An extended abstract [CR25a] of this article appears in the proceedings of IACR CRYPTO 2025. This is the full version [CR25b] of this article, which is available in the IACR eprint archive. A proof of concept implementation of the core building block by Maxim Grüninger [Grü25] is available on Github.

GURKE: Group Unidirectional Ratcheted Key Exchange

Daniel Collins[1] and Paul Rösler[2]

[1] Texas A&M University
danielpatcollins@gmail.com
[2] FAU Erlangen-Nürnberg
paul.roesler@fau.de

Abstract. Continuous Group Key Agreement (CGKA) is a primitive with which members of a group can continuously establish shared keys. With every interaction, these members also update their individual, local secrets such that temporary corruptions of these secrets only affect the security of shared keys established shortly before (Forward Security; FS) and after the corruption (Post-Compromise Security; PCS). Due to these interactive updates—possibly enriched by dynamic group membership changes—, CGKA is a very powerful but also very complex primitive.

In this work, we limit the power of CGKA to identify and analyze its core components. More concretely, we consider the case that all members of a group are always either senders or receivers. Thus, the interaction is strictly unidirectional from the former to the latter: a group of senders *Alice* establishes shared keys with a group of receivers *Bob*. With every shared key, Alice updates her local state to achieve FS and PCS; when receiving an established key, each Bob also updates their local state to achieve FS. This notion naturally lifts the so called *Unidirectional Ratcheted Key Exchange* concept (Bellare et al., Crypto 2017; Poettering and Rösler, Crypto 2018) to the group setting and, thereby, captures and generalizes Signal's Sender Key Mechanism, which is the core of WhatsApp and Signal's group chat protocols. We modularize this concept of *Group Unidirectional RKE* (GURKE) by considering either single or multiple senders, single or multiple receivers, and static or dynamic membership on each of both sides of the group.

To instantiate these new primitives, we develop a building block called Updatable Broadcast KEM (UB-KEM). Using UB-KEM, our GURKE constructions for static groups only use standard Key Encapsulation Mechanisms (KEMs) and induce only a *constant* communication overhead. Our GURKE constructions for dynamic groups are based on general Non-Interactive Key Exchange (NIKE) and offer a constant communication overhead as long as the set of members is unchanged; only for adding and removing users, a communication overhead logarithmic in the group size is induced. We discuss the benefits of replacing the Sender Key Mechanism in Signal and WhatsApp with our constructions, and demonstrate their practicality with a performance evaluation of our proof of concept UB-KEM implementation.

1 Introduction

Motivated by the standardization of the Messaging Layer Security (MLS) protocol [BBR+23], the academic research community has contributed a long list of articles that propose new design ideas for secure group messaging (e.g., [CCG+18, CHK21, ACDT20, KPPW+21, ACJM20, BDR20, WKHB21, AJM22, ACDT21, AAB+21, HKP+21, AHKM22, BDT22, AAN+22, BDG+22, BCV23, HKP22, WPBB23, AMT23, LGGR23, ANPPP23, ADJ24]). Before that, the academic community was similarly motivated by the success of Signal's two-party Double Ratchet messaging protocol [PM16, CCD+20, ACD19], and published a long list of articles with new two-party messaging design ideas.

Disassembling Messaging. Bellare et al. [BSJ+17] were the first to take one step back and, instead of developing new ideas, disassembled two-party messaging and reduced it to a primitive notion: a sender Alice continuously establishes symmetric keys for the use in external protocols (e.g., a secure channel) with receiver Bob, and neither ever switches roles. After each send operation, Alice updates her local key material, and after each receive operation, Bob updates his local key material, both aiming to reduce the harm of temporary key material corruptions on previously and subsequently established symmetric keys. Thus, this notion captures what is called Forward Security (FS; old symmetric keys remain secure) and Post-Compromise Security (PCS; future symmetric keys will be secure). Poettering and Rösler [PR18] called this primitive notion *Unidirectional Ratcheted Key Exchange* (URKE) and extended the underlying approach: they developed *Sesquidirectional* RKE, which is a variant that permits Bob to respond to Alice

for stronger FS and PCS; yet, these responses do not establish symmetric keys for the use in external protocols. Only their fully *Bidirectional* RKE notion treats Alice and Bob equally.

Understanding Messaging. These restricted notions of Ratcheted Key Exchange proved to be useful to study fundamental questions. For example, Balli et al. [BRV20] proved for URKE that, if adversaries can attack the random coins of Alice, strongly secure URKE constructions necessarily rely on expensive building blocks, called Key-Updatable KEM [RSS23]; Dowling et al. [DHRR22] consider URKE to define and fulfill a very strong notion of anonymity under state exposure. Studying messaging fundamentally is as least as meaningful for *groups* as for two-party communication, which motivates our work.

Ratcheted Key Exchange. Besides limiting the direction of interaction, the first implicit step by Bellare et al. [BSJ+17] for reducing complexity is to disregard the *message encryption* part of messaging, and only focus on *continuously establishing symmetric keys*; adding encryption on top of that can be done modularly. The second complexity reduction step is to consider continuous, aka. ratcheted, key exchange in an already established session; adding the initial key agreement that establishes the first shared secrets between the participants also works modularly [BFWW11]. Both of these modular simplifications are now standard in the literature.

In conclusion, the simplest form of *Ratcheted Key Exchange* (RKE) consists of algorithms init, snd, and rcv, where init $\rightarrow_{\$} (st_1, \ldots, st_n)$ initializes session states for all protocol participants, $\mathrm{snd}(st) \rightarrow_{\$} (st, k, c)$ establishes a symmetric key k, outputs a ciphertext c for all remaining participants, and updates the caller's local secret state st, and $\mathrm{rcv}(st, c) \rightarrow (st, k)$ computes the same symmetric key k and compatibly updates the caller's local secret state st. The most powerful and realistic notion of interaction treats all participants equally by letting them establish keys via algorithm snd and letting them compute the established keys of others via algorithm rcv. Restricting this *bidirectional* notion by assigning each participant a fixed sender or receiver role yields *Unidirectional* RKE (URKE).

Group Communication. The vast majority of articles on group messaging and its primary building block *Continuous Group Key Agreement* (CGKA) considers fully multi-directional interaction, many capturing dynamic group membership. In this work, instead of starting with these advanced notions and reducing complexity, we take two-party URKE as our starting point and incrementally add group features.

Our simplest *Group URKE* (GURKE) variant considers a single sender that establishes symmetric keys with a static group of multiple receivers (Static Single-Sender Multi-Receiver; S-SSMR). After this, we switch perspectives by considering a static group of senders who establish symmetric keys with a single receiver (S-MSSR). This leads to the final static case, in which multiple senders establish symmetric keys with multiple receivers (S-MSMR). Syntactically, the only difference between these three cases emerges in the initialization: algorithm init either outputs one or multiple sender states and one or multiple receiver states.

Signal's Sender Key Mechanism. By considering unidirectional communication, we do not only focus on identifying the fundamental essence of group messaging *in theory*. Instead, we capture the core of *real-world* group chats in WhatsApp, Signal and other widely deployed messengers that are based on Signal's Sender Key Mechanism [Mar14, BCG23]. This Sender Key Mechanism is a weakly secure instantiation of our above sketched concept of S-SSMR GURKE. More concretely, our notion of S-SSMR abstractly captures the Sender Key Mechanism and our concept of GURKE generalizes it.

