





The Key Lattice Framework for Concurrent Group Messaging

Kelong Cong¹ , Karim Eldefrawy² , Nigel P. Smart^{1,3} , and Ben Turner^{4*} 

¹ imec-COSIC, KU Leuven, Leuven, Belgium.

² SRI International, Menlo Park, U.S.A.

³ Zama Inc, France.

⁴ University of California Irvine, Irvine, U.S.A.

kelong.cong@esat.kuleuven.be, karim.eldefrawy@sri.com,

nigel.smart@kuleuven.be, bturner@uci.edu.

Abstract. Today, two-party secure messaging is well-understood and widely adopted, e.g., Signal and WhatsApp. Multiparty protocols for secure group messaging on the other hand 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. Proving security of group messaging protocols using the key lattice requires new game-based security definitions for both FS and PCS. Our new definitions are both simpler and more natural than previous ones, as our framework combines both FS and PCS into directional variants of the same abstraction, and additionally avoids dependence on time-based epochs. Additionally, we give a concrete, standalone instantiation of a concurrent group messaging protocol for dynamic groups. Our protocol provides both FS and PCS, supports concurrent updates, and only incurs $O(1)$ overhead for securing the messaging payload, $O(n)$ update cost and $O(n)$ healing costs, which are optimal.

* Part of this work was completed while at SRI International.

Table of Contents

The Key Lattice Framework for Concurrent Group Messaging	1
<i>Kelong Cong</i> ^{ID} , <i>Karim Eldefrawy</i> ^{ID} , <i>Nigel P. Smart</i> ^{ID} , and <i>Ben Terner</i> ^{ID}	
1 Introduction	3
1.1 Related Work	4
1.2 Technical Overview	7
1.3 Discussion	10
2 General Definitions and Notation	11
2.1 CCA Secure Encryption Scheme	13
2.2 Message Authentication Code (MAC)	13
2.3 Key Encapsulation Mechanism (KEM)	14
2.4 Authenticated Encryption with Associated Data (AEAD)	14
2.5 Public Key Authenticated Encryption with Associated Data (PKAEAD)	15
3 Key Lattice	16
3.1 Key Evolution	16
3.2 The Key Graph	17
3.3 Instantiation	18
3.4 Key Lattice as a Key Management Technique	20
4 Group Key Agreement	21
4.1 Security Definition	21
5 Group Randomness Messaging	23
5.1 Security	24
5.2 Correctness	25
5.3 Instantiation	26
5.4 Proof of Theorem 5.1	27
6 Group Messaging	29
6.1 Security Definition	29
6.2 GM from GRM and GKA	33
6.3 Main Theorem	35
6.4 Discussion	37
7 Full Proof of GM Construction	38
8 Extension to Dynamic Groups	45

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 [ACD19,BGB04,CCD⁺20,CCG16,KBB17], but multiparty protocols (for group messaging) are still an active research area. At the moment, the Message Layer Security (MLS) IETF working group⁵ 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 [ACDT20], 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 [AAN⁺22b,AHKM22,AAN⁺22a] that 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.⁶ 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. Next we clarify some important terminology.

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

⁵ <https://messaginglayersecurity.rocks/>

⁶ This approach bears some resemblance to the analysis of Fuchsbauer et al. [FKKP19] for public key re-encryption.

exchange. This is a well known problem even for two-party key agreement followed by composition with a secure channel, see for example [BFS⁺13,BFWW11]. 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 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 [BDR20] 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.

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 [ITW82,STW96]. Later work extended the security guarantees (e.g., by providing authentication, forward secrecy, and post-compromise security) [BCP01,BCPQ01,BM08,BMS07], and improved performance and added new features (e.g., support for dynamic groups) [BCP02].

Ratchet Trees, Propose & Commit: The family of key agreement protocols popularized by the Message Layer Security (MLS) working group [BBM⁺20], is based on binary trees. These protocols are efficient and secure; they require $O(\log(n))$ public key operations to update a shared key, and they achieve both forward secrecy (FS) and post-compromise security (PCS).

The first in this line of binary-tree protocols introduced asynchronous ratcheting trees (ART) [CCG⁺18,KPT04]. In ART, the authors constructed the first asynchronous GKA protocol with FS and PCS. The group initiator selects the secret keys for nodes on the tree, and allows the group members to update the secret. TreeKEM [Res19] evolved ART to introduce support for dynamic groups.

Alwen et al. [ACDT20] explained that TreeKEM does not provide adequate FS. Concretely, the authors formalized the security model and showed that, in the worst case, FS is only achieved if every group member updates their key material, which has a cost of $O(n \log n)$. To achieve optimal FS and reduce the complexity, the authors introduced a modification to TreeKEM, called Re-randomized TreeKEM (RTreeKEM), that uses updatable public key encryption to roll the group key with every encryption and decryption. This technique reduced the healing cost to $O(\log n)$.

Bienstock, Dodis, and Rösler [BDR20] give a tree-based construction that works with concurrent updates. The communication complexity varies between $O(\log n)$, when there is no concurrency, and $O(n)$, when the updates are fully concurrent. Alwen et al. [ACJM20,AJM22] added insider security to the family of TreeKEM protocols by considering the key schedule.

Recent evolutions of ratchet trees employ the “propose and commit” framework to achieve a nontrivial amount of concurrency. Specifically, the parties can concurrently propose updates, which

are resolved with a serial or ordered commit in the next round. CoCoA [AAN⁺22b], handles concurrent updates within one epoch with the help of a server. Their key idea is to apply all concurrent updates in one epoch by applying them in order determined by an ordering function that is a system parameter. This idea assumes a fully synchronous network; otherwise, consensus is required. Consequently, it may take up to $\log(n)$ rounds to complete all updates. DeCAF [AAN⁺22a] improves on CoCoA’s healing time, and requires a blockchain for ordering. SAIK [AHKM22] explicitly models the role of the server in group key agreement and improves on the upload cost to update the group key using multi-message multi-recipient PKE. CmPKE [HKP⁺21] is similar to SAIK in these regards, with tradeoffs on the communication costs compared to SAIK, and does not explicitly model the role of the server.

The closest work to ours is the recent paper by Weidner et al. [WKHB21], 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 [Lam78]. 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. [WKHB21], 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.

Other Protocols: There are many group key agreement and group messaging protocols that do not use the tree structure, e.g., generalized Diffie-Hellman protocols [ITW82,STW96]. Protocols in such early work often do not provide the strong security properties found in modern protocols or are not efficient (i.e., requiring $O(n)$ rounds of communication to establish a key). As such, we only discuss recent developments.

Secure group messaging can be implemented by running two-party Signal between all pairs in a group [CHK21,RMS18]. If a party wants to send a message to a group, it sends the message over all of its pairwise channels⁷. An advantage of this approach is that if two parties are in multiple groups, the same pairwise channel is reused. Forward secrecy and post-compromise security are guaranteed by the underlying Signal protocol. This approach works in a concurrent environment too since PCS updates do not need to be synchronized, they only happen in the pairwise channels. The disadvantage is that parties must always create n ciphertexts under n different keys for every message.

Sender Keys, currently deployed by WhatsApp [Wha21], 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 who corrupts a

⁷ In practice it is not as easy as simply creating a Signal instance between every two parties. Additional steps need to be added for the users to establish the group ID and perform group management tasks.

Protocol	Update Cost			PCS	FS	Active Server	Concurrent Updates	Proof	Adaptive
	Sender	Receiver	Healing Rounds						
Original TreeKEM [Res19]	$O(\log n)$	$O(1)$	n	yes	yes	Ordering	no	None	n/a
Causal TreeKEM [Wei19]	$O(\log n)$	$O(1)$	n	yes	yes	none	causal	StM	yes
RTreeKEM [ACDT20]	$O(\log n)$	$O(1)$	2	yes	yes	Ordering	no	ROM	yes
Concurrent TreeKEM [BDR20]	$O(n)$	$O(1)$	2	yes	no	none	yes	StM	yes
Signal group [CHK21,RMS18]	$O(n)$	$O(1)$	2	yes	yes	Prekeys	yes	None	n/a
Sender Keys [Wha21,RMS18]	$O(n^2)$	$O(n)$	2	yes	yes	Prekeys	yes	None	n/a
DCGKA [WKHB21]	$O(n)$ (\square)	$O(1)$	2	yes	yes	none	yes (\diamond)	ROM	no
CoCoA [AAN ⁺ 22b]	$O(\log n)$	$O(1)$	$\log(n)$	yes	yes	Process-Updates (\ddagger)	yes (\diamond)	ROM	yes
SAIK [AHKM22]	$O(\log n)$	$O(1)$	2	yes	yes	Process-Updates (\ddagger)	yes (Δ)	ROM	yes
DeCAF [AAN ⁺ 22a]	$O(\log t)$ (\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

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. (Δ) 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.

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 [BMS20,BM08,BMS07,PRSS21]. In this work we use the definition from [BMS07], which allows for asynchrony (as needed by our construction).

Group Randomness Messaging (GRM): We design a new primitive called GRM which abstracts the transport mechanism used to communicate key updates. This abstraction allows us to decouple the update mechanism from our messaging protocol, which makes our proof more modular. Specifically, GRM implements pairwise secure channels which are both forward secret and post-compromise secure, but is specially designated to only send random messages, as the update messages are always random. GRM is bootstrapped from the output of GKA, i.e., it requires agreement on an initial shared group secret key. It then creates a secure channel (which has FS & PCS properties) between every pair of group members to transport updates of the group key.

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 Section 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 Section 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.



(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 Figure 1a, 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 Figure 1b, 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.

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 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 Figure 1, every key is mapped to a point on the graph, and updates are mapped to edges in the graph. We color a vertex or edge black if it is not revealed to the adversary, and we color a vertex or edge red if it is revealed to the adversary. 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 Section 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 Figure 1b. The adversary wins the semantic security game if it can distinguish two ciphertexts encrypted under a fresh key.

Tracking Keys of Other Parties: Parties will maintain a local key lattice in order to track the group keys, but they do not (necessarily) need to maintain a full view of the key lattice. Each party

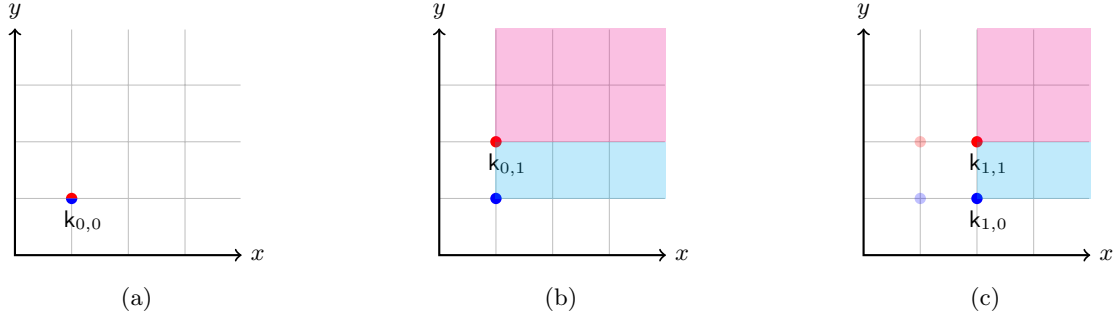


Fig. 2: An example of a local key lattice in an execution with two players (blue and red) from the perspective of the red party.

tracks only the keys that it may need in the future 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 the 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 Figure 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 Figure 2a, the red and blue parties initialize their local lattices with $k_{0,0}$.
2. In Figure 2b, red evolves the group key, which moves red’s latest key to $k_{0,1}$.
3. In Figure 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 ∞ in order to always guarantee decryption, but this approach loses FS.⁸ 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 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” [ACJM20] 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 new party joins the group, points become defined in the dimension corresponding to that party. When a party 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 new member. For simplicity, in the remainder of this paper, we define and construct our protocols without dynamic membership. We provide details of our dynamic group extension in Section 8.

1.3 Discussion

Fast Healing and Updates: Our key lattice and modular framework achieves a fast and intuitive healing mechanism. If any party is compromised, it must first heal its local GRM execution by calling its evolution function once (this refreshes the state of the channel as well as updates the group key). The next update that it receives from an uncompromised party yields an uncompromised key (including if the recovered party performs the second evolution itself). This means that healing requires 2 GRM messages. If all parties are simultaneously corrupted – meaning the adversary learns all of the keys in all parties’ local states – then all parties must refresh their GRM channels and then the next uncompromised key update yields an uncompromised group key.

Because we don’t require the propose-and-commit framework to complete an update, we reduce the complexity of every group update operation from 2 messages to 1 compared to propose-and-commit.

⁸ This tradeoff was similarly explored by [PP22]; 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.

Full Concurrency: Our approach to providing full concurrency is a foundational departure from the propose-and-commit framework. Propose-and-commit defines an execution as a series of epochs in which there is one group key per epoch, and somehow the parties achieve agree on a serial commit that defines the key for each epoch. (An infrastructure server implies this consensus by ordering messages; other protocols require an extra round of acknowledgements that still do not guarantee consensus without additional gadgetry.) Even DCGKA [WKHB21], the decentralized work closest to ours by eliminating the central server, requires that a dominating commitment is made in order to heal after compromises, but in the event of concurrent commitments there is no solution. Additionally, if multiple updating or committing parties encrypt group messages with respect to their own commitments, their messages are not guaranteed to be decryptable.

In contrast, our framework eliminates any notion of epochs and accepts that *there many be many simultaneous group keys*. It is possible that there is a different “latest” group key in the view of every party, and all parties may simultaneously update the group key(s). The key lattice framework tracks simultaneous keys while graphically representing the keys’ relationships to each other.

Partnering and Concurrent Sessions: In comparison to other recent work on group messaging [AAN⁺22b,ACDT20,AAN⁺22a,BDR20,WKHB21], our construction achieves security of concurrent *sessions* by considering *partnering*. Partnering [BMS07,Brz13,KY03] (also called matching) states that parties participating in concurrent sessions of group key agreement commonly distinguish the separate sessions. An entire line of work starting with ART [CCG⁺18] and continuing with extensions and improvements towards the MLS standard [ACDT20,AAN⁺22b,ACDT20,BDR20,Res19,WKHB21,Wei19] either explicitly or implicitly⁹ considers only one concurrent session at a time; other works explicitly model that only one CreateGroup instruction may be called [ACJM20].

