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

Joël Alwen
AWS Wickr
alwenjo@amazon.com

Sandro Coretti
IOHK
sandro.coretti@iohk.io

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

Yiannis Tselekounis
University of Edinburgh
y.tselekounis@ed.ac.uk

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.

CCS CONCEPTS

• **Security and privacy** → *Cryptography*.

KEYWORDS

Cryptographic protocols; Messaging Layer Security; MLS; TreeKEM; RTreeKEM; Secure Messaging; Forward Secrecy; Backward Secrecy

ACM Reference Format:
Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2021. Modular Design of Secure Group Messaging Protocols and the Security of MLS. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3460120.3484820

ACKNOWLEDGMENTS

This research is partially supported by gifts from VMware Labs and Google, NSF grants 1619158, 1319051, 1314568, and the "Quantum Computing for Modern Cryptography" project, funded by the UK Quantum Computing and Simulation Hub from the UKRI EPSRC grant EP/T001062/1.

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 [8]. 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

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484820

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, 6]. First introduced by Cohn-Gordon et al. [13] (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, 5], others for a more flexible communication model [10, 23], 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 [7, 26]. Although, it has undergone at least 2 major iterations before reaching its current version [8] 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, [6] 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 [6].

Two exceptions to the focus on CGKA are: [15], which focuses exclusively on how to optimize the bandwidth required to updated cryptographic state concurrent sessions with overlapping membership sets, and [11], 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* (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.

[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, metadata hiding and so called "external commits" where an external party makes changes to the group state.

[2] We note that, concurrently to our work, [9] introduced a security notion for SGM defined using formal verification tools which we discuss in the Related Work section below.

PKI. Another contribution to the analysis of MLS is a more accurate modeling of the PKI used by MLS. With the notable exceptions of [6, 15]), 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) [8] 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 [13] 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 [20]. Indeed, recent results in [17] 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) [5, 6].[3] Symbolic representations of an execution, equipped with safety predicates, have been used in [2, 14]. The GSD proof technique, introduced by Panjwani [25] and generalized by Jafargholi et al. [19], 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.* [13], focusing on adaptive security (informally sketched) for static groups. It inspired the initial version of TreeKEM [26] which was updated to provide secure group management in [7] and finally to its current form in [8]. Meanwhile, ART uses an older technique called "Tree-based DH groups" [27, 28, 30] which is also used by [21] to build key agreement. However, TreeKEM and ART differ significantly from [21], as discussed in [3]. TreeKEM is also related to schemes for (symmetric-key) broadcast encryption [16, 18] and multicast encryption [12, 24, 30]. Cremers *et al.* [15] note MLS/TreeKEM's disadvantages w.r.t. PCS for multiple groups, and Weider [29] suggests Causal TreeKEM, a variant that requires less ordering of protocol messages. Finally, in concurrent work appearing recently, [9] 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 FORMAL DEFINITION OF SECURE GROUP MESSAGING

In this section we formally define *secure group messaging* (SGM). Specifically, the formal syntax of SGM schemes is introduced in Section 3.1, and Section 3.2 introduces the security game for SGM schemes.

[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.

3.1 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 ok = false.

  Key bundles have the format $\mathsf{kb} = (\mathsf{wpk}, \mathsf{spk}, \mathsf{sig})$, 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.

3.2 Security

3.2.1 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.[4] 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 **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 V-Pt[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.

[4] 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.

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 WK-ID[·] and SK-ID[·] map wkid and skid values to the ID of the party that owns the corresponding secret keys; the arrays WK-PK[·] and SK-PK[·] map wkid and skid values to the corresponding public keys. Moreover, the array WK-SK[·] 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 WK-SK[wkid] ← skid.

The array CL-KB[·, ·] keeps track of the *contact list* for each party. Specifically, CL-KB[ID, ID′] is a *queue* of pairs (wkid, 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:* V-St[ID] contains pairs (vid, 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.[5] P-St[ID] contains the (IDs of) the proposals currently stored by ID. WK-St[·] and SK-St[·] contain the (IDs of) the welcome and signing keys, respectively, currently stored by ID. *Leaked values:* V-Lk is a set that contains triples (vid, ID). Each such tuple means that the attacker learned the state of ID in epoch vid, that for that state flag = true, which implies that leaked information can be used to infer information about subsequent epochs. WK-Lk and SK-Lk contain the (IDs of) the welcome and signing keys, respectively, for which the attacker has learned the secret keys. AM-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 Del[ID] = true) and which ones are not. Formally, a party with Del[ID] = 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: V-Tr (for undeleted information about epochs), AM-Tr (for undeleted information about application messages), WK-Tr (for undeleted information about welcome keys), and SK-Tr (for undeleted information about signature keys).

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

[5]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.

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 Ep-ID[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 BR[·]. 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 $r = \perp$, in which case uniformly random coins (unknown to $\mathcal{A}$) are used.

```
// General
b ← {0,1}
idCtr++
∀ID : s[ID] ← Init(ID)
// Communication
CM[·,·] ← ϵ
WM[·,·] ← ϵ
PM[·,·] ← ϵ
AM[·,·,·,·] ← ϵ
// History Graph
vid_root ← HG.init
V-Pt[·] ← vid_root
V-St[·] ← [(vid_root, false)]
V-Tr[·] ← ∅
V-Lk ← ∅
P-St[·] ← ∅
Ep-ID[·] ← ϵ
// App. Messages
AM-Tr[·] ← ∅
AM-Lk ← ∅
// Miscellaneous
Chall[·] ← ∅
Del[·] ← true
BR[·] ← false
// PKI
CL-KB[·,·] ← ϵ
WK-SK[·] ← ϵ
WK-ID[·] ← ϵ
WK-PK[·] ← ϵ
WK-St[·] ← ∅
WK-Tr ← ∅
WK-Lk ← ∅
SK-ID[·] ← ϵ
SK-PK[·] ← ϵ
SK-St[·] ← ∅
SK-Tr ← ∅
SK-Lk ← ∅
```

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

3.2.2 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 $s[\cdot]$.[6]

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_{root} = (.vid ← idCtr++, .orig ← ⊥, .data ← ⊥, .**pid** ← ⊥) and return $\text{vid}_{\text{root}} = v_{\text{root}}.\text{vid}$.

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

3.2.3 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 Section D of the Appendix. 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.

```
// 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)
```

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 Section D.