On the cryptographic level, every fully functional group chat in WhatsApp, Signal, and other messengers is composed of n Sender Key S-SSMR Group Unidirectional RKE instances. Thereby, each group member is a sender to all other members and, simultaneously, a receiver to each other member's sender role. Thus, each group member holds one sender state for sending to the group and $n-1$ receiver states for receiving from other members in the group. As we will elaborate on in Section 5.4, compared to our new GURKE constructions, the Sender Key Mechanism is, on balance, less secure and efficient when being used to build fully functional group chats via this parallel composition design.

Dynamic Membership. Based on the above three static membership variants, we develop a new perspective on dynamic groups: the single sender can dynamically change the set of receivers who the sender establishes keys with (D-SSMR); all senders can change the set of senders who can establish keys with the single receiver (D-MSSR); and all senders can change the sets of senders and receivers, where the former establish keys with the latter (D-MSMR). To capture dynamic membership syntactically, we add

algorithms add, rmv, and proc, which add users, remove users, and allow senders to process information about such membership changes; algorithms init, snd, rcv stay the same.

Security Requirements. For all variants of GURKE that we consider, we require that the symmetric keys output by the senders and receivers look random, even if the senders are *ever* corrupted (FS and PCS), and even if one of the current receivers will be corrupted in the *future* (FS). Note that receivers cannot recover from corruptions to achieve PCS since their computations are strictly deterministic. Nevertheless, we require that corruptions of every user, both senders and receivers, are harmless before they join and after they exit a group.

Adversaries in our definitions are permitted to actively manipulate the traffic on the network; against such attacks, the constructions are required to guarantee Implicit Authentication (IA). This means that, upon receiving a manipulated ciphertext, this ciphertext is either immediately rejected or it will diverge the receiver's local secrets irrevocably from those of the sender; the latter is, for example, enforced when the adversary *trivially impersonates*[3] the sender. With this notion of IA, we follow strong security definitions of RKA [PR18, JS18] and CGKA [ACJM20, AMT23] and we capture an implicit variant of *Recover Security* [DV19][4].

Static Group Constructions. Interestingly, we can use and generalize design ideas from two-party URKE [BSJ+17, PR18] to build efficient GURKE constructions for static groups. Concretely, we instantiate the simplest notion, namely S-SSMR GURKE, with a construction that generates an ordinary Key Encapsulation Mechanism (KEM) key pair upon initialization; the initial encapsulation key ek is given to the single sender, and the initial decapsulation key dk is given to *all* receivers.

Proc snd$((ek, ssk))$	**Proc** rcv$((dk, svk), (c, svk', \sigma))$
00 $(k', c) \leftarrow_{\$}$ K.enc(ek)	06 Require S.vfy$(svk, (c, svk'), \sigma)$
01 $(svk', ssk') \leftarrow_{\$}$ S.gen	07 $k' \leftarrow$ K.dec(dk, c)
02 $\sigma \leftarrow_{\$}$ S.sgn$(ssk, (c, svk'))$	08 $(k, s) \leftarrow$ H(k', c, svk')
03 $(k, s) \leftarrow$ H(k', c, svk')	09 $(_, dk') \leftarrow$ K.gen(s)
04 $(ek', _) \leftarrow$ K.gen(s)	10 Return $((dk', svk'), k)$
05 Return $((ek', ssk'), (c, svk', \sigma), k)$	
Proc snd$((PK, ssk))$	**Proc** rcv$((SK, svk), (pk^*, svk', \sigma))$
11 $(pk^*, sk^*) \leftarrow_{\$}$ N.gen	20 Require S.vfy$(vk, (c, svk'), \sigma)$
12 $(svk', ssk') \leftarrow_{\$}$ S.gen	21 For all sk in path SK (leaf to root):
13 $\sigma \leftarrow_{\$}$ S.sgn$(ssk, (pk^*, svk'))$	22 $k' \leftarrow$ N.key(sk, pk^*)
14 For all pk in tree PK (leaves to root):	23 $(k, s) \leftarrow$ H(k', pk^*, svk')
15 $k' \leftarrow$ N.key(sk^*, pk)	24 $(_, sk') \leftarrow$ K.gen(s)
16 $(k, s) \leftarrow$ H(k', pk^*, svk')	25 Add sk' to path SK'
17 $(pk', _) \leftarrow$ N.gen(s)	26 Return $((SK', svk'), k)$
18 Add pk' to tree PK'	
19 Return $((PK', ssk'), (pk^*, svk', \sigma), k)$	

Fig. 1. Simplified send and receive algorithms of our static (top) and dynamic (bottom) GURKE constructions.

When sending, a key k' is encapsulated via K.enc$(ek) \rightarrow_{\$} (k', c)$; key k' is fed into a random oracle H$(k', c) \rightarrow (k, s)$ that outputs the symmetric output key k that shall be established with all receivers; the second output s is used as a seed to deterministically generate a new KEM key pair (ek', dk'); the sender ignores dk' and uses ek' to replace the old encapsulation key ek for the next send operation. Although simplified, this protocol, shown in the upper half of Figure 1, illustrates the core mechanism with which the sender establishes a new symmetric key and updates their own key material.

The receiver decapsulates ciphertext c via K.dec$(dk, c) \rightarrow k'$ and performs the same update method via the random oracle. This yields the symmetric output key k as well as the seed for generating their updated decapsulation key dk'.

Based on this idea, the two remaining static GURKE cases, S-MSSR and S-MSMR, can be derived naturally: for the S-MSSR case, each sender generates their own KEM key pair, and the single receiver stores all corresponding decapsulation keys. Sending and receiving follows the same protocol. The S-MSMR case replicates the receiver state at initialization.

[3] An impersonation is "trivial" if the adversary corrupts the sender's key material in order to create a validly looking, forged ciphertext.

[4] R-Recover Security in the language of [CDV21, BCC+23a].

Intuitively, the random-oracle-based mechanism guarantees that key updates cannot be reverted; furthermore, when performed with different inputs due to a forged ciphertext, output key material irrevocably diverges. Our final construction, additionally, uses signatures, which, in total, guarantees FS, PCS, and IA.

We show that all three constructions are asymptotically optimal with respect to the state sizes and ciphertext sizes. Constant-sized ciphertexts and states (in the security parameter) are unsurprisingly optimal for S-SSMR: it has been shown that two-party URKE implies KEM [CDV21], and constant-sized parameters are clearly necessary for KEM. Observe that the receiver's state size is linear in the number of senders in our S-MSSR. We prove in Appendix E that this is asymptotically inevitable and, thereby, optimal. Intuitively, we show that multi-sender GURKE schemes can be used generically to build a Puncturable Encryption (PE) scheme; based on this, we can apply a recent lower bound on the size of PE secret keys by Barooti et al. [BCC+23a].

Dynamic Group Constructions. The constructions for the dynamic cases are substantially more sophisticated because they need to separate secrets between receivers: once a receiver is removed from the group, their local secrets should be harm- and useless with respect to future symmetric keys established in the group.

