**, if $\mathsf{ep_{new}} \neq \mathsf{ep}^*_\mathsf{h}$, it executes a **create-group** CGKA query query and reveals the newly created key (this is not affecting security as the newly created epoch is not ep^*_h). Then, uses the CGKA key to execute the rest of the oracle code while omitting $\mathsf{K\text{-}Create}$. If $\mathsf{ep_{new}} = \mathsf{ep}^*_\mathsf{h}$, the reveal query is substituted by a **chall** query. **commit** is similar. **prop-add-user**, substitutes the execution of Add, with the CGKA query **prop-add-user** and manages to generate a proposal without touching the CGKA state. **prop-rem-user**, **prop-up-user**, are similar. **dlv-CM**, **dlv-WM**, make the appropriate CGKA queries so as to make CGKA state evolve, and in addition, in the SGM level, use the CGKA keys, computed above, to fully simulate SGM operations. Simulation of **inj-PM**, **inj-CM**, **inj-WM**, **inj-AM**, **no-del**, is straightforward. **corr** sends a **corr** query to the CGKA game, incorporates the returned state to the SGM date of ID, and executes the code of **corr**(ID) in the SGM game. It is essential that the corruption will not affect security of the update secret for epoch ep^*_h. We have that $\mathsf{comp\text{-}sig} = \mathsf{true}$, therefore for a successful injection it is required that $\neg$***auth-compr**(ep^*). By definition, this implies ***PP-secure**(ep^*, ***Proj-PP**($\mathsf{SGM\text{-}Data}$)), which, by definition, implies that $\exists\ \mathsf{vid}' \succeq \mathsf{ep}^*$ s.t.

1. $\neg\mathsf{BI}[\mathsf{vid}']$
2. $\nexists\ \mathsf{vid}'' \in [\mathsf{vid}', \mathsf{vid}) : (\mathsf{vid}'', \cdot) \in \mathsf{V\text{-}Lk}$

By assumption ep^*_h is the last honestly generated commit, thus $\mathsf{vid}' = \mathsf{ep}^*_\mathsf{h}$, and the key generated in ep^*_h is a secure CGKA key, therefore the execution w.r.t. $\mathcal{B}_1$ satisfies Π_CGKA. Clearly, if the secret bit sampled by the CGKA game is 0, the CGKA key returned as a reply to the **chall** query is the actual CGKA key, and $\mathcal{B}_1$ simulates H_0, otherwise, it simulates H_1. The distinguishing advantage is bounded by the advantage of breaking CGKA security. □ □

The next hybrid, H_2, when computing welcome messages for epoch ep^*, instead of encrypting the PRF-PRNG state ($e \leftarrow \mathsf{E\text{-}Enc}(\mathbf{wpk}[i].\mathsf{epk}, \mathsf{s}.\sigma)$), encrypts the zero message $e \leftarrow \mathsf{E\text{-}Enc}(\mathbf{wpk}[i].\mathsf{epk}, \mathbf{0})$. Then, for all users that process that welcome message, omit decryption ($\mathsf{s}.\sigma \leftarrow \mathsf{E\text{-}Dec}(\mathsf{esk}, e)$) and PP computation ($(\mathsf{s}.\sigma, k_e, \mathsf{s.k_m}, \mathsf{s.C\text{-}epid}) \leftarrow \mathsf{PP}(\mathsf{s}.\sigma, I, \mathsf{v})$), and directly use the output of PP that has been computed by the commit operation when the welcome message was created. Then, we have the following lemma.

Lemma 34. *There exists a t'-attacker $\mathcal{B}_2$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_1,\Pi_\mathsf{SGM}}(\mathcal{A}) \ \leq\ \mathrm{Adv}^{\Gamma}_{H_2,\Pi_\mathsf{SGM}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{PKE}}_{\mathsf{CPA}}(\mathcal{B}_2)\ .$$

The proof of the above lemma is simply via a reduction to the CPA security of PKE, and therefore omitted.

The next hybrid, H_3, in epoch ep^*, substitutes the output of the PRF-PRNG, PP, with a uniformly random value. We formally prove the following lemma.

Lemma 35. *There exists a t'-attacker $\mathcal{B}_3$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_2,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \ \leq \ \mathrm{Adv}^{\Gamma}_{H_3,\Pi_{\mathsf{SGM}}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{PP}}_{\mathsf{PRF\text{-}PRNG},\Pi_{\mathsf{PP}}}(\mathcal{B}_3) \ .$$

Proof. $\mathcal{B}_3$ plays against the PRF-PRNG security game and simulates for $\mathcal{A}$ H_2, or H_3, depending on the secret bit of the PRF-PRNG game. $\mathcal{B}_3$ executes the following code.

- Run $\mathcal{A}$, receive $\mathsf{inj\text{-}type}$, ep^*, $\mathsf{ep}^*_{\mathsf{h}}$ and initialize H_2 with those parameters.
- For calls **gen-new-SigK**, **rem-SigK**, **get-SK**, **gen-new-KB**, **prop-add-user**, **prop-rem-user**, **prop-up-user**, **no-del**, execute the query normally. Other calls are simulated as described below.
- **get-KB**$(\mathsf{ID}, \mathsf{ID}', \mathsf{kb})$: execute normally and if $\mathsf{skid} \notin \mathsf{SK\text{-}Lk} \wedge \mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$, abort ($\mathcal{A}$ loses).
- **create-group**$(\mathsf{ID}, \mathsf{skid}, \mathsf{wkid}, r)$: Execute the oracle code and compute I.
  - If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*_{\mathsf{h}}$ and $r \neq \bot$ abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ and $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*_{\mathsf{h}}$, query **process** of the PRF-PRNG game with $(\mathsf{vid}_{\mathsf{root}}, I, 0)$.
  - If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*_{\mathsf{h}}$ and $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$, query **process** of the PRF-PRNG game with $(\mathsf{vid}_{\mathsf{root}}, \bot, 0)$.
  - If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*$, abort ($\mathcal{A}$ loses).