3.2.4 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.[7] 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.
  The oracle runs Gen-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 WK-ID, WK-PK, and WK-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 kb = (wpk, spk, 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., spk = SK-PK[skid] for some skid with SK-ID[skid] = 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 Get-KB is run on ID′ and kb. If the secret key corresponding to spk has not been leaked, i.e., skid ∉ SK-Lk, then Get-KB must only accept the key bundle if wpk exists and is bound to spk, i.e., WK-SK[wkid] = skid. If Get-KB outputs ok, skid ∉ SK-Lk, but WK-SK[wkid] ≠ 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., wkid = ⊥), the game generates a new wkid for wpk, updates WK-PK accordingly, and immediately marks wkid as leaked. Finally, ID's (symbolic) contact list is updated, by adding the pair (wkid, skid) to the end of queue CL-KB[ID, ID′].

3.2.5 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 (cf. Section D in the Appendix.).

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 HG.create(ID, skid) method. This causes HG to create a node v = (.vid ← idCtr++, .orig ← ID, .data ← skid, .**pid** ← ⊥) as a child of v_{root} and return vid = v.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 V-St[ID] is set to a queue containing

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

only (vid, 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.

```
// 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 ≠ ⊥
    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', ⊥)
    (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', ⊥)
    (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, ⊥, 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]
    ID_O ← PI.orig
    req ¬(*auth-compr(vid)
          ∧ *SK-compr(vid, ID_O))
    if PI ≠ ⊥
        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 Section D of the Appendix.

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.

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 = (.pid ← idCtr++, .vid ← V-Pt[ID], .op ← op, .orig ← ID, .data ← data), where op ∈ {add, rem, 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.[8] 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 ID_O of the proposal (as determined by the output of Proc-PM) is compromised (***SK-compr**(vid, ID_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) **pid**, the attacker calls the **commit** oracle. The specified **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 **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 **W**, and a commit message T being output. Bookkeeping calls HG.commit(ID, **pid**), which creates a new HG node

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

v = (.vid ← idCtr++, .orig ← ID, .data ← ⊥, .**pid** ← **pid**), as a child of ID's current epoch V-Pt[ID] and returns vid = v.vid. Then, the bad-randomness array is filled (depending on whether $\mathcal{A}$ specified coins r or not), and E is stored in array Ep-ID. Finally, the commit messages (for current group members) are stored in array CM, and the welcome messages (for new group members) in WM.

```
// ID commits proposals pid
commit(ID, pid, r)
    req *compat-commit(ID, pid)
    (s[ID], E, W, T) ← Commit(s[ID], PM[pid]; r)

    vid ← HG.commit(ID, pid)
    BR[vid] ← r ≠ ⊥
    Ep-ID[vid] ← E
    for ID′ ∈ HG.roster(V-Pt[ID])
      | CM[ID′, vid] ← T
    for ID′ ∈ Props.addedIDs(pid)
      | WM[ID′, vid] ← W[ID′]
    return vid
// Control msg. delivered to ID
dlv-CM(ID, vid)
    req *compat-dlv-CM(ID, vid)
    T ← CM[ID, vid]
    (s[ID], GI) ← Proc-CM(s[ID], T)
    if ¬HG.checkGI(vid, GI)
      | win
    if HG.isRemoved(ID, vid)
      | V-Pt[ID] ← vid_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′ injected to ID
inj-CM(ID, T′)
    req *compat-inj-CM(ID, T′)
    (s[ID], GI) ← Proc-CM(s[ID], T′)
    vid ← V-Pt[ID]
    ID_O ← GI.orig
    req ¬(*auth-compr(vid)
        ∧ *SK-compr(vid, ID_O))
    if GI ≠ ⊥
      | win
// Welcome msg. delivered to ID
dlv-WM(ID, vid)
    req *compat-dlv-WM(ID, vid)
    W ← WM[ID, vid]
    (s[ID], GI) ← Proc-WM(s[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[ID], GI) ← Proc-WM(s[ID], W′)
    E ← GI.epID
    req ∃ vid : Ep-ID[vid] = E
    ID_O ← GI.orig
    req ¬(*auth-compr(vid)
        ∧ *SK-compr(vid, ID_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 Section D of the Appendix.

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 V-Pt[ID], provided all proposals that lead to vid have been delivered to ID. Oracle **dlv-CM** runs SGM algorithm Proc-CM on T, which results in group information GI being output. This information is checked by function HG.checkGI. Function 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 HG.isRemoved), its pointer V-Pt[ID] is set to vid_{root}; otherwise, V-Pt[ID] is set to the new epoch vid. Furthermore, the index counter i[ID] is reset to 0, and P-St[ID], the set of proposals stored, is set to the empty set. If ID deletes old values, i.e., if Del[ID] = true, the game changes the flag in (·, flag) = V-St[ID].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 V-St[ID] is put into V-Tr[ID] (because it is about to be removed from the queue). Finally, the new epoch vid is added to the end of queue V-St[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 vid := V-Pt[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.

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 (S, i[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 (vid, S, i, 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', S', i', m'). These values are checked, and if R misidentifies any of them, the attacker immediately wins the security game. Subsequently, AM[vid, S, i, 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 AM-Tr[R]. Note that the oracle does not return any values since the attacker knows which values are output by Rcv.