For this, we make use of a tree of key pairs, which is a common approach in group messaging. Yet, in contrast to existing tree-based constructions [BBR18, BBR+23], we combine this tree with a method that only requires a constant sized ciphertext to update all nodes' keys simultaneously. This method is based on Non-Interactive Key Exchange (NIKE) with which a single key pair (pk^*, sk^*) can be used to establish symmetric keys k'_i with an arbitrary number of public keys pk_i. As shown in the simplified illustration in the lower half of Figure 1, a GURKE sender prepares such an NIKE key pair and derives a shared key k'_i with each node's public key pk_i in the tree. Every node's *updated* public key pk'_i is derived by feeding symmetric key k'_i to the random oracle $\mathrm{H}(k'_i, pk^*) \to (k_i, s)$ and using the output seed s for a subsequent key generation—akin to the static GURKE case described above. The final shared established GURKE group key k is the random oracle output k_{root} of the tree's root. The GURKE ciphertext sent to all receivers simply consists of public key pk^*. Each receiver then applies the same operations—NIKE key computation, key derivation via the random oracle, and seeded deterministic key-pair generation—for the path in the tree from their individual leaf to the shared root.

As long as the group remains unchanged, our dynamic GURKE constructions, thereby, have a constant communication overhead. Only for adding and removing group members, an overhead logarithmic in the group size is induced, which is caused by renewing keys on paths in the tree. Prior constructions of CGKA that achieve similar levels of FS, PCS, and IA rely on impractical tools like Hierarchical Identity-Based Encryption (HIBE) [ACJM20, AMT23]. To simplify the description of our constructions, we capture this design with an intermediate notion called Updatable Broadcast KEM (UB-KEM), which we believe to be a primitive of independent interest. Our constructions of UB-KEM, which support either a static or dynamic set of receivers, abstract away the KEM and random oracle operations in the upper half of Figure 1, and (in a simplified form) the NIKE, random oracle and tree operations in the lower half, respectively.

Dynamic Operations. Our full D-MSMR construction uses UB-KEM as a building block, yet itself is complex; we provide a detailed description in Section 6.3 (Figures 12 and 13). Each sender manages its own instance of UB-KEM and has their own (ratcheted) signature key pair, which at its core are used as in Figure 1. Receivers therefore maintain one UB-KEM secret key and signature verification key per sender.

Unlike the static case, sender-to-sender communication is needed to inform senders and receivers of group changes. When a sender S adds a receiver R, for example, all senders must be able to encapsulate to R. In this case, sender S embeds KEM secrets, one per sender, in R's initial state, so the other senders can add R to their UB-KEM instances by encrypting to R a new, up-to-date UB-KEM receiver secret key. In general, a sender may process several membership update messages from other senders before encapsulating their next key. In such cases, senders therefore queue and then apply all such operations when they next send, which is processed in the same order by receivers. We efficiently implement sender adds in particular through our UB-KEM's novel *fork* algorithm, which allows a sender S to locally spawn a 'child' instance of UB-KEM with the same set of receivers as S; these receivers then learn about the new sender via a constant-sized ciphertext from S.

Cases \ Sizes	$\lvert st_{\mathtt{S}}\rvert$	$\lvert st_{\mathtt{R}}\rvert$	$\lvert c_{\mathrm{snd}}\rvert$	$\lvert c_{\mathrm{mem},\mathtt{S}}\rvert$	$\lvert c_{\mathrm{mem},\mathtt{R}}\rvert$
S-SSMR	2	2	4		
S-MSSR	2	$2n_{\mathtt{S}}$	4		
S-MSMR	2	$2n_{\mathtt{S}}$	4		
D-SSMR	$3+2n_{\mathtt{R}}$	$2+\log n_{\mathtt{R}}$	4		Add R: $2+2\log n_{\mathtt{R}}$ Rmv R: $3+2\log n_{\mathtt{R}}$
D-MSSR	4	$3n_{\mathtt{S}}$	4	0	Add S: 6 Rmv S: 4
D-MSMR	$3+2n_{\mathtt{R}}$	$n_{\mathtt{S}}(2+\log n_{\mathtt{R}})$	4	Add R: $n_{\mathtt{S}}-1$ Else: 0	Add S/R: 6 / $2+2\log n_{\mathtt{R}}$ Rmv S/R: 4 / $3+2\log n_{\mathtt{R}}$

Table 1. Summary of concrete state sizes and ciphertext sizes of our constructions. c_{snd} is output by algorithm snd, and $(c_{\mathrm{mem},\mathtt{S}}, c_{\mathrm{mem},\mathtt{R}})$ are output by algorithms add or rmv and sent to the (remaining) senders and receivers, respectively. We neglect non-cryptographic information (e.g., bookkeeping variables in the local states); using Hashed ElGamal KEM, Hashed Diffie-Hellman key exchange, and Schnorr or ECDSA signatures, each unit corresponds to a single group element, exponent, or bit string in the length of the security parameter. "Add S" denotes the cost of adding a sender; this style is used naturally for other operations.

Crucially, communication between users is *asynchronous*—we assume messages from a given sender are delivered in-order, but messages can otherwise be interleaved and delayed in the network. This sets our construction apart from MLS and typical CGKA constructions that require users to agree on a total ordering of messages for group key and membership changes. In particular, our construction allows senders to concurrently establish keys. Due to the asynchrony we support, senders and receivers may have different views of the current set of senders and receivers over time. Therefore, a sender sends (a compressed form of) its current membership to receivers upon each encapsulation as well as to senders upon adding/removing another party. Overall, the construction requires careful bookkeeping to ensure correctness and security.

Performance. We summarize the performance of our GURKE constructions. For static groups, the intermediate UB-KEM primitive is based on a standard KEM; using ElGamal, a UB-KEM public key, secret key, and ciphertext consist of a single group element, exponent, and group element, respectively. For dynamic groups, the intermediate UB-KEM primitive is based on a tree of NIKE key pairs; using Hashed Diffie-Hellman key exchange [FHKP13], a UB-KEM public key and secret key consist of $2n$ group elements and $\log n$ exponents, respectively, standard UB-KEM ciphertexts consist of a single group element, group membership changes are communicated using $2\log n - 2$ (resp. $2\log n - 1$) group elements for adds (resp. removes), where n is the number of receivers, and forking produces a ciphertext containing a single group element.

Table 1 summarizes the resulting performance of our GURKE constructions. Based on the properties of UB-KEM, ciphertexts sent by algorithm snd are always of constant size; using Hashed Diffie-Hellman, this is a single group element.

As it is the core building block of our dynamic GURKE constructions, we provide a proof of concept implementation of our dynamic UB-KEM construction in [Grü25]. Furthermore, we present a performance evaluation thereof in Appendix B.1 that demonstrates its practicality.

Application to Full Group Chats. As motivated above, beyond studying unidirectional interaction in group messaging fundamentally, our results show how fully functional group messaging can be built with several attractive features over Signal's Sender Key Mechanism. Table 2 compares the communication overhead of Sender Key and MLS's messaging protocol with an approach based on our D-MSMR GURKE construction from Section 6.3; we provide a detailed explanation of our approach in Section 5.4.