  Receive vid', send a **reveal**(vid') query to the PRF-PRNG game challenger, and use the returned value as the output of PP to simulate the rest of the oracle execution for ID.
- **commit**$(\mathsf{ID}, \mathbf{pid}, r)$: Execute the oracle code and compute I, v.
  - If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*_{\mathsf{h}}$ and $r \neq \bot$ abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ and $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*_{\mathsf{h}}$, query **process** of the PRF-PRNG game with $(\mathsf{vid}, I, \mathsf{v})$.
  - If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*_{\mathsf{h}}$, query **process** of the PRF-PRNG game with $(\mathsf{vid}, \bot, \mathsf{v})$.
  - If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*$ and $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*_{\mathsf{h}}$, query the PRF-PRNG game with **chall**(vid).

  Receive vid', send a **reveal**(vid') query to the PRF-PRNG game challenger, and use the returned value as the output of PP to simulate the rest of the oracle execution for ID.
- **dlv-PM**$(\mathsf{ID}, \mathsf{pid})$: execute the code of the oracle but use the appropriate MAC key that was generated by the PRF-PRNG game to verify authenticity of the proposal.
- **dlv-CM**$(\mathsf{ID}, \mathsf{vid})$: execute the code of the oracle but instead of evaluating PP using the corresponding output that has been produced by the PRF-PRNG game via **commit**.
- **dlv-WM**$(\mathsf{ID}, \mathsf{vid})$: similar to **dlv-CM**.
- **inj-PM**(ID, P'):

  - If $\mathsf{inj\text{-}type} \neq \mathsf{pm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, execute the code of the oracle normally and use the appropriate MAC key that was generated by the PRF-PRNG game as the output of PP to verify authenticity of the proposal.

- **inj-CM**(ID, T'): abort ($\mathcal{A}$ loses the game).
  - If $\mathsf{inj\text{-}type} \neq \mathsf{cm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, execute the code of the oracle normally and use the output of the PRF-PRNG game as the output of PP in the rest of the execution.

- **inj-WM**(ID, W'): abort ($\mathcal{A}$ loses the game).
  - If $\mathsf{inj\text{-}type} \neq \mathsf{wm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, execute the code of the oracle normally and use the output of the PRF-PRNG game as the output of PP in the rest of the execution.

- **inj-AM**(a', e', R): abort ($\mathcal{A}$ loses the game).
  - If $\mathsf{inj\text{-}type} \neq \mathsf{am}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, execute the code of the oracle normally and use the output of the PRF-PRNG game as the output of PP in the rest of the execution.

- **send**(S, a, m): execute normally given the FS-GAEAD states that have been initialized via calls to the PRF-PRNG game.
- **dlv-AM**$(\mathsf{vid}, \mathsf{S}, i, \mathsf{R})$: same as **send**.
- **corr**(ID): simulate the oracle call and send a **corr** call to the PRF-PRNG game for vid.

We argue on the correctness of the above simulation. Clearly, **gen-new-SigK**, **rem-SigK**, **get-SK**, **gen-new-KB**, **prop-add-user**, **prop-rem-user**, **prop-up-user**, **no-del**, are either independent of the PP output, or they just use the output of PP that has been set by operations such as **commit**, **create-group** (such output is directly used by **prop-add-user**, **prop-rem-user**, **prop-up-user**), that use the uniformly random MAC key that has been computed after processing messages of epoch ep^*. In **get-KB**, simply check if the adversary tries to inject, in which case immediately loses the game as the $\mathcal{B}_3$ only deals with insecure signature keys ($\mathsf{comp\text{-}sig} = \mathsf{true}$). **create-group**, **commit** are similar: if the current epoch is the "healing epoch" ($\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*_{\mathsf{h}}$) and the operation uses bad randomness, the adversary loses. If the current epoch is not the healing epoch, nor the target epoch, simply make a query to **process** of the PRF-PRNG game with the computed CGKA key. If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*_{\mathsf{h}}$, query the PRF-PRNG game with **process**$(\mathsf{vid}, \perp, \mathsf{v})$, i.e., for this operation the state of PP receives good randomness that will be used to protect the target epoch ep^*. In epoch ep^* send a challenge query and use the reply to simulate the rest of the execution. Other operations use the output of PP as computed above. Finally, for **corr**(ID), simulate the oracle call and send a **corr** call to the PRF-PRNG game for vid. It is essential that the corruption will not affect the PRF-PRNG state in epoch ep^*. We have that $\mathsf{comp\text{-}sig} = \mathsf{true}$, therefore for a successful injection it is required that $\neg$***auth-compr**(ep^*). By definition, this implies ***PP-secure**(ep^*, ***Proj-PP**($\mathsf{SGM\text{-}Data}$)), which, by definition, implies that $\exists\ \mathsf{vid}' \succeq \mathsf{ep}^*$ s.t.