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:

```
// S sends message m with AD a
send(S, a, m)
    req *compat-send(S)
    (s[S], e) ← Send(s[S], a, m)
    i[S]++
    for R ∈ HG.roster(V-Pt[S]) \ {S}
        AM[V-Pt[S], S, i[S], R] ←
          (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) ← AM[S, vid, i, R]
    (s[R], E', S', i', m') ← Rcv(s[R], a, e)

    if (E', S', i', m') ≠
      (Ep-ID[vid], S, i, m)
        win
    if ¬Del[R]
        AM-Tr[R] ← (vid, S, i)
    AM[vid, S, i, R] ← received
// Attacker leaks state of ID
corr(ID)
    for (vid, flag) ∈ V-St[ID] ∪ V-Tr[ID]
        if flag
            V-Lk +← (vid, ID)
        AM-Rcvd ←
          {(S, i) | AM[vid, S, i, ID] =
          received}
        AM-Lk +←
          (vid, AM-Rcvd, AM-Tr[ID])
    W ← WK-St[ID] ∪ WK-Tr[ID]
    WK-Lk +← {wkid ∈ W |
      WK-ID[wkid] = ID}
    S ← SK-St[ID] ∪ SK-Tr[ID]
    SK-Lk +← {skid ∈ S |
      SK-ID[skid] = ID}
    HG.corrHanging(ID)
    return s[ID]
```

```
// Disable deletions for ID
no-del(ID)
    disable deletions for ID
    Del[ID] ← false
// Safety for privacy
*priv-safe
    return ∀vid, S, i : ((S, i) ∈
      Chall[vid] ⟹ *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) ← Send(s[S], a, m_b)
    i[S]++
    for R ∈ HG.roster(V-Pt[S]) \ {S}
        AM[V-Pt[S], S, i[S], R] ←
          (a, m, e)
    Chall[V-Pt[S]] +← (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') ← Rcv(s[R], a', e')

    if m' ≠ ⊥
        let vid' : Ep-ID[vid'] = E'
        if ¬(¬*AM-sec(vid', S', i')
          ∧ *SK-compr(vid', S'))
            ∨ AM[vid', S', i', R] =
          received
            win
        if ¬Del[vid]
            AM-Tr[R] ← (vid', S', i')
        AM[vid', S', i', R] ←
          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 Section D of the Appendix.

- *Epochs:* All epochs ID currently stores information for are kept track of by V-St[ID]; recall that for (vid, flag) ∈ 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, ·) ∈ 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 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] ← true resp. BR[vid] ← 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).

3.2.6 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.

3.2.7 Advantage. Let Π = {***AM-sec**, ***auth-compr**, ***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 $\mathrm{Adv}^{\Gamma}_{\mathrm{SGM},\Pi}(\mathcal{A})$.

Definition 3.1. An SGM scheme Γ is (t, q, ε)-*secure w.r.t. predicates* Π, if for all (t, q)-attackers,

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

4 MODULARIZING MLS AND PROVING ITS SECURITY

This section provides a summary of the MLS protocol, our modularization of it, as well as the security proofs for the full protocol—based only on the security of the *generic* components—and for the components themselves (based on specific common cryptographic assumptions).

Our modularization splits MLS into the following three component primitives: *continuous group key-agreement (CGKA), forward-secure group AEAD (FS-GAEAD)*, and *PRF-PRNGs*. FS-GAEAD corresponds to a single epoch of secure group messaging, the PRF-PRNG can be thought of as an entropy pool from which key material for FS-GAEAD is extracted, and the entropy pool itself is continually refreshed by values from CGKA.

4.1 Continuous Group Key Agreement

CGKA schemes have a structure that is very close to that of an SGM protocol: they proceed in epochs using the propose-and-commit paradigm and provide algorithms for group creation/joining, interaction with the PKI, issuing proposals, and creating/processing commits. However, instead of being used to send/receive messages, in each epoch, CGKA scheme outputs a high-entropy *update secret*. The full CGKA syntax can be found in Section A of the Appendix.

The part of MLS responsible for CGKA is called TreeKEM; an improved version called RTreeKEM is also known and considered in this work, despite not currently being part of the MLS standard.

Security definition. 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, both update proposals and commits must contribute to post-compromise security (PCS). CGKA schemes must also be able to deal with bad randomness 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 higher-level protocol using the CGKA scheme; it also greatly simplifies the security analysis.

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.

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 Section A of the Appendix.

Instantiation: (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 a commit operation 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.

Security results. 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: (1) 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; (2) no information about vid is known to the attacker as the result of *splitting* the group (cf. full version [4]); (3) good randomness was used by the commiter creating vid. More details can be found in the full version [4].

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 for standard public-key encryption to the IND-CPA security of the underlying scheme, and then we reduce the security of TreeKEM to that of GSD.

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, which allows to design completely deterministic schemes (apart from the initial key).

Security definition. 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 Section B of the Appendix and the full version [4].

Instantiation and security results. 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 as follows: messages and AD are encrypted/authenticated with a normal AEAD

scheme, where the key for the i^{th} message by party ID is derived using the label (ID, i).

The safety predicate ***FS-sec** achieved by our construction considers a message secure if no party's state is leaked before it receives the message. For more information see Section B of the Appendix and the full version [4].

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 pairs (I, C) *on the same state* are indistinguishable from random and independent values.

Security definition. The intuitive security properties for PRF-PRNGs mentioned above must also hold in the presence of state compromise. In particular, a PRF-PRNG must satisfy PCFS (cf. Section 3.2) and deal with splitting (cf. Section C of the Appendix and the full version [4]). 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 Section C of the Appendix.

Instantiation and security results. The MLS protocol uses HKDF [22] as PRF-PRNG. This work models HKDF as a random oracle and shows that it achieves achieves security w.r.t. safety predicate ***PP-secure**, which captures that in order for a value R to be considered secure in a particular epoch e, (1) e must have an ancestor e' (possibly itself) that was reached via a random input I not known to $\mathcal{A}$, and (2) there must have been no corruptions on the path from e' to e's parent. More details can be found in the full version [4].

4.4 Plugging things together

The protocol is based on the following primitives: A CGKA scheme K = (K-Gen-IK, K-Create, K-Add, K-Remove, K-Update, K-Commit, K-Proc-Com, K-Join), an FS-GAEAD scheme F = (F-Init, F-Send, F-Rcv), a CPA-secure public key encryption scheme PKE = (E-KeyGen, E-Enc, E-Dec), an existentially unforgeable signature scheme S = (S-KeyGen, S-Sign, S-Ver), a message authentication scheme M = (M-Tag, M-Ver), a PRF-PRNG PP, a collision resistant hash function H. The initialization and PKI algorithms of our construction are depicted on Figure 6. The helper functions are formally presented in the full version. Below we describe the main SGM algorithms depicted on Figure 7, 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 (s.G ← [ME]), executes the CGKA group creation operation, absorbs the output I into the PRF-PRNG PP. PP outputs the new PRF-RPNG state s.σ, the FS-GAEAD key k_e, a MAC key s.$\mathsf{k_m}$ (used to authenticate control messages), and the current epoch id, s.C-epid. 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 $\bar{\mathsf{P}}$, 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. Section E of the Appendix).

Committing. To commit to a set **P** of proposals, Commit calls the CGKA commit operation to obtain CGKA welcome messages W_{pub} and **WPrv** as well as CGKA control message $\bar{\mathsf{T}}$ and update secret I. $\bar{\mathsf{T}}$ 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 T′ = ("com", epid, ID, **h**, $\bar{\mathsf{T}}$). However, in order to obtain the MAC key, (1) the CGKA control message $\bar{\mathsf{T}}$ 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 Proc-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 s.C-epid of the current epoch. Receiving works analogously, with the caveat that the appropriate FS-GAEAD session must be used.

4.4.1 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*. In the full version of the paper [4] we prove that these properties together imply full SGM security. Besides modularity, simplified properties facilitate the transition from *selective* to *fully adaptive* security, as the individual "simplified" games, defined in the full version [4], 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 4.1), 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 8 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 8).

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**(vid, ID)), (2) injects a (proposal, welcome, commit) message when ***auth-compr**(vid) holds, which implies that ***PP-secure**(vid, ***Proj-PP**(SGM-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 ultimately prove the following theorem:

Theorem 4.1. *Let* SGM = SGM(K, F, PKE, S, M, PP, 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 a CGKA scheme with respect to predicate* Π_{CGKA}*, (2)* F *is an FS-GAEAD scheme with respect to predicate* Π_{FS}*, (3)* PKE *is a CPA secure public-key encryption scheme, (4)* S *is an existentially unforgeable signature scheme, (5)* M *is a message authentication code, (6)* PP *is a PRF-PRNG with respect to predicate* Π_{PP}*, (7)* H *is a collision resistant hash. Then,* SGM *is an SGM scheme with respect to predicate* Π_{SGM}*.*

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

