








# The Key Lattice Framework for Concurrent Group Messaging

Kelong Cong<sup>3</sup> , Karim Eldefrawy<sup>2</sup> , Nigel P. Smart<sup>1,3</sup>  ,  
and Ben Turner<sup>2</sup> 

<sup>1</sup> COSIC, KU Leuven, Leuven, Belgium  
`nigel.smart@kuleuven.be`

<sup>2</sup> SRI International, Menlo Park, USA  
{`karim.eldefrawy,ben.turner`}@confidential.io

<sup>3</sup> Zama Inc., Paris, France  
`kelong.cong@zama.ai`

**Abstract.** Today, two-party secure messaging is well-understood and widely adopted, e.g., Signal and WhatsApp. Multiparty protocols for secure group messaging are less mature and many protocols with different tradeoffs exist. Generally, such protocols require parties to first agree on a shared secret group key and then periodically update it while preserving forward secrecy (FS) and post compromise security (PCS).

We present a new framework, called a *key lattice*, for managing keys in concurrent group messaging. Our framework can be seen as a “key management” layer that enables concurrent group messaging when secure pairwise channels are available. Security of group messaging protocols defined using the key lattice incorporates both FS and PCS simply and naturally. Our framework combines both FS and PCS into directional variants of the same abstraction, and additionally avoids dependence on time-based epochs.

## 1 Introduction

End-to-end encrypted secure messaging systems such as Signal and WhatsApp are widely deployed and used. The case of two-party protocols is well-understood, and has been extensively analyzed in the literature [3, 8, 18, 20, 26], but multiparty protocols (for group messaging) are still an active research area. At the moment, the Message Layer Security (MLS) IETF working group<sup>1</sup> is developing a standard to define an efficient and secure group messaging protocol. The key building block of MLS is continuous group key agreement (CGKA), which lets a group of users securely agree on a shared secret key [4], evolve it continuously while ensuring forward secrecy (FS) and post compromise security (PCS).

Many existing CGKA protocols, and their extension to group messaging protocols, require an additional infrastructure server that guarantees availability and orders messages. Recent work reduces dependence on the additional infrastructure, but still depends on a propose-and-commit paradigm [1, 2, 6] that

<sup>1</sup> <https://messaginglayersecurity.rocks/>.

allows concurrent update proposals but requires serial commitments to accept the changes. This work develops abstractions and protocols to advance group messaging towards truly asynchronous channels and a decentralized environment where there is no central server to order messages. In such an environment, there may be a different “latest” group key in the view of every honest user—all of whom simultaneously encrypt messages, all of which must be decrypted.

Our main contribution is conceptual. We model the group keys used within the protocol via a key lattice, which can be seen as an  $n$ -dimensional grid if there are  $n$  participants. The key lattice tracks all the group keys that will ever be used by the parties. Each key evolution travels along a path in the lattice. Every party uses the key lattice to track not only its own view of the current group key(s), but also the information it has about the other parties’ views. To both permit concurrency (via the ability to swap the order of key updates) and to prevent the state space from exploding, we require that the key evolution functions are commutative.

By framing our (new) security definitions with respect to the key lattice, we intuitively find that the dual (and simultaneous) notions of FS and PCS become directional variants of the same simple notion, which states that the adversary cannot traverse the key lattice to learn keys which it has not yet compromised.<sup>2</sup> We also eliminate any dependence on epoch-based time from the analysis and solely focus on the keys’ relationships to each other. To ensure PCS, parties evolve the group key with random updates and define new points on the key lattice. To ensure FS, each party tracks other parties’ views of the group key, and deletes keys which it knows will never be used again. We also show how to trade FS for correctness when desired, since in a fully asynchronous network, the adversary may arbitrarily delay delivery of an encrypted application message in order to force one party to hold old keys.

Our secondary contribution is an instantiation of a novel group messaging protocol that uses the key lattice, and we prove its security.

**Group Key Agreement vs. Group Messaging:** It is not always straightforward to transform from group key agreement to group messaging. Key exchange protocols usually contain a key-confirmation step, but when the key exchange protocol is used as a building block in a larger protocol (e.g., secure messaging), this step breaks the key indistinguishability property of key exchange. This is a well known problem even for two-party key agreement followed by composition with a secure channel, see for example [15, 16]. We avoid this definitional problem by treating key-agreement and messaging together and directly analyzing the scheme for group messaging.

**Asynchrony vs. Concurrency:** An *asynchronous* group messaging protocol means that the adversary can arbitrarily reorder messages that are sent, as long

---

<sup>2</sup> This approach bears some resemblance to the analysis of Fuchsbauer et al. [24] for public key re-encryption.

as all are eventually delivered. This models a highly adversarial network, and subsumes the scenario that some parties can temporarily “go offline” (if the adversary does not deliver messages to them) and then receive messages later when they come back online. A *concurrent* protocol allows messages, including update messages, to be sent and processed concurrently. But messages are delivered within some round of execution. The work by Bienstock, Dodis and Rösler [7] studied the trade-off between PCS, concurrency, and communication complexity. They show an upper-bound in terms of communication overhead that increases from  $O(\log n)$  when there is no concurrency, to  $O(n)$  when the update messages are fully concurrent.

Concurrent group messaging is suitable for the *decentralized* setting where there does not exist a central party to order messages. Nevertheless, it is possible to use a central server as a broadcast station to improve the communication cost, this way parties no longer need to broadcast messages to the group by themselves.

## 1.1 Related Work

Group key agreement and group messaging protocols have a long history. Early work focused on generalizing the Diffie-Hellman key exchange protocol [25, 32]. Later work extended the security guarantees (e.g., by providing authentication, forward secrecy, and post-compromise security) [10, 12–14], and improved performance and added new features (e.g., support for dynamic groups) [11]. This section outlines a few of the related work that are similar to our work. For the full details on the related work, please see the full version [21].

The closest work to ours is the recent paper by Weidner et al. [33], who introduced “decentralized” continuous group key agreement (DCGKA). DCGKA makes progress on the concurrency problems in ART and RTreeKEM so that all group members converge to the same view if they receive the same set of messages (possibly in different orders). The key primitive that enables concurrent updates is authenticated causal broadcast, defined in a similar way as Lamport’s vector clocks [27]. Additionally, the authors made progress on how to manage group membership in an asynchronous network without a central server. However, their construction still requires a serial commitment.

In comparison to Weidner et al. [33], our construction does not require authenticated causal broadcast; we permit asynchronous messaging by buffering messages that are received out of order, and we authenticate via authenticated encryption. Our construction also does not require acknowledgements. This substantially reduces the cost of an update because DCGKA requires  $n-1$  broadcast acknowledgements for an update.

Sender Keys, currently deployed by WhatsApp [35], also builds group messaging from pairwise Signal. During initialization, each party sends a symmetric “sender” key to all the group members using the pairwise Signal protocol. This key is used for encrypting payload messages by that party. Every party keeps  $n$  “sender” keys in their state where  $n-1$  keys are used for decryption and 1 is used for encryption. Sender Keys does not provide PCS since an adversary

**Table 1.** Comparing our work and existing work. PCS denotes post compromise security, and FS denotes forward secrecy. ROM stands for the random oracle model, StM denotes the standard model. ( $\square$ ) an update for DCGKA requires  $n - 1$  broadcast acknowledgements, so the total complexity is  $O(n^2)$ , although the sender’s computational complexity is  $O(n)$ . ( $\diamond$ ) These works use the propose-and-commit paradigm, where assumes the existence of epochs and allows concurrent proposals but a serial commitment is required. ( $\dagger$ )  $t$  is the number of corrupt parties. ( $\ddagger$ ) The server in CoCoA and SAIK processes an update to send an individual packet to each participant. They also order messages. ( $\Delta$ ) The SAIK server arbitrarily chooses one of concurrent updates to be processed. Our work is the only one which supports concurrent updates, does not require an active server, is PCS and FS and has a proof of security against adaptive adversaries. In this table desired features are highlighted in blue and those which negative impact security are in red.

Protocol	Update Cost		PCS Healing Rounds	FS	Active Server	Concurrent Updates	Proof	Adaptive	
	Sender	Receiver							
Original TreeKEM [30]	$O(\log n)$	$O(1)$	$n$	yes	yes	Ordering	no	None	n/a
Causal TreeKEM [34]	$O(\log n)$	$O(1)$	$n$	yes	yes	none	causal	StM	yes
RTreeKEM [4]	$O(\log n)$	$O(1)$	2	yes	yes	Ordering	no	ROM	yes
Concurrent TreeKEM [7]	$O(n)$	$O(1)$	2	yes	no	none	yes	StM	yes
Signal group [22, 31]	$O(n)$	$O(1)$	2	yes	yes	Prekeys	yes	None	n/a
Sender Keys [31, 35]	$O(n^2)$	$O(n)$	2	yes	yes	Prekeys	yes	None	n/a
DCGKA [33]	$O(n)$ ( $\square$ )	$O(1)$	2	yes	yes	none	yes ( $\diamond$ )	ROM	no
CoCoA [2]	$O(\log n)$	$O(1)$	$\log(n)$	yes	yes	Process-Updates ( $\ddagger$ )	yes ( $\diamond$ )	ROM	yes
SAIK [6]	$O(\log n)$	$O(1)$	2	yes	yes	Process-Updates ( $\ddagger$ )	yes ( $\Delta$ )	ROM	yes
DeCAF [1]	$O(\log t)$ ( $\dagger$ )	$O(1)$	$\log(t)$	yes	yes	blockchain	yes ( $\diamond$ )	ROM	yes
Our work	$O(n)$	$O(1)$	2	yes	yes	none	yes	StM	yes

who corrupts a party will learn all the symmetric keys and decrypt future messages sent to all parties. Fully healing the state therefore requires every party to update its symmetric key, which has a cost of  $O(n^2)$ .

Our work can be viewed as a generalization of Sender Keys with improved security and functionality, where parties update the key lattice instead of holding symmetric keys for each party. The group session heals once a corrupted party’s pairwise channels heal because the next update it sends or receives is indecipherable to the adversary. This requires  $O(n)$  public key operations (also  $O(n)$  communication complexity) after one corruption.

**Summary:** Table 1 summarizes a representative sample of recent literature on group key agreement and group messaging. “Update Cost” gives the communication complexity to update a shared or pairwise key, for the sender and the receiver, and “Healing rounds” describes the round complexity of healing the session after a corruption. “Active Server” is a server that provides additional functionalities other than a PKI, such as ordering messages or post-processing updates. For example, the Signal servers need to store single-use pre-keys and the TreeKEM servers need to order messages. “Adaptive” means whether the adversary can adaptively pick which oracles to query during the security game.

Our work, on the last row, carves out a new trade-off in the group messaging design space. Specifically, we use pairwise channels which results in  $O(n)$  update cost and, in contrast to prior work, maintain a set of evolving shared group key without compromising security, i.e., allowing adaptive queries.

## 1.2 Technical Overview

Our group messaging (GM) protocol consists of three building blocks: (1) an initial group key agreement (GKA) protocol, (2) a group randomness messaging (GRM) protocol used to transport key updates, and (3) a key lattice. We overview all blocks but focus on the key lattice as it is our primary contribution.

**Group Key Agreement (GKA):** Our GKA assumes existence of a public key infrastructure (PKI). In other words, each party knows the other party’s long-term public key. The protocol takes as input the identities and public keys of the group members and outputs a symmetric key shared by those members. This symmetric key is used by the other two building blocks detailed below. We use the GKA as a black box and thus are not concerned with the exact construction in this work. Nevertheless, we require that it is forward secure, i.e., if the long-term secret key is compromised after agreeing on a shared key, the adversary still learns nothing about the shared key. Note that many GKA protocols exist in the literature [9, 13, 14, 29]. In this work we use the definition from [14], which allows for asynchrony (as needed by our construction).