1. $\neg\mathsf{BI}[\mathsf{vid}']$
2. $\nexists\ \mathsf{vid}'' \in [\mathsf{vid}', \mathsf{vid}) : (\mathsf{vid}'', \cdot) \in \mathsf{V\text{-}Lk}$

By assumption ep^*_h is the last honestly generated commit, thus $\mathsf{vid}' = \mathsf{ep}^*_\mathsf{h}$. In addition, ***PP-secure** guarantees that there is no compromised node in $[\mathsf{vid}', \mathsf{vid})$, and by the collision resistance property of H, we have that in the presence of group splits (or forks in HG), the PRF-PRNG state on different branches are independent. This ensures that corruptions on other branches do not compromise security of the PRF-PRNG state in $[\mathsf{vid}', \mathsf{vid})$, and the safety predicate of the PRF-PRNG game, Π_{PP}, is satisfied, therefore, $\mathcal{B}_3$ queries the game with a valid challenge. If the secret bit sampled by the PRF-PRNG game is 0, $\mathcal{B}_3$ receives the actual PP output and simulates H_2 for $\mathcal{A}$, otherwise it receives a uniformly random value and simulates H_3. Therefore, the distinguishing advantage is bounded by the advantage of breaking the PRF-PRNG security. □ □

Next our proof splits into two cases; below we consider reduction to the security of the MAC scheme, while the latter reduction is to the security of authenticity property of the FS-GAEAD scheme.

H.6.3 Case 2a: $\mathsf{comp\text{-}sig} = \mathsf{true}$, $\mathsf{inj\text{-}type} \in \{\mathsf{pm}, \mathsf{cm}, \mathsf{wm}\}$.

We prove the following lemma.

Lemma 36. *There exists a t'-attacker $\mathcal{B}_4$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_3,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \ \leq\ \mathrm{Adv}^{\mathsf{M}}_{\mathsf{MAC}}(\mathcal{B}_4)\ .$$

Proof. $\mathcal{B}_4$ simulates H_3 for $\mathcal{A}$ while playing against the MAC security game. $\mathcal{B}_4$ is defined as follows.

- Run $\mathcal{A}$, receive $\mathsf{inj\text{-}type}$, ep^*, ep^*_h, and initialize H_3 with those parameters.
- For calls to **gen-new-SigK**, **rem-SigK**, **get-SK**, **gen-new-KB**, **create-group**, **send**, **dlv-AM**, **no-del**, execute the code of the oracles and return the output to $\mathcal{A}$. Other calls are simulated as described below.
- **get-KB**($\mathsf{ID}, \mathsf{ID}', \mathsf{kb}$): execute normally and if $\mathsf{skid} \notin \mathsf{SK\text{-}Lk} \wedge \mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$, abort ($\mathcal{A}$ loses).
- **prop-add-user**($\mathsf{ID}, \mathsf{ID}'$):
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, instead of computing $t \leftarrow \mathsf{M\text{-}Tag}(\mathsf{s.k_m}, \mathrm{P}')$, query oracle of the MAC security game with P' and use the returned value to simulate the rest of the computation.
- **prop-rem-user**($\mathsf{ID}, \mathsf{ID}'$): similar to **prop-add-user**.
- **prop-up-user**($\mathsf{ID}, \mathsf{skid}, r$): similar to **prop-add-user**.
- **dlv-PM**($\mathsf{ID}, \mathsf{pid}$):

  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, instead of computing $\mathsf{M\text{-}Ver}(\mathsf{s.k_m}, \mathrm{P}', t)$, send a verification query (P', t) to the MAC security game and simulate the rest of the computation w.r.t. that output.

- **commit**$(\mathsf{ID}, \mathbf{pid}, r)$: similar to **prop-add-user**.
- **dlv-CM**$(\mathsf{ID}, \mathsf{vid})$: similar to **dlv-PM**.
- **dlv-WM**$(\mathsf{ID}, \mathsf{vid})$: similar to **dlv-PM**.
- **inj-PM**(ID, P'):
  - If $\mathsf{inj\text{-}type} \neq \mathsf{pm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, parse $(\mathrm{P}', t, \mathrm{sig}) \leftarrow P'$ and output (P', t) as a forgery to the MAC security game.
- **inj-CM**(ID, T'):
  - If $\mathsf{inj\text{-}type} \neq \mathsf{cm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, parse $(\mathrm{T}', t, \mathrm{sig}) \leftarrow T'$ and output (T', t) as a forgery to the MAC security game.
- **inj-WM**(ID, W'):
  - If $\mathsf{inj\text{-}type} \neq \mathsf{wm}$, abort ($\mathcal{A}$ loses).
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or the last honest commit is not in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, parse $(\mathrm{W}', t, \mathrm{sig}) \leftarrow W'$ and output (W', t) as a forgery to the MAC security game.
- **inj-AM**(a', e', R): abort ($\mathcal{A}$ loses).
- **corr**(ID): if compromises the MAC key of epoch ep^*, abort ($\mathcal{A}$ loses). Otherwise execute the query normally.

By inspection we see that the above simulation is correct and $\mathcal{B}_4$ manages to simulate H_3 for $\mathcal{A}$ by requesting MAC tag creation and verification queries from the MAC security game w.r.t. M. Observe that $\mathcal{B}_4$ (as well as $\mathcal{A}$) will never get to see the MAC key for epoch ep^*, as it would violate the security predicate: since ***PP-secure** holds in $\mathsf{ep}^*_{\mathsf{h}}$, the output of ***PP-secure** is securely substituted by a uniformly value (including the MAC key), therefore no compromise that exposes that key is allowed. □ □

From the above we derive that for any adversary complying to Case 2a

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth}, \Pi_{\mathsf{SGM}}}(\mathcal{A}) \leq \varepsilon_{\mathsf{CR}} + \varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{PKE}} + \varepsilon_{\mathsf{PP}} + \varepsilon_{\mathsf{mac}} \ .$$

H.6.4 Case 2b: $\mathsf{comp\text{-}sig} = \mathsf{true}$, $\mathsf{inj\text{-}type} = \mathsf{am}$.