```
// Generate Signature Keys
Gen-SK
    (spk, ssk) ← S-KeyGen
    SK-sk[spk] ← ssk
    return spk
// Delete Signing Key
Rem-SK(spk)
    SK-sk[spk] ← ε
// 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 6: The SGM Construction : Initialization and PKI Algorithms.

```
// Create a group
Create(spk, wpk; r)
    s.G ← [ME]
    wsk ← WK-wk[wpk]
    (·, isk) ← wsk
    (s.γ, I) ← K-Create(ME, isk; r)
    (s.σ, k_e, s.k_m, s.C-epid) ← PP(0, I, 0)
    s.Ep-SPK[s.C-epid, ME] ← spk
    s.υ[s.C-epid] ← F-Init(k_e, 1, ME)
// Add proposal
Add(ID_a)
    kb' ← CL-KB[ID_a]
    ((epk, ipk), spk) ← kb'
    (s.γ, P̄) ← K-Add(s.γ, ID_a, ipk)
    P' ← ("add", s.C-epid, ME, (ID_a, kb'), P̄)
    t ← M-Tag(s.k_m, P')
    sig ← S-Sign(s.C-ssk, (P', t))
    return (P', t, sig)
// Remove proposal
Remove(ID_r)
    (s.γ, P̄) ← K-Remove(s.γ, ID_r)
    P' ← ("rem", s.C-epid, ME, ID_r, P̄)
    t ← M-Tag(s.k_m, P')
    sig ← S-Sign(s.C-ssk, (P', t))
    return (P', t, sig)
// Update proposal
Update(spk; r)
    req SK-sk[spk] ≠ ε
    (s.γ, P̄) ← K-Update(s.γ; r)
    P' ← ("upd", s.C-epid, ME, spk, P̄)
    t ← M-Tag(s.k_m, P')
    sig ← S-Sign(s.C-ssk, (P', t))
    return (P', t, sig)
// Receive A Message
Rcv(a, e)
    (e', sig) ← e
    (epid, a, ē) ← e'
    (s.υ[epid], ID_S, i, m) ← F-Rcv(s.υ[epid], (epid, a), ē)

    req ID_S ≠ ⊥
    spk ← s.Ep-SPK[epid, ID_S]
    req S-Ver(spk, e', sig)
    return (epid, ID_S, i, m)
// Commit
Commit(P; r)
    req P ⊆ s.Props
    // Get CGKA Proposals
    P̄ ← P.P̄
    //  – Prepare Commit Message –
    (s.γ, W_pub, WPrv, T̄, I) ← K-Commit(s.γ, P̄; r)

    T' ← ("com", s.C-epid, ME, H(P), T̄)
    v ← H(T')
    // New MAC Key & Epoch ID
    (·, ·, k_m, epid) ← PP(s.σ, I, v)
    t ← M-Tag(k_m, T')
    sig ← S-Sign(s.C-ssk, (T', t))
    // Commit Message
    T ← (T', t, sig)
    //  – Prepare Welcome Messages –
    (ID, wpk) ← *added(P)
    spk ←
      *new-spks(s.Ep-SPK[s.C-epid, ·], P)
    for i ∈ ID
        e ← E-Enc(wpk[i].epk, s.σ)
        W' ←
          ("wel", ME, ID[i], v, W_pub, ...
        ..., WPrv[i], e, wpk[i], spk)
        t ← M-Tag(k_m, (W', epid))
        sig ← S-Sign(spk[ME], (W', t))
        // Welcome Message
        W[i] ← (W', t, sig)
    return (epid, W, T)
```

```
// Process a proposal
Proc-PM(P)
    (P', t, sig) ← P
    (., epid, ID, ., .) ← P'
    req epid = s.C-epid
    spk ← s.Ep-SPK[epid, ID]
    req M-Ver(s.k_m, P', t)
    req S-Ver(spk, (P', t), sig)
    s.Props +← P'
    return *get-propInfo(P')
// Send A Message
Send(a, m)
    E ← s.C-epid
    (s.υ[E], ē) ← F-Send(s.υ[E], (E, a), m)
    e' ← (E, a, ē)
    sig ← S-Sign(s.C-ssk, e')
    return (e', sig)
// Process Commit Message
Proc-CM(T)
    (T', t, sig) ← T
    (., epid, ID, h, T̄) ← T'
    // Matching Epochs?
    req epid = s.C-epid
    spk ← s.Ep-SPK[epid, ID]
    req S-Ver(spk, (T', t), sig)
    P ← s.Props[h]
    // Call CGKA
    (s.γ, GI, I) ← K-Proc-Com(s.γ, T̄)
    (s.σ, k_e, s.k_m, s.C-epid) ← PP(s.σ, I, H(T'))

    req M-Ver(s.k_m, T', t)
    s.G ← GI.G
    pos ← *roster-pos(ME, s.G)
    // Start FS-GAEAD
    s.υ[s.C-epid] ← F-Init(k_e, |s.G|, pos)
    s.Ep-SPK[s.C-epid, .] ←
      *new-spks(s.Ep-SPK[epid, ·], P)
    spk ← s.Ep-SPK[s.C-epid, ME]
    s.C-ssk ← SK-sk[spk]
    return GI
// Join a Group
Proc-WM(W)
    (W', t, sig) ← W
    (., ID_S, ., v, W_pub, W_priv, e, wpk, spk) ←
      W'
    spk ← spk[ID_S]
    req
      spk ∈ CL-S[ID_S] ∧ S-Ver(spk, W', sig)
    (esk, isk) ← WK-wk[wpk]
    s.σ ← E-Dec(esk, e)
    (s.γ, GI, I) ← K-Join(ME, ID_S, W_pub, W_priv, isk)

    (s.σ, k_e, s.k_m, s.C-epid) ← PP(s.σ, I, v)
    req M-Ver(s.k_m, (W', s.C-epid), t)
    s.G ← GI.G
    pos ← *roster-pos(ME, s.G)
    s.υ[s.C-epid] ← F-Init(k_e, |s.G|, pos)
    s.Ep-SPK[s.C-epid, .] ← spk
    spk ← s.Ep-SPK[s.C-epid, ME]
    s.C-ssk ← SK-sk[spk]
    s.Props ← ε
```