**Group Randomness Messaging (GRM):** GRM abstracts the transport mechanism used to communicate key updates and make our proof more modular. Because GRM requires pairwise channels with FS & PCS, it could be implemented using pairwise 2-party secure messaging e.g., pairwise Signal or another double-ratchet-based protocol. We provide a custom instantiation of GRM in Sect. 5 that better fits our assumptions (specifically, we assume only a public key infrastructure and do not require a server to distribute pre-key bundles), is conceptually simpler than a double-ratchet, and is easier to prove secure. Nevertheless, we give an outline of how to build a concurrent group messaging protocol from black-box primitives in Sect. 3.4.

Our GRM protocol is intuitively simple. Whenever a party  $U$  sends a random message  $x$  to party  $V$ ,  $U$  samples a fresh key pair  $(\mathbf{pk}', \mathbf{sk}')$ , and encrypts  $(x, \mathbf{pk}')$  under the public key  $\mathbf{pk}_V$  that  $U$  holds for  $V$ . When  $V$  receives  $(x, \mathbf{pk}')$ , it assigns  $\mathbf{pk}'$  as its latest public key for  $U$  and outputs  $x$  as  $U$ ’s message. Future messages sent by  $V$  to  $U$  must be encrypted under the latest ephemeral public key that  $V$  holds for  $U$ . The scheme achieves both FS and PCS because all secret keys are independently sampled with every message sent, and therefore leaking one secret key never reveals information about another. The scheme uses a public key AEAD scheme for all encrypted messages, where the associated data are bookkeeping material on the order of updates.

**Key Lattice:** We now explain our key lattice framework, including our security game and its representation of FS and PCS.

*Framework:* Every group key in a group messaging protocol is associated with a coordinate in a discrete  $n$ -dimensional space, where  $n$  is the number of players



(a) The red vertices and edges are explicitly revealed to the adversary.

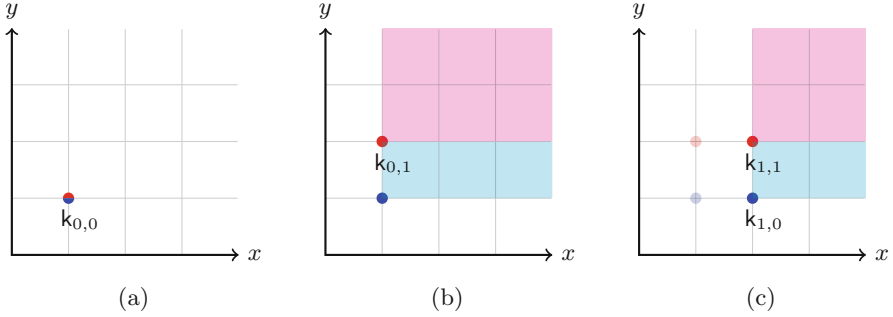
(b) The full set of information that an adversary can compute from 1a.

**Fig. 1.** In Fig. a, the red vertices and edges are explicitly revealed to the adversary. If PCS holds, then the adversary cannot compute the key  $k_{2,2}$  because there is no path of red edges from a red vertex to  $k_{2,2}$ . In Fig. b, the adversary can compute the keys  $k_{0,1}$ , and  $k_{0,1}$ , and  $k_{1,1}$  by starting at  $k_{0,0}$  and following a path of red edges. FS can analogously be visualized by (preventing) traversing the directed graph “backwards” from a compromised vertex. (Color figure online)

in the group. When parties update the group key (at some index), the new key produced is mapped to a larger index. For example, for  $n = 2$ , a key  $k_{1,0}$  at coordinate  $(1, 0)$  may be updated to a new key with an associated coordinate  $k_{1,1}$ . We also provide a graphical explanation of a key lattice in which the indices in the discrete  $n$ -dimensional space are vertices, and each vertex is labeled with a key. In the graph, edges between vertices represent key updates.

*FS & PCS:* Our key lattice allows us to discuss FS & PCS in a unified and simple manner, as directional variants of the same abstraction. In Fig. 1, every key is mapped to a point on the graph, and updates are mapped to edges in the graph. Black vertices and edges are not revealed to the adversary, and red vertices and edges are revealed. A party that “knows” both the key corresponding to a vertex and an edge leaving that vertex will also “know” the vertex’s neighbor. FS & PCS mean that the *only* way the adversary can learn a key  $k^*$  at some target vertex  $v^*$  is by starting with a red vertex on the graph and following a path of red edges to  $v^*$ . In the traditional definition of FS, this would mean that given a vertex  $v$ , without following (in reverse) a path of red edges, the adversary cannot learn a predecessor of  $v$ . In the traditional definition of PCS, this would mean that given a vertex  $v$ , without following a path of red edges, the adversary cannot learn a successor of  $v$ . The key lattice is described in full in Sect. 3.

*Security Game and Freshness:* Our security game is an oracle game in which the adversary activates oracles corresponding to parties running a polynomial number of protocol executions. The adversary plays a semantic security game against a “fresh” key on one of the lattices. A key is “fresh” precisely if the adversary cannot derive that key from its view of the execution thus far; graphically, this means that the key is black in the corresponding graph akin to Fig. 1b. The



**Fig. 2.** An example of a local key lattice in an execution with two players (blue and red) from the perspective of the red party. (Color figure online)

adversary wins the semantic security game if it can distinguish two ciphertexts encrypted under a fresh key.

*Tracking Keys of Other Parties:* Each party maintains a local key lattice to track the group keys, but does not (necessarily) need to maintain a full view of the key lattice. Each party tracks only the keys it needs in order to decrypt a message that it has not yet received. This permits the construction to achieve the best possible FS while also achieving correctness; as soon as some party knows it no longer needs some key, it deletes the key from its view (in order to prevent an adversary from learning the key after it has become deprecated).

We illustrate our approach in Fig. 2. For simplicity, we only consider two parties labelled with the colors red and blue. The shaded regions, assigned by color, indicate the set of points towards which the corresponding party may define a new group key in the future. Any point in a totally unshaded region represents an index of a key that can be deleted. In our construction, when any party updates the key, it moves the latest group key towards a point in the  $n$ -dimensional space along an axis that has been assigned uniquely to it. Blue and red update the key towards higher indices on the  $x$  axis and  $y$  axis, respectively.

1. In Fig. 2a, the red and blue parties initialize their local lattices with  $k_{0,0}$ .
2. In Fig. 2b, red evolves the group key, which moves red’s latest key to  $k_{0,1}$ .
3. In Fig. 2c, suppose red received an update message from blue. Red applies the update and evolves its own index from  $k_{0,1}$  to  $k_{1,1}$ . Because red knows that blue evolved its key, red updates its view of blue’s index  $k_{0,0}$  to  $k_{1,0}$ . Specifically, red’s perspective of the latest key for blue becomes  $k_{1,0}$ . Since  $k_{0,0}$  and  $k_{0,1}$  are outside the shaded region, these keys are removed.

*Windowing to Limit State Expansion:* In addition to the state reduction described above, we also apply a state “window” that prevents the state from blowing up in case encrypted messages are delayed over the network, at the expense of the ability to decrypt long-delayed messages. Consider that if one

party makes  $m$  updates to the shared group key, resulting in  $m$  possible different group keys, then parties must keep  $O(m)$  states in case another party sends a message using one of those  $m$  keys. In our windowing scheme, each party maintains *at most* the latest  $w$  key evolutions from every other party, which provides the ability to compute at most  $w^n$  total keys on the key lattice at any time.

When using this scheme, there are situations in which parties may send messages such that some application messages are not decryptable. Suppose sender  $S$  sends an application message  $m$  encrypted under key  $k$ , and then suppose  $S$  updates the group key  $w$  times starting with  $k$ . If  $S$ 's message  $m$  is delayed until after receiver  $R$  receives  $S$ 's key updates, then  $R$  will delete the key material describing how to decrypt  $m$ . In synchronous networks, the window can be set such that parties update their keys once per epoch, and the window can be set large enough (by setting  $w$  is equal to the number of epochs that measure the network delay) for sent messages to always be received in time to be decrypted. In the general asynchronous case, the window can be set to  $\infty$  in order to always guarantee decryption, but this approach loses FS.<sup>3</sup> Thus, windowing allows us to trade between security and correctness.

**Group Messaging (GM):** In our construction, parties who wish to participate in a GM instance begin by running a GKA protocol to obtain a shared symmetric key  $k$ . They use  $k$  to initialize their key lattice, and then use GRM to securely communicate update messages that can be applied to the key lattice to evolve the shared group key. When a party encrypts an application (payload) message, it always uses the latest key in its key lattice.

**Dynamic Membership:** We provide an informal extension of our framework that permits dynamic group membership “for free,” and additionally handles simultaneous adds and removals with no additional effort, thus completely avoiding “splitting” [5] issues in synchronous protocols where multiple parties make competing simultaneous updates. The intuitive understanding is to view our representation of a key lattice as a lossless compression of an  $n$ -dimensional space in which only a finite number of points are defined, where  $n$  is the number of all possible identities. Each dimension in the key lattice represents a party that belongs to the group, and all other dimensions in the lattice are defined to contain points set to  $\perp$ . When a party joins the group, points become defined in its corresponding dimension. When it leaves the group, its future group updates become invalid.

Treating dynamic membership in this way averts all of the problems of concurrency incurred by other works, including with respect to insider attacks, since groups including the new members are only defined in the lattice as successor points of the addition operation, and we incur no conflicts by maintaining multiple copies of the lattice that correspond to groups both with and without the

<sup>3</sup> This tradeoff was similarly explored by [28]; our asynchronous security model specifically accounts for the attacks they describe by withholding some ciphertexts and corrupting a party days later to recover the messages.



new member. Dynamic membership is not the main focus of our work and a formal definition and analysis is needed before it can be considered for practical use, which we leave for future work. Nevertheless, we provide more details of our dynamic group extension in the full version [21].

## 2 General Definitions and Notation

We denote by  $\mathbb{N}$  the natural numbers. For a list  $\ell$ ,  $\ell[i]$  denotes the  $i$ th element of  $\ell$ . We write  $[m] = \{1, \dots, m\}$ , and  $[a, b] = \{a, a + 1, \dots, b - 1, b\}$  where  $b > a$ .  $\mathcal{P}$  is the set of all possible parties, and  $n = |\mathcal{P}|$ . We define a function  $\phi : \mathcal{P} \rightarrow [n]$  that assigns a canonical ordering of  $\mathcal{P}$ , i.e., to each  $U \in \mathcal{P}$ ,  $\phi(U)$  assigns a unique index between 1 and  $n$ .

Let  $\mathbf{i} \in \mathbb{N}^n$  denote an *index vector*. All keys will be indexed by index vectors, i.e., we will always write the secret keys as  $\mathbf{k}_i$ . The  $j$ -th element of index vector  $\mathbf{i}$  will be denoted by  $\mathbf{i}^{(j)}$ . We introduce a function  $\text{increment}(\mathbf{i}, j)$  with inputs an index vector  $\mathbf{i}$  and an integer  $j \in [n]$  and returns an index vector  $\mathbf{i}'$  such that for  $i \neq j$ ,  $\mathbf{i}'^{(i)} = \mathbf{i}^{(i)}$ , and  $\mathbf{i}'^{(j)} = \mathbf{i}^{(j)} + 1$ . Similarly,  $\text{decrement}(\mathbf{i}, j)$  returns an index vector  $\mathbf{i}'$  such that for  $i \neq j$ ,  $\mathbf{i}'^{(i)} = \mathbf{i}^{(i)}$ , and  $\mathbf{i}'^{(j)} = \mathbf{i}^{(j)} - 1$ . We define a partial ordering over index vectors by saying  $\mathbf{i} \geq \mathbf{c}$  if  $\mathbf{i}^{(j)} \geq \mathbf{c}^{(j)}$  for all  $j$ .  $\mathcal{H}_{\geq \mathbf{c}}$  for a constant index vector  $\mathbf{c} \in \mathbb{N}^n$  denotes the  $n$ -dimensional hyperplane of all index vectors  $\mathbf{i}$  such that  $\mathbf{i}^{(j)} \geq \mathbf{c}^{(j)}$  for all  $j \in [n]$ .