We conclude security against injections for the adversaries such that $\mathsf{comp\text{-}sig} = \mathsf{true}$, $\mathsf{inj\text{-}type} = \mathsf{am}$, via a reduction to the FS-GAED authenticity property. More formally, we prove the following lemma.

Lemma 37. *There exists a t'-attacker $\mathcal{B}_5$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\mathsf{\Gamma}}_{H_3,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \;\le\; \mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Auth},\Pi_{\mathsf{FS}}}(\mathcal{B}_5)\;.$$

Proof. $\mathcal{B}_5$ simulates H_3 for $\mathcal{A}$ while playing against the FS-GAEAD authenticity game.

- Run $\mathcal{A}$, receive $\mathsf{inj\text{-}type}(=\mathsf{am})$, ep^*, $\mathsf{ep}^*_{\mathsf{h}}$, ID^*, i^* and initialize H_3 game with those parameters.
- For calls to **gen-new-SigK**, **rem-SigK**, **get-SK**, **gen-new-KB**, **prop-add-user**, **prop-rem-user**, **prop-up-user**, **dlv-PM**, execute the queries normally. Calls to other oracles are described below.
- **get-KB**$(\mathsf{ID}, \mathsf{ID}', \mathsf{kb})$: execute normally and if $\mathsf{skid} \notin \mathsf{SK\text{-}Lk} \wedge \mathsf{WK\text{-}SK}[\mathsf{wkid}] \neq \mathsf{skid}$, abort ($\mathcal{A}$ loses).
- **create-group**$(\mathsf{ID}, \mathsf{skid}, \mathsf{wkid}, r)$:
  - If $\mathsf{ep}_{\mathsf{new}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, execute the query but substitute the output of PP with a uniformly random value and omit the FS-GAEAD state initialization, i.e., omit $\mathsf{F\text{-}Init}$, and initialize the FS-GAEAD security game with $G = \{\mathsf{ID}\}$ (i.e., execute **init**(G)) .
- **inj-PM**(ID, P'): abort ($\mathcal{A}$ loses).
- **commit**$(\mathsf{ID}, \mathbf{pid}, r)$:
  - If $\mathsf{ep}_{\mathsf{new}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, substitute the output of PP with a uniformly random value.
- **dlv-CM**$(\mathsf{ID}, \mathsf{vid})$:
  - If $\mathsf{ep}_{\mathsf{new}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, execute the query but substitute the output of PP with the uniformly random value used above by **commit** (or **create-group**) and omit the FS-GAEAD state initialization, i.e., omit $\mathsf{F\text{-}Init}$, and initialize the FS-GAEAD security game with $\mathsf{s.G}$ (i.e., execute **init**$(\mathsf{s.G})$) .
- **inj-CM**(ID, T'): abort ($\mathcal{A}$ loses).
- **dlv-WM**$(\mathsf{ID}, \mathsf{vid})$: similar to **dlv-CM**.
- **inj-WM**(ID, W'): abort ($\mathcal{A}$ loses).
- **send**(S, a, m):
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$, execute the query normally.

  - Otherwise, omit $\mathsf{F\text{-}Send}$ and query the FS-GAEAD game's **send** oracle with (S, a, m) and use the reply $\bar{e}$ to simulate the rest of the **send** execution.

- **dlv-AM**$(\mathsf{vid}, \mathsf{S}, i, \mathsf{R})$:
  - If $\mathsf{ep_{cur}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, omit $\mathsf{F\text{-}Rcv}$ and query the FS-GAEAD game with a **dlv-AM**$(\mathsf{S}, i, \mathsf{R})$ call.

- **inj-AM**(a', e', R): parse $(\mathsf{S}, i, e) \leftarrow e'$ and
  - If $\mathsf{ep_{cur}} \neq \mathsf{ep}^*$ or $\mathsf{S} \neq \mathsf{ID}^*$ or $i \neq i^*$ or ep^*_h is not the epoch of the last good commit, abort ($\mathcal{A}$ loses).
  - Otherwise, send an **inj-AM** query to the FS-GAEAD security game with input $(a', \mathsf{S}, i, e, \mathsf{R})$ and halt.

- **corr**(ID):
  - If ID's state contains information that enables processing of the $i^{*\text{th}}$ message sent by ID^* in epoch ep^*, abort ($\mathcal{A}$ loses).
  - Otherwise, send a **corr**(ID) query to the FS-GAEAD game, incorporate the FS-GAEAD state to the SGM state of ID, execute **corr** of the SGM game and return the output to $\mathcal{A}$.

- **no-del**(ID): execute normally. If $\mathsf{ep_{cur}} = \mathsf{ep}^*$, send a **no-del**$(\mathsf{ID})$ query to the FS-GAEAD game challenger.