Figure 7: The SGM Construction: main algorithms.

```
// Determines if authenticity of epoch vid compromised
*auth-compr(vid)
    return ¬*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) ∩ SK-Lk = ∅
// 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 = (𝒱, 𝒫, V-Lk, AM-Lk, BR, WK-Lk))
    for (vid, orig, data, pid) ∈ 𝒱
        𝒱' +← vid
    for vid ∈ 𝒱'
        BI[vid] ← ¬*CGKA-priv(vid', *Proj-CGKA(SGM-Data))
    return (𝒱', V-Lk, BI)
// Project History Graph to CGKA Game
*Proj-CGKA (SGM-Data = (𝒱, 𝒫, V-Lk, AM-Lk, BR, WK-Lk))
    for (vid, orig, data, pid) ∈ 𝒱
        𝒱' +← (vid, orig, pid)
    for p = (pid, vid, orig, op, data) ∈ 𝒱
        if op = add
            let data = (ID', wkid', skid')
            data' ← (ID', wkid')
            p' ← (pid, vid, op, orig, data')
        if op = rem
            p' ← p
        if op = upd
            p' ← ⊥
        𝒫' +← p'
    return (𝒱', 𝒫', V-Lk, BR, WK-Lk)
// Project History Graph to FS-GAEAD Game
*Proj-FS (SGM-Data = (𝒱, 𝒫, V-Lk, AM-Lk, BR, WK-Lk), vid)
    for (vid, ID, AM-Rcvd[ID], AM-Tr[ID]) ∈ AM-Lk
        AM-Lk' +← (ID, AM-Rcvd[ID], AM-Tr[ID])
    return AM-Lk'
```

Figure 8: Safety oracles and safety predicate of the security game for SGM schemes.

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, Y. Dodis, and Y. Tselekounis. Modular design of secure group messaging protocols and the security of mls. Cryptology ePrint Archive, Report 2021/1083, 2021. https://ia.cr/2021/1083.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, Aug. 2010.

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

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

[25] 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.

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

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

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

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

[30] 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 CONTINUOUS GROUP KEY AGREEMENT

A.1 Syntax

This section introduces the formal syntax of a CGKA scheme. Parties are identified by unique party IDs ID chosen from an arbitrary fixed set. In the following, γ and γ' denote the internal state of the CGKA scheme before and after an operation, respectively. A CGKA scheme CGKA consists of eight algorithms:

- For PKI:
  - $(\mathsf{ipk}, \mathsf{isk}) \leftarrow$ Gen-IK: generates and outputs a new initial key pair.
- For group creation:
  - $\gamma' \leftarrow \mathsf{Create}(\mathsf{ID}, \mathsf{isk})$: initializes the state and creates group with self, using as init secret key isk.
- For proposals:
  - $(\gamma', P) \leftarrow \mathsf{Add}(\gamma, \mathsf{ID}', \mathsf{ipk}')$: generates a proposal to add party ID' using initial public key ipk';
  - $(\gamma', P) \leftarrow \mathsf{Remove}(\gamma, \mathsf{ID}')$: generates a proposal to remove party ID';
  - $(\gamma', P) \leftarrow \mathsf{Update}(\gamma)$: generates a proposal for self to update.
- For commits:
  - $(\gamma', I, W_{\mathsf{pub}}, \mathbf{W}_{\mathsf{priv}}, T) \leftarrow \mathsf{Commit}(\gamma, \mathbf{P})$: creates a commit corresponding to a vector $\mathbf{P}$ of proposals and outputs the resulting *commit secret* I, a *public* welcome message W_{pub} as well as a vector $\mathbf{W}_{\mathsf{priv}}$ of *private* welcome messages (for newly added parties), and a control message T (for existing group members);
  - $(\gamma', \mathsf{GI}, I) \leftarrow \mathsf{Proc\text{-}Com}(\gamma, T, \mathbf{P})$: used by existing group members to process control message T w.r.t. vector of proposals $\mathbf{P}$, and reach new epoch; outputs updated group information GI (where $\mathsf{GI} = \perp$ if T is considered invalid) and a commit secret I;
  - $(\gamma, \mathsf{GI}, I) \leftarrow \mathsf{Join}(\gamma, \mathsf{orig}, W_{\mathsf{pub}}, W_{\mathsf{priv}}, \mathsf{isk})$: used by newly added group members to process welcome message W_{pub} and W_{priv}, generated via a commit operation by orig, and join a group using initial secret key isk; outputs group information GI (where $\mathsf{GI} = \perp$ if W_{pub} or W_{priv} are considered invalid) and a commit secret I.

A.2 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.

A.2.1 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,
- **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 V-Pt[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 (IK-PK) and secret init key (IK-SK). Moreover, IK-St records the IKs stored by each party, and IK-Tr and IK-Lk keep track of "trash" IKs (old secret init keys not deleted) and leaked IKs.

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.

A.2.2 Initialization. The initialization of the CGKA game proceeds along very similar lines to that of the SGM game. It is depicted in Figure 9.