Comparison to a Simplified Session Model As an example of the complexity introduced by concurrent sessions, we note that [ACDT21] recently showed how to build group messaging from group key agreement (with clear abstraction boundaries, as opposed to our construction) by omitting a key confirmation step. Their protocol exists in a fully authenticated model where the adversary is not allowed to inject messages, and the analysis only considers isolated sessions. The model therefore disallows any possible attacks that could break key confirmation. They then defer authentication to the higher-level secure messaging protocol that uses a CGKA underlying primitive.

2 General Definitions and Notation

We denote by \mathbb{N} the natural numbers. For a list ℓ , we denote by $\ell[i]$ the i th element of ℓ . We write $[m] = \{1, \dots, m\}$, and write $[a, b] = \{a, a + 1, \dots, b - 1, b\}$ where $b > a$. We assume a set of all possible parties \mathcal{P} and let $n = |\mathcal{P}|$. For ease of notation, 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 . No specific representation is used to identify every player $U \in \mathcal{P}$, as long as the representation is unique, such as a public key.

⁹ Cohn-Gordon [Coh18] explains that ART adopts a session-identifier model that obviates this issue by essentially assuming that different sessions are distinguished by the participants.

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 $k_{\mathbf{i}}$. The j -th element of index vector \mathbf{i} will be denoted by $\mathbf{i}^{(j)}$. For ease of notation, 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, the function $\text{decrement}(\mathbf{i}, j)$ 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 also define a partial ordering on the index vectors by saying $\mathbf{i} \geq \mathbf{c}$ if $\mathbf{i}^{(j)} \geq \mathbf{c}^{(j)}$ for all j . We write $\mathcal{H}_{\geq \mathbf{c}}$ for a constant index vector $\mathbf{c} \in \mathbb{N}^n$ to be 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: We assume parties are connected via pairwise channels such that both parties know the identity of the party on the other end. We assume a PKI exists that provides a mapping between an identity $U \in \mathcal{P}$ and a 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 [CGR14]. 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 (PKAEAD or CCA Encryption).

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.¹⁰

Insider Security The adversary can “take over” a party by first learning its GRM key (via a separate corruption query than leaks the group keys) and then evolving the group key on the party’s behalf. This is an insider attack, as the party has become impersonated, and it is not considered recoverable *in any known scheme* if the same party ever issues a competing key update. However, if the adversary only uses the discovered state to send a message early which would have been sent later by the party as in [ACJM20], then the attack is naturally covered by our security framework “for free,” as this attack is equivalent in our game to calling an oracle’s evolution function early, revealing the edge, and delaying delivery of the message to other parties. This is also equivalent to setting the randomness for the party’s next key evolution; in either case, the adversary simply learns the party’s next key evolution.

The techniques of [AJM22] for insider security require incorporating another protocol key into the key schedule, which might not be revealed alongside a party’s other local state, as well as the simulator’s ability to learn RO calls. The former is beyond the scope of the key lattice but could be included in a comprehensive system, and it is unclear that the latter is possible in the standard model.¹¹

¹⁰ Some authentication schemes require parties to sign messages with their long-term keys [DGP22] but adapting this to concurrent group messaging is non-trivial, and not the focus of this work.

¹¹ When [ACJM20] provide a construction without their RO, they achieve only static security.

2.1 CCA Secure Encryption Scheme

Definition 2.1 (Symmetric Key Encryption Scheme). A symmetric key encryption scheme consists of three algorithms:

- $\text{KeyGen}(1^\lambda)$: Output a symmetric key with security parameter λ .
- $\text{Enc}(m; k)$: On plaintext input m , output a ciphertext c encrypted under the symmetric key k .
- $\text{Dec}(c; k)$: Decrypt the ciphertext input c using k and output the plaintext m if successful, otherwise output \perp .

Definition 2.2 (Symmetric Encryption Scheme IND-CCA Security). The security of an IND-CCA symmetric encryption scheme is defined by a game between a challenger and an adversary \mathcal{A} as follows:

1. Challenger samples a symmetric key $k \xleftarrow{\$} \text{KeyGen}(1^\lambda)$.
2. The adversary \mathcal{A} outputs two messages m_0, m_1 .
3. The challenger selects $b \xleftarrow{\$} \{0, 1\}$ and computes $c^* \leftarrow \text{Enc}(m_b; k)$.
4. The challenger sends c^* to \mathcal{A} .
5. \mathcal{A} outputs b' .

The adversary has access to an encryption oracle and a decryption oracle. On input x , the former outputs $\text{Enc}(x; k)$ and the latter outputs $\text{Dec}(x; k)$. After the adversary learns c^* , it is not allowed to query the decryption oracle on c^* . The advantage of the adversary is

$$2 \cdot |\Pr[b = b'] - 1/2|.$$

We say an encryption scheme described in Definition 2.1 is secure if, for any polynomial-time adversary \mathcal{A} , the advantage of the game above is negligible in the security parameter λ .

2.2 Message Authentication Code (MAC)

Definition 2.3 (MAC). A MAC consists of three algorithms

- $k \leftarrow \text{MAC.KeyGen}(1^\lambda)$,
- $t \leftarrow \text{MAC}(m; k)$, and
- $b \leftarrow \text{MAC.Verify}(m, t; k)$.

For correctness we require for every λ , every key k and every $m \in \{0, 1\}^*$ it holds that $\text{MAC.Verify}(m, \text{MAC}(m; k); k) = 1$.

Definition 2.4 (MAC EUF-CMA Security). The security of a MAC is modelled using the existentially unforgeable under an adaptive chosen-message attack (EUF-CMA) game between challenger \mathcal{C} and adversary \mathcal{A} .

- \mathcal{C} generates $k \leftarrow \text{MAC.KeyGen}(1^\lambda)$.
- \mathcal{A} is allowed to query the MAC oracle (i.e., $\text{MAC}(m; k)$) on for any message m of his choice. All the queried messages are stored in a table T . Additionally, \mathcal{A} is also allowed the verification oracle $\text{MAC.Verify}(m, t; k)$ on his input (m, t) .
- Eventually, \mathcal{A} outputs (m^*, t^*) to \mathcal{C} .
- \mathcal{A} wins the game if $\text{MAC.Verify}(m^*, t^*; k) = 1$ and $m^* \notin T$.

The advantage of the adversary is given as

$$\text{Adv}_{\mathcal{A}}^{\text{mac}} = \Pr[\mathcal{A} \text{ wins EUF-CMA}].$$

The scheme is secure if, for any polynomial-time adversary \mathcal{A} , the advantage is negligible in the security parameter λ .

2.3 Key Encapsulation Mechanism (KEM)

Definition 2.5 (KEM). A KEM consists of three algorithms

- $(\text{pk}, \text{sk}) \leftarrow \text{KEM.KeyGen}(1^\lambda)$,
- $(c, k) \leftarrow \text{KEM.Encap}(\text{pk})$, and
- $k \leftarrow \text{KEM.Decap}(c; \text{sk})$.

For correctness we require if $(c, k) \leftarrow \text{KEM.Encap}(\text{pk})$ then $k = \text{KEM.Decap}(c; \text{sk})$ for all $(\text{pk}, \text{sk}) \leftarrow \text{KEM.KeyGen}(1^\lambda)$.

Definition 2.6 (KEM IND-CCA Security). The security of a KEM is modelled using a game between challenger \mathcal{C} and adversary \mathcal{A} .

- \mathcal{C} generates $(\text{pk}, \text{sk}) \leftarrow \text{KEM.KeyGen}(1^\lambda)$.
- \mathcal{C} generates a random key k_0 from the symmetric key space.
- \mathcal{C} runs encapsulation algorithm $(c^*, k_1) \leftarrow \text{KEM.Encap}(\text{pk})$
- \mathcal{C} samples $b \xleftarrow{\$} \{0, 1\}$ and outputs (c^*, k_b) to \mathcal{A} .
- Finally \mathcal{A} outputs a bit b' .

During the game, \mathcal{A} is allowed to query the decapsulation oracle $\text{KEM.Decap}(c; \text{sk})$ on any c that is not c^* . The advantage of the adversary is given as

$$\text{Adv}_{\mathcal{A}}^{\text{kem}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any polynomial-time adversary \mathcal{A} , the advantage is negligible in the security parameter λ .

2.4 Authenticated Encryption with Associated Data (AEAD)

Definition 2.7 (AEAD). An AEAD scheme consists of three algorithms:

- $k \leftarrow \text{AEAD.KeyGen}(1^\lambda)$,
- $(c, t) \leftarrow \text{AEAD.Enc}(m, d; k)$, and
- $\{m, \perp\} \leftarrow \text{AEAD.Dec}(c, d, t; k)$.

For correctness, we require that if $(c, t) \leftarrow \text{AEAD.Enc}(m, d; k)$ then we will obtain $m = \text{AEAD.Dec}(c, d, t; k)$ for all $(\text{pk}, \text{sk}) \leftarrow \text{AEAD.KeyGen}(1^\lambda)$, $m \leftarrow \{0, 1\}^*$ and $d \leftarrow \{0, 1\}^*$ where m is the message and d is the associated data.

Definition 2.8 (AEAD IND-CCA Security). The security of AEAD is described using a game between a challenger \mathcal{C} and an adversary \mathcal{A} .

1. \mathcal{C} generates $k \leftarrow \text{AEAD.KeyGen}(1^\lambda)$.

2. \mathcal{C} samples $b \xleftarrow{\$} \{0, 1\}$.
3. \mathcal{A} calls the test query $\text{Test}((m_0, d_0), (m_1, d_1))$.
4. The challenger returns $(c^*, t^*) \leftarrow \text{AEAD.Enc}(m_b, d_b; k)$.
5. \mathcal{A} outputs a bit b' .

\mathcal{A} is allowed to query the encryption oracle $\text{AEAD.Enc}(m, d; k)$ for any (m, d) and the decryption oracle $\text{AEAD.Dec}(c, d, t; k)$ for any c, d, t except when $c = c^*$ or $t = t^*$. The advantage of the adversary is given as

$$\text{Adv}_{\mathcal{A}}^{\text{aead}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any polynomial-time adversary \mathcal{A} , the advantage is negligible in the security parameter λ .

2.5 Public Key Authenticated Encryption with Associated Data (PKAEAD)

Definition 2.9 (PKAEAD). A PKAEAD scheme consists of the following algorithms:

- $(\text{pk}, \text{sk}) \leftarrow \text{PKAEAD.KeyGen}(1^\lambda)$: generate the key pair.
- $(c, t) \leftarrow \text{PKAEAD.Enc}(m, d; \text{pk})$: encrypt the plaintext m and authenticate the associated data d under the public key pk . The ciphertext c and the authentication tag t is returned.
- $\{m, \perp\} \leftarrow \text{PKAEAD.Dec}(c, d, t; \text{sk})$: decrypt the ciphertext c using sk and then return the plaintext m . This procedure fails if t is not a valid authentication tag for c or d .

For correctness we require that if $(c, t) \leftarrow \text{PKAEAD.Enc}(m, d; \text{pk})$ then, for all $m \leftarrow \{0, 1\}^*$, $d \leftarrow \{0, 1\}^*$, and $(\text{pk}, \text{sk}) \leftarrow \text{PKAEAD.KeyGen}(1^\lambda)$, we will obtain $m = \text{PKAEAD.Dec}(c, d, t; \text{sk})$.

Definition 2.10 (PKAEAD IND-CCA Security). Here we describe a typical IND-CCA security adapted to PKAEAD.

1. The challenger generates $(\text{pk}, \text{sk}) \leftarrow \text{PKAEAD.KeyGen}(1^\lambda)$ and sends pk to the adversary.
2. The challenger samples $b \xleftarrow{\$} \{0, 1\}$.
3. \mathcal{A} calls the test query $\text{Test}((m_0, d_0), (m_1, d_1))$.
4. The challenger returns $\text{PKAEAD.Enc}(m_b, d_b; \text{pk})$.
5. \mathcal{A} outputs a bit b' .

The adversary is allowed to query the the decryption oracle $\text{PKAEAD.Dec}(c, d, t; \text{sk})$ both before and after the Test query, for any c, d, t except when $(c = c^*$ or $t = t^*)$. The advantage of the adversary is given as

$$\text{Adv}_{\mathcal{A}}^{\text{pkaead}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any polynomial-time adversary \mathcal{A} , the advantage is negligible in the security parameter λ .

Below we instantiate PKAEAD scheme using a KEM and a symmetric key AEAD.

- $\text{PKAEAD.KeyGen}(1^\lambda)$: return $\text{KEM.KeyGen}(1^\lambda)$.
- $\text{PKAEAD.Enc}(m, d; \text{pk})$: $(k, e) \leftarrow \text{KEM.Encap}(\text{pk})$, $(c, t) \leftarrow \text{AEAD.Enc}(m, d; k)$, return $((e, c), t)$.
- $\text{PKAEAD.Dec}((e, c), d, t; \text{sk})$: $k \leftarrow \text{KEM.Decap}(\text{sk}; e)$, return the plaintext via $\text{AEAD.Dec}(c, d, t; k)$.

The security of this construction can be proven in a similar way as hybrid ciphers [CS03]. We sketch the proof below. In G_0 , \mathcal{A} plays the standard PKAEAD IND-CCA game. In G_1 , we modify the test query to use a different symmetric key to encrypt and authenticate the message in the AEAD component. This allows us to build an adversary \mathcal{A}_1 for the KEM game by asking its challenger for a key which might or might not be the key that corresponds to the encapsulation. Additionally, we can build an adversary \mathcal{A}_2 that plays the AEAD game. Namely, \mathcal{A}_2 forwards $((m_0, d_0), (m_1, d_1))$ to the AEAD challenger and creates a “fake” KEM encapsulation for \mathcal{A} . The security of PKAEAD holds from the security of AEAD and KEM.

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 Section 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 KeyRoll function. First, we require that KeyRoll is commutative, i.e. $\text{KeyRoll}(\text{KeyRoll}(k, x), x') = \text{KeyRoll}(\text{KeyRoll}(k, x'), x)$ for all $k \in \mathbb{K}$ and $x, x' \in \mathcal{X}$.

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 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 negl such that for every probabilistic polynomial time adversary \mathcal{A} and every λ :*

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

\mathcal{F} 's inverse is unpredictable if there exists a negligible function negl such that for any polynomial time adversary \mathcal{A} and every λ :

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

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 $\text{KeyRoll}(k, x) = \text{PRF}_x(k)$ ¹². 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 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 KeyRoll is one-way in the first input, then the construction inherits additional useful properties, which we describe in Section 6.4. We now present our definitions for one-wayness on the second input.¹³

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 negl such that for every probabilistic polynomial-time adversary \mathcal{A} and every λ

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

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

ℓ -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 Section 6.4.