By inspection we see that the above simulation is correct and $\mathcal{B}_5$ manages to simulate H_3 for $\mathcal{A}$ using the oracles of the FS-GAEAD security game. Whenever $\mathcal{B}_5$ needs to execute Init, if the current epoch is not ep^*, it executes the code locally, otherwise it sends a **init** query to the FS-GAEAD security game. In addition, whenever it needs to send, or deliver, a message, if the message is for the target epoch ep^* it users the oracles of the FS-GAEAD game, otherwise it executes the operation based on local information. Security against this type of injection, requires either the FS-GAEAD state in epoch ep^* to be secure, of if it is compromised, the compromised party should have already received the message for the target sender ID^* and message index i^*. By the SGM security game we have that ***AM-sec**$(\mathsf{ep}^*, \mathsf{ID}^*, i^*)$ holds, which implies security of the PRF-PRNG output in ep^*, as well as ***FS-sec**(ID^*, i^*), which is the safety predicate of the FS-GAEAD game. Therefore, a valid injection against **inj-AM** implies a valid injection against the FS-GAEAD target session, and the proof is concluded. □ □

From the above lemmas we derive that for any adversary $\mathcal{A}$,

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Auth},\Pi_{\mathsf{SGM}}}(\mathcal{A}) \leq \varepsilon_{\mathsf{sig}} + \varepsilon_{\mathsf{CR}} + \varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{PKE}} + \varepsilon_{\mathsf{PP}} + \varepsilon_{\mathsf{mac}} + \varepsilon_{\mathsf{FS}} \ ,$$

and the authenticity proof is concluded.

□ □

H.7 SGM Privacy

In this section we prove the following theorem.

Theorem 38. *The scheme Γ is (t', ε')-private w.r.t. Π_{SGM}, for $\varepsilon' = (\varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{CR}} + \varepsilon_{\mathsf{PKE}} + \varepsilon_{\mathsf{PP}} + \varepsilon_{\mathsf{FS}})$, and $t' \approx t$.*

Our proof proceeds in hybrids, similar to the ones we used for proving authenticity of $\mathsf{\Gamma}$.

H_0 executes as the original game, but in addition, keeps a list of hash/pre-image pairs $(\mathsf{v}, \mathrm{T}')$, computed by the commit operation, and if it finds T', T'', $\mathrm{T}' \neq \mathrm{T}''$, such that $\mathsf{H}(\mathrm{T}') = \mathsf{H}(\mathrm{T}'')$, execution aborts. Clearly, indistinguishability between the H_0 and the original game, follows by the collision resistance property of H. More formally, we have the following lemma.

Lemma 39. *There exists a t'-attacker $\mathcal{B}_0$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\mathsf{\Gamma}}_{\mathsf{SGM\text{-}Priv},\Pi_{\mathsf{SGM}}}(\mathcal{A}) \ \leq\ \mathrm{Adv}^{\mathsf{\Gamma}}_{H_0,\Pi_{\mathsf{SGM}}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{H}}_{\mathsf{CR}}(\mathcal{B}_0)\ .$$

H_1 is identical to H_0, except that in epoch $\mathsf{ep}^*_{\mathsf{h}}$, the CGKA update secret, I, is substituted with a uniformly random and independent value. We formally prove the following lemma.

Lemma 40. *There exists a t'-attacker $\mathcal{B}_1$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\mathsf{\Gamma}}_{H_0,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \ \leq\ \mathrm{Adv}^{\mathsf{\Gamma}}_{H_1,\Pi_{\mathsf{SGM}}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{K}}_{\mathsf{CGKA},\Pi_{\mathsf{CGKA}}}(\mathcal{B}_1)\ .$$

Proof. $\mathcal{B}_1$ plays against the CGKA security game w.r.t. K, and simulates for $\mathcal{A}$ H_0, or H_1, depending on the secret bit sampled by the CGKA game.

- Run $\mathcal{A}$, receive ep^*, $\mathsf{ep}^*_{\mathsf{h}}$, ID^*, i^*, initialize the SGM privacy game with those parameters, and set $\mathsf{WK\text{-}IK}[\cdot] \leftarrow \varepsilon$.
- For calls to **gen-new-SigK**, **rem-SigK**, **get-SK**, **get-KB**, **dlv-PM**, **send**, **dlv-AM**, **red-inj-AM**, execute the code of the oracle normally and return the output to $\mathcal{A}$.
- **gen-new-KB**$(\mathsf{ID}, \mathsf{skid})$: execute the code of the oracle with the following differences: in $\mathsf{Gen\text{-}KB}$ instead of executing $\mathsf{K\text{-}Gen\text{-}IK}$, make a query to the CGKA oracle **gen-new-ik** with input ID, receive ikid, ipk, and execute the remaining code of $\mathsf{Gen\text{-}KB}$ without setting wsk, and return $(\mathsf{wkid}, \mathsf{kb})$ to $\mathcal{A}$. Set $\mathsf{WK\text{-}IK}[\mathsf{wkid}] \leftarrow \mathsf{ikid}$.
- **create-group**$(\mathsf{ID}, \mathsf{skid}, \mathsf{wkid}, r)$:
  - If $\mathsf{ep}_{\mathsf{new}} \neq \mathsf{ep}^*_{\mathsf{h}}$, send the query **create-group**$(\mathsf{ID}, \mathsf{WK\text{-}IK}[\mathsf{wkid}], r)$ to the CGKA game challenger, and if **create-group** returns vid, send a **reveal** query for vid (the epoch created by the group creation operation) to the CGKA challenger and receive I. Then, execute $\mathsf{Create}(s[\mathsf{ID}], \mathsf{SK\text{-}PK}[\mathsf{skid}], \mathsf{WK\text{-}PK}[\mathsf{wkid}]; r)$, however, omit the execution of $\mathsf{K\text{-}Create}$, and use I as being the output of $\mathsf{K\text{-}Create}$ for the rest of the computation.
  - If $\mathsf{ep}_{\mathsf{new}} = \mathsf{ep}^*_{\mathsf{h}}$, execute as above, however instead of a reveal query, send a challenge query **chall**(vid), for vid that corresponds to epoch $\mathsf{ep}^*_{\mathsf{h}}$, and use the reply I^*, to simulate the rest of the computation.
- **prop-add-user**$(\mathsf{ID}, \mathsf{ID}')$: execute the code of **prop-add-user**, however instead of executing $\mathsf{Add}(s[\mathsf{ID}], \mathsf{ID}')$, send the query **prop-add-user**$(\mathsf{ID}, \mathsf{ID}', \mathsf{ikid}')$ to the CGKA game, where ikid' is the id of ipk used by the add operation. If the query is successful, **prop-add-user** returns (pid, P). Use P to simulate the rest of the execution of **prop-add-user**.
- **prop-rem-user**$(\mathsf{ID}, \mathsf{ID}')$: Similar to **prop-add-user** but substitute the call to Remove with a CGKA oracle query to **prop-rem-user**.