// General
$b \leftarrow \{0,1\}$
idCtr++
$\forall$ID : γ[ID] $\leftarrow$ Init(ID)
// Communication
CM[$\cdot,\cdot$] $\leftarrow \epsilon$
PM[$\cdot,\cdot$] $\leftarrow \epsilon$
$\text{WM}_{\text{pub}}[\cdot,\cdot] \leftarrow \epsilon$
$\text{WM}_{\text{priv}}[\cdot,\cdot] \leftarrow \epsilon$
WM-Lk $\leftarrow \emptyset$
// History Graph
$\text{vid}_{\text{root}} \leftarrow$ HG.init
V-Pt[$\cdot$] $\leftarrow \text{vid}_{\text{root}}$
V-Tr[$\cdot$] $\leftarrow \emptyset$
V-Lk $\leftarrow \emptyset$
P-St[$\cdot$] $\leftarrow \emptyset$
// Miscellaneous
Key[$\cdot$] $\leftarrow \emptyset$
Reveal[$\cdot$] $\leftarrow \emptyset$
Chall[$\cdot$] $\leftarrow \emptyset$
Del[$\cdot$] $\leftarrow$ true
BR[$\cdot$] $\leftarrow$ false
// PKI
IK-PK[$\cdot$] $\leftarrow \epsilon$
IK-SK[$\cdot$] $\leftarrow \epsilon$
IK-St[$\cdot$] $\leftarrow \emptyset$
IK-Tr $\leftarrow \emptyset$
IK-Lk $\leftarrow \emptyset$

Figure 9: Initialization of the security game for CGKA schemes.

```
// Instruct ID to generate a new initial key
gen-new-ik(ID)
    (γ[ID], (ipk, isk)) ← Gen-IK(γ[ID])
    ikid ← idCtr++
    IK-PK[ikid] ← ipk
    IK-SK[ikid] ← isk
    IK-St[ID] +← ikid
    return (ikid, ipk)
// Allows attacker to register IKs with game
reg-ik(ipk)
    req ∄ ikid : IK-PK[ikid] = ipk
    ikid ← idCtr++
    IK-Lk +← ikid
    IK-PK[ikid] ← ipk
    return ikid
```

Figure 10: PKI-related oracles of the security game for continuous group key agreement schemes.

A.2.3 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 14. 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.

A.2.4 PKI.. The spirit of the PKI (Figure 10) 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).

A.2.5 Main oracles. The main oracles of the CGKA game are split into two figures: oracles related to (i) group creation, proposals, and commits (Figure 11), and (ii) message processing, corruption, and challenges (Figure 13). The validity of all oracle calls is checked by the corresponding compatibility functions (Figure 14).

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 HG.create(ID, skid) method. This causes HG to create a node

$$v = \begin{pmatrix} \text{.vid} & \leftarrow & \text{idCtr++} \\ \text{.orig} & \leftarrow & \text{ID} \\ \textbf{.pid} & \leftarrow & \perp \end{pmatrix}$$

as a child of v_{root} and return vid = v.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 11) 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 Props.new(op, ID, data), which records proposal data

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

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

- *(add proposals)* data = (ID′, ikid′), where ikid′ is the (ID of the) init key ID′ is to be added with;
- *(remove proposals)* data = ID′;
- *(update proposals)* data = $\perp$.

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 11) allows $\mathcal{A}$ to instruct a party ID to execute a commit operations. It works analogously to its SGM-game counterpart. Bookkeeping calls HG.commit(ID, **pid**), which creates a new HG node

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

```
// ID creates new group
create-group(ID, ikid, r)
    req *compat-create(ID, ikid)
    isk ← IK-SK[ikid]
    (γ[ID], I) ← Create(γ[ID], isk; r)
    vid ← HG.create(ID, r)
    Key[vid] ← I
    BR[vid] ← (r ≠ ⊥)
    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']
    (γ[ID], P) ← Add(γ[ID], ID', 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', ⊥)
    (γ[ID], P) ← Remove(γ[ID], 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, ⊥, ⊥)
    (γ[ID], P) ← Update(γ[ID]; r)
    pid ← Props.new(upd, ID, ⊥)
    BR[vid] ← (r ≠ ⊥)
    PM[pid] ← P
    return (pid, P)
// ID commits proposals pid
commit(ID, pid, r)
    req *compat-commit(ID, pid)
    vid ← HG.commit(ID, pid)
    (γ[ID], I, W_pub, W_priv, T) ← Commit(γ[ID], PM[pid]; r)
    Key[vid] ← I
    BR[vid] ← (r ≠ ⊥)
    for ID' ∈ HG.roster(V-Pt[ID])
        CM[vid, ID'] ← T
    (ID_1, ..., ID_m) ← Props.addedIDs(pid)
    for i = 1, ..., m
        WM_pub[vid, ID_i] ← W_pub
        WM_priv[vid, ID_i] ← W_priv[ID_i]
    return (vid, T, W_pub, W_priv)
```

Figure 11: 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 14.

```
// Process commit msg for ID and epoch vid
process(vid, ID)
    req *compat-process(vid, ID)
    T ← CM[vid, ID]
    P ← PM[HG[vid].pid]
    (γ[ID], GI, I) ← Proc-Com(γ[ID], T, P)
    if ¬HG.checkGI(vid, GI) ∨ Key[vid] ≠ I
        win
    if ¬Del[ID]
        V-Tr[ID] ← V-Pt[ID]
    if HG.isRemoved(vid, ID)
        V-Pt[ID] ← vid_root
    else
        V-Pt[ID] ← vid
    i[ID] ← 0
    P-St[ID] ← ∅
    return GI
// Reveal the update secret of epoch vid
reveal(vid)
    req Key[vid] ≠ ϵ
    req ¬(Reveal[vid] ∨ Chall[vid])
    Reveal[vid] ← true
    return Key[vid]
// Challenge the update secret of epoch vid
chall(vid)
    req Key[vid] ≠ ϵ
    req ¬(Reveal[vid] ∨ Chall[vid])
    I_0 ← Key[vid]
    I_1 ← 𝓘
    Chall[vid] ← true
    return I_b
// Welcome msg. of epoch vid delivered to ID
dlv-WM(vid, ID)
    req *compat-dlv-WM(vid, ID)
    W_pub ← WM-pub[vid, ID]
    W_priv ← WM-priv[vid, ID]
    ikid ← HG.addedIK(vid, ID)
    isk ← IK-SK[ikid]
    let v s.t. v.vid = vid
    orig ← v.orig = vid
    (γ[ID], GI, I) ← Join(ID, orig, W_pub, W_priv, isk)
    if ¬HG.checkGI(vid, GI) ∨ Key[vid] ≠ I
        win
    if ¬Del[ID]
        IK-Tr[ID] +← ikid
    IK-St[ID] −← ikid
    V-Pt[ID] ← vid
    return GI
```

Figure 12: Part one of 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 14.

```
// Corrupt ID
corr(ID)
    V-Lk +← {(vid, ID) | vid ∈ {V-Pt[ID]} ∪ V-Tr[ID]}
    IK-Lk +← IK-St[ID] ∪ IK-Tr[ID]
    HG.corrHanging(ID)
    return γ[ID]
// Enable no-delete for ID
no-del(ID)
    disable deletions for ID
    Del[ID] ← false
// Safety for privacy
*priv-safe
    return ∀vid : Chall[vid] ⟹ *CGKA-priv(vid)
```

Figure 13: Part two of 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 14.

as a child of ID's current epoch V-Pt[ID] and returns vid = v.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 13) 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 13), 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.