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.¹⁴ Specifically, we define a key

¹² In practice we cannot use the PRF construction because it is not commutative.

¹³ 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.

¹⁴ In this work, every graph is a directed acyclic graph.

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* ρ of length ℓ 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 our group messaging protocol, which is presented in Section 6.2, allows parties to 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 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 σ , which may allow us to derive the keys at other lattice points \mathbf{j} such that $\mathbf{j}^{(d)} = \mathbf{i}^{(d)}$.

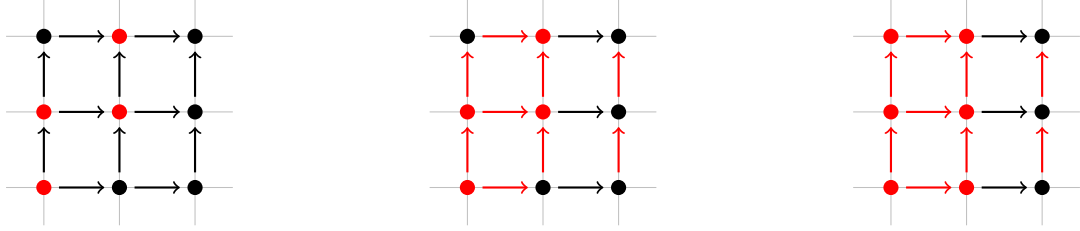


Fig. 3: Suppose the red keys in the figure on the left are revealed in a key lattice. If the KeyRoll function is unpredictable but not one-way, then knowledge of a pair of adjacent keys would reveal all edges (updates) in the corresponding row or column, as shown in the middle figure. These inferred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

The function $\text{Computable}(L, E) \rightarrow L'$ 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. Indeed, one-wayness restricts how many lattice points that we can compute without additional information on the edges.

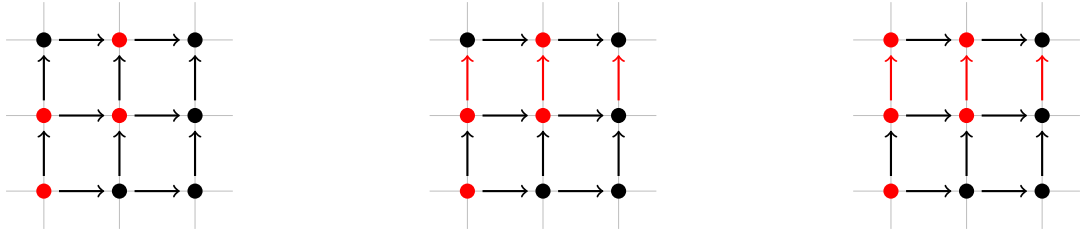


Fig. 4: Begin with the same lattice as in Figure 3 but assume that KeyRoll is one-way. The lattice points in the left figure do not allow us to compute a new lattice with more keys. However, given additional information on the edges in the middle figure, it is possible to compute one additional lattice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

The function $\text{Computable}(L, E)$ 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 .
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 the function $\text{Update}(L, e) \rightarrow L'$ 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 the tuple $(\text{increment}(\mathbf{i}), \text{KeyRoll}(\mathbf{k}_i, x))$ 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. In order 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.

Recall that we need windowing to limit state expansion and provide FS (Section 1.2). As such, 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 $d \in [n]$ wants to update 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 $\mathbf{i}' \leftarrow \text{increment}(\mathbf{i}, d)$.

- If a party at index $d \in [n]$ wants to send 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 the secure pairwise channels). The ciphertext is encrypted using AEAD where the associated data is the lattice index that corresponds to the key that was 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.
- Of course, storing all the keys that are in the key lattice is expensive and trades off forward security. Every party also 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 [BMS07] 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 [BMS07].

In this section, we reproduce the definition and introduce a few syntactic tweaks. Namely, the partnering definition is modified so that it is more natural in the context of a group. The modification details are explained below. The GKA will be used to construct our GM protocol in Section 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 $(\mathbf{pk}_U, \mathbf{sk}_U)$. 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 represented by 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 may output done with a key k to notify to the party that the protocol is completed.

4.1 Security Definition

Defining security requires additional terminology. We use $\Pi_{U,i}^{\text{gka}}$ to denote an oracle which models the i -th instance of party U engaging in the group key agreement protocol. Below, we will also use the notation (U, i) to refer to this oracle instance. We write $n = |\mathcal{P}|$ to define the total number of participants, and we assume that each participant U can engage in at at most n_S sessions, i.e. $i \in [1, \dots, n_S]$.

An oracle $\Pi_{U,i}^{\text{gka}}$ maintains a number of variables $(\delta_{U,i}, \kappa_{U,i}, \text{gid}_{U,i}, \text{sid}_{U,i}, \mathbf{k}_{U,i})$.

- The value $\delta_{U,i}$ denotes the current state of the oracle, which can be one of the following
 - **pending**: this is the initial state of each oracle. It signals that the oracle has not yet determined a key.

- **accept**: this state indicates that the oracle has determined a key.
- **abort**: this indicates that for some reason the oracle has aborted.
- The value $\kappa_{U,i} \in \{\perp, \text{corrupted}\}$ indicates the corruption state of the oracle. It is initially set to be \perp .
- The value $\text{gid}_{U,i} \subseteq \mathcal{P}$ denotes the intended group with which the oracle intends to engage in a group discussion. For convenience we assume $U \in \text{gid}_{U,i}$.
- The value $\text{sid}_{U,i}$ is a session identifier. Note that the index i in $\Pi_{U,i}^{\text{gka}}$ is not the same as $\text{sid}_{U,i}$. The value i acts as an internal session identifier. $\text{sid}_{U,i}$ is a global session identifier which the protocol needs to establish. Once established, all group members share the same sid .
- Finally $k_{U,i}$ is a key for the group $\text{gid}_{U,i}$, which is initially set to \perp .

Several functions can be called on the oracles $\Pi_{U,i}^{\text{gka}}$, which allow us to model the protocol and respond to messages. The adversary has complete control of the network and so can decide what messages to send to parties, and when.

- $\Pi_{U,i}^{\text{gka}}.\text{Init}(G)$: Initialize an instance of the GKA protocol for the group members in G where $U \in G$. Sets $\text{gid}_{U,i} \leftarrow G$ and return a set of messages $\{M_V\}_{V \in G}$, where M_V is a message intended to be passed to an oracle associated with party V . Set $\delta_{U,i} = \text{pending}$.
- $\Pi_{U,i}^{\text{gka}}.\text{Recv}(M)$:
 - If $\delta_{U,i} = \text{abort}$ then this call does nothing.
 - Otherwise, the oracle $\Pi_{U,i}^{\text{gka}}$ responds with a set of messages $\{M_V\}_{V \in G}$, where M_V is a message intended to be passed to an oracle associated with party V .
 If $\Pi_{U,i}^{\text{gka}}$ outputs a group key, then set $\delta_{U,i} = \text{accept}$ and $\Pi_{U,i}^{\text{gka}}$ outputs done .
- $\Pi_{U,i}^{\text{gka}}.\text{Corrupt}()$: This sets $\kappa_{U,i} = \text{corrupted}$ for all i associated with identity U . This command will return sk_U . Since the long-term keys are associated with the party U across all instances, calling $\Pi_{U,i}^{\text{gka}}.\text{Corrupt}$ is the same as $\Pi_{U,j}^{\text{gka}}.\text{Corrupt}$ where $i \neq j$.
- $\Pi_{U,i}^{\text{gka}}.\text{Reveal}()$: If $\delta_{U,i} \neq \text{accept}$, do nothing. Otherwise, return the shared group key $k_{U,i}$.
- $\Pi_{U,i}^{\text{gka}}.\text{StateReveal}()$: Return the internal state $\text{state}_{U,i}$.
- $\Pi_{U,i}^{\text{gka}}.\text{Test}()$: If $\delta_{U,i} \neq \text{accept}$, abort the protocol. Otherwise, sample either output the shared group key from the GKA protocol, or a random key, depending on the challenger's choice.

Using these definitions, we can give the security definition of a GKA protocol. As is usual the key definitions to define security for key agreement are *partnering* and *freshness*.

For our partnering definition we slightly deviate from the formalism of [BMS07], in that we define partnering to be defined only for the whole group; whereas [BMS07] does this in a pairwise manner. By transitivity of the pairwise partnering relation the two are essentially equivalent.

Definition 4.2 (Partnering of GKA). *Given a group $G \subseteq \mathcal{P}$ and a set of pairs $Q = (U, i_U)_{U \in G}$ (there is one pair per group member) defining associated oracles Π_{U,i_U}^{gka} , we say the oracles corresponding to $Q' \subseteq Q$ are partnered if the following conditions hold:*

1. For all $(U, i_U) \in Q'$ we have $\delta_U^{i_U} = \text{accept}$.
2. For all $(U, i_U) \in Q'$ we have $\text{gid}_{U,i_U} = G$.
3. There is a single value $\text{sid}_{Q'}$ such that for all $(U, i_U) \in Q'$ we have $\text{sid}_U^{i_U} = \text{sid}_{Q'}$.
4. No oracles, apart from Π_{U,i_U}^{gm} for $(U, i_U) \in Q'$, accept with session identifier $\text{sid}_{Q'}$.

The freshness definition below describes the state where an oracle is unaffected by the adversary. It is a form of “adversary restriction” which stops the adversary from winning the game using trivial attacks, e.g., revealing the shared key and then immediately making a test query. Freshness also helps us define forward secrecy implicitly as we will see in the security definition in Definition 4.4. This freshness definition is different to one in Section 6 because GKA does not have the concept of a key lattice so we use the traditional freshness definition.

Definition 4.3 (Freshness of GKA). *An oracle $\Pi_{U,i}^{\text{gka}}$ is considered fresh if*

- No $(U, i) \in \text{gid}_{U,i}$ is asked for a **Corrupt** query prior to a $\Pi_{V,j}^{\text{gka}}.\text{Recv}(M)$ such that $(V, j) \in \text{gid}_{U,i}$ before the partners of $\Pi_{U,i}^{\text{gka}}$ are in the accept state.
- Neither (U, i) or its partners are asked for a **StateReveal** query before they are in the accept state.
- Neither (U, i) or its partners are asked for a **Reveal** query after having accepted.

Definition 4.4 (Security of GKA). *Security of Group Key Agreement is defined by the following sequence of steps:*

1. All queries can be executed without restriction.
2. The adversary selects a fresh target (U, i) and calls $\Pi_{U,i}^{\text{gka}}.\text{Test}()$. The challenger samples a bit $b \xleftarrow{\$} \{0, 1\}$ and outputs the real shared group key k or a random key r sampled uniformly at random.
3. Continue interacting with the GKA oracles.
4. The adversary outputs a bit b' and terminates.

Any time that a session is accepted, the **sid** and the **gid** are passed to the adversary. The advantage of the adversary \mathcal{A} in this game is

$$\text{Adv}_{\mathcal{A}}^{\text{gka}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The correctness definition is described below. It is similar to the definition in [BMS07] except we modify it to use our group-based partnering definition.

Definition 4.5 (Correctness of GKA). *A key agreement protocol is said to be correct if for a group $G \subseteq \mathcal{P}$ and a set of pairs $Q = (U, i_U)_{U \in G}$ (with one pair per group member) giving associated oracles Π_{U,i_U}^{gka} , then the oracles being partnered implies that each oracle has the same shared group key k_{U,i_U} , i.e. for all $(U, i_U), (V, i_V) \in Q$ we have $k_{U,i_U} = k_{V,i_V}$.*

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 $x \in \mathcal{X}$.

5.1 Security

Security for GRM is defined in Definition 5.2. Compared to the syntax in Definition 5.1, we add `StateReveal` and `Test` queries. Furthermore, `Init`, `Evolve` and `Recv` are modified as follows: `Init` does not take a key k because these would allow the adversary to trivially win the security game described later. Additionally, `Init` takes a set of oracles Q instead of a set of players G . This change is required because the challenger needs to make sure the adversary initializes the oracles that correspond to the same session, using the same key k . The `Evolve` oracle does not output the plaintext anymore, for the same reason. Finally, `Recv` takes an additional flag `dec_flag` which allows the adversary to see the plaintext messages of the updates it uses to evolve the oracle's states. In other words, `Recv` can be used as a decryption oracle.

Intuitively, our security definition aims to capture FS and PCS. Namely, the adversary is allowed to reveal the state either before or after the test query. Nevertheless, as long as the group member under attack had a chance to recover from the corruption or deleted its old state, the adversary should learn nothing about the plaintexts that the group member sends or sent. We assume out-of-order messages, including repetition, do not happen in the point-to-point channels. In practice, this kind of attack can be detected using sequence numbers or hash chains.

Definition 5.2 (GRM Security). *The security of a GRM scheme is defined by a game between the adversary and a challenger. A mapping QtK between a group of oracles Q and a key k is kept so that the challenger uses the same key for oracles in the same Q during initialization. This mapping is not revealed to the adversary. The adversary has access to oracles $\Pi_{U,i}^{\text{grm}}$, each of which maintains internal state $\text{state}_{U,i}$ and can be invoked as follows:*

- $\{c_{U,V}\}_{(V,i) \in Q} \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Init}(w, Q)$: initializes GRM for the window size w and oracles Q . If $\text{QtK}[Q] = \perp$, sample a symmetric key k and set $\text{QtK}[Q] \leftarrow k$. Finally, return $\text{GRM}.\text{Init}(\text{QtK}[Q], w, G)$.
- $\{c_{U,V}\}_{V \in G} \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Evolve}()$: return $\text{GRM}.\text{Evolve}()$.
- $\sigma \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Recv}(c, \text{dec_flag})$: process the ciphertext c using $\text{GRM}.\text{Recv}(c)$. If `dec_flag` = 1, output the plaintext message σ , otherwise output $\sigma = \perp$.
- $\text{state}_{U,i} \leftarrow \Pi_{U,i}^{\text{grm}}.\text{StateReveal}()$: returns $\text{state}_{U,i}$ to the caller.
- $(x_0, x_1) \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Test}(c^*)$: described in the game below.