**Network Model:** Parties are connected via pairwise channels such that both parties know the identity of the party on the other end. A PKI provides a mapping between an identity  $U \in \mathcal{P}$  and its long-term public key. Every  $U \in \mathcal{P}$  also has its own long-term private key.

**Adversarial Model:** In our security game, the adversary is responsible for delivering all messages to its oracles. It may reorder messages arbitrarily, as per the definition of an asynchronous network [17]. Proper ordering of messages *within a subprotocol* is enforced by sequence numbers on our updates and encrypted messages, and therefore in the exposition we assume that each subprotocol's messages are ordered, but messages sent by different subprotocols (such as GKA, GRM, and GM application messages) are not ordered with respect to each other.

The adversary may call its oracles on messages that have not been sent by honest parties. This is an injection attack. However, because all messages in our constructions are authenticated, successfully changing the state of an oracle without knowledge of a party's underlying key would break the security of an authenticated cryptographic primitive (e.g., AEAD).

The adversary can corrupt parties to learn protocol keys, and in some cases may inject messages based on those keys. For example, learning a group key

allows the adversary to inject application messages, but these injections do not affect the security of other keys.<sup>4</sup>

We defer a discussion of insider security in our model to the full version [21].

**Encryption:** In the full version [21] we give the standard definitions for encryption, key encapsulation mechanisms (KEMs) and authenticated encryption with associated data (AEAD) that we use in this paper. Notably, we use a variant of public key encryption—public key encryption with additional data (PKEAD). It is similar to an IND-CCA secure public key encryption scheme that allows additional plaintext data to be appended, where the additional data binds to the ciphertext.

### 3 Key Lattice

The key lattice is our central idea for managing concurrent key updates. Because the key lattice tracks the set of group keys generated during a group messaging execution, we additionally define security of group messaging with respect to the key lattice. We now formally define a key lattice.

**Definition 3.1 (Key Lattice).** *We define  $\mathbb{K}$  to be the space of keys, and we define  $\mathbb{L}$  to be the lattice of  $\mathbb{N}^n$  where the ordering is defined by  $\mathbf{i}_a \leq \mathbf{i}_b$  if all elements in  $\mathbf{i}_a$  are less or equal to  $\mathbf{i}_b$ , and  $\mathbf{i} \in \mathbb{N}^n$  denotes a point on the lattice. A key lattice  $L = \{(\mathbf{i}, \mathbf{k}_i)\}_{\mathbf{i} \in \mathbb{L}}$  where  $\mathbf{k}_i \in \mathbb{K} \cup \{\perp\}$  is a discrete lattice for which every point  $\mathbf{i} \in \mathbb{L}$  is associated with either a single key or  $\perp$ .*

We denote the association by letting  $\mathbf{k}_i$  be the key associated with  $\mathbf{i}$ . We also say that the key for an index  $\mathbf{i}$  is *defined* if  $\mathbf{k}_i \neq \perp$ . Intuitively, parties will compute and agree on many pairs  $(\mathbf{i}, \mathbf{k}_i)$ .

Given a key lattice, a key  $\mathbf{k}_i$  is  $j$ -maximal if there is no  $\mathbf{j} \in \mathbb{N}^n$  for which  $\mathbf{j}^{(j)} > \mathbf{i}^{(j)}$  and  $\mathbf{k}_j \neq \perp$ . If a key is  $j$  maximal for all  $j \in [n]$ , we say the key is maximal in the lattice. Looking ahead, in each party’s local lattice there is always a maximal key, computed by all applying all updates that the party knows.

#### 3.1 Key Evolution

When a party evolves the group key, it adds a new key (or, as in our construction in Sect. 3.3, a group of keys), to the key lattice. Key evolution is described by a function  $\text{KeyRoll} : \mathbb{K} \times \mathcal{X} \rightarrow \mathbb{K}$ , where  $\mathbb{K}$  is the key space and  $\mathcal{X}$  is the *update space*, which encodes the data applied to the key during evolution. In our construction, we will require a few properties of the  $\text{KeyRoll}$  function. First, we require that  $\text{KeyRoll}$  is commutative, i.e.  $\text{KeyRoll}(\text{KeyRoll}(\mathbf{k}, x), x') = \text{KeyRoll}(\text{KeyRoll}(\mathbf{k}, x'), x)$  for all  $\mathbf{k} \in \mathbb{K}$  and  $x, x' \in \mathcal{X}$ .

<sup>4</sup> Some authentication schemes require parties to sign messages with their long-term keys [23] but adapting this to concurrent group messaging is non-trivial, and not the focus of this work.

In addition to commutativity, we require that  $\text{KeyRoll} : \mathbb{K} \times \mathcal{X} \rightarrow \mathbb{K}$  is *unpredictable* in its second input. Intuitively, knowing only the first input (a key from  $\mathbb{K}$ ), no adversary can “predict” the output (another key from  $\mathbb{K}$ ), if the second input (an update from  $\mathcal{X}$ ) is sampled at random. Similarly, we say that  $\text{KeyRoll}$ ’s inverse is unpredictable if given only  $k' \leftarrow \text{KeyRoll}(k, x)$ , no adversary can “guess” the input  $k$ . More formally, we have the following.

**Definition 3.2 (Unpredictability).** *A family of functions  $\mathcal{F} = \{F_\lambda\}_\lambda$  where  $F_\lambda : \mathbb{K}_\lambda \times \mathcal{X}_\lambda \rightarrow \mathbb{K}_\lambda$  is unpredictable in its second input if there exists a negligible function  $\text{negl}$  such that for every probabilistic polynomial time adversary  $\mathcal{A}$  and every  $\lambda$ :*

$$\Pr[y = F_\lambda(k, x) : k \leftarrow \mathbb{K}_\lambda, x \leftarrow \mathcal{X}_\lambda, y \leftarrow \mathcal{A}(1^\lambda, k)] \leq \text{negl}(\lambda)$$

*$\mathcal{F}$ ’s inverse is unpredictable if there exists a negligible function  $\text{negl}$  such that for any polynomial time adversary  $\mathcal{A}$  and every  $\lambda$ :*

$$\Pr[k' = k : k \leftarrow \mathbb{K}_\lambda, x \leftarrow \mathcal{X}_\lambda, k' \leftarrow \mathcal{A}(1^\lambda, F_\lambda(k, x))] \leq \text{negl}(\lambda)$$

*where in each experiment,  $k$  and  $x$  are sampled uniformly at random from their respective domains.*

We remark that there are many families of unpredictable functions. For instance,  $\text{KeyRoll}(k, x) = k \oplus x$  satisfies the unpredictability definition, as well as  $\text{KeyRoll}(k, x) = \text{PRF}_x(k)$ <sup>5</sup>. In both cases, it is not possible to predict the output without knowing the key. The difference between the first construction and the second is that in the first case, knowing the first input and the output completely leaks the update material  $x$ . This property is not critical to our construction; we can prove security for our main protocol assuming only that  $\text{KeyRoll}$  is unpredictable. However, for completeness (and for situations where unpredictability is not enough), one can define a variant of one-wayness.

*One-Wayness.* We introduce a non-standard form of one-wayness to analyze the properties of our scheme. Intuitively, a function is one-way on a challenge (first or second) input if, given  $F(k, x)$  and the other input, it is hard for any adversary to compute the challenge input. Below we provide definitions of one-wayness on the second input. Although we do not use it in our construction, it is also possible to define one-way-ness in the first input analogously to one-way-ness in the second input. Intuitively, given  $x$  and  $F(k, x)$ , it should be hard to compute  $k$ . If  $\text{KeyRoll}$  is one-way in the first input, then the construction inherits additional useful properties, which we describe in the full version [21]. We now present our definitions for one-wayness on the second input.<sup>6</sup>

<sup>5</sup> In practice we cannot use the PRF construction because it is not commutative.

<sup>6</sup> We remark that the standard definition of one-wayness requires the adversary to find an equivalent pre-image of the function, and not the exact same pre-image.

**Definition 3.3 (One-Wayness (on the Second Input)).** A family of functions  $\mathcal{F} = \{F_\lambda\}_\lambda$  where  $F_\lambda: \mathbb{K}_\lambda \times \mathcal{X}_\lambda \rightarrow \mathbb{K}_\lambda$  is one-way on its second input if there exists a negligible function  $\text{negl}$  such that for every probabilistic polynomial-time adversary  $\mathcal{A}$  and every  $\lambda$

$$\Pr[x' = x : k \leftarrow \mathbb{K}_\lambda, x \leftarrow \mathcal{X}_\lambda, x' \leftarrow \mathcal{A}(1^\lambda, k, F_\lambda(k, x))] \leq \text{negl}(\lambda).$$

where  $k$  and  $x$  are sampled randomly from their respective domains.

*$\ell$ -Point One-Wayness.* The definition above can be generalized to the setting where  $\mathcal{A}$  obtains polynomially many (in the security parameter) samples of  $(k, F_\lambda(k, x))$  pairs for different randomly sampled  $k$  but the same  $x$ . This additional property allows us to further constrain the power of the adversary. We defer the definition and discussion to the full version [21].

### 3.2 The Key Graph

In our construction, parties track the group key(s) by assigning each key to a point on the lattice. When a party evolves the group key, it defines the transition from one point on the lattice to another. In fact, our construction defines the transitions from a family of points to another family of points. Therefore, it is useful to describe the key lattice as a directed acyclic graph, where the vertices are labeled with keys, and the edges encode key evolutions.<sup>7</sup> Specifically, we define a key graph  $\mathcal{G}$ , where each lattice point  $\mathbf{i} \in \mathbb{N}^n$  is a vertex, and each vertex is labeled with a single key or with  $\perp$ . In our discussion, we refer to vertices by the lattice points they represent. There exists a directed edge from vertex  $\mathbf{i}$  to  $\mathbf{j}$  if  $\mathbf{j} = \text{increment}(\mathbf{i}, k)$  for some  $k \in [n]$ , and we say that  $\mathbf{i}$  *precedes*  $\mathbf{j}$ , or  $\mathbf{j}$  *succeeds*  $\mathbf{i}$ , if there is an edge from  $\mathbf{i}$  to  $\mathbf{j}$ . Edges in a key graph are labeled with the key evolutions that they represent. We say there exists a *path*  $\rho$  of length  $\ell$  between two vertices  $\mathbf{i}$  and  $\mathbf{i}'$  if there exists a sequence of edges  $(v_1, v_2), (v_2, v_3), \dots, (v_{\ell-1}, v_\ell)$  such that (a)  $v_1 = \mathbf{i}$ , (b)  $v_\ell = \mathbf{i}'$ , and (c)  $v_{j-1}$  precedes  $v_j$  for all  $j \in [2, \ell]$ . The local state held by each party in our protocol is a pair  $(L, E)$ , where  $L$  denotes the key lattice held by the party and  $E$  represents the edges representing the transformation between keys.

### 3.3 Instantiation

We now describe how parties manipulate a key lattice.