Corruption oracles. The oracles related to corruption in the CGKA game (Figure 13) are:

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

A.2.6 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.

A.2.7 Advantage. Let Π = ***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 $\mathrm{Adv}^{\mathsf{K}}_{\mathrm{CGKA},\Pi}(\mathcal{A})$.

Definition A.1. A CGKA scheme K is (t, q, ε)-*secure w.r.t. predicate* Π, if for all (t, q)-attackers,

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

```
*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 14: Compatibility oracles of the security game for continuous group key agreement schemes.

B FORWARD-SECURE GROUP AEAD

B.1 Syntax

- $v \leftarrow \mathsf{Init}(k_e, n, \mathsf{ID})$: takes as input a key k_e, the group size n, as well as a party ID ID and generates the initial state.
- $(v', e) \leftarrow \mathsf{Send}(v, a, m)$ generates ciphertext e encrypting plaintext m and authenticating associated data a;
- $(v', \mathsf{S}, i, m) \leftarrow \mathsf{Rcv}(v, a, e)$: decrypts ciphertext e to plaintext m and verifies associated data a; also outputs a pair (S, i) consisting of sender ID S and message index i.

B.2 Security

B.2.1 Main oracles. The oracles of the FS-GAEAD game are depicted in Figure 15.

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 i^{th} message sent by S was a challenge,
- 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 $\mathsf{AM}[\mathsf{S}, i, \mathsf{R}]$ 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 $(\mathsf{S}, i[\mathsf{S}])$ is recorded as being a challenge.

Delivering and injecting ciphertexts. Oracle **dlv-AM**$(\mathsf{S}, i, \mathsf{R})$ 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 $\mathsf{AM}[\mathsf{S}, i, \mathsf{R}] \leftarrow \texttt{received}$ 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.

```
// Initialize group
init(G)
    b ← {0,1}
    k_e ← 𝒦
    for ID ∈ G
        v[ID] ← Init(k_e, |G|, ID)
        i[ID] ← 0
    Chall ← ∅
    AM-Tr[·] ← ∅
    AM-Lk ← ∅
    AM[·,·,·] ← ϵ
    Del[·] ← ϵ
// S sends m with AD a
send(S, a, m)
    (v[S], e) ← Send(v[S], a, m)
    i[S]++
    for R ∈ G \ {S}
        AM[S, i[S], R] ← (a, m, e)
    return e
// Corruption of ID
corr(ID)
    AM-Rcvd ← {(S, i) | AM[S, i, ID] = received}
    AM-Lk +← (ID, AM-Rcvd, AM-Tr[ID])
    return v[ID]
// Message delivery
dlv-AM(S, i, R)
    req AM[S, i, R] ≠ {ϵ, received}
    (a, m, e) ← AM[S, i, R]
    (v[R], S', i', m') ← Rcv(v[R], a, e)
    if (S', i', m') ≠ (S, i, m)
        win
    if ¬Del[R]
        AM-Tr[R] +← (S, i)
    AM[S, i, R] ← received
// stop deletions for ID
no-del(ID)
    Del[ID] ← false
// Message injection
inj-AM(a', e', R)
    req ∀S, i, i : AM[S, i, R] ≠ (a', i, e')
    (v[S], S', i', m') ← Rcv(v[R], a, e)
    if m' ≠ ⊥
        if (*AM-sec(S', i')) ∨ AM[S', i', R] = received
            win
    AM[S', i', R] ← 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) ← Send(v[S], a, m_b)
    i[S]++
    for R ∈ G \ {S}
        AM[S, i[S], R] ← (a, m, e)
    Chall +← (S, i)
    return e
// Safety predicate
*priv-safe
    if ∀(S, i) : (S, i) ∈ Chall ⟹ *FS-sec(S, i)
        return true
    return false
```

Figure 15: The main oracles of the FS-GAEAD security game.

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 (ID, AM-Rcvd, AM-Tr[ID]) to the set AM-Lk to indicate that ID's state is now compromised, but the messages recorded in AM-Rcvd are supposed to remain secure (because the corresponding key material must have been deleted); the game also adds potential trash, AM-Tr[ID], stored by ID to the same set.

B.2.2 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.

B.2.3 Advantage. Let Π = ***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 $\mathrm{Adv}^{\mathsf{FS}}_{\mathsf{fs},\Pi}(\mathcal{A})$.

Definition B.1. An FS-GAEAD scheme F is (t, q, ε)-*secure w.r.t. predicate* Π, if for all (t, q)-attackers,

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

C PRF-PRNGS

C.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.

C.2 Security

A PRF-PRNG must satisfy PCFS (cf. Section 3.2) and be resilient to splitting-attacks. Therefore, the security game for PRF-PRNGs (cf. Figure 16) 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 σ[vid] stores the corresponding state of the PRF-PRNG,
  - the value R[vid] stores the corresponding output of the PRF-PRNG, and
  - the value BI[vid] is a flag indicating whether the input I absorbed to reach the state σ[vid] is known to the attacker.
- The set V-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 BI[vid] is set accordingly.
- He may reveal or challenge outputs R[vid] corresponding to arbitrary nodes $\mathsf{vid} \neq \mathsf{vid}_{\mathsf{root}}$ by calling the corresponding oracles **reveal** and **chall**, respectively. The flags Reveal[vid] resp. store Chall[vid] whether a reveal resp. a challenge has been requested for vid.
- Finally, $\mathcal{A}$ can also leak the state σ[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**.

C.2.1 Advantage. Let Π = ***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 PRF-PRNG w.r.t. to predicate Π is denoted by $\mathrm{Adv}^{\mathsf{PP}}_{\mathsf{PRF\text{-}PRNG},\Pi}(\mathcal{A})$.

init
  $b \leftarrow_{\mathsf{R}} \{0, 1\}$
  $\mathsf{vid}_{\mathsf{root}} \leftarrow \mathsf{HG.init}$
  $\sigma[\mathsf{vid}_{\mathsf{root}}] \leftarrow 0$
  V-Lk $\leftarrow \emptyset$
  $R[\cdot] \leftarrow \emptyset$
  Reveal[$\cdot$] $\leftarrow \emptyset$
  Chall[$\cdot$] $\leftarrow \emptyset$
  BI[$\cdot$] $\leftarrow \emptyset$
corr (vid)
  V-Lk $+\!\leftarrow$ vid
  return σ[vid]
process(vid, I, C)
  req $\nexists$ child of vid for (I, C)
  $\mathsf{vid}' \leftarrow \mathsf{HG.create}(\mathsf{vid}, I)$
  BI[vid'] $\leftarrow (I \neq \perp)$
  $(\sigma[\mathsf{vid}'], R[\mathsf{vid}']) \leftarrow \mathsf{PP}(\sigma[\mathsf{vid}], I, C)$
  return vid'
reveal(vid)
  req R[vid] $\neq \varepsilon$
  req $\neg$(Reveal[vid] $\vee$ Chall[vid])
  Reveal[vid] $\leftarrow$ true
  return R[vid]
chall(vid)
  req R[vid] $\neq \varepsilon$
  req $\neg$(Reveal[vid] $\vee$ Chall[vid])
  $R_0 \leftarrow R$[vid]
  $R_1 \leftarrow \mathcal{R}$
  Chall[vid] $\leftarrow$ true
  return R_b
safe
  return $\forall$vid : Chall[vid] $\Longrightarrow$ ***PP-secure**(vid)

Figure 16: PRF-PRNG security game.

Definition C.1. A PRF-PRNG scheme PP is (t, ε)-*secure w.r.t. predicate* Π, if for all t-attackers,

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

D SECURE GROUP MESSAGING

In this section we define compatibility helpers for SGM.