The security game is divided into phases, separated by the adversary's `Test()` query, as follows:

1. All queries can be executed without restriction.

2. The adversary calls the test query, $\Pi_{U,i}^{\text{grm}}.\text{Test}(c^*)$. The challenger samples a bit $b \xleftarrow{\$} \{0,1\}$, samples a value $x_0 \xleftarrow{\$} \mathcal{X}$ and then computes $(V, j, x_1) \leftarrow \text{GRM}.\text{Recv}(c^*)$. The adversary is given the output $(V, j, x_b), (V, j, x_{1-b})$.
3. All queries can be executed without restriction.
4. At any time the adversary can stop making queries and output a bit b' and win the game if $b' = b$.

The game above is additionally constrained by the following restrictions, which prevent trivial attacks:

- The adversary is not allowed to call Recv with $\text{dec_flag} = 1$ on c^* , the ciphertext given to the test oracle.
- Let c^* be the test ciphertext. There may be no other c' such that c^* and c' were both output from the same call to Evolve , for which c' has already been an input to any oracle query $\Pi_{V,i}^{\text{grm}}.\text{Recv}(c', \text{dec_flag} = 1)$.
- Further, consider the call to $\Pi_{V,i}^{\text{grm}}.\text{Test}(c^*)$, where c^* is taken from a call to $\Pi_{U,i}^{\text{grm}}.\text{Evolve}$. We do not allow $\Pi_{V,i}^{\text{grm}}.\text{StateReveal}$ to be called until $w + 1$ calls have been made to $\Pi_{V,i}^{\text{grm}}.\text{Evolve}$ from the time that $\Pi_{V,i}^{\text{grm}}.\text{Test}(c^*)$ is called. This condition ensures that oracle $\Pi_{V,i}^{\text{grm}}$ has refreshed its state.

The advantage of the adversary \mathcal{A} in this game is

$$\text{Adv}_{\mathcal{A}}^{\text{grm}} = 2 \cdot |\Pr[b = b'] - 1/2|.$$

The scheme is secure if, for any probabilistic polynomial-time adversary, the advantage is negligible in the security parameter.

5.2 Correctness

The correctness definition for GRM requires correct decryption of all key evolutions under two conditions which already appeared in the security definition (Definition 5.2).

1. Messages in the point-to-point channels are not reordered, i.e., these channels are modelled as FIFO queues.
2. Each point-to-point channel can buffer at most w messages, this is similar to the final restriction that prevents trivial attacks from Definition 5.2.

The constraints above can be viewed as a ω -well-ordered execution from Section 6.1 when $\omega = 0$.

Definition 5.3 (GRM Correctness). *A GRM protocol is correct if in every infinite execution by every PPT adversary \mathcal{A} who must deliver all messages, for all $U \in G$, for all $\{c_{U,V}\}_{V \in G} \leftarrow \text{GRM}_U.\text{Evolve}()$, there exists a σ and for all $V \in G$ there exists an oracle call $\sigma_{V,U} \leftarrow \text{GRM}_V.\text{Recv}(c_{U,V})$ such that $\sigma_{V,U} = \sigma$ and $\sigma \neq \perp$.*

Correctness (Definition 5.3) of the protocol described in Section 5.3 holds by construction. That is, the secret keys used for decryption are guaranteed to be available as long as there are no more than w messages in the FIFO queue.

5.3 Instantiation

We instantiate GRM using PKAEAD. 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 (\mathbf{pk}_U^0, m) , where \mathbf{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 $(\mathbf{pk}_U^0, \mathbf{sk}_U^0)$. Initialize $\text{state}_U.\text{sks} = \{\mathbf{sk}_U^0\}$ and $\text{state}_U.\text{pks} = \emptyset$, and save w as the window parameter. Compute $m \leftarrow \text{MAC}(\mathbf{pk}_U^0; k)$, where k is the input key, \mathbf{pk}_U^0 is the message and `MAC` is a cryptographic MAC scheme. Send the same message (\mathbf{pk}_U^0, m) to every member in G .
- $\text{GRM}_U.\text{Evolve}()$:
 1. A new private key \mathbf{sk}_U^{j+1} is generated, along with its public key \mathbf{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 $(c, t) \leftarrow \text{PKAEAD}.\text{Enc}(\mathbf{pk}_U^{j+1} \parallel \sigma, j_V; \mathbf{pk}_V^{j_V})$ and then set $c_{U,V} \leftarrow (c, t, j_V)$. Note that $\mathbf{pk}_V^{j_V}$ can be found in $\text{state}_U.\text{pks}$ and j_V is the index of the public key associated with V .
 4. state_U is updated as follows.
 - Add \mathbf{sk}_U^{j+1} to $\text{state}_U.\text{sks}$
 - If $|\text{state}_U.\text{sks}| > w$, remove the oldest one (i.e., \mathbf{sk}_U^{j-w}).
- $\text{GRM}_U.\text{Recv}(c_{V,U})$: There are two possible message formats. The message output by `Init` is an ephemeral public key \mathbf{pk}_V^0 with a `Mac`; if the message is this type, then verify the `Mac` using the key k provided to `Init`¹⁵ and then set V 's public key in $\text{state}_U.\text{pks}$ to be $(0, \mathbf{pk}_V^0)$. 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 \mathbf{sk}_U^j . Abort the protocol if it does not exist.
 3. $\mathbf{pk}_V^{j_V} \parallel \sigma_{V,U} \leftarrow \text{PKAEAD}.\text{Dec}(c, t, j; \mathbf{sk}_U^j)$, abort if this step returns \perp .
 4. Add or update V 's public key in $\text{state}_U.\text{pks}$ to be $(j, \mathbf{pk}_V^{j_V})$.
 5. Let j_{\min} be the smallest j in $\{(j, \mathbf{pk}_V^{j_V}) : V \in G\}$.
 6. Delete all secret keys \mathbf{sk}_U^j where $j < j_{\min}$.
 7. Return $\sigma_{V,U}$

¹⁵ If verification fails due to trying the wrong key from multiple concurrent sessions, return \perp and process the incoming message via the `Recv` function of a different session.

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

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

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.

5.4 Proof of Theorem 5.1

For the proof of Theorem 5.1 we briefly define a restricted variant of the EUF-CMA game for MAC forgery. The challenger samples a signing key k , which it does not provide to the adversary. It provides the adversary with oracle access for MACs on randomly sampled messages (by the challenger) which are guaranteed to verify under the key k . (Note the difference between this game and the EUF-CMA game, that the adversary does not sample the messages that it asks of the oracle.) The adversary wins the game if it can construct a message m^* that verifies using k . We require that in this game, the challenger samples messages from the domain of public keys for a public-key AEAD scheme.

Proof. In the zeroth game G_0 , we handle all the queries normally, as prescribed by the GRM protocol. Note that MAC forgery is possible in this game. This happens when \mathcal{A} outputs $(\text{pk}_V, c_V) \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Recv}(c, 0)$, where pk_V is a public key that did not output by any oracle call to $\text{Init}()$ but c_V is a valid MAC on pk_V verified using the key k used to initialize some oracle.

The game G_1 is the same as G_0 except we add a forgery check as follows.

- On $\Pi_{U,i}^{\text{grm}}.\text{Init}(w, Q)$: Generate $\{(\text{pk}_V, \text{sk}_V)\}_{V \in Q}$ and compute the MAC for $c_V \leftarrow \text{MAC}(\text{pk}_V)$ for every $V \in Q$. Output $\{(\text{pk}_V, c_V) : V \in Q\}$.
- On $\Pi_{U,i}^{\text{grm}}.\text{Recv}(c, \text{dec.flag})$: If c has the format (pk_V, c_V) and pk_V is not one of the public keys generated then we consider two cases.
 1. If c_V is an invalid MAC on pk_V , abort the game as prescribed.
 2. Otherwise, also abort the game and denote this event as F . This is the forgery event.

Observe that only difference between G_0 and G_1 is event F , the forgery event.

Claim. There exists an adversary \mathcal{C} against the restricted EUM-CMA game described above that uses the adversary \mathcal{A} for GRM such that $\text{Adv}^{\text{mac}}(\mathcal{C}) \geq \Pr[\mathcal{A} \text{ forges a MAC}]$.

Proof. We first build an adversary \mathcal{C} for the modified EUF-CMA game above as follows. In the security game for the GRM game, \mathcal{A} invokes oracles via their $\text{Init}()$ query, but the key k used to initialize a set of oracles Q is not exposed to the adversary. When an oracle U within a group Q is called with Init by the adversary, the game initializes that oracle and outputs a key pk_U along with a MAC c_U on pk_U which is guaranteed to verify with k .

\mathcal{C} first guesses which session i^* of a maximum of n_S GRM sessions (where n_S is allowed to be polynomial in λ). When the adversary \mathcal{A} for the GRM game calls Init on an oracle in a group Q in the i^* th session, the \mathcal{C} forwards the query to its challenger as a request for a signed message. \mathcal{C} stores the response (pk, c) and forwards it to \mathcal{A} . (Note that the \mathcal{A} will make only up to $|Q|$ such

queries because it can only call `Init` once per oracle in the group per session.) For every other session \mathcal{C} samples a MAC and simulates the GRM session for \mathcal{A} perfectly as if it were \mathcal{A} 's challenger.

In the i^* th session, whenever \mathcal{A} calls `Recv` using a message (pk, c) that \mathcal{C} has already forwarded to \mathcal{A} in the response to an `Init` query, \mathcal{C} continues the game as normal. If \mathcal{A} never calls `Recv` using a message (pk, c) that \mathcal{C} has not forwarded to \mathcal{A} , then \mathcal{C} returns a random message and MAC to its challenger. When \mathcal{A} calls `Recv` using a message (pk, c) that \mathcal{C} has not forwarded to \mathcal{A} , \mathcal{C} forwards the message to its challenger. \mathcal{C} wins its game with at least the probability that \mathcal{A} constructs a valid forgery, conditioned on guessing i^* correctly. □

We bound $\Pr[\mathcal{A} \text{ wins } G_1]$ by designing adversary \mathcal{B} against the PKAEAD game. At the start, \mathcal{B} guesses the instance that \mathcal{A} will query. This guess is correct with probability $1/|Q|_{\max}$. Additionally, \mathcal{B} guesses the index of the ciphertext that \mathcal{A} will use in `Test`. Specifically, there will be a critical query to \mathcal{B} 's `PKAEAD.Enc` oracle and the output of this query is used by \mathcal{A} in the test query. The guess will be correct with probability $1/n_Q$, where 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. \mathcal{B} sets $c^* \leftarrow \perp$.

Since forgery does not occur in G_1 , we can replace the MACs by ideal MACs. That is, \mathcal{B} stores a lookup table \mathcal{M} that maps messages to MACs. Every time a MAC needs to be generated for message m , either $\mathcal{M}[m]$ is returned if it exists, otherwise a MAC v sampled uniformly at random is returned and $\mathcal{M}[m] \leftarrow v$ is stored.

Concretely, the queries are handled as follows.

- $\Pi_{U,i}^{\text{grm}}.\text{Init}(w, Q)$: Follow the steps in Definition 5.2, i.e., call the function `GRMU.Init(k, w, G)` using the same k for the same Q .
- $\Pi_{U,i}^{\text{grm}}.\text{Evolve}()$: Perform the steps described in the protocol instantiation except substitute `PKAEAD.Enc` to calls to \mathcal{B} 's oracle. If one of the calls to `PKAEAD.Enc` is the critical query, then instead of calling `PKAEAD.Enc`, \mathcal{B} samples $(r_0^*, r_1^*) \xleftarrow{\$} R^2$ and then calls

$$c^* \leftarrow \text{PKAEAD.Test}(m_0^*, m_1^*),$$

where $m_0^* = \text{pk}_{U,i}^{j+1} \| r_0^*$ and $m_1^* = \text{pk}_{U,i}^{j+1} \| r_1^*$.

- $\Pi_{U,i}^{\text{grm}}.\text{Recv}(c, \text{dec.flag})$: If $c = c^*$, abort. Otherwise, perform the steps described in the instantiation except \mathcal{B} substitutes `PKAEAD.Dec` to calls to \mathcal{B} 's oracle.
- $\Pi_{U,i}^{\text{grm}}.\text{StateReveal}()$: return `stateU,i`.
- $\Pi_{U,i}^{\text{grm}}.\text{Test}(c)$: If $c = c^*$, return (m_0^*, m_1^*) . Otherwise, abort.

Finally, \mathcal{A} will output a bit b' to \mathcal{B} and \mathcal{B} outputs the same bit b' to the challenger.

Observe that although \mathcal{A} is allowed to call `StateReveal`, it is not allowed to call it if one of the secret keys can be used to decrypt c^* or any ciphertext from the same `Evolve` call due to our security definition. So \mathcal{A} gains no advantage from having access to `StateReveal`. Specifically, consider $\Pi_{U,i}^{\text{grm}}.\text{Test}(c^*)$ where $\text{pk}_{U,i}^*$ is used to encrypt the plaintext m^* such that $(c^*, t^*) \leftarrow \text{PKAEAD.Enc}(m^*, d^*; \text{pk}_{U,i}^*)$, then there are four cases.

1. $\Pi_{U,i}^{\text{grm}}.\text{StateReveal}$ is called before $\Pi_{U,i}^{\text{grm}}.\text{Test}$: the time when `StateReveal` is called would not have $\text{sk}_{U,i}^*$ to decrypt the key.
2. $\Pi_{U,i}^{\text{grm}}.\text{StateReveal}$ is called after $\Pi_{U,i}^{\text{grm}}.\text{Test}$: the key $\text{sk}_{U,i}^*$ would have been deleted when `StateReveal` is called.

3. $\Pi_{U,i}^{\text{grm}}$.StateReveal is called before $\Pi_{V,j}^{\text{grm}}$.Test where $(U, i) \neq (V, j)$: only the oracle $\Pi_{V,j}^{\text{grm}}$ has the private key to decrypt c^* , so revealing the state of (U, i) does not give \mathcal{A} an advantage.
4. $\Pi_{U,i}^{\text{grm}}$.StateReveal is called after $\Pi_{V,j}^{\text{grm}}$.Test where $(U, i) \neq (V, j)$: only the oracle $\Pi_{V,j}^{\text{grm}}$ has the private key to decrypt c^* , so revealing the state of (U, i) does not give \mathcal{A} an advantage.