*Generating a Set of Key Evolutions.* In our construction, each party updates the group key in its own “direction” in  $L$ ; the  $d$ th party ( $U \in \mathcal{P}$  for which  $\phi(U) = d$ ) always updates the group key towards larger indices in the  $d$ th dimension on the lattice. A key update  $\sigma \in \Sigma$  sent by one party to another is therefore a tuple  $(d, j, x)$ , where  $d$  is a dimension in the key lattice (generated by the party

<sup>7</sup> In this work, every graph is a directed acyclic graph.

$U$  such that  $\phi(U) = d$ ),  $j \in \mathbb{N}$  is an index that annotates how many times the updating party has updated the group key, and  $x \in \mathcal{X}$  is data that describes how to update the key (for **KeyRoll**). In other words,  $\Sigma = [n] \times \mathbb{N} \times \mathcal{X}$ . The  $j$ th key evolution generated by any party therefore defines the transition from every index  $\mathbf{i}$  to index  $\mathbf{i}'$  such that  $\mathbf{i}^{(d)} = j$  and  $\mathbf{i}' = \text{increment}(\mathbf{i}, d)$ , and it defines the evolution to use update data  $x$ . In our construction, the space  $\mathcal{X}$  is the same as described in Definitions 3.2 and 3.3.

Observe that each key update in our construction defines a group of key evolutions, which can be described in our graphical representation as a group of edges. We require commutativity of **KeyRoll** to guarantee that when transitioning from key  $k$  to key  $k'$  (over one or more edges), where  $k$  is represented by vertex  $u$ ,  $k'$  is represented by vertex  $v$ , and there are multiple paths between  $u$  and  $v$  in some party's key lattice, it does not matter which path is taken.

*Our KeyRoll Function.* Our construction depends on the discrete logarithm assumption to instantiate **KeyRoll**( $k, x$ ) as  $k^x$ . That is to say, let key space  $\mathbb{K}$  be a prime-order group  $G$  in which the discrete log problem is hard, and let update space  $\mathcal{X}$  be  $\mathbb{Z}_{|G|-1}$ . This construction easily satisfies our commutativity requirement since  $(k^x)^{x'} = (k^{x'})^x$ . For appropriately chosen parameters, the construction is trivially unpredictable. If the discrete logarithm problem is hard in  $G$ , then **KeyRoll** is also one-way on its second input.

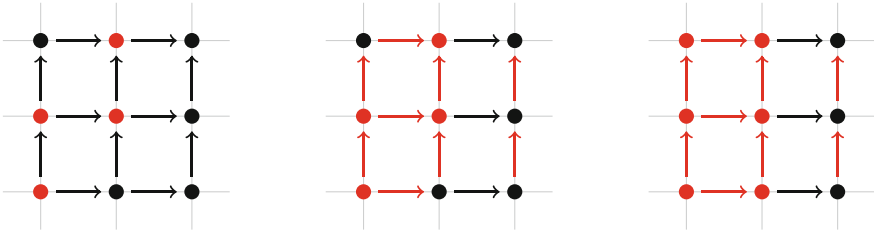
**Computable Lattice:** The description of a key lattice  $L$  may not be “complete” in the sense that given a set  $L = \{(\mathbf{i}, k)\}$  representing a key lattice, it may be possible to infer the keys assigned to other indices on the lattice (i.e., points not in  $L$ ). Below we illustrate the possible inferences depend on the choice of the **KeyRoll** function. Consider the case where **KeyRoll** is defined using XOR, then knowing the key at  $\mathbf{i}$  and a succeeding key at  $\mathbf{i}' = \text{increment}(\mathbf{i}, d)$  allows us to derive the update  $\sigma$ , which may allow us to derive the keys at other lattice points  $\mathbf{j}$  such that  $\mathbf{j}^{(d)} = \mathbf{i}^{(d)}$ .

The function **Computable**( $L, E$ )  $\rightarrow L'$  outputs all the computable lattice points  $L'$  given the original lattice  $L$  and a set of updates  $E = \{(d, j, x)\}$ , where  $d \in [n]$  is the dimension,  $j$  is an index and  $x$  is the argument to **KeyRoll**.

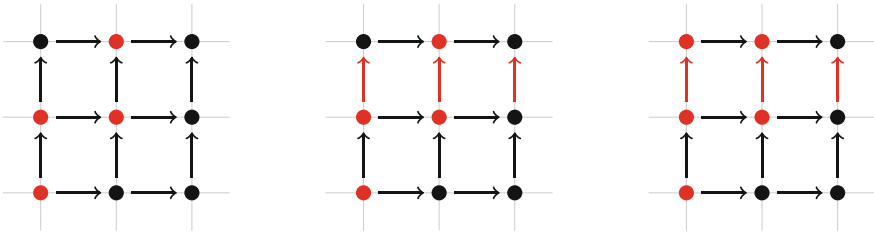
Two examples below illustrate the dependence of **Computable** on the properties of **KeyRoll**. Figure 3 illustrates how **Computable** works if a **KeyRoll** function is not one-way. Figure 4 illustrates the difference when **KeyRoll** is one-way.

The function **Computable**( $L, E$ ) can be realized as follows:

1. Interpret the lattice  $L$  as a directed graph  $\mathcal{G}$ . Initially this graph has no edges, only vertices from  $L$ .
2. Add every edge from  $E$  to the graph. Recall that every edge in  $E$  corresponds to multiple edges in  $\mathcal{G}$ . Specifically,  $e = (d, j, x)$  describes all edges that begin with a vertex  $(\dots, j, \dots)$  and end with a vertex  $(\dots, j + 1, \dots)$  where  $j$  and  $j + 1$  are on the  $d$ th position, and each edge is labeled with the update  $x$ .



**Fig. 3.** Suppose the red keys in the figure on the left are revealed in a key lattice. If the KeyRoll function is unpredictable but not one-way, then knowledge of a pair of adjacent keys would reveal all edges (updates) in the corresponding row or column, as shown in the middle figure. These inferred edges lead to additional computable keys (colored in red) in the right figure. (Color figure online)



**Fig. 4.** Begin with the same lattice as in Fig. 3 but assume that KeyRoll is one-way. The lattice points in the left figure do not allow us to compute a new lattice with more keys. Given additional information on the edges in the middle figure, one additional lattice point is computable (top left in the right figure).

3. Traverse  $\mathcal{G}$  from the origin. For every pair of predecessor-successor vertices  $(u, v)$  where  $u \neq \perp$  and  $v = \perp$ , if there exists an edge labeled with  $x$  connecting  $u$  to  $v$ , then compute  $k_v \leftarrow \text{KeyRoll}(k_u, x)$ .
4. Similar as above, but traverse  $\mathcal{G}$  backwards and if there exist two predecessor-successor vertices  $(u, v)$  where  $k_u = \perp$  and  $k_v \neq \perp$  then compute  $k_u \leftarrow \text{KeyRoll}^{-1}(k_v, x)$ , where  $x$  is the label on the edge between  $u$  and  $v$ . Note that, if KeyRoll is one-way on its first input, then this step is omitted, as it is hard to compute  $u$  given  $x$  and  $v$ .

**Adding Keys:** Parties may update the key lattice using  $\text{Update}(L, e) \rightarrow L'$  which takes a key lattice  $L$  and an update  $e = (d, j, x)$  and returns a new key lattice  $L'$  as follows:

- Let  $D = \{\mathbf{i}_m\}$  be all  $d$ -maximal index vectors in  $L$ .
- Output a new lattice  $L'$  with additional points defined by  $(\text{increment}(\mathbf{i}), \text{KeyRoll}(k_{\mathbf{i}}, x))$  for all  $\mathbf{i} \in D$ .

Note that since the lattice points included in  $D$  are  $d$ -maximal, all keys in  $\text{increment}(\mathbf{i}, d+1)$  are  $\perp$  in the original lattice  $L$ . One can think of this operation as (possibly) adding keys to the lattice based on  $e$ .

In the key graph interpretation of the lattice, **Update** looks at the largest index  $i$  for which a key is defined in dimension  $d$ , and labels every edge from  $i$  to  $i+1$  in dimension  $d$  (holding every other dimension constant) with update  $e$ .

**Forgetting Keys:** A key lattice is an infinite object. To manage memory requirements, (and looking ahead, to provide FS) we remove keys from a party's local version of the key lattice. The function  $\text{Forget}(L, \mathbf{i}) \rightarrow L'$  takes a key lattice  $L$  and an index vector  $\mathbf{i}$ , and returns a new lattice  $L'$  such that all keys in index vectors  $\mathbf{i}'$  such that  $\mathbf{i}' < \mathbf{i}$ , are set to  $\perp$ . Implicitly, **Forget** also deletes from a party's state all of the edges leading to vertices that have been forgotten.

We use windowing to limit state expansion and provide FS (Sect. 1.2). When we write  $\text{Forget}(L, w) \rightarrow L'$ , then **Forget** works as follows, where  $w$  is the window parameter. We call  $\mathbf{i}_w$  below the *threshold index vector*.

- For every dimension  $d \in [n]$ , let  $i_d$  the maximum  $j$  such that there is a key defined in  $L$  at index  $j$  in dimension  $d$ .
- Let  $\mathbf{i}_w$  be an index vector such that for every  $d \in [n]$ ,  $\mathbf{i}_w^{(d)} = \max(0, i_d - w)$ .
- Execute  $\text{Forget}(L, \mathbf{i}_w)$  and return the new lattice  $L'$ .

### 3.4 Key Lattice as a Key Management Technique

The key lattice is enough to build a concurrent group messaging protocol from existing primitives such as pairwise channels. The following generic approach uses a key lattice to build concurrent group messaging using three building blocks: (1) an initial group key, (2) secure pairwise channels between all parties in a group and (3) an AEAD scheme for sending payload messages.

- Given the initial group key  $k_0$ , the parties initialize their key lattice with  $(\mathbf{0}, k_0)$ , and assign  $\perp$  to the key at every other lattice point.
- If a party at index  $d \in [n]$  updates the key for the  $j$ th time, it samples  $x \xleftarrow{\$} \mathcal{X}$  and sends  $(d, j, x)$  using the secure pairwise channels.
- Upon receiving  $(d, j, x)$  the receiver adds key  $k' \leftarrow \text{KeyRoll}(k, x)$  to the lattice at point  $\mathbf{i}'$ , where  $k$  is the maximal key in the lattice and is located at point  $\mathbf{i}$ , and  $\mathbf{i}' \leftarrow \text{increment}(\mathbf{i}, d)$ .
- When a party at index  $d \in [n]$  sends an application message, it encrypts the message using the maximal key  $k$  in its local key lattice and sends the ciphertext to the group members (without using secure pairwise channels). The ciphertext is encrypted using AEAD where the associated data is the lattice index corresponding to the key used to encrypt the message.
- Upon receiving the ciphertext encrypting a payload message, the receiver checks whether it has the key in the key lattice required to decrypt. If so, then the receiver decrypts it immediately. Otherwise, the receiver buffers the message until it receives sufficient information to decrypt.

- Storing all the keys that are in the key lattice is expensive and trades off forward security. Every party runs  $\text{Forget}(L, w)$  for its lattice  $L$  and the window parameter  $w$  every time the party processes an update message.

## 4 Group Key Agreement

To agree on the very first shared key we use an existing group key agreement (GKA) protocol. There are many definitions of security of GKA protocols; for our purposes we adapt the one from [14] as it captures strong-forward secrecy and a strong corruption model. For our GM protocol to be asynchronous, the GKA subprotocol must also be asynchronous; this is true for the model of [14].

In this section, we reproduce the definition and introduce a few syntactic tweaks. For the full security definition we refer the reader to the full version [21]. The GKA will be used to construct our GM protocol in Sect. 6.