- **prop-up-user**$(\mathsf{ID}, \mathsf{skid}, r)$: Similar to **prop-add-user** but substitute the call to Update with a CGKA oracle query to **prop-up-user**.
- **commit**$(\mathsf{ID}, \mathbf{pid}, r)$:
  - If $\mathsf{ep}_{\mathsf{new}} \neq \mathsf{ep}^*_{\mathsf{h}}$, send the query **commit**$(\mathsf{ID}, \mathbf{pid}, r)$ to CGKA game challenger, and let W_{pub}, $\mathbf{W}_{\mathsf{priv}}$, T, be the reply. Send a **reveal** query for vid (the epoch created by the commit operation) to the CGKA challenger and receive I. Then, execute **commit**, however omit the execution of K-Commit, and use W_{pub}, $\mathbf{W}_{\mathsf{priv}}$, T, I, as being the output of K-Commit for the rest of the computation.
  - If $\mathsf{ep}_{\mathsf{new}} = \mathsf{ep}^*_{\mathsf{h}}$, execute as above, however instead of a reveal query, send a challenge query, **chall**(vid), to the CGKA game challenger, for vid that corresponds to epoch $\mathsf{ep}^*_{\mathsf{h}}$, and use the reply I^*, together with W_{pub}, $\mathbf{W}_{\mathsf{priv}}$, T (as computed above), to simulate the rest of the commit operation.
- **dlv-CM**$(\mathsf{ID}, \mathsf{vid})$:
  - Query the oracle **process** of the CGKA game with input $(\mathsf{vid}, \mathsf{ID})$, and let GI be the reply (if the call fails, abort).
  - Execute the code of **dlv-CM**, and when computing $\mathsf{Proc\text{-}CM}(s[\mathsf{ID}], T)$, omit the execution of K-Proc-Com, and use GI and the corresponding CGKA key I used in the **commit** that created vid (see above), to simulate the rest of the execution.
- **dlv-WM**$(\mathsf{ID}, \mathsf{vid})$:
  - Query the oracle **dlv-WM** of the CGKA game with input $(\mathsf{vid}, \mathsf{ID})$, and let GI be the reply (if the call fails, abort).
  - Execute the code of the SGM **dlv-WM**, and when computing $\mathsf{Proc\text{-}WM}(s[\mathsf{ID}], W)$, omit the execution of K-Join, and use GI and the corresponding CGKA key I used in the **commit** that created vid (see above), to simulate the rest of the execution.
- **chall**$(\mathsf{S}, a, m_0, m_1)$
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or $\mathsf{S} \neq \mathsf{ID}^*$ or $i[\mathsf{S}] \neq i^*$ or if the last good commit wasn't in epoch $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, execute the query normally.
- **corr**(ID): query **corr** of the CGKA game with ID, incorporate the returned CGKA state in the SGM state of ID, execute the code of **corr**(ID) in the SGM game and send the output to $\mathcal{A}$. If $\mathcal{A}$ violates the saftey predicate w.r.t. $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
- **no-del**(ID): execute SGM's **no-del** and query the CGKA game with **no-del**(ID).

$\mathcal{B}_1$ outputs the bit that is output by $\mathcal{A}$.

Simulation correctness follows by inspection and using similar arguments to the ones used in the proof of Lemma 33. □ □

The next hybrid, H_2, when computing welcome messages for epoch ep^*, instead of encrypting the PRF-PRNG state ($e \leftarrow \mathsf{E\text{-}Enc}(\mathbf{wpk}[i].\mathsf{epk}, \mathsf{s}.\sigma)$), encrypts the zero message $e \leftarrow \mathsf{E\text{-}Enc}(\mathbf{wpk}[i].\mathsf{epk}, \mathbf{0})$. Then, for all users that process that welcome message, omit decryption ($\mathsf{s}.\sigma \leftarrow \mathsf{E\text{-}Dec}(\mathsf{esk}, e)$) and

PP computation $((\mathsf{s}.\sigma, k_e, \mathsf{s.k_m}, \mathsf{s.C\text{-}epid}) \leftarrow \mathsf{PP}(\mathsf{s}.\sigma, I, \mathsf{v}))$, and directly use the output of PP that has been computed by the commit operation when the welcome message was created. Then, we have the following lemma.

Lemma 41. *There exists a t'-attacker $\mathcal{B}_2$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_1,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\Gamma}_{H_2,\Pi_{\mathsf{SGM}}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{PKE}}_{\mathsf{CPA}}(\mathcal{B}_2)\ .$$

The proof of the above lemma is simply via a reduction to the CPA security of PKE, and therefore omitted.

H_3 substitutes the output of the PRF-PRNG, PP, in epoch ep^*, with a uniformly random value. Formally, we have the following lemma.