Contributions. Our results can be summarized as follows: We develop a generic and modular syntactic perspective on unidirectional ratcheted key exchange in static and dynamic groups of users (Section 3); based on this, we specify security that is almost as strong as possible under temporary state corruptions, which entails Forward Security (FS), Post-Compromise Security (PCS), and Implicit Authentication (IA); although our definitions are simpler than those of general CGKA, the remaining level of complexity seems unavoidable, which might be a valuable insight, too. By extending techniques from two-party ratcheted key exchange [BSJ+17, PR18], we design secure constructions for the static group case (Section 5); while comparably efficient, our design outperforms the security guarantees of the Sender Key

Via \ Operation		Add	Remove	Send	IA	FS	PCS
Sender Key	$\|c\|$	$3n$	$3n^2$	3	○	●	◑
	Exp.	$3n$	$3n^2$	0			
D-MSMR	$\|c\|$	$4n \log n + \mathcal{O}(n)$	$2n \log n + \mathcal{O}(n)$	4	●	●	◑
	Exp.	$4n^2 + \mathcal{O}(n \log n)$	$4n^2 + \mathcal{O}(n \log n)$	4			
MLS	$\|c\|$	$2n \log n + \mathcal{O}(n)$	$2n \log n + \mathcal{O}(n)$	$2n \log n + \mathcal{O}(n)$	○	◑	●
	Exp.	$n + \mathcal{O}(\log n)$	$n + \mathcal{O}(\log n)$	$n + \mathcal{O}(\log n)$			

Table 2. Communication overhead and number of exponentiations executed for fully functional group messaging (adding and removing members and sending payload), as well as security in terms of Implicit Authentication (IA), Forward Security (FS) and Post-Compromise Security (PCS), which is achieved fully (●), partially (◑) or not (○), based on parallel Sender Key Mechanism instances, a single D-MSMR instance, and MLS as follows: we consider commit messages with a single proposal for Add/Remove and no proposal for Send, and consider *best-case* performance (namely we assume an unblanked tree). Costs are counted as in Table 1. Full details are in Sections 4.3 and 5.4.

Mechanism [Mar14] that is used in Signal, WhatsApp, and other messengers. Particularly for dynamic groups (Section 6), we define Updatable Broadcast KEM, which might be a primitive of independent interest (Section 4); for its construction, we develop a new tree-based design with remarkable performance, which is demonstrated by our implementation [Grü25]: while comparable levels of FS, PCS, and IA were so far only achieved with impractical identity-based building blocks [ACJM20, AMT23], we use efficient standard techniques. Based on these dynamic group constructions, we informally propose an attractive alternative to building full group messaging via parallel Signal Sender Key sessions (Section 5.4). For some of our constructions, we prove optimality via lower bounds (Appendix E).

Future Work. For coordination between senders and for the initial contact with new users, we assume secure underlying pairwise channels, which indeed exist in real-world messengers; we transparently make use of them as little as possible, but avoiding this assumption entirely would likely increase the complexity substantially (e.g., by adding the need for public key distribution and explicit initial key exchange) with little anticipated gain in insight. Avoiding this assumption in a larger communication model remains an open problem. Another question that our work provokes is how to extend our definitions and constructions to capture the receivers' ability to respond to senders. In particular, two natural extensions would be (1) allowing receiver(s) to respond for better PCS (which could be called "Group *Sesquidirectional* RKE" in the notation of [PR18]) and (2) letting receiver(s) establish keys with these responses (which could be called "Group *Bidirectional* RKE" in the notation of [PR18]). Both of these steps may have the six flavors that we propose here (i.e., static/dynamic group, single/multi sender/receiver). An orthogonal aspect that we do not cover is the security of GURKE against attacks that go beyond state corruptions. In particular, we do not consider attacks against the senders' random coins. For the type of definitions specified in this work it was proven by Balli et al. [BRV20] that weak randomness imposes the use of inefficient HIBE-like building blocks [RSS23]. Consequently, we decided that assuming *good* randomness is a reasonable starting point for this line of research; moreover, we believe that considering bad randomness only has substantial value in the dynamic multi-sender case. Nevertheless, we informally discuss the effect of weak randomness in Appendix B.2. Further functional features like administration privileges or immediate decryption under out-of-order delivery as well as developing extensions that yield post-quantum security are also left as open problems. Overall, after the CGKA literature developed many sophisticated solutions, we aim to build a new foundational perspective on group messaging here, based on which more advanced questions can be studied in the future.

Further Related Work. Hashimoto et al. [HKP+21] build a CGKA from multi-receiver public-key encryption; their focus is on achieving post-quantum security and low communication costs on the receiver side at the cost of linear communication overhead on the sender side. Bienstock et al. [BDT22] analyze Multicast Key Agreement as an alternative unidirectional building block for group messaging resembling D-SSMR and propose highly efficient constructions at the cost of relatively weak security guarantees; for example, they do not consider fully active adversaries and, thereby, do not require authenticity. A deeper discussion on different levels of FS, PCS, and IA for prominent TreeKEM-like constructions [BBR18, ACDT20, ACJM20, AMT23] is at the end of Section 4.3. An even simpler approach to building group messaging than using GURKE is proposed by Bienstock et al. [BDRW24]: they add update mechanisms

to KEMs to achieve FS and PCS. Yet, a major drawback is that the induced communication overhead is linear in the number of group members since they do not propose a mechanism for encrypting to multiple receivers simultaneously besides trivially encrypting to each receiver individually. Furthermore, they do not offer immediate FS due to coarse-grained key updates. Finally, Balbás et al. [BCG23] provide the first formal security analysis of Signal's Sender Key Mechanism; their definitional framework mainly focuses on the peculiarities of this protocol and how it is composed with other building blocks to offer fully functional group messaging.

2 Notation

To denote the set of integers up to n, we use $[n] = \{1, \ldots, n\}$ and let $[0] = \emptyset$; the power-set of set X is denoted $\mathcal{P}(X)$. We make use of queues that can be accessed and modified with algorithms Q.enqueue that adds an element at the end, Q.dequeue that retrieves and removes an element from the first position, Q.first that only retrieves the first element without removing it, Q.last that retrieves the element at the end, and Q.setlast that overwrites the last element. When presenting security games, we use the command "Require P" for a predicate P, which is such that if the predicate evaluates to false, the oracle immediately outputs $\perp$.

Standard definitions of Pseudorandom Functions, Digital Signatures, Key Encapsulation Mechanisms, and Non-Interactive Key Exchange are provided in Appendix A

3 Definitions

3.1 Static Single-Sender Multi-Receiver

We introduce six notions of Group Unidirectional Ratcheted Key Exchange (GURKE). These notions differ in the number of senders (single vs. multi) and receivers (single vs. multi) and the ability to change the sets of sender and receivers, respectively (static vs. dynamic). For simplicity, we begin with the simplest case, Static Single-Sender Multi-Receiver (S-SSMR), in which one sender continuously establishes symmetric keys with a static group of receivers. After giving an intuition for the correctness and security guarantees we require for that simple case, we switch to our most powerful notion, Dynamic Multi-Sender Multi-Receiver (D-MSMR): a dynamically modifiable set of senders continuously establishes symmetric keys with a dynamically modifiable set of receivers. For this case, we formally define *syntax*, *correctness*, and *security*. These three definitions can be easily simplified and relaxed to obtain corresponding formal definitions for all other cases of GURKE.