**Definition 4.1 (Group Key Agreement).** *We use  $G \subseteq \mathcal{P}$  to denote some group of players that participate in the protocol. Each party  $U \in \mathcal{P}$  is assumed to already have a long term public/private key pair  $(\text{pk}_U, \text{sk}_U)$ . We assume a PKI exists and the public keys are available to all parties.*

*The protocol consist of two stateful algorithms.*

- $\{m_V\}_{V \in G} \leftarrow \text{GKA.Init}(G)$ : Initialize an instance of the GKA protocol for a group  $G$  and return a set of responses, one for every party in  $G$ .
- $\{m_V\}_{V \in G} \leftarrow \text{GKA.Recv}(M)$ : Process message  $M$  and return a set of responses.

*The GKA outputs done with a key  $k$  to notify that the protocol completed.*

## 5 Group Randomness Messaging

We present the group randomness messaging (GRM) abstraction through which the parties communicate update messages. The main functionality is to send authenticated data and a ciphertext encrypting a random key update to all members in the group using pairwise channels. We require the pairwise channels to have FS & PCS properties.

**Definition 5.1 (Group Randomness Messaging (GRM)).** *Consider the player executing the protocol is  $U$ , a GRM scheme consists of three stateful algorithms.*

- $\{c_{U,V}\}_{V \in G} \leftarrow \text{GRM}_U.\text{Init}(k, w, G)$ : initialize the GRM instance using the initial key  $k$ , the window size  $w$ , and the group members  $G$ .  
*This step initializes the internal state  $\text{state}_{U,i}$ . The output is a set of ciphertexts, one for every player in  $G$ .*
- $\{c_{U,V}\}_{V \in G} \leftarrow \text{GRM}_U.\text{Evolve}()$ : output a ciphertext  $c_{U,V}$  for every  $V \in G$ .



- $\sigma_{V,U} \leftarrow \text{GRM}_U.\text{Recv}(c_{V,U})$ : process the ciphertext  $c_{V,U}$ , update the internal state and return the plaintext  $\sigma_{V,U}$  if the decryption is successful. If decryption is unsuccessful, return  $\perp$ .

In the above definition,  $\sigma_{V,U}$  is a triple  $(U, j, x)$  where  $U$  is the identity of the sender,  $j$  is a positive integer and  $x \in \mathcal{X}$ . The full version [21] discusses the correctness and security definitions for GRM.

## 5.1 Instantiation

We instantiate GRM using PKEAD. In essence, every party keeps a queue of  $w$  public and secret key-pairs. This queue is updated every time the party calls Evolve by dropping the oldest keypair and adding a new one. Each party  $U$  also maintains a public key for every other party  $V$  which is updated whenever  $U$  receives the output of  $V$ 's Evolve.  $U$  uses this public key in order to encrypt messages to  $V$ .  $U$  also maintains an integer  $j_V$  that tracks the index of the latest public key  $U$  has received from  $V$ .

This initial message sent by each party is a pair  $(\text{pk}_U^0, m)$ , where  $\text{pk}_U^0$  is the party's initial ephemeral public key,  $m$  is a MAC on the public key using the key  $k$  provided as input to Init. Where  $k$  is the key output by a GKA execution, this effectively “ties” a GRM to the GM application that uses it, as the MAC links the output  $k$  of a GKA session with the GRM session that will be used to evolve the key.

On a high level, the protocol achieves PCS because public keys are cycled over time and FS because old keys are dropped. Our construction is detailed below. Let the set  $\mathcal{X}$  to be domain from which updates are randomly sampled.

- $\text{GRM}_U.\text{Init}(k, w, G)$ : Generate an ephemeral key pair  $(\text{pk}_U^0, \text{sk}_U^0)$ . Initialize  $\text{state}_U.\text{sks} = \{\text{sk}_U^0\}$  and  $\text{state}_U.\text{pks} = \emptyset$ , and save  $w$  as the window parameter. Compute  $m \leftarrow \text{MAC}(\text{pk}_U^0; k)$ , where  $k$  is the input key,  $\text{pk}_U^0$  is the message and MAC is a cryptographic MAC scheme. Send the same message  $(\text{pk}_U^0, m)$  to every member in  $G$ .
- $\text{GRM}_U.\text{Evolve}()$ :
  1. A new private key  $\text{sk}_U^{j+1}$  is generated, along with its public key  $\text{pk}_U^{j+1}$ .
  2. Sample  $x \xleftarrow{\$} \mathcal{X}$  and let  $\sigma \leftarrow (U, j+1, x)$ , where  $j$  is the index of the latest secret key in  $\text{state}_U.\text{sks}$ .
  3. Repeat the steps below for every  $V \in G$  (including  $U$ ).
    - If the public key of the receiver  $V$  is not known, abort.
    - Call  $(c, t) \leftarrow \text{PKEAD}.\text{Enc}(\text{pk}_U^{j+1} \parallel \sigma, j_V; \text{pk}_V^{j_V})$  and then set  $c_{U,V} \leftarrow (c, t, j_V)$ . Note that  $\text{pk}_V^{j_V}$  can be found in  $\text{state}_U.\text{pks}$  and  $j_V$  is the index of the public key associated with  $V$ .
  4.  $\text{state}_U$  is updated as follows.
    - Add  $\text{sk}_U^{j+1}$  to  $\text{state}_U.i.\text{sks}$
    - If  $|\text{state}_U.\text{sks}| > w$ , remove the oldest one (i.e.,  $\text{sk}_U^{j-w}$ ).

- $\text{GRM}_U.\text{Recv}(c_{V,U})$ : There are two possible message formats. The message output by  $\text{Init}$  is an ephemeral public key  $\text{pk}_V^0$  with a  $\text{Mac}$ ; if the message is this type, then verify the  $\text{Mac}$  using the key  $k$  provided to  $\text{Init}$ <sup>8</sup> and then set  $V$ 's public key in  $\text{state}_U.\text{pks}$  to be  $(0, \text{pk}_V^0)$ . All other messages are handled as follows.
  1. Parse the message  $c_{V,U}$  as  $(c, t, j)$ , where  $j$  is an index into the current user  $U$ 's secret key.
  2. Find secret key  $\text{sk}_U^j$ . Abort the protocol if it does not exist.
  3.  $\text{pk}_V^{j_V} \parallel \sigma_{V,U} \leftarrow \text{PKEAD.Dec}(c, t, j; \text{sk}_U^j)$ , abort if this step returns  $\perp$ .
  4. Add or update  $V$ 's public key in  $\text{state}_U.\text{pks}$  to be  $(j, \text{pk}_V^{j_V})$ .
  5. Let  $j_{\min}$  be the smallest  $j$  in  $\{(j, \text{pk}_V^{j_V}) : V \in G\}$ .
  6. Delete all secret keys  $\text{sk}_U^j$  where  $j < j_{\min}$ .
  7. Return  $\sigma_{V,U}$

**Theorem 5.1.** *Let  $\mathcal{A}$  be an adversary against the GRM game, let  $\mathcal{B}$  be an adversary against the PKEAD game, and let  $\mathcal{C}$  be an adversary against the MAC EUF-CMA game. Then*

$$\text{Adv}_{\mathcal{A}}^{\text{grm}} \leq n_S \cdot \text{Adv}_{\mathcal{C}}^{\text{mac}} + 2 \cdot |Q|_{\max} \cdot n_Q \cdot \text{Adv}_{\mathcal{B}}^{\text{pkead}}.$$

where  $|Q|_{\max}$  is the upperbound for the number of oracles in a group,  $n_Q$  is the upperbound of the number of queries to the encryption oracle that  $\mathcal{B}$  makes on behalf of  $\mathcal{A}$  for the instance under test, and  $n_S = \text{poly}(\lambda)$  is the maximum number of concurrent GRM sessions that  $\mathcal{A}$  is allowed to invoke in its security game.

For a proof of this theorem see the full version [21].

## 6 Group Messaging

We define group messaging as a protocol which establishes and evolves a lattice of keys. Parties may additionally send messages encrypted under the group keys, which must be decrypted successfully by the other group members.

Our definition of group messaging assumes the existence of a Group Key Agreement (GKA) primitive (Sect. 4).

**Definition 6.1 (Group Messaging).** *A group messaging protocol consists of five stateful algorithms defined as follows:*

- $\text{GM.Init}(G, w)$ : Initialize the protocol with group  $G \subseteq \mathcal{P}$  and the windows size  $w$ . Output a set of messages, one for each party in  $G$ .
- $\text{GM.Evolve}()$ : Outputs a set of update messages, one for each party in  $G$ .
- $\text{GM.Recv}(M)$ : Processes the message  $M$  (e.g., from the network), and outputs a response.
- $\text{GM.Enc}(m)$ : Encrypts a plaintext  $m$  and outputs a ciphertext.
- $\text{GM.Dec}(c)$ : Decrypts ciphertext  $c$  and outputs a plaintext.

<sup>8</sup> If verification fails due to trying the wrong key from multiple concurrent sessions, return  $\perp$  and process the incoming message via  $\text{Recv}$  of a different session.

## 6.1 Security Definition

The security of GM is modeled via a game between a challenger and an adversary, where the key lattice tracks the evolution of the group key(s) over time. Our freshness definition specifies the conditions under which a particular state (in our case the state is a key in the key lattice) is not compromised by the adversary. Contrary to the definitions of freshness in other key agreement works (e.g., [4, 19]), we state freshness below with respect to a specific lattice point.

The adversary invokes oracles  $\Pi_{U,i}^{\text{gm}}$  where  $U$  is a group member and  $i \in [1, \dots, n_S]$ , where the subscript  $i$  denotes a specific instance of the oracle that belongs to party  $U$ . Different instances that belong to the same party may share long-term keys, e.g., identity keys. The adversary invokes the oracles arbitrarily as long as it follows the constraints described in Sect. 2.

We assume there is an instance of the GKA oracle running under every GM oracle. This method allows us to inherit the partnering definition and many oracle queries. Nevertheless, our description is self-contained since we reproduce the common oracle queries in the GM definition. Additional details of the GKA can be found in Sect. 4.

Each oracle  $\Pi_{U,i}^{\text{gm}}$  maintains internal variables to track each party's view of the key lattice and the group messages that have been received by that party. They also collectively maintain global state that tracks which elements of the key lattice and which key updates have been explicitly revealed to the adversary. We denote by  $L_{\text{sid}}^{\text{rev}}$  the key lattice describing all keys (points on the lattice) which are revealed to the adversary, and we denote by  $E_{\text{sid}}^{\text{rev}}$  the set of key updates, modeled as edges in the graphical interpretation of the key lattice, which are revealed to the adversary.  $S_{\text{sid}}^{\text{rev}} = (L_{\text{sid}}^{\text{rev}}, E_{\text{sid}}^{\text{rev}})$  denotes all of the key material that is revealed to the adversary in some session  $\text{sid}$ . The session ID  $\text{sid}$  is a unique identifier for the group members who have successfully completed the initial group key agreement and established a session (described in detail in Sect. 4 since it is a property inherited from GKA). Indeed,  $\text{sid}$  is not defined when a GKA session begins, but this is not an issue since the session's lattice is instantiated only after the session is established. The full information on the key lattice available to the adversary is given by  $\text{Computable}(L_{\text{sid}}^{\text{rev}}, E_{\text{sid}}^{\text{rev}})$ . We remark that the session ID ( $\text{sid}$ ) is not the same as the instance ID. The instance of an oracle, e.g.,  $(U, i)$ , is established when the oracles are initialized, but the session ID is only established some time later, after the oracles are ready to evolve keys.