Additionally, due to our security definition, \mathcal{A} is not allowed to use the decryption oracle on c^* or any ciphertext c' that came from the same call to Evolve as c^* . So \mathcal{A} gains no advantage in distinguishing the two plaintexts on top of its existing advantage from PKAEAD.

Our simulator perfectly simulates the view of \mathcal{A} with probability $\frac{1}{|Q|_{\max} \cdot n_Q}$. Thus we have

$$\text{Adv}_{\mathcal{A}}^{\text{grm}'} \leq |Q|_{\max} \cdot n_Q \cdot \text{Adv}_{\mathcal{B}}^{\text{pkaead}},$$

where $\text{Adv}_{\mathcal{A}}^{\text{grm}'} = 2|\Pr[\mathcal{A} \text{ wins } G_1] - 1/2|$.

Recall that the adversaries of Game 0 and Game 1 attack disjoint events of the probability space of \mathcal{A} 's security game. Summing the inequalities of the adversaries' advantages, we obtain

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

□

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 (Section 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.

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., [ACDT20, CCG⁺18]), 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 Section 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 the variables, of GKA. Below we explain the states that the GM oracle must keep, described the inheritance, and finally give the full details of the GM oracle and the security definition.

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 sid . The session ID 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 Section 4 since it is a property inherited from GKA). Indeed, 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 (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. This is inherited from the GKA oracle and we use it in the GM oracle.
- $L_{U,i}$ represents the key lattice maintained by oracle $\Pi_{U,i}^{\text{gm}}$. We use the language from Section 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 sid .

Many of the oracles are similar to the ones in GKA but extended to fit the needs of GM. Specifically, **Corrupt**, **Reveal**, **StateReveal** and **Recv** carry the same syntax and serve the same purpose. **Init** is extended to incorporate a window parameter. The remaining oracles are new. Since the GKA oracles do not have a concept of updating group keys, we introduce **Evolve** in the GM oracle for this purpose. Additionally, the GKA oracles cannot send and receive payload messages, this is the purpose of **Enc** and **Dec**. 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 $L_{U,i}$.¹⁶
- $\Pi_{U,i}^{\text{gm}}.\text{Evolve}()$: 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 Recv or 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 **Corrupt**, **Reveal** and **StateReveal** queries the adversary can learn the entire secret internal state of the oracle $\Pi_{U,i}^{\text{gm}}$. Specifically, **Reveal** gives the party’s current group keys, and **StateReveal** gives the party’s internal state except for what is provided by the former two queries. **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 **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.

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 **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 **Recv** or **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.

¹⁶ For our construction, this adds all of the edges in $E_{U,i}$ to $E_{\text{sid}}^{\text{rev}}$.

The adversary may additionally invoke `Recv` or `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. 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 Π_{U, i_U}^{gm} , we say the oracles are partnered if the underlying GKA oracles $\Pi_{U, i_U}^{\text{gka}}$ (see Definition 4.2) are partnered.*

For some security parameter λ 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 λ , 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 λ . The adversary \mathcal{A} is given at the start of the game all the public keys 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 `Reveal` queries, or implicitly, via a combination of `Reveal` and `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 sid , a key k_{i^*} with at index 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 that are 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 ct^* to the adversary. We denote the test oracle by $\Pi_{U^*, i^*}^{\text{gm}}$. We call i^* the test index.
- **Phase 2:** All queries can be executed except for:
 1. Any query that would add k_{i^*} to the set of keys computable from $S_{\text{sid}}^{\text{rev}}$.
 2. If the message ct^* is at any point processed by $\text{Dec}(\text{ct}^*)$, by the oracles, then the result is not returned to the adversary (however 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 λ .

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.

ω -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 *ω -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 *ω -well-ordered* if all pairwise channels are ω well-ordered.

We claim that when windowing with our protocol, for any ω -well-ordered execution, if the window parameter w is greater than or equal to ω , 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 Δ implies Δ -well-ordered channels; therefore, setting $w = \Delta$ implies correctness. If the network is asynchronous, then w must be set to ∞ in order 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 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, when a party wishes to evolve the group key, it invokes `GRM.Evolve()` to receive a random key update σ along with an encryptions of the update to send to each other party via pairwise channel. The calling party adds σ 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.¹⁷ When a party encrypts a message, it provides the hashed

¹⁷ 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.

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 $\text{Computable}(\mathcal{L}, \mathcal{E})$. 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 $\text{Forget}(\mathcal{L}, \mathbf{i})$. 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 $\text{GM.Init}()$, run $\text{GKA.Init}(G)$ and output the result. Note that U holds the long-term key pair $(\text{pk}_U^{\text{tt}}, \text{sk}_U^{\text{tt}})$.

Fig. 5: Algorithm for $\text{GM.Init}(G, w)$

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 $\text{GM.Evolve}()$

6.3 Main Theorem

We now state our main theorem. The proof is in Section 7.

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 $\text{Adv}^{\text{gm}}(\mathcal{A}) \leq 2n_S \text{Adv}^{\text{gka}}(\mathcal{B}) + 2n_S n \text{Adv}^{\text{grm}}(\mathcal{C}) +$*

- If M is a GKA message:
 - Compute $\{m_{U,V}\}_{V \in G} \leftarrow \text{GKA.Recv}(M)$, and outputs $m_{U,V}$ 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 $\mathcal{L} \leftarrow \text{Computable}(\mathcal{L}, \mathcal{E})$.^a
 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 $\text{GM.Recv}(M)$

^a 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 $(\text{ct}, t) \leftarrow \text{AEAD.Enc}(m, U \parallel \mathbf{i}; \text{H}(k_{\mathbf{i}}))$, and then returns $(\text{ct}, U \parallel \mathbf{i}, t)$.

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

Parse M as $(\text{ct}, V \parallel \mathbf{i}, t)$. 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 $\mathbf{i} < \mathcal{I}[\phi(V)]$, then 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 \mathbf{B} and return \perp .
- If $k_{\mathbf{i}} \neq \perp$, compute $m \leftarrow \text{AEAD.Dec}(\text{ct}, V \parallel \mathbf{i}, t; \text{H}(k_{\mathbf{i}}))$. If $m = \perp$, abort the protocol. Otherwise, return m .

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

$n_S n_q \text{Adv}^{\text{cca}}(\mathcal{D})$, 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.

Proof Sketch. The proof proceeds via three reductions. In the first, we present an adversary \mathcal{B} for the GKA game. \mathcal{B} attacks the initial key established by GKA for GM, and it uses \mathcal{A} to distinguish between the output of a GKA scheme and a randomly sampled key. Intuitively, \mathcal{B} simulates a GM execution for \mathcal{A} , and \mathcal{B} itself generates all of the keys and edges in the lattice for \mathcal{A} after the initial key is produced by GKA. In the test session (which \mathcal{B} must guess), \mathcal{B} targets the initial key by invoking the GRM oracle's $\text{Test}()$ query to be provided either the key output by GKA or a random key. On all other sessions, \mathcal{B} uses its own Reveal query to learn the GKA key. In either case, this GKA key is used as the initial key for the session's key lattice. \mathcal{B} then internally simulates GRM for \mathcal{A} and, because it knows all of the keys, can perform any requested encryption and decryption. The core idea is that \mathcal{A} will attack a specific key k^* in the lattice. However, because \mathcal{B} knows the

transformations from the initial key to k^* , \mathcal{B} can also perform the inverse computation to attack the initial GKA key.

In the second reduction, we present an adversary \mathcal{C} for the GRM game. Similar to the GKA adversary \mathcal{B} , \mathcal{C} simulates an execution of GM and tracks the keys and updates for \mathcal{A} in the key lattice defined by the execution. \mathcal{C} learns every edge except for one, which it uses to call its own `Test()` query. \mathcal{C} guesses its test edge such that with some (polynomial) probability, the key that \mathcal{A} tests will depend on the update represented by \mathcal{C} 's chosen edge. If \mathcal{A} is able to distinguish between games in which this edge is faithful to the GRM protocol and a random edge, then \mathcal{C} directly inherits the advantage to distinguishing whether the decryption output by its challenger was a faithful decryption of the update or random.

In the final reduction, we present an adversary \mathcal{D} that attacks the CCA-security of an AEAD scheme. \mathcal{D} again simulates GM for \mathcal{A} , and in this case \mathcal{D} knows *every key and update except for the test key*. \mathcal{D} therefore answers every encryption and decryption query that \mathcal{A} asks using its knowledge of the key lattice; for queries on the test key, \mathcal{D} forwards \mathcal{A} 's requests to its own challenger. The one difficulty of the proof is that \mathcal{D} does not know which key \mathcal{A} will attack in GM, and \mathcal{A} is able to ask for encryptions under that key before it makes its `Test` query, which is \mathcal{D} 's only indicator of the test key. Therefore, \mathcal{D} adaptively guesses which key will be the one tested. Observe that if \mathcal{A} makes n evolutions, 2^n keys are defined. If $n = \text{poly}(\lambda)$, then this is exponential in the security parameter. However, \mathcal{A} can only run in polynomial time (in the security parameter), and therefore can only explore a polynomial number n_q of them. \mathcal{D} adaptively guesses that the *next* key explored by \mathcal{A} will be the test key with probability $\frac{1}{n_q}$. We show that by this strategy, \mathcal{D} chooses the correct key with some probability greater than the inverse of a polynomial in the security parameter.

6.4 Discussion

Forward Secrecy, and Post-Compromise Secrecy, and Traversing the Key Lattice We frame our analyses of the above reductions in terms of the graph that represents the collective key lattice defined by the execution. Specifically underpinning our analyses, we require that GKA is forward secure and that GRM updates are both forward secure and post-compromise secure. Consider the conceptualization in which vertices and edges on the key lattice are black if they are not revealed to the adversary, and red if they are revealed to the adversary. Also color red any vertex that is discoverable by the adversary by starting at a revealed vertex and following a path of revealed edges. If GRM is both forward secure and post-compromise secure, then these are the only edges that are computable from those that are revealed, and therefore the adversary cannot learn new edges – and as a result, new keys – on the graph.

This analysis additionally requires that `KeyRoll` and its inverse are unpredictable, and therefore the adversary cannot learn arbitrary vertices by revealing a single vertex in the lattice. The properties of `KeyRoll` do not appear in the reduction because they directly imply the definition of *freshness*, which rules on which `Reveal` and `StateReveal` queries that \mathcal{A} may make in the game to enforce that \mathcal{A} may never explicitly or implicitly reveal the test key. If `KeyRoll` is not unpredictable, then if the adversary reveals any vertex, the entire lattice is revealed. If `KeyRoll` is one-way, the the adversary is permitted to learn more information about vertices and edges around the test vertex than if `KeyRoll` is only unpredictable. We next discuss how the properties of `KeyRoll` impact the information that the adversary learns from its corruption queries.

Properties of the Evolution Function. In depth, to enforce the fact that the adversary cannot learn additional components of the key lattice (which preserve the definition of freshness), without explicitly corrupting vertices, we depend on properties of the key evolution function KeyRoll, described in Section 3.3. In the following discussion, we refer to KeyRoll simply as F .

If F is *unpredictable* in its second input (Definition 3.2), then given only the key k corresponding to a vertex v , the adversary cannot learn the key k' at any successor of v for which the connecting edge is unrevealed. More granularly, given only k , \mathcal{A} cannot learn $F(k, x)$ when x was sampled at random (as the protocol specifies). Similarly, given only $F(k, x)$, where k and x were sampled at random, the adversary cannot “traverse the graph backwards” to learn the preceding key k . This is the only property of KeyRoll which our proof requires. We next describe the impact of additional desirable properties of KeyRoll on the ability of the adversary to infer points on the key lattice.

If F is *one-way on the second input* (Definition 3.3), then given a pair of vertices (u, v) , \mathcal{A} cannot learn the transformation that describes the key evolution between them. Letting k be the key associated with the vertex u and $F(k, x)$ be the key associated with v , \mathcal{A} therefore cannot learn x . Recall that in our construction, each key evolution corresponds to a group of edges in the key graph. Therefore, it may be the case that the adversary learns a number of vertices $(u_1, u_2, \dots, u_\ell)$ and $(v_1, v_2, \dots, v_\ell)$ (for $\ell < n$), where all u_i correspond to lattice points with the same index j in dimension d , and all v_i correspond to points with index $j + 1$ in dimension d . Then the adversary may attempt to use multiple points in order to learn the transformation, which is guaranteed to be the same between each u_i and v_i . We would like to prevent \mathcal{A} from learning some target key k^* which also has index $j + 1$ in dimension d , even if \mathcal{A} already knows the preceding key (the key with index j in dimension d but the same index in all other dimensions, which corresponds to the key before applying transformation x). We therefore call on an ℓ -point version of the one-wayness definition (Definition 6.6), which says that given the keys (k_{u_i}, k_{v_i}) corresponding to predecessor-successor vertices (u_i, v_i) for $i \in [1, \ell]$ and the key $k_{u_{\ell+1}}$ corresponding to the vertex $u_{\ell+1}$, the adversary still cannot learn the key $k_{v_{\ell+1}}$ corresponding to vertex $v_{\ell+1}$, even if all u_i follow the same transition (the same key evolution) to v_i .

Definition 6.6 (ℓ -Point One-Wayness (on the Second Input)). $\mathcal{F} = \{F_\lambda\}_\lambda$ is ℓ -point one-way on its second input if there exists a negligible function negl such that for every probabilistic polynomial-time adversary \mathcal{A} and every λ

$$\Pr[x' = x : \mathbf{k} \leftarrow \mathbb{K}_\lambda^\ell, x \leftarrow \mathcal{X}_\lambda, x' \leftarrow \mathcal{A}(1^\lambda, \mathbf{k}, [F_\lambda(k_1, x), \dots, F_\lambda(k_\ell, x)])] \leq \text{negl}(\lambda).$$

where x and $\mathbf{k} = (k_1, \dots, k_\ell)$ are sampled randomly from their respective domains.

One-Wayness on the First Input Although not used in our construction, it is still illustrative to explain the properties of the revealed lattice if F is *one-way on the first input* (Definition 3.3). Given given the update x on an edge (u, v) and the key corresponding to vertex v , the adversary still cannot learn the key corresponding to vertex u . For multiple points, the discussion is analogous to the previous discussion of ℓ -point one-wayness on the second input.

7 Full Proof of GM Construction

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 $\text{Adv}^{\text{gm}}(\mathcal{A}) \leq 2n_S \text{Adv}^{\text{gka}}(\mathcal{B}) + 2n_{Sn} \text{Adv}^{\text{grm}}(\mathcal{C}) +$*

$n_S n_q \text{Adv}^{\text{cca}}(\mathcal{D})$, 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.