Syntax. GURKE in the Static Single-Sender Multi-Receiver (S-SSMR) case is a tuple of three algorithms $\mathrm{R} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$ with sender state space $\mathcal{ST}_{\mathtt{S}}$, receiver state space $\mathcal{ST}_{\mathtt{R}}$, associated data space $\mathcal{AD}$, ciphertext space $\mathcal{C}_{\mathtt{R}}$, key space $\mathcal{K}$, and key identifier space $\mathcal{KID}$, such that:

- $\mathrm{init} : \mathbb{N} \to_{\$} \mathcal{ST}_{\mathtt{S}} \times (\mathcal{ST}_{\mathtt{R}})^{+}$ Initializes a state variables for the sender and all receivers.
- $\mathrm{snd} : \mathcal{ST}_{\mathtt{S}} \times \mathcal{AD} \to_{\$} \mathcal{ST}_{\mathtt{S}} \times \mathcal{C}_{\mathtt{R}} \times \mathcal{K} \times \mathcal{KID}$ Establishes a fresh key with all receivers.
- $\mathrm{rcv} : \mathcal{ST}_{\mathtt{R}} \times \mathcal{AD} \times \mathcal{C}_{\mathtt{R}} \to \mathcal{ST}_{\mathtt{R}} \times \mathcal{K} \times \mathcal{KID}$ Computes the key established by the sender.

We carefully define the syntax by extending established notions for two-party unidirectional ratcheted key exchange in the literature [BSJ+17, PR18, BRV20, DHRR22]. Thereby, we only naturally increase the number of receiver states that are initialized; we refrain from adding public-key generation and modeling initial key exchange explicitly, as this would increase complexity with only little anticipated gain. Furthermore, we add a key identifier output.

Key Identifier. Although not necessarily relevant for the S-SSMR setting, algorithms snd and rcv output a key identifier $kid \in \mathcal{KID}$. This output is inspired by a survey of definitions in the group key exchange literature by Poettering et al. [PRSS21]; the identifier can be used publicly and it serves as a unique reference to the key it was output with. Furthermore, it can contain extra metadata about the key, such as an identifier for the user who initially established that key or the set of members who the key is shared with, which is valuable information for higher level applications that use the key. As we will see, this feature is particularly useful in the dynamic group setting, in which concurrently initiated membership

changes can lead to diverging views of the set of group members. For that reason, we require that functions

$$\text{ini} : \mathcal{KID} \to \mathcal{UID}, \quad \text{mem}_\mathtt{S} : \mathcal{KID} \to \mathcal{P}(\mathcal{UID}), \quad \text{mem}_\mathtt{R} : \mathcal{KID} \to \mathcal{P}(\mathcal{UID})$$

on input of a key identifier output the initiator of the key establishment, the set of senders, and the set of receivers, respectively.[5] For simplicity, we use integers to identify users such that $\mathcal{UID} = \mathbb{N}$. In the S-SSMR setting, this means that $\text{ini}(kid) = \text{mem}_\mathtt{S}(kid) = 1$ and $\text{mem}_\mathtt{R}(kid) = [n]$ for every output key identifier kid, where n is the initial number of receivers.

Correctness. We call a GURKE construction for the S-SSMR case correct iff every key k has a unique key-identifier kid and each key–key-identifier tuple (k, kid) computed by a receiver equals the corresponding tuple (k', kid') computed by the sender, as long as all ciphertexts received by that receiver, so far, were honestly transmitted from the sender. The formal definition for the D-MSMR case in Section 3.2 below captures this requirement for all considered cases, including S-SSMR.

Security. We define security using games $\text{IND}^b_\mathtt{R}, b \in \{0, 1\}$ played by an adversary $\mathcal{A}$ who begins with specifying the number of receivers $n_\mathtt{R}$ and, afterwards, has access to the following oracles:

- $\text{Snd}(ad, ch) \to (c_\mathtt{R}, k, kid)$ Lets the sender execute algorithm snd and either outputs the resulting real key or a random key k, depending on whether the adversary queried a challenge (i.e., $ch = \mathtt{T}$) and depending on challenge bit b.
- $\text{Rcv}(i, ad, c) \to (k, kid)/kid$ Lets receiver i process ciphertext c via algorithm rcv and either outputs the real, a random, or no key, depending on challenge bit b and the trivial attack strategies that we describe below. Upon receipt, receiver i is marked *trivially impersonated* iff (1) c was successfully processed but not sent in that order by the sender, and (2) the sender state was exposed (via oracle $\text{Expose}_\mathtt{S}$) right after the last ciphertext received by i was sent; otherwise, if only condition (1) holds, receiver i is marked *out of sync*; if condition (1) does not hold, receiver i remains *in sync*.
- $\text{Expose}_\mathtt{S} \to st_\mathtt{S}$ Outputs the sender's local state.
- $\text{Expose}_\mathtt{R}(i) \to st^i_\mathtt{R}$ Outputs the local state of receiver i.

The adversary terminates by outputting guess b'. To forbid trivial attack strategies, we silence the oracles based on the following conditions:

1. After one of the receivers was exposed while still being *in sync* (i.e., everything they received was honestly transmitted from the sender), oracle Snd will abort if it was queried as a challenge (i.e., with $ch = \mathtt{T}$); otherwise, the challenge could be solved trivially.
2. Oracle Rcv will not output any key for receivers who are still *in sync*; the corresponding (challenge) key was already output on the sender side.
3. Oracle Rcv will only output a challenge key (i.e., real or random key, depending on bit b) if the receiver is *out of sync* (i.e., neither *in sync* nor *trivially impersonated*); *in sync* keys were already output on the sender side, and the adversary can predict keys computed by receivers after a successful *trivial impersonation*.
4. If receiver i is *in sync*, oracle $\text{Expose}_\mathtt{R}$ only outputs i's state if i already received all challenges sent so far; otherwise, all unreceived challenges are trivially solved.

We provide a pseudo-code definition of games $\text{IND}^b_\mathtt{R}$ with all oracles and the above restrictions of trivial attack strategies for the D-MSMR case in Section 3.2. The S-SSMR case can be easily extracted from this by ignoring dynamic membership changes and by having only a single sender.

The advantage of adversary $\mathcal{A}$ in winning games $\text{IND}^b_\mathtt{R}$ is defined as

$$\text{Adv}^{\text{ind}}_\mathtt{R}(\mathcal{A}) = |\Pr[\text{IND}^0_\mathtt{R}(\mathcal{A}) = 1] - \Pr[\text{IND}^1_\mathtt{R}(\mathcal{A}) = 1]|.$$

3.2 Dynamic Multi-Sender Multi-Receiver

Our most powerful GURKE notion allows a set of senders to send to a set of receivers concurrently. Furthermore, the senders can change the sets of senders and receivers dynamically via additional algorithms add, rmv, and proc. We color our syntax definition to highlight relaxations that we explain hereafter.

[5] Note that this implies that *kid* grows linearly in the number of senders and receivers.