Specifically, the oracles maintain the following state:

- $\delta_{U,i} \in \{\text{pending}, \text{accept}, \text{abort}\}$  indicates whether the oracle is ready to start evolving keys.
- $L_{U,i}$  represents the key lattice maintained by oracle  $\Pi_{U,i}^{\text{gm}}$ . We use the language from Sect. 3 to describe the key lattice.
- $\text{state}_{U,i}$  is the remaining state that the implementation may keep. (For our protocol, this includes  $E_{U,i}$ , a set of edges between lattice points, as well as the state held by underlying subprotocols.)
- $S_{\text{sid}}^{\text{rev}} = (L_{\text{sid}}^{\text{rev}}, E_{\text{sid}}^{\text{rev}})$  represents the key lattice  $L_{\text{sid}}^{\text{rev}}$  containing all the revealed keys by the adversary as well as the revealed updates  $E_{\text{sid}}^{\text{rev}}$  in session  $\text{sid}$ .

The full details of the GM oracles are specified below.

- $\Pi_{U,i}^{\text{gm}}.\text{Init}(G, w)$ : Initialize an instance of the GM protocol for the group members in  $G$  where  $U \in G$  and  $w$  is the window size. Set  $\delta_{U,i} = \text{pending}$  and return a hash function  $H$ . The response is returned to the adversary.
- $\Pi_{U,i}^{\text{gm}}.\text{Corrupt}()$ : Return the long-term secret to the adversary.
- $\Pi_{U,i}^{\text{gm}}.\text{Reveal}()$ : If  $\delta_{U,i} \neq \text{accept}$  then return  $\perp$ . Otherwise, return the set of keys that are computable from  $L_{U,i}$ , and add these keys to  $L_{\text{sid}}^{\text{rev}}$
- $\Pi_{U,i}^{\text{gm}}.\text{StateReveal}()$ : If  $\delta_{U,i} \neq \text{accept}$  then return  $\perp$ . Else, return the internal state  $\text{state}_{U,i}$ , excluding the computable keys  $L_{U,i}$ .<sup>9</sup>
- $\Pi_{U,i}^{\text{gm}}.\text{Evolve}()$ : If  $\delta_{U,i} = \text{abort}$  then return  $\perp$ . Else, return a set of message  $\{M_V\}_{V \in G}$ .
- $\Pi_{U,i}^{\text{gm}}.\text{Recv}(M)$ :
  - If  $\delta_{U,i} = \text{abort}$  then this call does nothing.
  - Otherwise process the message, optionally update the state  $\text{state}_{U,i}$  and the key lattice  $L_{U,i}$ . Return a set of messages  $\{M_V\}_{V \in G}$ . The input  $M$  should be from either the output of  $\text{Recv}$  or  $\text{Evolve}$ .
- $\Pi_{U,i}^{\text{gm}}.\text{Dec}(c)$ : Use the available internal state to decrypt the ciphertext  $c$  and output the plaintext. If the oracle does not have enough information to decrypt the message, then it is buffered.
- $\Pi_{U,i}^{\text{gm}}.\text{Enc}(m)$ : Encrypts the plaintext  $m$  using the maximal key in  $L_{U,i}$  and returns a ciphertext.
- $\Pi_{U,i}^{\text{gm}}.\text{Test}(m_0, m_1)$ : This is defined in the security game below.

By execution of  $\text{Corrupt}$ ,  $\text{Reveal}$  and  $\text{StateReveal}$  queries the adversary can learn the entire secret internal state of the oracle  $\Pi_{U,i}^{\text{gm}}$ . Specifically,  $\text{Reveal}$  gives the party’s current group keys, and  $\text{StateReveal}$  gives the party’s internal state except for what is provided by the former two queries.  $\text{Corrupt}$  gives the party’s long-term public key and secret key (from the PKI); because this is only used for the GKA protocol, which we require to be forward secure, this reveals the initial group keys in *future* GKA executions. Also note that the above gives the adversary a decryption oracle via  $\text{Dec}$ .

**Modeling Pairwise Channels in the Oracle Game:** In our general oracle game, the adversary is permitted to invoke the oracles in any order, which models an asynchronous network. However, to describe the guarantees that the protocol achieves when windowing, we define a syntactic model to describe the messages sent “between parties” in the oracle game. Specifically, between every ordered pair of parties  $(U, V)$  the adversary maintains a special buffer  $\mathcal{C}_{U,V}$  called a *channel* representing the pairwise connection between  $U$  and  $V$ . When an oracle query returns a message  $c$  to be sent from  $U$  to  $V$ , the adversary places  $(c, n)$  into  $\mathcal{C}_{U,V}$ , where  $n$  is an integer recording that  $c$  is the  $n$ th message placed into the channel.

<sup>9</sup> For our construction, this adds all of the edges in  $E_{U,i}$  to  $E_{\text{sid}}^{\text{rev}}$ .

In the above game description, each oracle provides three queries to generate messages to other parties.  $\Pi_{U,i}^{\text{gm}}.\text{Enc}(m)$  encrypts a message using the oracle's latest key and returns a ciphertext which is forwarded to all other parties. Whenever a  $\Pi_{U,i}^{\text{gm}}.\text{Enc}(m)$  query is made, the returned message  $c$  is simultaneously put into the channels  $\mathcal{C}_{U,V}$  for all  $V \in G$ .  $\Pi_{U,i}^{\text{gm}}.\text{Evolve}()$  generates a key evolution, but returns a *different message* for each other party in the execution. Similarly,  $\Pi_{U,i}^{\text{gm}}.\text{Recv}(M)$  may output a different message for every other party in the execution, but it may also output no messages. Whenever a  $\Pi_{U,i}^{\text{gm}}.\text{Evolve}()$  or  $\Pi_{U,i}^{\text{gm}}.\text{Recv}(M)$  query is made, the oracle returns a list of ciphertexts  $c_V$ , one for each  $V \in G$ . Each of these messages is immediately placed into the corresponding channel  $\mathcal{C}_{U,V}$  along with its index.

A message  $c$  generated by an  $\text{Enc}$  query is removed from its corresponding buffer only when it is input to a corresponding oracle  $\Pi_{V,j}^{\text{gm}}.\text{Dec}(c)$ . A message  $c$  generated by an  $\text{Recv}$  or  $\text{Evolve}$  query is removed from its corresponding buffer only when it is input to a corresponding oracle  $\Pi_{V,j}^{\text{gm}}.\text{Recv}(c)$ . Note that if an oracle receives a message that it cannot yet process due to reordering of messages over a pairwise channel, then the oracle is expected to buffer the message until it can process the message, and return the result once it can process the message.

The adversary may additionally invoke  $\text{Recv}$  or  $\text{Dec}$  oracles on messages that have not been placed in channels but instead were adversarially generated. These actions do not affect the channels.

**Partnering:** For group messaging, *partnering* is analogous to the case for GKA. Intuitively, a group in a GKA protocol is partnered if the parties participate in the same session and agreed on the same group key. For group messaging, parties are partnered if they are running a protocol with each other to agree on a lattice of group keys.

**Definition 6.2 (Partnering).** *Given a group  $G \subseteq \mathcal{P}$  and a set of pairs  $Q = (U, i_U)_{U \in G}$  defining associated oracles  $\Pi_{U, i_U}^{\text{gm}}$ , we say the oracles are partnered if the underlying GKA oracles  $\Pi_{U, i_U}^{\text{gka}}$  are partnered.*

For some security parameter  $\lambda$  we define a security game for the adversary  $\mathcal{A}$ , this consists of the set of participants  $\mathcal{P}$  where  $n$  (the number of participants) is a polynomial function of  $\lambda$ , as is the maximum number of sessions per participant  $n_S$ . Thus the number of oracles  $\Pi_{U, i}^{\text{gm}}$  is also a polynomial function of  $\lambda$ . The adversary  $\mathcal{A}$  is given at the start of the game all the public keys  $\text{pk}_U$  for  $\text{pk} \in \mathcal{P}$  and it interacts with the oracles  $\Pi_{U, i}^{\text{gm}}$  via the sequence of oracle queries as above.

**Freshness:** We now define freshness for our game. Intuitively, we say that a key is *fresh* if it has not been revealed to the adversary, either explicitly via  $\text{Reveal}$  queries, or implicitly, via a combination of  $\text{Reveal}$  and  $\text{StateReveal}$  queries. The global state  $S_{\text{sid}}^{\text{rev}}$  tracks the keys computable by the adversary, and a key is fresh if and only if it is not computable from  $S_{\text{sid}}^{\text{rev}}$ .

**Definition 6.3 (Freshness).** *In a session  $\text{sid}$ , a key  $k_{\mathbf{i}^*}$  with at index  $\mathbf{i}^*$  is fresh if and only if it is not computable from  $S_{\text{sid}}^{\text{rev}}$  using the Computable function, as defined in the group messaging definition (Definition 6.1).*

Depending on when the adversary invokes **Corrupt** on a party and learns its long-term secret key, the adversary might learn *all* messages delivered to that party, and any such key or update material is included in  $S_{\text{sid}}^{\text{rev}}$ . Therefore, keys that the adversary can learn from messages delivered to this party are not fresh.

**Security Game:** The security game tries to break the semantic security of a message sent between the parties. It runs in two phases, the division between the two phases is given by the point in which the adversary executes a **Test** query.

- **Phase 1:** All queries can be executed without restriction.
- **Test Query:**  $\Pi_{U,i}^{\text{gm}}.\text{Test}(m_0, m_1)$ : Given two equal length messages  $m_0$  and  $m_1$ , if  $k_U$  is fresh, where  $k_U$  is the maximal key of instance  $(U, i)$ , then the challenger selects a bit  $b \in \{0, 1\}$  and applies  $\Pi_{U,i}^{\text{gm}}.\text{Enc}(m_b)$ , returning the output  $\text{ct}^*$  to the adversary. We denote the test oracle by  $\Pi_{U^*,i^*}^{\text{gm}}$ . We call  $\mathbf{i}^*$  the test index.
- **Phase 2:** All queries can be executed except for:
  1. Any query that would add  $k_{\mathbf{i}^*}$  to the set of keys computable from  $S_{\text{sid}}^{\text{rev}}$ .
  2. If  $\text{ct}^*$  is at any point processed by  $\text{Dec}(\text{ct}^*)$ , by the oracles, then the result is not returned to the adversary but the game still continues.

At the end of the game, the adversary  $\mathcal{A}$  needs to output its guess  $b'$ , and wins the game if  $b = b'$ . We define  $\text{Adv}_{\mathcal{A}}(\lambda) = 2 \cdot |\Pr[b = b'] - 1/2|$ .

**Definition 6.4 (Security of Group Messaging).** *A GM scheme is secure if for any probabilistic polynomial time adversary  $\mathcal{A}$  the advantage  $\text{Adv}_{\mathcal{A}}(\lambda)$  is negligible in the security parameter  $\lambda$ .*

Thanks to the underlying key lattice, our security game captures FS and PCS at the same time in a natural way. Specifically, queries to **Reveal**, **StateReveal** or **Corrupt** during **Phase 1** are used for capturing PCS and queries to these oracles during **Phase 2** captures FS. Unlike prior definitions [4, 33] we do not need to use epochs or separate definitions for PCS and FS.

**Correctness.** Intuitively, a GM protocol is correct if every message that is encrypted with the group key is correctly decrypted by every recipient. We write the formal definition with respect to the oracles defined for our security game. Our definition of correctness requires all encrypted messages must eventually be correctly decrypted under a property called “well-ordered execution” which we define as well.