Proof. We begin by defining three games which will allow us to complete the reduction. The first game **Game0** is the group messaging game. The second game, **Game1**, is like the group messaging game, except that in the beginning of the game, the challenger selects a random initial group key. The third game, **Game2**, is like **Game1**, except that the challenger switches the update σ corresponding to some party's key evolution to a random update. We will show that if \mathcal{A} is an adversary against the GM game, then we use these games to construct adversaries \mathcal{B} for the GKA game, \mathcal{C} for the GRM game, and \mathcal{D} for the AEAD-CCA game that use \mathcal{A} to win their respective games, with the advantages given in the theorem statement.

Lemma 7.1. *There exists an adversary \mathcal{B} for GKA such that $\text{Adv}^{\text{gka}}(\mathcal{B}) \geq \frac{1}{n_S} |\Pr[A \text{ wins Game0}] - \Pr[A \text{ wins Game1}]|$.*

Proof. We show how to construct \mathcal{B} such that it uses \mathcal{A} to win the GKA game. Specifically, the difference between the two games is that **Game0** is the GM game, and in **Game1**, the GKA key on the oracle under test is switched for a random key. \mathcal{B} uses \mathcal{A} 's ability to distinguish between these two games in order to win its key indistinguishability game, which is exactly whether, on the **Test** instance, \mathcal{B} is given the correct output of GKA or a random key.

\mathcal{B} 's challenger samples long-term keypairs $(\text{pk}_U, \text{sk}_U)$ for every parties U in the game. It provides \mathcal{B} with the parties' public keys. It also samples a bit $b_{\text{gka}} \leftarrow \{0, 1\}$ which serves as the challenge bit.

We now describe how \mathcal{B} emulates the environment for \mathcal{A} . \mathcal{B} begins by guessing the instance $\hat{\text{sid}}$ which \mathcal{A} will test. Whenever \mathcal{A} makes an oracle query that corresponds to a **gka** query, \mathcal{B} forwards the query to its own oracle and returns the response directly to \mathcal{A} . For every instance $\text{sid} \neq \hat{\text{sid}}$, as a first step after any **gka** subprotocol outputs a key, \mathcal{B} queries its oracle $\Pi_{U,j}^{\text{gka}}.\text{Reveal}()$ (for any instance (U, j) which has the initial group key for sid in its state) to learn the group key. For instance $\hat{\text{sid}}$, \mathcal{B} queries $\Pi_{U,i}^{\text{gka}}.\text{Test}()$ immediately after the **gka** session outputs a key (for appropriate (U, i) mapped to that instance), and receives either the group key or a random key. \mathcal{B} denotes its initial group key in session sid by $k_{\text{sid}}^{(0)}$.

After GKA has been executed for a session, \mathcal{B} emulates the key evolutions and encrypted messaging of GM for \mathcal{A} . Whenever \mathcal{A} makes an oracle query that corresponds to a GRM query, \mathcal{B} emulates the GRM oracle internally. Specifically, once it has an initial group key $k_{\text{sid}}^{(0)}$ for session sid , \mathcal{B} begins to simulate a GRM instance and tracks every key update σ generated by every party. It applies these updates to $k_{\text{sid}}^{(0)}$ as necessary in order to fulfill \mathcal{A} 's requests. Whenever \mathcal{A} makes an encryption query $\Pi_{U,j}^{\text{gm}}.\text{Enc}()$, \mathcal{B} evolves $k_{\text{sid}}^{(0)}$ (for the appropriate corresponding sid) to the appropriate key (which is exactly the key corresponding to the maximal index in U 's lattice in the instance mapped to sid). \mathcal{B} similarly responds to decryption queries, first by looking up the key index referenced by the decryption message and then evolving the initial key using the appropriate updates.

We now provide the full details of the reduction. \mathcal{B} receives all parties' long-term public keys as input in its game. Throughout the game, it maintains state for each party in the GM execution in order to emulate GM internally. It maintains the variables $\delta_{U,i} \in \{\text{pending}, \text{accept}, \text{abort}\}$ to denote party U 's pairing status in instance (U, i) (which maps to some session sid in which other (V, j) are

present; \mathcal{B} internally computes this mapping and we elide the details), and $\kappa_{U,i} \in \{\text{corrupted}, \perp\}$ to denote U 's corruption status.

In addition to the local state of all parties, \mathcal{B} maintains additional state to track the simulation. L^{sid} is a key lattice of all the keys and updates defined in session sid . The vertices and edges that are colored red in L^{sid} correspond to $L_{\text{sid}}^{\text{rev}}$. We note that in the description of the simulator, we do not have \mathcal{B} track the GM buffer \mathbf{B}_U for each party U . The adversary \mathcal{A} learns every message M that would be put in the buffer, and can invoke its oracles in order to mimic the behavior of the GM protocol, if it chooses.

\mathcal{B} responds to oracle queries by \mathcal{A} as follows:

- $\Pi_{U,i}^{\text{gm}}.\text{Init}(G)$: \mathcal{B} initializes local states for party $U \in G$ for instance i by setting $\delta_{U,i} \leftarrow \text{pending}$ and $\rho_{U,i} \leftarrow \perp$. If U has not been initialized before, then \mathcal{B} sets $\kappa_U \leftarrow \perp$. \mathcal{B} initializes the GKA protocol for party i by forwarding \mathcal{A} 's query to its own oracle $\Pi_{U,i}^{\text{gka}}.\text{Init}(G)$ and returns the output to \mathcal{A} .
- $\Pi_{U,i}^{\text{gm}}.\text{Corrupt}()$:
 - \mathcal{B} sets $\kappa_U \leftarrow \text{corrupted}$.
 - \mathcal{B} forwards the corruption query to $\Pi_{U,i}^{\text{gka}}.\text{Corrupt}()$ and returns to \mathcal{A} the output.
 - Every future GKA message received by $\Pi_{U,j}^{\text{gm}}$ for any j is passed to \mathcal{A} , and corresponding vertices and edges that it learns are colored red in the respective L^{sid} (for the sid mapped by (U, j)).
- $\Pi_{U,i}^{\text{gm}}.\text{Evolve}()$: \mathcal{B} internally executes $\{(c_{U,V}, x_{U,V})\}_{V \in G} \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Evolve}()$ to generate a message x (where each $x_{U,V}$ is equal to x) and a set of ciphertexts \mathbf{c} encoding x . \mathcal{B} updates L^{sid} (for sid mapped by (U, i)) by labeling the edges corresponding to the key evolution with x , and returns \mathbf{c} to \mathcal{A} .
- $\Pi_{U,i}^{\text{gm}}.\text{Recv}(M)$:
 - If $\delta_{U,i} = \text{abort}$, \mathcal{B} does nothing.
 - Else if M is a GKA message (contains a header gka) and $\delta_{U,i} = \text{pending}$, \mathcal{B} queries $\Pi_{U,i}^{\text{gka}}.\text{Recv}(M)$ and returns the output to \mathcal{A} . If $\Pi_{U,i}^{\text{gka}}$ outputs done , \mathcal{B} sets $\delta_{U,i} = \text{accept}$. If U is the first party in the session for which $\delta_{U,i} = \text{accept}$ (meaning $\Pi_{U,i}^{\text{gka}}$ is not yet partnered with any other oracles), then:
 - * if the sid mapped by (U, i) is the test session sid^* , \mathcal{B} queries $\Pi_{U,i}^{\text{gka}}.\text{Test}()$ and assigns the output to $k_{\text{sid}}^{(0)}$, and updates L^{sid} by assigning $k_{\text{sid}}^{(0)}$ to the vertex at $\mathbf{0}$.
 - * if the sid mapped by (U, i) is not the test session sid^* , \mathcal{B} queries $\Pi_{U,i}^{\text{gka}}.\text{Reveal}()$, assigns the output to $k_{\text{sid}}^{(0)}$, and updates L^{sid} by assigning $k_{\text{sid}}^{(0)}$ to the vertex at $\mathbf{0}$.
 - Else if M is a GRM message (contains a header grm) and $\delta_{U,i} = \text{accept}$, \mathcal{B} internally runs $\Pi_{U,i}^{\text{grm}}.\text{Recv}(M)$ and forwards to \mathcal{A} anything that is returned
- $\Pi_{U,i}^{\text{gm}}.\text{Dec}(M)$:
 - If M is not of the form $(V \parallel \mathbf{i} \parallel \text{ct})$ or $\delta_{U,i} \neq \text{accept}$, then \mathcal{B} sets $\delta_{U,i} = \text{abort}$ and returns \perp to \mathcal{A} .
 - Otherwise, \mathcal{B} if there is no key at index \mathbf{i} in U 's lattice (for instance i), \mathcal{B} returns \perp to \mathcal{A} . Otherwise, \mathcal{B} returns $m \leftarrow \text{Dec}(k, \text{ct})$ to \mathcal{A} , where k is the key at index \mathbf{i} in U 's lattice.
- $\Pi_{U,i}^{\text{gm}}.\text{Enc}(M)$:
 - If $\delta_{U,i} \neq \text{accept}$ then set $\delta_{U,i} \leftarrow \text{abort}$ and return \perp .

- Otherwise, \mathcal{B} computes \mathbf{i}_U as the maximal index in U 's local lattice and \mathbf{k} as the key corresponding to \mathbf{i}_U in U 's lattice. \mathcal{B} computes $\text{ct} \leftarrow \text{Enc}(\mathbf{k}, M)$, and returns $(U \parallel \mathbf{i}_U \parallel \text{ct})$ to \mathcal{A} .
- $\Pi_{U,i}^{\text{gm}}.\text{Reveal}()$: If $\delta_{U,i} \neq \text{accept}$ then \mathcal{B} does nothing. If this query is called after $\text{Test}()$ on the test session sid^* and the key with index \mathbf{i}^* (defined in Test) is computable from U 's local state (in instance (U, i)), then \mathcal{B} does nothing. Otherwise \mathcal{B} computes K as the set of all keys computable from (U, i) 's local state, and marks every vertex in K as red in $L_{\text{sid}}^{\text{rev}}$ (for the sid mapped by (U, i)). Finally, \mathcal{B} computes $s \leftarrow \Pi_{U,i}^{\text{gka}}.\text{Reveal}()$ and returns the values (s, K) to \mathcal{A} .
- $\Pi_{U,i}^{\text{gm}}.\text{StateReveal}()$: \mathcal{B} computes $s \leftarrow \Pi_{U,i}^{\text{gka}}.\text{StateReveal}()$ and returns $(s, \text{state}_{U,i})$ to \mathcal{A} , where $\text{state}_{U,i}$ is the state that \mathcal{B} maintains for (U, i) in its GRM instance. \mathcal{B} also marks all edges in $L_{\text{sid}}^{\text{rev}}$ as red that are revealed by $\text{state}_{U,i}$ or in $\mathcal{E}_{U,i}$.
- $\Pi_{U,i}^{\text{gm}}.\text{Test}(m_0, m_1)$: \mathcal{B} computes \mathbf{i}_U as the maximal index in U 's local lattice and \mathbf{k}_U as the key corresponding to \mathbf{i}_U in U 's lattice. If \mathbf{k}_U is computable from $L_{\text{sid}}^{\text{rev}}$ (in the sid mapped by (U, i)), then \mathcal{B} ignores the request, as this key is not fresh. Otherwise, \mathcal{B} sets $\mathbf{i}^* \leftarrow \mathbf{i}_U$, samples $b \leftarrow \{0, 1\}$ and returns $c \leftarrow \text{Enc}(\mathbf{k}_U, m_b)$ to \mathcal{A} .

Note that \mathcal{B} ignores \mathcal{A} 's Test query only if \mathcal{A} is testing a key which has already been revealed to it; in this case, \mathcal{A} 's query is disallowed by the game.

The advantage of \mathcal{B} in the GKA game follows directly from the advantage of \mathcal{A} in its own game. Assume that \mathcal{B} correctly guesses the instance sid^* which \mathcal{A} tests (meaning $\hat{\text{sid}} = \text{sid}^*$); this occurs with probability at least $\frac{1}{n_S}$. When \mathcal{B} 's challenger's bit $b_{\text{gka}} = 0$, \mathcal{A} 's environment is exactly Game0 . When \mathcal{B} 's challenger's bit $b_{\text{gka}} = 1$, \mathcal{A} 's environment is exactly Game1 . When \mathcal{A} outputs $b' = b$, \mathcal{B} outputs 1. When \mathcal{A} outputs $b' \neq b$, \mathcal{B} outputs 0. It follows that \mathcal{B} 's advantage is lowerbounded by the probability that \mathcal{B} guesses the instance correctly times the advantage of \mathcal{A} in distinguishing Game0 and Game1 .

$$\text{Adv}^{\text{grm}}(\mathcal{B}) \geq \frac{1}{n_S} |\Pr[\mathcal{A} \text{ wins Game1}] - \Pr[\mathcal{A} \text{ wins Game0}]| \quad (1)$$

□

We next proceed to the difference between Game1 and Game2 .

Lemma 7.2. *There exists an adversary \mathcal{C} for GRM such that $\text{Adv}^{\text{grm}}(\mathcal{C}) \geq \frac{1}{n_S n} |\Pr[\mathcal{A} \text{ wins Game2}] - \Pr[\mathcal{A} \text{ wins Game1}]|$.*

Proof. Recall that Game1 is the GM game, except that the initial group key is replaced with a random key on the test instance. Moreover, Game2 is just like Game1 , except that some key update for the key under Test is swapped for a random update.