Syntax. GURKE in the Dynamic Multi-Sender Multi-Receiver (D-MSMR) case is a tuple of algorithms $\mathrm{R} = (\mathsf{init}, \mathsf{snd}, \mathsf{add}, \mathsf{rmv}, \mathsf{proc}, \mathsf{rcv})$ with sender state space $\mathcal{ST}_{\mathtt{S}}$, receiver state space $\mathcal{ST}_{\mathtt{R}}$, associated data space $\mathcal{AD}$, sender-to-sender ciphertext space $\mathcal{C}_{\mathtt{S}}$, sender-to-receiver ciphertext space $\mathcal{C}_{\mathtt{R}}$, key space $\mathcal{K}$, key identifier space $\mathcal{KID}$, and user identity space $\mathcal{UID}$, such that:

- $\mathsf{init} : \mathbb{N} \times \mathbb{N} \to_{\$} (\mathcal{ST}_{\mathtt{S}})^{+} \times (\mathcal{ST}_{\mathtt{R}})^{+}$ Initializes state variables for all initial senders and receivers.
- $\mathsf{snd} : \mathcal{ST}_{\mathtt{S}} \times \mathcal{AD} \to_{\$} \mathcal{ST}_{\mathtt{S}} \times \mathcal{C}_{\mathtt{R}} \times \mathcal{K} \times \mathcal{KID}$ Just as S-SSMR.
- $\mathsf{add} : \mathcal{ST}_{\mathtt{S}} \times \mathcal{AD} \times \{\mathtt{S}, \mathtt{R}\} \times \mathcal{UID} \to_{\$} \mathcal{ST}_{\mathtt{S}} \times (\mathcal{ST}_{\mathtt{S}} \cup \mathcal{ST}_{\mathtt{R}}) \times \mathcal{C}_{\mathtt{S}} \times \mathcal{C}_{\mathtt{R}} \times \mathcal{K} \times \mathcal{KID}$ Adds a new sender, resp. receiver, by creating their new state variable $st^{*} \in (\mathcal{ST}_{\mathtt{S}} \cup \mathcal{ST}_{\mathtt{R}})$, informing other senders about this operation via $c_{\mathtt{S}} \in \mathcal{C}_{\mathtt{S}}$, and establishing a key $k \in \mathcal{K}$ with all subsequent receivers via $c_{\mathtt{R}} \in \mathcal{C}_{\mathtt{R}}$.
- $\mathsf{rmv} : \mathcal{ST}_{\mathtt{S}} \times \mathcal{AD} \times \{\mathtt{S}, \mathtt{R}\} \times \mathcal{UID} \to_{\$} \mathcal{ST}_{\mathtt{S}} \times \mathcal{C}_{\mathtt{S}} \times \mathcal{C}_{\mathtt{R}} \times \mathcal{K} \times \mathcal{KID}$ Removes a sender, resp. receiver, by informing other senders about this operation via $c_{\mathtt{S}} \in \mathcal{C}_{\mathtt{S}}$ and establishing a key $k \in \mathcal{K}$ with all subsequent receivers via $c_{\mathtt{R}} \in \mathcal{C}_{\mathtt{R}}$.
- $\mathsf{proc} : \mathcal{ST}_{\mathtt{S}} \times \mathcal{AD} \times \mathcal{C}_{\mathtt{S}} \to \mathcal{ST}_{\mathtt{S}}$ Processes sender-to-sender information about a membership operation.
- $\mathsf{rcv} : \mathcal{ST}_{\mathtt{R}} \times \mathcal{AD} \times \mathcal{C}_{\mathtt{R}} \to \mathcal{ST}_{\mathtt{R}} \times \mathcal{K} \times \mathcal{KID}$ Just as S-SSMR.

Relaxations. Input $\{\mathtt{S}, \mathtt{R}\}$ specifies the role of a user that is added to, respectively removed from, the set of senders, respectively receivers. Notably, this information is only relevant if both of these member sets can be modified dynamically (D-MSMR). In case the set of senders (and receivers) can be modified dynamically by multiple senders (D-*MS*SR or D-*MS*MR), sharing the view of the member sets between senders is important. Therefore, dynamic membership changes can be announced via sender-to-sender ciphertexts $c \in \mathcal{C}_{\mathtt{S}}$ that are processed by algorithm proc. Furthermore, only if the set of senders can be modified (D-*MS*SR or D-*MS*MR), algorithm add outputs a second sender state $st \in \mathcal{ST}_{\mathtt{S}}$ that the newly added sender uses thereafter; correspondingly, only if the set of receivers can be modified (D-SS*MR* or D-MS*MR*), algorithm add outputs a receiver state $st \in \mathcal{ST}_{\mathtt{R}}$ for that newly added receiver. Algorithms add and rmv are only relevant in the dynamic setting (*D*-SSMR, *D*-MSSR, or *D*-MSMR). Finally, algorithm init only takes two input integers $\mathbb{N} \times \mathbb{N}$ for multiple senders and multiple receivers (S/D-*MSMR*), which equally affects the number output sender states and receiver states $(\mathcal{ST}_{\mathtt{S}})^{+} \times (\mathcal{ST}_{\mathtt{R}})^{+}$. We give the concrete relaxed syntax definitions for all static and dynamic cases with single and multiple senders and receivers, respectively, in Appendix C.

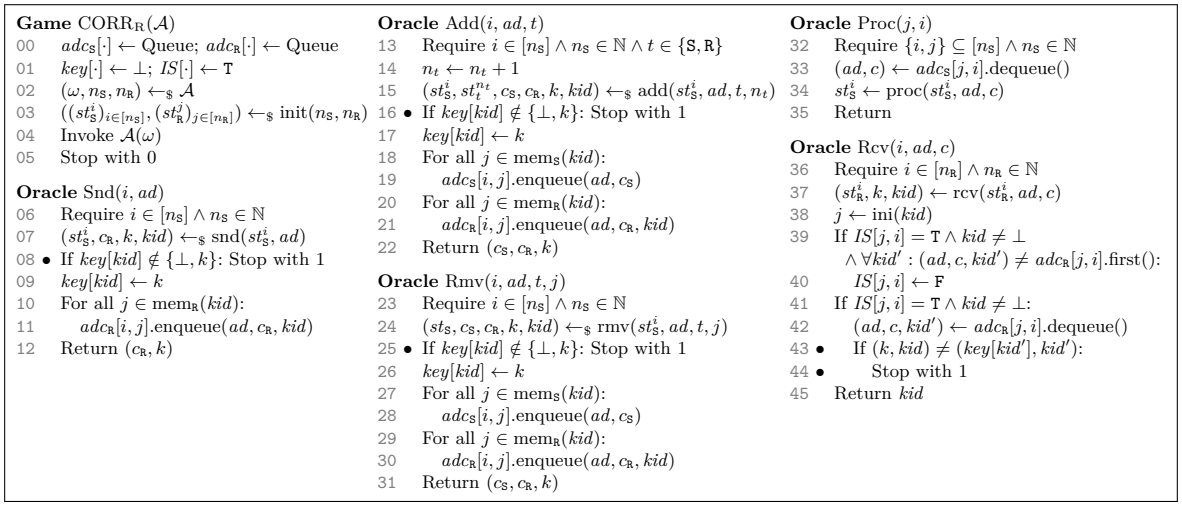


Fig. 2. Game CORR for D-MSMR scheme R. Symbol '•' highlights the crucial correctness requirement.

Correctness. The intuition for our correctness requirements remains unchanged compared to the S-SSMR case. This is formalized in Figure 2, where adversary $\mathcal{A}$ can query oracles Snd, Add, Rmv, Proc, and Rcv to let the senders and receivers execute the corresponding algorithms.

Once a key is computed in oracles Snd, Add, or Rmv, we require that its key-identifier does not collide with that of a different key (lines 08,16,25). Then, for each received ciphertext, oracle Rcv traces whether receiver i remains *in sync* with the corresponding sender (lines 39-40). As soon as i computes a non-trivial key identifier for a ciphertext that was not sent in that order by the indicated sender j, i and j are marked *out of sync*. For all sender-receiver pairs that stay *in sync*, the key–key-identifier tuples computed by the receiver must equal those computed by the sender (lines 43-44).[6] Based on this, a GURKE scheme R is defined *correct* iff $\Pr[\mathrm{CORR}_{\mathrm{R}}(\mathcal{A}) = 0] = 1$.