**Definition 6.5 (Correctness of Group Messaging).** *A GM protocol is correct if in every infinite execution by every PPT adversary  $\mathcal{A}$  who is allowed to query the GM oracles except **Corrupt**, **StateReveal**, **Reveal** and **Test** and must*

deliver all messages, for all  $U, i$ , for all  $c \leftarrow \Pi_{U,i}^{\text{gm}}.\text{Enc}(m)$ , and for all  $V \in G \setminus \{U\}$  there exists a  $j$  and an oracle call  $m' \leftarrow \Pi_{V,j}^{\text{gm}}.\text{Dec}(c)$  such that  $(U, i)$  is partnered with  $(V, j)$  and  $m' = m$ .

Recall that when we apply windowing, some party may be forced by the protocol to discard the group key used to decrypt a message that has still not been delivered to it. To facilitate our analysis of correctness when windowing, we define an ordering property of an execution that describes how many times a party may evolve the group key between the moment it sends a message and that message is delivered.

*$\omega$ -Well-Ordered Execution.* Recall that our oracle game tracks the order in which messages are returned from oracles to be sent to other parties via our abstraction of pairwise channels, and that the adversary may delay and reorder messages sent via the pairwise channels. A channel is  *$\omega$ -well-ordered* if the  $n$ th message sent over  $C$  is removed from the channel before the  $(n + \omega)$ th message (via delivery to the correct oracle), for all  $n \in \mathbb{N}$ . An execution is  *$\omega$ -well-ordered* if all pairwise channels are  $\omega$  well-ordered.

We claim that when windowing with our protocol, for any  $\omega$ -well-ordered execution, if the window parameter  $w$  is greater than or equal to  $\omega$ , then the protocol is correct. The proof is trivial by construction of the protocol. When  $w < \omega$ , windowing may force some decryption keys to be purged before the corresponding message is delivered.

*Remark 6.1 (Well Ordering and Network Synchrony).* Well-ordering is a strict relaxation of network synchrony that depends on ordering messages rather than on time. In a synchronous network, a delay parameter of  $\Delta$  implies  $\Delta$ -well-ordered channels; therefore, setting  $w = \Delta$  implies correctness. If the network is asynchronous, then  $w$  must be set to  $\infty$  to guarantee correctness. However, this sacrifices forward secrecy, as parties may store old group keys indefinitely.

## 6.2 GM from GRM and GKA

We first present our construction of GM from GKA, GRM, and a CCA-secure AEAD scheme; we then prove security of GM based on the underlying primitives.

**Protocol Overview.** In our construction of a group messaging protocol, parties maintain local versions of a global key lattice in order to track the group key. They then encrypt and decrypt messages using keys from the lattice, and they update the group key by adding new keys on the key lattice. Our protocol uses the above primitives to initialize their key lattices, encrypt and decrypt messages using the keys in the lattice, send updates to the group key, and remove keys from their lattices. Specifically, each party maintains a local key lattice  $\mathcal{L}$ , a local set of key updates  $\mathcal{E}$ , and a buffer  $\mathbf{B}$  of unprocessed messages, which contains both GRM messages that it cannot yet process and application messages that it is not yet able to decrypt. Every update  $e \in \mathcal{E}$  has the form  $(d, i, x)$  where

$d \in [n]$  corresponds to the dimension of the party that generates the update,  $i$  is an index and  $x$  is key transformation data. Parties also maintain a list of index vectors  $\mathcal{I} \in (\mathbb{N}^n)^n$  that tracks each party's view of the current key of every other party, which is used to optimistically exclude keys from its state.

*Message Headers and the Recv Subprotocol.* We make the distinction between *protocol messages* and *application messages*. Protocol messages in GM are either GKA messages (to agree on an initial group key) or GRM messages (to evolve the group key). Application messages are encryptions under some group key.

Our construction uses a single `Recv` function to process every incoming protocol message, provided in Fig. 7, which directs the incoming message to the appropriate subprotocol (either GKA or GRM). To help distinguish between GKA protocol messages and GRM protocol messages in the descriptions of the protocols and the proofs, we say that a message is a “GKA message” if it contains a prefix `gka`, and a message is a “GRM message” if it contains a prefix `grm`. In an implementation, these headers can be encoded as flags. Where the context is clear, we elide these prefixes from the exposition.

**Initialization:** When a group of parties begin a GM protocol, they initialize the execution via `GM.Init()`, which is described in Fig. 5. Each party saves the set of other parties in the protocol and the window parameter. They also agree on a hash function  $H$  described below, which is a public parameter. The parties then run GKA in order to agree on an initial group key. Note that the key lattice and GRM is *not* initialized yet; they can only be initialized after the GKA outputs the initial key as shown in Fig. 6.

**Sending and Receiving Key Updates:** Our GM construction uses GRM as a transport for generating and communicating random key updates. In Fig. 6 and Fig. 7 we specify how parties generate new key updates and process updates from other parties, respectively.

Specifically, parties invoke `GRM.Evolve()` to receive a random key update  $\sigma$  along with an encryptions of the update to send to each other party via pairwise channel. The calling party adds  $\sigma$  to its set of edges  $\mathcal{E}$  and computes any possible new points in  $\mathcal{L}$ . When a party receives a key update, it calls `GRM.Recv()` on the update, and if a key update is returned then it adds the update as an edge in  $\mathcal{E}$  and computes any possible new keys in  $\mathcal{L}$ . If it cannot yet decrypt the key update, it buffers the message.

**Encrypting and Decrypting a Message:** Whenever a party wishes to encrypt a message  $m$  using the group key, it calls `GM.Enc` using the maximal key in its key store. Specifically, we require a hash function  $H: \mathbb{K} \rightarrow \mathbb{K}$ , that maps from the keyspace of the key lattice to the keyspace for a CCA-secure



AEAD encryption scheme.<sup>10</sup> When a party encrypts a message, it provides the hashed key corresponding to the maximal index  $\mathbf{i}$  in its key lattice  $\mathcal{L}$  as input to `AEAD.Enc`, and it includes the index  $\mathbf{i}$  as associated data. The encrypting party then forwards the encrypted message to every other party.

When a party seeks to decrypt a message, it looks up the corresponding key (the index of which is found in associated data), and supplies the hashed key to `AEAD.Dec`. When a party receives an encrypted message, it checks whether the index of the key used to encrypt is in `Computable`( $\mathcal{L}, \mathcal{E}$ ). If so, it uses the key at that index to decrypt the message. If not, it adds the message to the buffer `B`. The implementations of encryption and decryption are given in Fig. 8 and Fig. 9.

**Pruning the Key Lattice:** Parties continuously attempt to prune elements from their local state, both in order to manage the size of the state they keep, and also because deleting old keys facilitates forward secrecy. When a party knows that it will no longer receive any messages encrypted with keys below a particular key index  $\mathbf{i}$ , it optimistically prunes all such keys from its lattice via `Forget`( $\mathcal{L}, \mathbf{i}$ ). Additionally, if ever a key index exceeds the key window (keys whose index vector that are less than the threshold index vector  $\mathbf{i}_w$ ) it purges the key (and relevant updates) from  $\mathcal{L}$  (and  $\mathcal{E}$ ).

Whenever a party receives an encryption from a party  $V$ , it updates its index vector  $\mathcal{I}[\phi(V)]$  tracking the keys used by  $V$ . Recall that because our construction requires key updates to move toward higher lattice indices, the set of future indices is the union of the  $n$ -dimensional hyperplanes  $\mathcal{H}^* = \bigcup_{\mathbf{i}_V \in \mathcal{I}} \mathcal{H}_{\geq \mathbf{i}_V}$ . Any index *outside* this union represents an obsolete key, and the related keys are deleted via `Forget` in Fig. 9.

In summary, keys and edges that fall outside the window parameter are deleted as specified in Fig. 7. Keys and edges that will not be used in the future are deleted as specified in Fig. 9. This is possible because parties also send their maximal lattice point along with their message (in Fig. 8) so that the receiving party can compute the minimum view (lattice point) of all parties and delete keys and edges that are smaller than the minimum view.

On execution of `GM.Init()`, run `GKA.Init( $G$ )` and output the result. Note that  $U$  holds the long-term key pair  $(\mathbf{pk}_U^{lt}, \mathbf{sk}_U^{lt})$ .

**Fig. 5.** Algorithm for `GM.Init( $G, w$ )`

<sup>10</sup> This hash function's purpose is semantic to convert between types. We only require (informally) that if the adversary does not know  $k$  then it does not know  $H(k)$ . We elide discussion of  $H$  in the proof.

$U$  calls  $\{c_{U,X}\}_{X \in G} \leftarrow \text{GRM.Evolve}()$ , and outputs  $c_{U,X}$  to  $X$  for  $X \in G$ .

**Fig. 6.** Algorithm for  $\text{GM.Evolve}()$

If  $M$  is a GKA message:

- Compute  $\{m_{U,V}\}_{V \in G} \leftarrow \text{GKA.Recv}(M)$ , and output  $m_{U,V}$  to party  $V$  for  $V \in G$ .
- If GKA outputs done with a key  $k$ :
  - Initialize  $\mathcal{L}$  with the point  $(\mathbf{0}, k)$ .
  - Initialize a GRM execution via  $\{c_{U,V}\}_{V \in G} \leftarrow \text{GRM.Init}(k, w, G)$  and send  $c_{U,V}$  to  $V$  for  $V \in G$ .
  - Initialize an empty message buffer  $\mathbf{B} \leftarrow \emptyset$ .

If  $M$  is a GRM message received from party  $V$ :

1. Compute  $\sigma \leftarrow \text{GRM.Recv}(M)$ . If  $\sigma = \perp$ , then add  $M$  to  $\mathbf{B}$  and return. Otherwise, let  $(d, j, x) \leftarrow \sigma$ , add  $(d, j, x)$  to the set of edges  $\mathcal{E}$  and then compute  $\mathcal{L} \leftarrow \text{Computable}(\mathcal{L}, \mathcal{E})$ .<sup>a</sup>
2. Delete deprecated keys using  $\mathcal{L} \leftarrow \text{Forget}(\mathcal{L}, w)$ .
3. Delete deprecated edges from  $\mathcal{E}$  that precede the corresponding index in the threshold index vector (see Section 3.3). Specifically, suppose the threshold index vector is  $\mathbf{i}_w = (i_1, \dots, i_{n_S})$  and  $E = \{(d_k, j_k, x_k)\}_k$ , then remove all edges  $(d_k, j_k, x_k)$  where  $j_k < i_{d_k}$ .
4. While  $\mathbf{B}$  is not empty or  $\mathbf{B}$  has not changed from the previous iteration:
  - For every message  $M \in \mathbf{B}$ , execute  $\text{GM.Recv}(M)$

<sup>a</sup> A sanity check would be that  $d = \phi(V)$  and  $j$  should equal the  $d$ th element of the maximal index vector of  $\mathcal{L}$ .

**Fig. 7.** Algorithm for  $\text{GM.Recv}(M)$

Player  $U$  finds the  $\phi(U)$ -maximal lattice point  $\mathbf{i}$  in its local lattice  $\mathcal{L}$ , computes  $(\text{ct}, t) \leftarrow \text{AEAD.Enc}(m, U \parallel \mathbf{i}; H(k_i))$ , and then returns  $(\text{ct}, U \parallel \mathbf{i}, t)$ .