The GRM adversary \mathcal{C} simulates the GM oracle queries for \mathcal{A} by (a) internally simulating a GKA execution and (b) forwarding all GRM queries to its own challenger. Note that because the GRM adversary \mathcal{C} now internally simulates all GKA instances for \mathcal{A} , it therefore knows the random initial group key for GM. (This is because \mathcal{C} samples the long term keys for \mathcal{A} 's GM game, and therefore knows them for the execution of GKA.) Starting with the initial random key $k_{\text{sid}}^{(0)}$ for each session sid , \mathcal{C} initializes its GRM oracles and plays the GRM game. \mathcal{C} responds to \mathcal{A} 's encryption queries by tracking all of the keys and key updates requested by \mathcal{A} , and encrypting messages under the appropriate keys. When \mathcal{A} makes a query $\mathbf{c} \leftarrow \Pi_{U,i}^{\text{gm}}.\text{Evolve}()$, \mathcal{C} forwards the query to $\Pi_{U,i}^{\text{grm}}.\text{Evolve}()$. When any ciphertext $c_V \in \mathbf{c}$ is submitted to $\Pi_{V,i}^{\text{gm}}.\text{Recv}(c_V)$, \mathcal{C} invokes

$x \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Recv}(c, \text{dec_flag} = 1)$, and \mathcal{C} then uses this value of x as the decryption of every $c' \in \mathbf{c}$. Because \mathcal{C} knows the initial group key $k_{\text{sid}}^{(0)}$ for session sid and every x , it can “fill out” the entire key lattice $L_{\text{sid}}^{\text{rev}}$ defined by the execution, and therefore it can derive every encryption key used in the session.

\mathcal{C} adapts this strategy slightly in order to tie its $\text{Test}()$ query to \mathcal{A} 's, and therefore derives an advantage in its game from \mathcal{A} 's advantage. Recall that \mathcal{A} sends a pair of messages (m_0, m_1) to its challenger, receives the encryption of m_b under some party U^* 's latest key (where b is the bit sampled by the challenger), and must guess whether $b = 0$ or $b = 1$. Let \mathbf{i}^* be the maximal index of a defined key in U^* 's state; this is the key k^* for U^* 's challenge. Let sid^* be the session in which \mathcal{A} queries its $\text{Test}()$ oracle. To respond to \mathcal{A} 's query, \mathcal{C} must encrypt one of \mathcal{A} 's two messages under k^* . However, instead of faithfully encrypting \mathcal{A} 's query with the appropriate key \mathcal{C} chooses an edge e along a path from $\mathbf{0}$ to \mathbf{i}^* , and calls $(x_0, x_1) \leftarrow \Pi_{U^*, \text{sid}^*}^{\text{grm}}.\text{Test}(c)$, where c corresponds to the ciphertext encrypting U^* 's true update along the edge e . \mathcal{C} samples x_β for a random $\beta \in \{0, 1\}$, and replaces the true update along e with x_β . If \mathcal{C} guesses the correct update (β is the same as its challenger's test bit), then \mathcal{A} 's environment is exactly **Game1**. If \mathcal{C} guesses the random update, then \mathcal{A} 's environment is exactly **Game2**.

\mathcal{C} 's full strategy therefore deviates from learning every update as follows. At the beginning of the game, \mathcal{C} uniformly at random chooses some $\hat{U} \in \mathcal{P}$ and some $\hat{\text{sid}} \in n_S$. When \mathcal{A} makes its first call to $\Pi_{\hat{U}, \hat{\text{sid}}}^{\text{grm}}.\text{Evolve}()$, \mathcal{C} queries $c \leftarrow \Pi_{\hat{U}, \hat{\text{sid}}}^{\text{grm}}.\text{Evolve}()$. \mathcal{C} then immediately makes its $\text{Test}()$ query by choosing a $c \in \mathbf{c}$ and invoking $(x_0, x_1) \leftarrow \Pi_{\hat{U}, \hat{\text{sid}}}^{\text{grm}}.\text{Test}(c)$. \mathcal{C} then randomly samples $\beta \in \{0, 1\}$ and sets x_β as the value of the update corresponding to U 's evolution along that edge in L^i for the duration of the game. If \mathcal{C} 's Test query corresponds to an update on the path from $\mathbf{0}$ to \mathbf{i}^* , then when \mathcal{A} outputs a bit $b \in \{0, 1\}$ for its game, \mathcal{C} responds with the same bit. If \mathcal{C} 's Test query does not correspond to an update on the path to \mathbf{i}^* or \mathcal{C} guesses the wrong instance $\hat{\text{sid}}$ ($\hat{\text{sid}} \neq \text{sid}^*$), then \mathcal{C} outputs a uniformly random bit.

We remark here that for technical reasons, because **Game1** requires that the $\text{Test}()$ instance have a random $k^{(0)}$, \mathcal{C} must guess the session sid^* on which \mathcal{A} will call its $\text{Test}()$ query at the beginning of the simulation. \mathcal{C} samples a random session $\hat{\text{sid}}$, and samples a random key $k_{\hat{\text{sid}}}^{(0)}$ independent of the execution of GKA for that session. (\mathcal{C} is correct if $\hat{\text{sid}} = \text{sid}^*$.)

We now provide a full description of \mathcal{C} . \mathcal{C} uniformly at random chooses some $\hat{U} \in \mathcal{P}$ and some $\hat{\text{sid}} \in n_S$. It then proceeds as follows:

- $\Pi_{U,i}^{\text{grm}}.\text{Init}(G, w)$:
 - \mathcal{C} sets $\delta_{U,i} \leftarrow \text{pending}$, $\rho_{U,i} \leftarrow \perp$ and $\kappa_{U,i} \leftarrow \perp$.
 - \mathcal{C} simulates $\Pi_{U,i}^{\text{gka}}.\text{Init}(G)$ and forwards to \mathcal{A} anything that is returned
- $\Pi_{U,i}^{\text{grm}}.\text{Corrupt}()$:
 - \mathcal{C} sets $\kappa_U \leftarrow \text{corrupted}$.
 - \mathcal{C} returns sk_U to \mathcal{A} . If no key has yet been derived (it must be the case that this party has not yet completed GKA in session i), then once this party derives the initial group key, mark it as red in $L_{\text{sid}}^{\text{rev}}$. Every future GKA key learned by U in another session is revealed in the corresponding $L_{\text{sid}}^{\text{rev}}$.
- $\Pi_{U,i}^{\text{grm}}.\text{Evolve}()$:
 - \mathcal{C} calls $c \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Send}()$
 - if $U = \hat{U}$ and (U, i) maps to $\hat{\text{sid}}$, then \mathcal{C} chooses an appropriate $c \in \mathbf{c}$ (corresponding to the ciphertext intended for any $V \neq U$) and computes $(x_0, x_1) \leftarrow \Pi_{V,i}^{\text{grm}}.\text{Test}(c)$. \mathcal{C} samples

- $\beta \leftarrow \{0, 1\}$ uniformly at random, and applies x_β to the lattice $L_{\text{sid}}^{\text{rev}}$ on the edge corresponding to U 's update. (This edge is still black in $L_{\text{sid}}^{\text{rev}}$.)
- Otherwise, U chooses the appropriate $c \in \mathcal{c}$ (corresponding to the encryption of the update for U), and computes $x \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Recv}(c, \text{dec_flag} = 1)$. \mathcal{C} then applies x to the lattice $L_{\text{sid}}^{\text{rev}}$ (for sid mapped by (U, i)) on the edge corresponding to U 's update.
- $\Pi_{U,i}^{\text{gm}}.\text{Reveal}()$:
- If $\delta_{U,i} = \text{abort}$, then \mathcal{C} returns \perp .
 - If $\delta_{U,i} = \text{pending}$ then \mathcal{C} emulates the call $\Pi_{U,i}^{\text{gka}}.\text{Reveal}()$ for its simulation of U 's view in GKA instance (U, i) and returns the response to \mathcal{A} .
 - If $\delta_{U,i} = \text{accept}$, then \mathcal{C} computes the set of pairs (\mathbf{i}, \mathbf{k}) from \mathcal{L}_U (for sid mapped by (U, i)) corresponding to the vertices and keys in (U, i) 's local state. Let this set be R . If $\text{Test}()$ has already been called and $(\mathbf{i}^*, \mathbf{k}^*)$ is included in R , then \mathcal{C} ignores the query. Otherwise, \mathcal{C} returns R to \mathcal{A} and colors all of the vertices in R as red in $L_{\text{sid}}^{\text{rev}}$.
- $\Pi_{U,i}^{\text{gm}}.\text{StateReveal}()$:
- If $\delta_{U,i} = \text{abort}$, then \mathcal{C} returns \perp .
 - If $\delta_{U,i} = \text{pending}$ then \mathcal{C} emulates the call $\Pi_{U,i}^{\text{gka}}.\text{StateReveal}()$ for its simulation of U 's view in GKA instance (U, i) and returns the response to \mathcal{A} .
 - If $\delta_{U,i} = \text{accept}$, then \mathcal{C} computes the set of edges \mathcal{E}_U which are defined in (U, i) 's current state. If $\text{Test}()$ has already been called and coloring all of the edges of \mathcal{E}_U red in $L_{\text{sid}}^{\text{rev}}$ would also color the vertex at \mathbf{i}^* red, then \mathcal{C} ignores the call. Otherwise, \mathcal{C} returns E to \mathcal{A} and colors all of the edges in E red in $L_{\text{sid}}^{\text{rev}}$ (for the session sid mapped by (U, i)).
- $\Pi_{U,i}^{\text{gm}}.\text{Enc}(M)$: If $\delta_U^i \neq \text{accept}$ then set $\delta_U^i = \text{abort}$ and return. Otherwise, let \mathbf{i} the maximal index in U 's local lattice (in instance (U, i)), and let \mathbf{k}_i be the key defined at that index. \mathcal{C} computes $(\text{ct}, t) \leftarrow \text{AEAD}.\text{Enc}(m, U \parallel \mathbf{i}, \mathbf{k}_i)$, and returns $(\text{ct}, U \parallel \mathbf{i}, t)$ to \mathcal{A} .
- $\Pi_{U,i}^{\text{gm}}.\text{Dec}(M)$: If $\delta_U^i = \text{abort}$, then \mathcal{C} ignores the query. \mathcal{C} parses M as $(\text{ct}, V \parallel \mathbf{i}, t)$. If M is not of this form, \mathcal{C} returns \perp . Let \mathbf{k} be the key at index \mathbf{i} in U 's local state. If \mathbf{k} is not computable, then buffer M . If \mathbf{k} is computable from U 's state, then \mathcal{C} computes $m \leftarrow \text{AEAD}.\text{Dec}(\text{ct}, V \parallel \mathbf{i}, t; \mathbf{k}_i)$. If $m = \perp$, \mathcal{C} sets $\delta_U^i = \text{abort}$. Otherwise, \mathcal{C} returns m to \mathcal{A} .
- $\Pi_{U,i}^{\text{gm}}.\text{Recv}(M)$:
- If $\delta_{U,i} = \text{abort}$, \mathcal{B} does nothing.
 - Else if M is a GKA message (contains a header gka) and $\delta_{U,i} = \text{pending}$, \mathcal{C} simulates the execution of $\Pi_{U,i}^{\text{gka}}.\text{Recv}(M)$. If $\Pi_{U,i}^{\text{gka}}.\text{Recv}(M)$ does not output done , then \mathcal{C} returns to \mathcal{A} anything that is returned. If done is returned, then \mathcal{C} derives $k_{\text{sid}}^{(0)}$ (for the session sid mapped by (U, i)). Because \mathcal{C} simulated this execution, it knows $k_{\text{sid}}^{(0)}$; recall as well that if $\text{sid} = \hat{\text{sid}}$, then \mathcal{C} samples a uniformly random key $k_{\text{sid}}^{(0)}$ independent of the simulated GKA execution. \mathcal{C} then calls $\Pi_{U,i}^{\text{grm}}.\text{Init}(k_{\text{sid}}^{(0)}, w)$, and returns to \mathcal{A} any messages that were returned, except for the initial group key $k_{\text{sid}}^{(0)}$.
 - Else if M is a GRM message (contains a header grm) and $\delta_{U,i} = \text{accept}$:
 - * if $\text{Test}()$ has been called, and M is a ciphertext c' which was output in the same set of ciphertexts as the ciphertext on which $\text{Test}()$ was called, then \mathcal{C} updates the state of U (in instance (U, i)) as if calling $\text{GM}.\text{Recv}$ where x_β is returned from $\text{GRM}.\text{Recv}$, where x_β was the update chosen by \mathcal{C} after receiving its challenge.
 - * Otherwise, \mathcal{C} compute $x \leftarrow \Pi_{U,i}^{\text{grm}}.\text{Recv}(M)$. If $x = \perp$, then do nothing. Otherwise, update U 's local state (in instance (U, i)) as in the description of $\text{GM}.\text{Recv}$ letting x be

the decrypted update by adding x to U 's set of edges E , propagating the changes by computing all additional keys in L , and forgetting keys as described in GM.Recv .

- $\Pi_{U,i}^{\text{gm}}.\text{Test}(m_0, m_1)$:
 - if $\delta_{U,i} \neq \text{accept}$ then return \perp
 - Let $\mathbf{i}^* = \mathbf{i}_U$ such that \mathbf{i}_U is the maximal index in U 's local state. If \mathbf{i}^* is red in L^{sid} (for sid mapped by (U, i)) then ignore the query. Otherwise, \mathcal{C} computes $\mathbf{k}^* = \mathbf{k}_U$ such that \mathbf{k}_U is the key corresponding to \mathbf{i}_U .
 - \mathcal{C} samples $b \xleftarrow{\$} \{0, 1\}$, computes $(\text{ct}, t) \leftarrow \text{AEAD.Enc}(m_b, U || \mathbf{i}^*; \mathbf{k}^*)$ and returns $(\text{ct}, U || \mathbf{i}^*, t)$ to \mathcal{A} .