Membership Consistency. Our correctness definition does not require that senders compute *the right*™ set of members because, naturally, there is no unique definition for this. Consider concurrently initiated membership changes that contradict each other (e.g., sender i adds receiver r and, concurrently, sender j removes i; whether r is a group member afterwards seems to be a matter of taste). We do not want to enforce a specific conflict resolution policy and, instead, leave it to the construction to do so. For all cases in which such conflicts are naturally avoided (i.e., all static cases as well as D-SSMR; also if concurrency between senders is prevented or resolved externally), it is obvious that our constructions indeed maintain *the right* sets of members.

Security. As in the S-SSMR case, we define security using games $\mathrm{IND}^b_{\mathrm{R}}, b \in \{0,1\}$ played by an adversary $\mathcal{A}$, which is formally defined in Figure 3. During this game, $\mathcal{A}$ can query the following oracles:

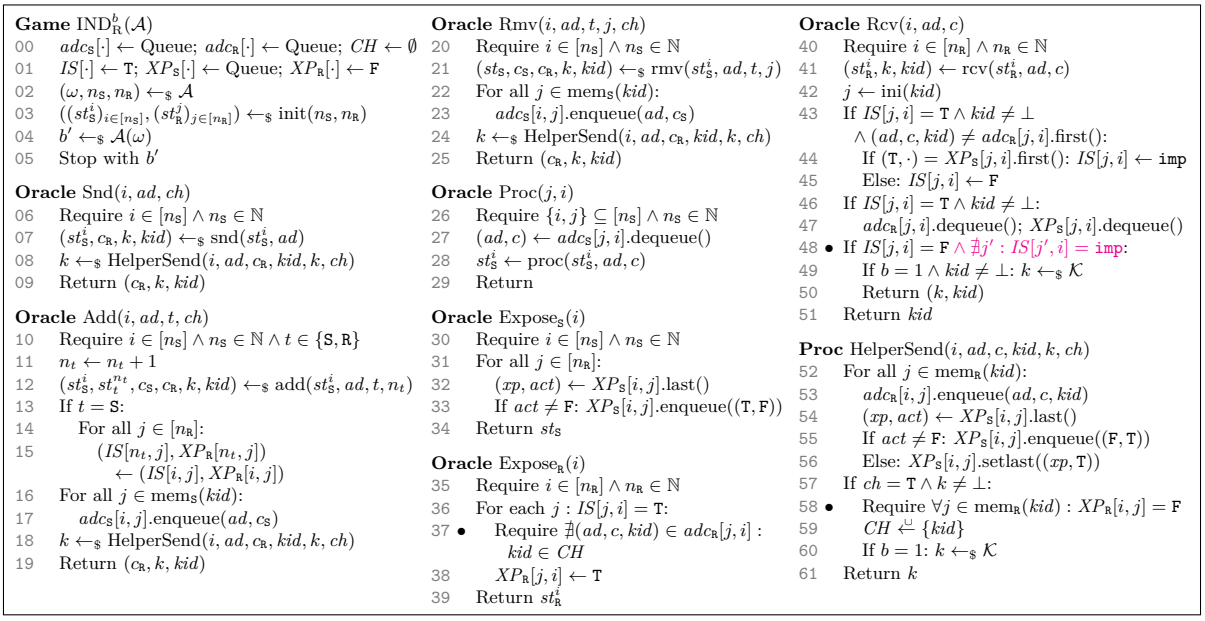


Fig. 3. Games IND^b, $b \in \{0,1\}$, for D-MSMR scheme R. Symbol '•' highlights the restrictions that prevent trivial attack strategies.

- $\mathrm{Snd}(i, ad, ch)$, $\mathrm{Add}(i, ad, t, ch)$, $\mathrm{Rmv}(i, ad, t, j, ch)$ all output $(c_{\mathtt{R}}, k, kid)$: Lets sender i send, add either a new sender or receiver, or remove either sender or receiver j, where $t \in \{\mathtt{S}, \mathtt{R}\}$ specifies which type of user is added, resp. removed. The output key is either the real or a random key, depending on bits $ch \in \{\mathtt{T}, \mathtt{F}\}$ and $b \in \{0,1\}$.
- $\mathrm{Proc}(j, i)$: Lets sender i receive and process the next ciphertext from sender j. To focus on the security of keys established by senders with receivers, and for simplicity and clarity, we assume that sender-to-sender communication is transmitted via an authenticated confidential in-order channel.[7]

[6] This level of correctness follows the flavor of "safety" instead of "liveness", known from the distributed systems literature.

[7] Formally investigating how a D-SSMR GURKE scheme can be used generically to build such a channel remains a question for future work.

– $\text{Rcv}(i, ad, c) \to (k, kid)/kid$: Lets receiver i process ciphertext c and, depending on challenge bit b and the trivial attack strategies that we describe below, outputs the real, a random, or no key. Based on the same policy as in the S-SSMR case, sender-receiver pairs are marked *in sync*, *out of sync*, or *trivially impersonated*. That is, if sender j was not exposed for a manipulated ciphertext accepted by receiver i, (j, i) are marked *out of sync*. Otherwise, if sender j was exposed immediately after sending the last ciphertext accepted by i, and the subsequent ciphertext is successfully manipulated, (j, i) are marked *impersonated*. For this, we trace sender exposures via queue $XP_{\mathtt{S}}$ (lines 32-33,44,47,54-56) that observes the sender-receiver pair in parallel to queue $adc_{\mathtt{R}}$: After exposing sender j, the last entry in queue $XP_{\mathtt{S}}[j, i]$ stores that j is currently exposed via $xp = \mathtt{T}$ (lines 32-33). Once sender j sends to receiver i, this queue entry is activated by flipping act from $\mathtt{F}$ to $\mathtt{T}$ (lines 54-56).
– $\text{Expose}_{\mathtt{S}}(i) \to st^i_{\mathtt{S}}$, $\text{Expose}_{\mathtt{R}}(i) \to st^i_{\mathtt{R}}$: Output the local state of sender, resp., receiver i.

The adversary terminates by outputting guess b'. The forbidden trivial attack strategies described for the S-SSMR case remain almost unchanged:

1. Oracles Snd, Add, and Rmv only output challenge keys if none of the designated receivers was exposed while being *in sync* so far (lines 57-60).
2. Oracle Rcv will not output a key for receivers who are *in sync* (lines 48-51).
3. For challenge keys output on the receiver side, we add a special condition for the dynamic multi-sender cases (D-MSSR and D-MSMR): challenge keys are only output if the receiver is *out of sync* with the respective sender *and* no sender was impersonated towards the receiver (lines 48-50); otherwise, the adversary could have used that impersonation to add more malicious senders to the group. This special condition limits the adversary slightly more than theoretically necessary; the minimally strengthened security requirement would, however, induce far more complexity in the definition, which we elaborate on below.
4. Oracle $\text{Expose}_{\mathtt{R}}$ only outputs states of receivers who received all challenges that were sent by senders who they are *in sync* with (lines 36-37).