**Fig. 8.** Algorithm for  $\text{GM.Enc}(M)$

### 6.3 Concrete Costs

We give an estimate of our concrete communication cost for 128-bit security. Since the payload ciphertext form is  $(\text{ct}, U \parallel i, t)$ , the concrete communication cost for 32 bytes payload is  $32 + 3 \cdot 16 = 80$  bytes, assuming the identity  $U$ , the lattice point  $i$ , and the AEAD tag  $t$  are 128 bits. Additionally, the update ciphertext has the form  $(c, t, j)$  where  $c$  is a ciphertext encrypting  $\text{pk} \parallel U \parallel i \parallel x$ , under a public key encryption scheme, where  $\text{pk}$  is assumed to be 32 bytes and

Parse  $M$  as  $(\text{ct}, V\|\mathbf{i}, t)$ . If  $M$  is not of this form, return  $\perp$ . Then:

- If  $\mathbf{i} < \mathbf{i}_w$ , where  $\mathbf{i}_w$  is the threshold index vector, or if  $\mathbf{i} < \mathcal{I}[\phi(V)]$ , return  $\perp$ .
- Update  $\mathcal{I}[\phi(V)] \leftarrow \mathbf{i}$ , compute  $\mathbf{i}_{\min}$  as the index vector of the element-wise minimum of all  $\mathbf{i} \in \mathcal{I}$ , and then execute  $\mathcal{L} \leftarrow \text{Forget}(\mathcal{L}, \mathbf{i}_{\min})$ .
- Find the key at  $\mathbf{i}$  in  $\mathcal{L}$  using  $\text{Computable}(\mathcal{L}, \mathcal{E})$ , if  $k_{\mathbf{i}} = \perp$ , then add  $M$  to  $\mathcal{B}$  and return  $\perp$ .
- If  $k_{\mathbf{i}} \neq \perp$ , compute  $m \leftarrow \text{AEAD.Dec}(\text{ct}, V\|\mathbf{i}, t; H(k_{\mathbf{i}}))$ . If  $m = \perp$ , abort the protocol. Otherwise, return  $m$ .

**Fig. 9.** Algorithm for  $\text{GM.Dec}(M)$

$x$  is from the update space assumed to be 16 bytes. One update message is needed for every party in the group, for a group size of 128, the update cost is  $128 \cdot (32 + 5 \cdot 16) = 14.3$  KB. Our scheme uses less communication than Weidner et al. [33] which has a payload ciphertext cost of 139 bytes and an update cost of 39.6 KB in the same setting.

The storage overhead comes from the window parameter  $w$  and the group size  $N$ . Specifically, we need to maintain at most  $w$  update messages per party and only one key in the lattice at the minimum view. For a window size of 1,000 and 128 parties, the storage requirement would be just over 14 MB in the worst case which is insignificant in today’s devices.

### 6.4 Main Theorem

We now state our main theorem. The proof is in the full version [21].

**Theorem 6.1 (Security of Group Messaging).** *If  $\mathcal{A}$  is an adversary against the GM game, then there exist adversaries  $\mathcal{B}$ ,  $\mathcal{C}$ , and  $\mathcal{D}$  such that*

$$\text{Adv}^{\text{gm}}(\mathcal{A}) \leq 2n_S \text{Adv}^{\text{gka}}(\mathcal{B}) + 2n_S n_q \text{Adv}^{\text{grm}}(\mathcal{C}) + n_S n_q \text{Adv}^{\text{cca}}(\mathcal{D}),$$

where  $n_S = \text{poly}(\lambda)$  is the maximum the number of GM sessions  $\mathcal{A}$  may invoke, and  $n_q = \text{poly}(\lambda)$  is the maximum number of keys that  $\mathcal{A}$  may query in a session.

**Acknowledgments.** This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. FA8750-19-C-0502 (Approved for Public Release, Distribution Unlimited). The first and third author would also like to thank the FWO under an Odysseus project GOH9718N, and by CyberSecurity Research Flanders with reference number VR20192203.

The work of the first author was conducted whilst he was at KU Leuven, the third author whilst he was at SRI International, and the fourth author whilst he was a student at UC Irvine.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the funders. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

## References

1. Alwen, J., Auerbach, B., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K.: DeCAF: decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559 (2022). <https://eprint.iacr.org/2022/559>
2. Alwen, J., et al.: CoCoA: concurrent continuous group key agreement. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, May–June 2022, vol. 13276, pp. 815–844. Springer, Heidelberg (2022). [https://doi.org/10.1007/978-3-031-07085-3\\_28](https://doi.org/10.1007/978-3-031-07085-3_28)
3. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: security notions, proofs, and modularization for the signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-17653-2\\_5](https://doi.org/10.1007/978-3-030-17653-2_5)
4. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-56784-2\\_9](https://doi.org/10.1007/978-3-030-56784-2_9)
5. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-64378-2\\_10](https://doi.org/10.1007/978-3-030-64378-2_10)
6. Alwen, J., Hartmann, D., Kiltz, E., Mularczyk, M.: Server-aided continuous group key agreement. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022, November 2022, pp. 69–82. ACM Press (2022). <https://doi.org/10.1145/3548606.3560632>
7. Bienstock, A., Dodis, Y., Rösler, P.: On the price of concurrency in group ratcheting protocols. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 198–228. Springer, Cham (2020). [https://doi.org/10.1007/978-3-030-64378-2\\_8](https://doi.org/10.1007/978-3-030-64378-2_8)
8. Borisov, N., Goldberg, I., Brewer, E.: Off-the-record communication, or, why not to use PGP. In: Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, pp. 77–84 (2004)
9. Boyd, C., Mathuria, A., Stebila, D.: Protocols for Authentication and Key Establishment. Information Security and Cryptography. Springer, Heidelberg (2020). <https://doi.org/10.1007/978-3-662-58146-9>
10. Bresson, E., Chevassut, O., Pointcheval, D.: Provably authenticated group Diffie-Hellman key exchange—the dynamic case. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 290–309. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45682-1\\_18](https://doi.org/10.1007/3-540-45682-1_18)
11. Bresson, E., Chevassut, O., Pointcheval, D.: Dynamic group Diffie-Hellman key exchange under standard assumptions. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 321–336. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-46035-7\\_21](https://doi.org/10.1007/3-540-46035-7_21)
12. Bresson, E., Chevassut, O., Pointcheval, D., Quisquater, J.J.: Provably authenticated group Diffie-Hellman key exchange. In: Reiter, M.K., Samarati, P. (eds.) ACM CCS 2001, November 2001, pp. 255–264. ACM Press (2001). <https://doi.org/10.1145/501983.502018>
13. Bresson, E., Manulis, M.: Securing group key exchange against strong corruptions. In: Abe, M., Gligor, V. (eds.) ASIACCS 2008, March 2008. pp. 249–260. ACM Press (2008)

14. Bresson, E., Manulis, M., Schwenk, J.: On security models and compilers for group key exchange protocols. In: Miyaji, A., Kikuchi, H., Rannenberg, K. (eds.) IWSEC 2007. LNCS, vol. 4752, pp. 292–307. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-75651-4\\_20](https://doi.org/10.1007/978-3-540-75651-4_20)
15. Brzuska, C., Fischlin, M., Smart, N.P., Warinschi, B., Williams, S.C.: Less is more: relaxed yet composable security notions for key exchange. *Int. J. Inf. Sec.* **12**(4), 267–297 (2013). <https://doi.org/10.1007/s10207-013-0192-y>
16. Brzuska, C., Fischlin, M., Warinschi, B., Williams, S.C.: Composability of Bellare-Rogaway key exchange protocols. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM CCS 2011, October 2011, pp. 51–62. ACM Press (2011). <https://doi.org/10.1145/2046707.2046716>
17. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming, 2nd edn. Springer, Heidelberg (2014). <https://doi.org/10.1007/978-3-642-15260-3>
18. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. *J. Cryptol.* **33**(4), 1914–1983 (2020). <https://doi.org/10.1007/s00145-020-09360-1>
19. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: asynchronous group messaging with strong security guarantees. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018, October 2018, pp. 1802–1819. ACM Press (2018). <https://doi.org/10.1145/3243734.3243747>
20. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: Hicks, M., Köpf, B. (eds.) Computer Security Foundations Symposium, CSF 2016, pp. 164–178. IEEE Computer Society Press (2016). <https://doi.org/10.1109/CSF.2016.19>
21. Cong, K., Eldefrawy, K., Smart, N.P., Terner, B.: The key lattice framework for concurrent group messaging. *Cryptology ePrint Archive*, Report 2022/1531 (2022). <https://eprint.iacr.org/2022/1531>
22. Cremers, C., Hale, B., Kohbrok, K.: The complexities of healing in secure group messaging: why cross-group effects matter. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021, August 2021, pp. 1847–1864. USENIX Association (2021)
23. Dowling, B., Günther, F., Poirrier, A.: Continuous authentication in secure messaging. In: Atluri, V., Di Pietro, R., Jensen, C.D., Meng, W. (eds.) ESORICS 2022, Part II. LNCS, September 2022, vol. 13555, pp. 361–381. Springer, Heidelberg (2022). [https://doi.org/10.1007/978-3-031-17146-8\\_18](https://doi.org/10.1007/978-3-031-17146-8_18)
24. Fuchsbauer, G., Kamath, C., Klein, K., Pietrzak, K.: Adaptively secure proxy re-encryption. In: Lin, D., Sako, K. (eds.) PKC 2019. LNCS, vol. 11443, pp. 317–346. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-17259-6\\_11](https://doi.org/10.1007/978-3-030-17259-6_11)
25. Ingemarsson, I., Tang, D.T., Wong, C.K.: A conference key distribution system. *IEEE Trans. Inf. Theor.* **28**(5), 714–719 (1982). <https://doi.org/10.1109/TIT.1982.1056542>
26. Kobeissi, N., Bhargavan, K., Blanchet, B.: Automated verification for secure messaging protocols and their implementations: a symbolic and computational approach. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 435–450. IEEE (2017)
27. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978). <https://doi.org/10.1145/359545.359563>
28. Pijenburg, J., Poettering, B.: On secure ratcheting with immediate decryption. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part III. LNCS, December 2022, vol. 13793, pp. 89–118. Springer, Heidelberg (2022). [https://doi.org/10.1007/978-3-031-22969-5\\_4](https://doi.org/10.1007/978-3-031-22969-5_4)

29. Poettering, B., Rösler, P., Schwenk, J., Stebila, D.: SoK: game-based security models for group key exchange. In: Paterson, K.G. (ed.) CT-RSA 2021. LNCS, May 2021, vol. 12704, pp. 148–176. Springer, Heidelberg (2021). [https://doi.org/10.1007/978-3-030-75539-3\\_7](https://doi.org/10.1007/978-3-030-75539-3_7)
30. Rescorla, E.: Subject: [MLS] TreeKEM: an alternative to ART. MLS Mailing List (2019). <https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/>. Accessed 19 Jan 2022
31. Rösler, P., Mainka, C., Schwenk, J.: More is less: on the end-to-end security of group chats in Signal, Whatsapp, and Threema. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, 24–26 April 2018, London, United Kingdom, pp. 415–429. IEEE (2018). <https://doi.org/10.1109/EuroSP.2018.00036>
32. Steiner, M., Tsudik, G., Waidner, M.: Diffie-Hellman key distribution extended to group communication. In: Gong, L., Stern, J. (eds.) ACM CCS 1996, March 1996, pp. 31–37. ACM Press (1996). <https://doi.org/10.1145/238168.238182>
33. Weidner, M., Kleppmann, M., Hugenhroth, D., Beresford, A.R.: Key agreement for decentralized secure group messaging with strong security guarantees. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021, November 2021, pp. 2024–2045. ACM Press (2021). <https://doi.org/10.1145/3460120.3484542>
34. Weidner, M.A.: Group messaging for secure asynchronous collaboration. M. Phil thesis, University of Cambridge, June 2019. <https://mattweidner.com/acs-dissertation.pdf>
35. WhatsApp Inc.: Whatsapp encryption overview. Online, September 2021. <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>. Accessed 19 Jan 2022