Lemma 42. *There exists a t'-attacker $\mathcal{B}_3$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_2,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\Gamma}_{H_3,\Pi_{\mathsf{SGM}}}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{PP}}_{\mathsf{PRF\text{-}PRNG},\Pi_{\mathsf{PP}}}(\mathcal{B}_3)\ .$$

The proof of the above lemma is similar to the one given for Lemma 35, modified appropriately to capture the privacy game w.r.t. Γ.

Finally we prove the following lemma.

Lemma 43. *There exists a t'-attacker $\mathcal{B}_4$, with $t' \approx t$, such that*

$$\mathrm{Adv}^{\Gamma}_{H_3,\Pi_{\mathsf{SGM}}}(\mathcal{A}) \;\leq\; \mathrm{Adv}^{\mathsf{F}}_{\mathsf{FSAE\text{-}Priv},\Pi_{\mathsf{FS}}}(\mathcal{B}_4)\ .$$

Proof. $\mathcal{B}_4$ simulates H_3 for $\mathcal{A}$ while playing the privacy game against F. $\mathcal{B}_4$ is defined as follows.

- Run $\mathcal{A}$, receive ep^*, $\mathsf{ep}^*_{\mathsf{h}}$, ID^*, i^*, and initialize H_3 game with those parameters.
- For calls to **gen-new-SigK**, **rem-SigK**, **get-SK**, **gen-new-KB**, **get-KB**, **prop-add-user**, **prop-rem-user**, **prop-up-user**, **dlv-PM**, execute the query normally. Other calls are simulated as described below.
- **create-group**$(\mathsf{ID}, \mathsf{skid}, \mathsf{wkid}, r)$:
  - If $\mathsf{ep_{new}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, execute the query but substitute the output of PP with a uniformly random value and omit the FS-GAEAD state initialization, i.e., omit $\mathsf{F\text{-}Init}$, and initialize the FS-GAEAD security game with $G = \{\mathsf{ID}\}$ (i.e., execute **init**(G)) .
- **commit**$(\mathsf{ID}, \textbf{pid}, r)$:
  - If $\mathsf{ep_{new}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, substitute the output of PP with a uniformly random value.
- **dlv-CM**$(\mathsf{ID}, \mathsf{vid})$:
  - If $\mathsf{ep_{new}} \neq \mathsf{ep}^*$, execute the query normally.

  - Otherwise, execute the query but substitute the output of PP with the uniformly random value used above by **commit** (or **create-group**) and omit the FS-GAEAD state initialization, i.e., omit $\mathsf{F\text{-}Init}$, and initialize the FS-GAEAD security game with $\mathsf{s.G}$ (i.e., execute **init**($\mathsf{s.G}$)) .

- **dlv-WM**($\mathsf{ID}, \mathsf{vid}$): similar to **dlv-CM**.

- **send**(S, a, m):
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, omit $\mathsf{F\text{-}Send}$ and query the FS-GAEAD game's **send** oracle with (S, a, m) and use the reply $\bar{e}$ to simulate the rest of the **send** execution.

- **dlv-AM**($\mathsf{vid}, \mathsf{S}, i, \mathsf{R}$):
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$, execute the query normally.
  - Otherwise, substitute the call to $\mathsf{F\text{-}Rcv}$ with a **dlv-AM**($\mathsf{S}, i, \mathsf{R}$) call to the FS-GAEAD game.

- **chall**(S, a, m_0, m_1)
  - If $\mathsf{ep}_{\mathsf{cur}} \neq \mathsf{ep}^*$ or $\mathsf{S} \neq \mathsf{ID}^*$ or $i[\mathsf{S}] \neq i^*$ or the last honest commit is not in $\mathsf{ep}^*_{\mathsf{h}}$, abort ($\mathcal{A}$ loses).
  - Otherwise, send a **chall** query to the FS-GAEAD security game with input $(\mathsf{S}, a, m_0, m_1)$ and use the returned ciphertext to simulate the rest of **chall** in SGM.

- **red-inj-AM**(a', e', R): execute the query normally.

- **corr**(ID): If ID's state contains information that enables processing of the i^* message sent by ID^* in epoch ep^*, abort ($\mathcal{A}$ loses). Otherwise send a **corr**(ID) query to the FS-GAEAD game, incorporate the FS-GAEAD state to the SGM state of ID, execute **corr** of the SGM game and return the output to $\mathcal{A}$.

- **no-del**(ID): execute normally. If $\mathsf{ep}_{\mathsf{cur}} = \mathsf{ep}^*$, send a **no-del**($\mathsf{ID}$) query to the FS-GAEAD game challenger.

$\mathcal{B}_4$ outputs the bit output by $\mathcal{A}$.

By inspection we see that the above simulation is correct and $\mathcal{B}_4$ manages to simulate H_3 for $\mathcal{A}$ using the oracles of the FS-GAEAD security game. Whenever $\mathcal{B}_4$ needs to execute Init, if the current epoch is not ep^*, it executes the code locally, otherwise it sends a **init** query to the FS-GAEAD security game. In addition, whenever it needs to send, or deliver, a message, if the message is for the target epoch ep^* it users the oracles of the FS-GAEAD game, otherwise it executes the operation based on local information. Privacy requires that either the FS-GAEAD state in epoch ep^* is secure, of if it is compromised, the compromised party should have already received the message for the target sender ID^* and message index i^*. As we argued when proving authenticity of our SGM scheme, by a reduction to FS-GAEAD authenticity, our safety predicate enforces that behavior, therefore, a valid challenge in H_3 implies a valid challenge against the FS-GAEAD target session. If the secret bit sampled by the FS-GAEAD security game is 0 (resp. 1), $\mathcal{B}_4$ simulates for $\mathcal{A}$ H_3 with secret bit 0 (resp. 1). Therefore, the winning advantage of $\mathcal{A}$ in H_3 is bounded by the winning advantage of $\mathcal{B}_4$ against the FS-GAEAD security game, and the proof is concluded. $\square$ $\square$