We now analyze the advantage of \mathcal{C} in winning the GRM game. \mathcal{C} derives an advantage from \mathcal{A} 's ability to distinguish between Game1 and Game2 precisely when the edge that \mathcal{C} chooses for its $\text{Test}()$ query corresponds to an update used to compute \mathbf{k}^* . We note that it may be the case that \mathcal{A} never evolves the GM key, and always invokes $\text{Test}()$ on the initial group key. This case is irrelevant to the lemma, as \mathcal{A} 's advantage can be shown to break either GKA (the first game hop) or CCA (the final game hop). Assume that \mathcal{A} chooses to evolve the key at least once. Then there must be at least one party U on whose oracle \mathcal{A} calls $\Pi_{U,i}^{\text{gm}}.\text{Evolve}()$. \mathcal{C} derives an advantage from \mathcal{A} if \mathcal{C} correctly guesses this party, which it does with probability at least $\frac{1}{n}$. Additionally, \mathcal{C} must correctly guess the session on which \mathcal{A} will execute its $\text{Test}()$ query; this occurs with probability at least $\frac{1}{n_S}$. It follows that

$$\text{Adv}^{\text{grm}}(\mathcal{C}) \geq \frac{1}{n \cdot n_S} |\Pr[\mathcal{A} \text{ wins Game2}] - \Pr[\mathcal{A} \text{ wins Game1}]| \quad (2)$$

Lemma 7.3. *There exists an adversary \mathcal{D} for CCA such that $\text{Adv}^{\text{gka}}(\mathcal{D}) = \frac{1}{n_S n_q} |\Pr[\mathcal{A} \text{ wins Game2}] - \frac{1}{2}|$.*

Proof. \mathcal{D} uses \mathcal{A} 's advantage in the GM game in order to break the CCA security of an encryption scheme as follows. \mathcal{D} emulates the entire execution of GM for \mathcal{A} , and forwards only encryptions and decryptions on its own challenge key to \mathcal{A} . Specifically, \mathcal{D} samples long term public and private keys for all parties, emulates each GKA execution for \mathcal{A} , and learns the initial key $k^{(0)}$ for every group of oracles. \mathcal{D} then simulates the execution of GRM internally, and learns every key evolution output by GRM. \mathcal{D} uses this information to construct a key lattice L for each execution and label every vertex and edge on the lattice with the appropriate key and update, respectively. When \mathcal{A} requests an encryption or decryption, \mathcal{D} uses its knowledge of the key lattice to compute the encryption; specifically, computes the lattice key at the index that \mathcal{A} queries, and evaluates \mathbf{H} in order to compute the encryption key corresponding to that point. Similarly, when \mathcal{A} requests a decryption of a message, \mathcal{D} uses its knowledge of the key to decrypt \mathcal{A} 's message. For technical reasons as in the previous lemma, \mathcal{D} also chooses some update output by $\text{Evolve}()$ to be replaced with a random update; this is chosen as described below such that the test key depends on the random update.

\mathcal{D} only does not answer encryption queries on its own when \mathcal{A} issues queries to Enc, Dec , or Test on \mathbf{i}^* , where \mathbf{i}^* is defined as the current lattice index of the oracle on which \mathcal{A} issues $\Pi_{U,i}^{\text{gm}}.\text{Test}()$. When \mathcal{A} makes encryption or decryption requests for $\mathbf{k}_{\mathbf{i}^*}$, \mathcal{D} forwards the requests to its own challenger. Note that because the key at $\mathbf{k}_{\mathbf{i}^*}$ is randomly distributed from the view of \mathcal{A} , and because \mathcal{D} 's challenger selects the encryption key at random, the responses of \mathcal{D} 's challenger are distributed just as \mathcal{A} 's challenger in its game. Specifically, when \mathcal{A} makes its Test query, \mathcal{D} forwards

the messages (m_0, m_1) to its own challenger, and it returns the resulting encryption to \mathcal{A} ; when \mathcal{A} outputs a bit $b' \in \{0, 1\}$, \mathcal{D} outputs the same bit, and wins with the same probability that \mathcal{A} wins.

However, \mathcal{D} does not know the key on which \mathcal{A} will call $\text{Test}()$, and moreover it cannot wait for \mathcal{A} to call $\text{Test}()$ to begin forwarding queries to its own adversary, because \mathcal{A} may request encryptions under some key before it calls $\text{Test}()$ on that key. Therefore, \mathcal{D} must guess the key on which \mathcal{A} will call $\text{Test}()$. First, \mathcal{D} uniformly at random guesses the session on which \mathcal{A} will call $\text{Test}()$, and guesses correctly with probability $\frac{1}{n_S}$. Second, \mathcal{D} guesses the key within that session on which \mathcal{A} will guess $\text{Test}()$. Observe that when \mathcal{A} makes n $\text{Evolve}()$ queries to define new keys, in fact there are 2^n keys defined. This is an exponential number of keys. However, \mathcal{A} must be a polynomial-time adversary and therefore can only explore n_q keys by making queries to them. \mathcal{D} therefore guesses that *this* key will be the one tested by \mathcal{A} with probability $\frac{1}{n_q - l}$ each time that \mathcal{C} issues a query to a new key, where l is a counter that tracks how many times \mathcal{A} has challenged a key. This scheme adaptively guesses the next key uniformly at random with total probability $\frac{1}{n_q}$ for each key.

It follows that

$$\text{Adv}^{\text{cca}}(\mathcal{D}) \geq \frac{1}{n_S \cdot n_q} \left| \Pr[\mathcal{A} \text{ wins Game2}] - \frac{1}{2} \right| \quad (3)$$

□

Using the above lemmas we complete the proof by computing the advantage of the adversaries \mathcal{B} , \mathcal{C} , and \mathcal{D} with respect to \mathcal{A} .

$$\begin{aligned} \text{Adv}^{\text{gm}}(\mathcal{A}) &= 2 \cdot \left| \Pr[\mathcal{A} \text{ wins } G_0] - \frac{1}{2} \right| \\ &= 2 \cdot \left| \Pr[\mathcal{A} \text{ wins } G_0] - \frac{1}{2} + \Pr[\mathcal{A} \text{ wins } G_1] - \Pr[\mathcal{A} \text{ wins } G_1] \right. \\ &\quad \left. + \Pr[\mathcal{A} \text{ wins } G_2] - \Pr[\mathcal{A} \text{ wins } G_2] \right| \\ &\leq 2 \cdot \left| \Pr[\mathcal{A} \text{ wins } G_0] - \Pr[\mathcal{A} \text{ wins } G_1] \right| \\ &\quad + 2 \cdot \left| \Pr[\mathcal{A} \text{ wins } G_1] - \Pr[\mathcal{A} \text{ wins } G_2] \right| \\ &\quad + 2 \cdot \left| \Pr[\mathcal{A} \text{ wins } G_2] - \frac{1}{2} \right| \\ &\leq 2 \cdot n_S \cdot \text{Adv}^{\text{gka}}(\mathcal{B}) + 2 \cdot n_S \cdot n \cdot \text{Adv}^{\text{gm}}(\mathcal{C}) + 2 \cdot n_S \cdot n_q \cdot \text{Adv}^{\text{cca}}(\mathcal{D}) \end{aligned}$$

□

8 Extension to Dynamic Groups

In this section we describe the extension of our framework to dynamic groups. We note that the extension is a feature of the key lattice, and group messaging protocols that are defined with respect to a key lattice such as ours should be able to adapt the protocol to permit dynamic membership with little cost, save application-specific rules. We will not re-prove our theorems for security of our protocol with dynamic groups as the extensions are straightforward.

Our key lattice abstraction naturally extends to the dynamic setting. Recall that our general definition of the key lattice is an n -dimensional space, where n is the set of all players. Each player is mapped to a dimension via a canonical ordering function ϕ . Players who evolve the group key only do so in their respective dimensions. Adding or removing parties simply corresponds to defining updates in more or fewer dimensions.

In our extension, parties running the protocol maintain a lossless compression of the lattice by tracking only indices corresponding to participants in the protocol. Let \mathcal{P} be the set of all identities. The function $\phi: \mathcal{P} \rightarrow \mathbb{N}$ assigns a canonical ordering to \mathcal{P} . Observe that an uncompressed index vector that describes the index of every party is very large if the set of identities is large. Rather than sending messages containing a full index vector, parties compress their representations by only including entries for parties who are part of the execution, listed in order according to ϕ .

Adding Members For a group G that starts an execution, all of the members of G are initialized at point $\mathbf{0}$ in their dimension. In other words, there exists a key at the lattice point $\mathbf{0}$ which is the initial group key. All other parties not in G are initialized at point \perp (which is compressed by omitting the dimension from any index vectors). At any point, a member can add a new party to the group with a message $(\text{AddMember}, U, \mathbf{i})$, where U is the identity of the newly added member and \mathbf{i} is the maximal lattice point of the adding party (which includes the new dimension). The adding party also sends $k_{\mathbf{i}}$ to party U . In response, all parties that receive the message will begin receiving from and sending messages to U .

When a new member is added to the group, that member is defined to belong to the group only for messages (and key updates) that *succeed* the **AddMember** message on the key lattice. (The newly added party is defined to *not be in the group* with respect to messages that are concurrent to the **AddMember** message.) When U is added to the group, the adding party must send the party a key with which to initialize its lattice (at \mathbf{i}). To prevent V from learning group messages from before it was a member, the adding party must associate the addition with a key update, which it sends along with the **AddMember** message to the other parties in the group. If two parties add the same group member concurrently, the new member is defined to be part of the group for all messages that succeed *either* of the corresponding **AddMember** messages. Where the two concurrent updates meet in the lattice, the two key updates provided by the adding parties compose naturally by the commutativity of **KeyRoll**. Note that since we assume PKI, the long-term public key of U can be fetched by all the parties.

Removing Members Similarly to adding members, any member of a group can remove another member with a message $(\text{RemoveMember}, U, \mathbf{i})$, where U is identity of the member to be deleted, and \mathbf{i} is the lattice point for which U should no longer receive updates. (The deleting party sets \mathbf{i} as its maximal lattice point at the time it sends the **RemoveMember** message.) A group member is defined to be removed from the group for all messages that *succeed* the **RemoveMember** message, meaning all lattice points that are greater than \mathbf{i} . For messages that are concurrent to **RemoveMember** in the key lattice, the party is still in the group.

If two parties attempt to concurrently remove the same member from the group (and that party is already in the group), then the party is removed for all messages that succeed either **RemoveMember** message. If two parties attempt to concurrently add and remove a member from the group who is not yet in the group, then the **RemoveMember** message must be invalid and only the **AddMember** is processed. If two parties attempt to concurrently add and remove a member who is already in the group, then the **AddMember** message must be invalid and only the **RemoveMember** message is processed.

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.

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

- AAN⁺22a. Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559, 2022. <https://eprint.iacr.org/2022/559>.
- AAN⁺22b. Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 815–844. Springer, Heidelberg, May / June 2022.
- ACD19. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.
- ACDT20. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.
- ACDT21. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1463–1483. ACM Press, November 2021.
- ACJM20. Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, November 2020.
- AHKM22. Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 69–82. ACM Press, November 2022.
- AJM22. Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 34–68. Springer, Heidelberg, August 2022.
- BBM⁺20. Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-11, Internet Engineering Task Force, December 2020. Work in Progress.
- BCP01. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Provably authenticated group Diffie-Hellman key exchange – the dynamic case. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 290–309. Springer, Heidelberg, December 2001.
- BCP02. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Dynamic group Diffie-Hellman key exchange under standard assumptions. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 321–336. Springer, Heidelberg, April / May 2002.
- BCPQ01. Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. Provably authenticated group Diffie-Hellman key exchange. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 255–264. ACM Press, November 2001.

- BDR20. Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, November 2020.
- BFS⁺13. Christina Brzuska, Marc Fischlin, Nigel P. Smart, Bogdan Warinschi, and Stephen C. Williams. Less is more: relaxed yet composable security notions for key exchange. *Int. J. Inf. Sec.*, 12(4):267–297, 2013.
- BFWW11. Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 51–62. ACM Press, October 2011.
- BGB04. Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84, 2004.
- BM08. Emmanuel Bresson and Mark Manulis. Securing group key exchange against strong corruptions. In Masayuki Abe and Virgil Gligor, editors, *ASIACCS 08*, pages 249–260. ACM Press, March 2008.
- BMS07. Emmanuel Bresson, Mark Manulis, and Jörg Schwenk. On security models and compilers for group key exchange protocols. In Atsuko Miyaji, Hiroaki Kikuchi, and Kai Rannenberg, editors, *IWSEC 07*, volume 4752 of *LNCS*, pages 292–307. Springer, Heidelberg, October 2007.
- BMS20. Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for Authentication and Key Establishment*. Information Security and Cryptography. Springer Berlin Heidelberg, Berlin, Heidelberg, 2020.
- Brz13. Christina Brzuska. *On the foundations of key exchange*. PhD thesis, Darmstadt University of Technology, Germany, 2013.
- CCD⁺20. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020.
- CCG16. Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In Michael Hicks and Boris Köpf, editors, *CSF 2016 Computer Security Foundations Symposium*, pages 164–178. IEEE Computer Society Press, 2016.
- CCG⁺18. Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018.
- CGR14. Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2014.
- CHK21. Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1847–1864. USENIX Association, August 2021.
- Coh18. Katriel Cohn-Gordon. *On secure messaging*. PhD thesis, University of Oxford, UK, 2018.
- CS03. Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.
- DGP22. Benjamin Dowling, Felix Günther, and Alexandre Poirrier. Continuous authentication in secure messaging. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part II*, volume 13555 of *LNCS*, pages 361–381. Springer, Heidelberg, September 2022.
- FKKP19. Georg Fuchsbauer, Chethan Kamath, Karen Klein, and Krzysztof Pietrzak. Adaptively secure proxy re-encryption. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 317–346. Springer, Heidelberg, April 2019.
- HKP⁺21. Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, November 2021.
- ITW82. Ingemar Ingemarsson, Donald T. Tang, and C. K. Wong. A conference key distribution system. *IEEE Trans. Inf. Theory*, 28(5):714–719, 1982.
- KBB17. Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. 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)*, pages 435–450. IEEE, 2017.
- KPT04. Y. Kim, A. Perrig, and G. Tsudik. Group key agreement efficient in communication. *IEEE Transactions on Computers*, 53(7):905–921, July 2004.
- KY03. Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 110–125. Springer, Heidelberg, August 2003.
- Lam78. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

- PP22. Jeroen Pijnenburg and Bertram Poettering. On secure ratcheting with immediate decryption. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part III*, volume 13793 of *LNCS*, pages 89–118. Springer, Heidelberg, December 2022.
- PRSS21. Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 148–176. Springer, Heidelberg, May 2021.
- Res19. Eric Rescorla. Subject: [MLS] TreeKEM: An alternative to ART. MLS Mailing List, 2019. <https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/>, Accessed 2022-01-19.
- RMS18. Paul Rösler, Christian Mainka, and Jörg Schwenk. 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, London, United Kingdom, April 24-26, 2018*, pages 415–429. IEEE, 2018.
- STW96. Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-Hellman key distribution extended to group communication. In Li Gong and Jacques Stern, editors, *ACM CCS 96*, pages 31–37. ACM Press, March 1996.
- Wei19. Matthew A. Weidner. Group messaging for secure asynchronous collaboration. M.phil thesis, University of Cambridge, 6 2019. <https://mattweidner.com/acs-dissertation.pdf>.
- Wha21. WhatsApp Inc. Whatsapp encryption overview. Online, Sep 2021. <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>, Accessed 2022-01-19.
- WKHB21. Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2024–2045. ACM Press, November 2021.