```
*compat-create(ID, skid, wkid)
    chk V-Pt[ID] = vid_root
    chk SK-ID[skid] = ID
    chk WK-SK[wkid] = skid
    chk skid ∈ SK-St[ID]
    chk wkid ∈ WK-St[ID]
    return true
*compat-prop(op, ID, ID′, skid)
    vid ← V-Pt[ID]
    chk vid ≠ vid_root
    G ← HG.roster(vid)
    select op
        case add do
            chk
              CL-KB[ID, ID′] ≠ ∅
            chk ID′ ∉ G
        case rem do
            chk ID′ ∈ G
        case upd do
            chk skid ∈ 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 ← V-Pt[ID]
    chk vid ≠ vid_root
    chk ∄ pid :
          Props[pid].vid = vid
        ∧ P′ = PM[pid]
    return true
*compat-commit(ID, pid)
    vid ← V-Pt[ID]
    chk vid ≠ vid_root
    chk pid ⊆ P-St[ID]
    G ← HG.roster(vid)
    for pid ∈ pid
        G ← *app-prop(G, pid)
        chk G ≠ ⊥
    return true
*app-prop(G, pid)
    p ← Props[pid]
    req p.orig ∈ G
    select 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-CM(ID, vid)
    chk HG.isChild(V-Pt[ID], vid)
    chk HG[vid].pid ⊆ P-St[ID]
    return true
*compat-inj-CM(ID, T′)
    vid ← V-Pt[ID]
    chk vid ≠ vid_root
    chk ∀vid′ ∈ HG.children(vid) :
          T′ ≠ CM[vid′, ID]
    return true
*compat-dlv-CM(ID, vid)
    chk HG.isChild(V-Pt[ID], vid)
    chk HG[vid].pid ⊆ P-St[ID]
    return true
*compat-inj-CM(ID, T′)
    vid ← V-Pt[ID]
    chk vid ≠ vid_root
    chk ∀vid′ ∈ HG.children(vid) :
          T′ ≠ CM[vid′, ID]
    return true
*compat-dlv-WM(ID, vid)
    chk V-Pt[ID] = vid_root
    wkid ← HG.addedWK(ID, vid)
    chk wkid ≠ ⊥
          ∧ wkid ∈ WK-St[ID]
    return true
*compat-inj-WM(ID, W′)
    vid ← V-Pt[ID]
    chk vid = vid_root
    chk ∀vid′ :
          W′ ≠ CM[vid′, ID]
    return true
*compat-send(S)
    chk V-Pt[ID] ≠ vid_root
    return true
*compat-chall(S, m_0, m_1)
    chk V-Pt[ID] ≠ vid_root
    chk |m_0|= |m_1|
    return true
*compat-dlv-AM(vid, S, i, R)
    chk vid ∈ V-St[R]
    chk AM[vid, S, i, R] ∉
      {ϵ, received}
    return true
*compat-inj-AM(a, e, R)
    chk ∀S, vid, i, m :
      AM[vid, S, i, R] ≠ (a, m, e)
    return true
```

Figure 17: Compatibility oracles of the security game for secure group-messaging schemes.

E SGM CONSTRUCTION

In this section we formally define the helper functions of our SGM construction.

```
// Returns ids, wpks, of new parties
*added(P)
    ID[·] ← ε
    wpk[·] ← ε
    i ← 1 for P ∈ P
        if P = ("add", ·, ·, (ID_a, kb′), ·)
            (wpk, ·) ← kb′
            ID[i] ← ID_a
            wpk[i] ← wpk
            i++
    return (ID, wpk)
// Returns spks after applying props
*new-spks(epid, P)
    spk ← s.Ep-SPK[epid]
    for P ∈ P
        if P = ("add", ·, ·, (ID_a, kb′), ·)
            (·, spk) ← kb′
            spk[ID_a] ← spk
        if P = ("rem", ·, ·, ID_r, ·)
            spk[ID_r] ← ε
        if P = ("upd", ·, ID_u, spk, ·)
            spk[ID_u] ← spk
    return spk
// Returns position of ID in roster
*roster-pos(ID, G)
    [ID_1, ..., ID_n] ← G
    for i ∈ [1, n]
        if ID_i = ID
            return i
// Returns Proposal Info
*get-propInfo(P′)
    if P′ = ("add", ·, ID_s, (ID_a, kb′), P̄)
        op = add
        orig = ID_s
        (wpk, spk) ← kb′; data = (ID_a, wpk, spk)
    if P′ = ("rem", ·, ID_s, ID_r, P̄)
        op = rem
        orig = ID_s
        data = ID_r
    if P′ = ("upd", ·, ID_s, spk, P̄)
        op = upd
        orig = ID_s
        data = spk
    return (op, orig, data)
```

Figure 18: The SGM Construction : Helper Algorithm.

Helpers. *added receives a vector of proposals **P** and returns the ids and welcome public keys of newly added members. *new-spks receives an epoch id and vector of proposals, (epid, **P**), and returns the public verification keys for the ids affected by the proposals in **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 op = add, data holds the id, ID_a, welcome key material, wpk, and the signature verification key spk, of the newly added member. If op = rem, data holds the id of the removed party ID_r, and if op = upd, data holds the updated verification key spk.