From the above lemmas we derive that

$$\mathrm{Adv}^{\Gamma}_{\mathsf{SGM\text{-}Priv},\Pi_{\mathsf{SGM}}}(\mathcal{A}) \leq \varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{CR}} + \varepsilon_{\mathsf{PKE}} + \varepsilon_{\mathsf{PP}} + \varepsilon_{\mathsf{FS}} \ .$$

H.8 Proof of Correctness

In this subsection we prove the following Lemma.

Lemma 44. *The scheme* Γ *is* $(t', 2\varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{FS}})$*-correct w.r.t.* Π_{SGM}.

Proof. Recall that, compared to the SGM game, the SGM-Corr correctness game has no **chall** oracle nor inject oracle for welcome, proposal or commit messages. The only injections permitted are application messages but using the **red-inj-AM**, defined in Figure 30. In particular, the view of $\mathcal{A}$ is independent of bit b and the only way to win SGM-Corr is to trigger the **win** instruction in one of the delivery oracles for a proposal, commit, welcome or application message. We address each of these cases in turn.

Claim 45. *Triggering* **win** *in* **dlv-PM** *is impossible.*

Proof. Assuming that the signature scheme S and the MAC scheme M, are perfectly correctly, i.e., honestly generated signatures and tags, respectively, verify correctly, to trigger **win** in **dlv-PM** it must be that Props.checkPI(pid, PI) = false. However, this never happens. The SGM-Corr game ensures that the proposal messages passed to Proc-PM are the same ones created by Γ when proposal pid was created. Moreover, the mapping from properties of a proposal to the attributes (op, orig, data) implemented by Props.new is the same as the one computed by *get-propInfo. Thus for any pid passed as an argument to **dlv-PM** the output PI of *get-propInfo will always cause checkPI(pid, PI) to be true. □ □

Claim 46. $\Pr[$**win** *in* **dlv-CM** *is triggered*$] \leq \varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{CR}}$.

Proof. To Trigger the **win** in **dlv-CM** requires HG.checkGI(vid, GI) = false. Here too, SGM-Corr ensures that the same commit message is delivered to K-Proc-Com as was generated when vid was created and its attributes set, unless there is a collision against H, in which case the receiver maps the received hash **h** to a different set of proposals from the one it has been already computed over. In that case we have a reduction to the collision resistance property of H that simply simulates the execution and outputs as a collision the pair of proposals. Assuming the probability of that event to be negligible, for checkGI to return false would require breaking the correctness of the underlying CGKA construction K. That is we can use $\mathcal{A}$ to trigger the **win** condition in the **process** oracle of CGKA.

A bit more formally, we can build a reduction to CGKA(K) by simulating the rest of construction Γ and the SGM-Corr security game to $\mathcal{A}$. We use the CGKA(K) oracles to interact with the CGKA component in Γ. For example, to simulate oracle **gen-new-KB** of SGM-Corr we need to simulate a call to algorithm Gen-KB(s[ID], spk) of Γ. For this we run Γ as designed except we replace the call to K-Gen-IK with a call to **gen-new-ik**(ID). The same idea works when simulating the oracle **get-KB** using **reg-ik** as well as the oracles for sending and delivering proposals, commit messages and welcome messages and a **no-del** corruption. The CGKA is not needed to send or deliver application messages. Finally, to simulate a **corr** in SGM-Corr we corrupt the same party id CGKA to obtain the CGKA component of their local state in SGM-Corr.

It remains only to observe that if a call to **dlv-CM** in $\mathsf{SGM\text{-}Corr}$ would trigger **win** then the condition in to trigger **win** in **process** are also met. (It is actually, if anything, even easier to trigger the later.) In the above we assume perfect correctness of M and S. □ □

An essentially identical argument mapping a trigger of **win** in **dlv-WM** to triggering **win** in **dlv-WM** results in the follow claim.

Claim 47. $\Pr[$**win** *in* **dlv-WM** *is triggered*$] \leq \varepsilon_{\mathsf{CGKA}}$.

Finally, an analogous reduction from triggering the **win** in **dlv-AM** in the $\mathsf{SGM\text{-}Corr}$ game to triggering **win** in the **dlv-AM** in the $\mathsf{FS\text{-}GAEAD}$ game implies the following claim.

Claim 48. $\Pr[$**win** *in* **dlv-AM** *is triggered*$] \leq \varepsilon_{\mathsf{FS}}$.

To conclude the proof of the lemma we consider 5 hybrid games. The first is simply $\mathsf{SGM\text{-}Corr}$ and in each of the next we replace one more of the **win** instructions with an instruction that does nothing letting the game continue. In particular, the final game has no **win** instruction and is independent of the bit b so the an advarsary's advantage is always 0. Thus the above 4 claims imply that the advantage of any t-time adversary can be at most $2\varepsilon_{\mathsf{CGKA}} + \varepsilon_{\mathsf{FS}} + \varepsilon_{\mathsf{CR}}$, assuming perfect correctness of M, S and PKE. □ □

Putting things together. By Theorems 25,30,38, and Lemma 44, we derive the bounds of Theorem 4.