The advantage of adversary $\mathcal{A}$ in winning games IND^b_{R} is defined as

$$\text{Adv}^{\text{ind}}_{\text{R}}(\mathcal{A}) = |\Pr[\text{IND}^0_{\text{R}}(\mathcal{A}) = 1] - \Pr[\text{IND}^1_{\text{R}}(\mathcal{A}) = 1]|.$$

Non-Trivial Attacks. Our definition imposes strong security requirements. Yet, for clarity and comprehensibility, we refrain from defining *strongest possible* security with respect to the correctness requirements and the specified oracles. With the following restrictions, we limit the adversary minimally stronger than theoretically necessary.

Inherited impersonation: For the dynamic multi-sender cases, once one of the senders was *impersonated* towards a receiver, our definition never requires this receiver to compute secure keys *out of sync* anymore. Intuitively, an adversary can extend an impersonation by adding further malicious senders to the group, whose established keys the adversary may know. Yet, a more fine-grained definition could precisely trace the history and relation of membership changes to identify and distinguish between honest senders and senders who were added due to such malicious impersonations.

Impersonation towards new receivers: After a new receiver j is added, their *chain of trust* towards existing senders is rooted in sender i^* who added them. To impersonate one of the senders $i' \neq i^*$ towards j with the first i'-to-j ciphertext, an adversary would actually need to expose i' after i' received the respective sender-to-sender ciphertext from i^* that added j. In our definition, we slightly extend this window of harmful exposures of i': instead of only considering exposures of i' harmful that happen after i' received i^*'s ciphertext, we consider exposures of i' harmful that happen immediately after i^* added j.

Uniform randomness: All random coins used by senders are considered to be sampled secretly from a uniform distribution. Changing this (e.g., by letting the adversary manipulate random coins) would imply that HIBE-like building blocks are necessary to build GURKE [BRV20, RSS23]. Yet, modeling that the senders' random coins are always *good* assumes that senders always behave honestly as long as they are group members. This assumption disregards the so-called *double-join* issue[8]: a dishonest sender could maliciously choose or simply store their random coins during their group membership in order to adversarially re-join the group after having been removed. Although, this threat is only relevant in the

[8] See https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/.

dynamic multi-sender cases (as, otherwise, bad randomness only affects their own established secrets), we consider extending our notions to also capture this threat an interesting, orthogonal open problem for future work.

Intuition for Security Guarantees. Overall, by allowing sender state exposures before and after challenges as well as receiver exposures after challenges, we require strong forms of Forward Secrecy and Post-Compromise Security. Furthermore, due to tagging receivers *out of sync* or *trivially impersonated* as soon as they accept the first manipulated ciphertext from the adversary, we require a strong form of Implicit Authentication. Concretely, keys of senders and receivers as well as their local state variables are required to become independent of each other after such active attacks against the traffic between them occur. The practical motivation for this requirement is that successful active attacks should not be concealable; by requiring that sender and receiver states diverge upon such attacks, we capture a strong implicit variant of *Recover Security* [DV19, CDV21, BCC+23a]. Although not required by our definition, our constructions offer explicit authentication for out-of-sync receivers.

4 Updatable Broadcast KEM

We introduce the main building block of our GURKE constructions that we call Updatable Broadcast KEM (UB-KEM). This building block encapsulates symmetric keys k with a single sender's public key ek to the secret keys dk of multiple receivers efficiently. Via dynamic membership changes, UB-KEM allows for adding and removing receiver secret keys. Furthermore, the sender's UB-KEM public key can be forked to add new senders on the GURKE level. Each encapsulation triggers an update of the sender's public key and the receivers' secret keys to achieve *Forward Security* (FS)—secret key corruption does not affect earlier decapsulated symmetric keys—and *Implicit Authentication* (IA)—active attacks against ciphertexts sent from the sender alters the receivers' secret keys incompatibly.

4.1 Definition

We formally define UB-KEM for dynamic groups. Applying our dynamic group definitions to the static group construction from Section 4.2 is straightforward.

Syntax. An Updatable Broadcast KEM (UB-KEM) is a tuple of algorithms $\mathsf{BK} = (\mathsf{BK.gen}, \mathsf{BK.enc}, \mathsf{BK.fin}, \mathsf{BK.dec}, \mathsf{BK.add}, \mathsf{BK.rmv}, \mathsf{BK.fork}, \mathsf{BK.proc})$ with encapsulation key space $\mathcal{EK}$, decapsulation key space $\mathcal{DK}$, update token space $\mathcal{UP}$, associated data space $\mathcal{AD}$, ciphertext space $\mathcal{C}$, key space $\mathcal{K}$, and key identifier space $\mathcal{KID}$, such that:

- $\mathsf{BK.gen} : \mathbb{N} \to_\$ \mathcal{EK} \times (\mathcal{DK})^+$ Generates sender public encapsulation key and an initial number of secret receiver decapsulation keys.
- $\mathsf{BK.enc} : \mathcal{EK} \to_\$ \mathcal{UP} \times \mathcal{C}$ Prepares a ciphertext for encapsulation.
- $\mathsf{BK.fin} : \mathcal{UP} \times \mathcal{AD} \to \mathcal{EK} \times \mathcal{K}$ Finalizes the encapsulation and updates the encapsulation key with respect to associated data.
- $\mathsf{BK.dec} : \mathcal{DK} \times \mathcal{AD} \times \mathcal{C} \to \mathcal{DK} \times \mathcal{K}$ Decapsulates symmetric key and updates decapsulation key.
- $\mathsf{BK.add} : \mathcal{EK} \to_\$ \mathcal{EK} \times \mathcal{DK} \times \mathcal{C}$ Adds new receiver.
- $\mathsf{BK.rmv} : \mathcal{EK} \times \mathbb{N} \to_\$ \mathcal{EK} \times \mathcal{C}$ Removes receiver.
- $\mathsf{BK.fork} : \mathcal{EK} \to_\$ \mathcal{EK} \times \mathcal{EK} \times \mathcal{C}$ Forks an encapsulation key instance.
- $\mathsf{BK.proc} : \mathcal{DK} \times \mathcal{C} \to \mathcal{DK} \times (\mathcal{DK} \cup \{\bot\})$ Processes ciphertext of a membership change or a fork.

Static group UB-KEM comprises algorithms $(\mathsf{BK.gen}, \mathsf{BK.enc}, \mathsf{BK.fin}, \mathsf{BK.dec})$.

Intuition. The primary idea of UB-KEM is to encapsulate efficiently to a set of receivers and, thereby, split the creation of the ciphertext (via BK.enc) from the computation of the encapsulated symmetric key as well as the update of the encapsulation key (via BK.fin). By splitting these two operations, a higher level application can perform additional intermediate operations that *depend on the ciphertext* and that can *influence* both the *update* of the key pair *and the en- resp. decapsulation* of the symmetric key. As mentioned above, updates of encapsulation keys and decapsulation keys have two purposes: they should guarantee *Forward Security* (FS) and *diverge keys* once encapsulation and decapsulation processed

different ciphertext–associated-data pairs. Thus, if a late decapsulation key is exposed to an adversary, this should neither affect any previously decapsulated symmetric key, nor a symmetric key that was encapsulated after the interaction between the sender on the encapsulation side and the receiver on the decapsulation side was disrupted (e.g., due to an active attack).

In the dynamic group setting, besides having the ability to change the set of decapsulation key members, we add an (efficient) fork mechanism: the forked encapsulation key can be computed locally on the sender side, and the ciphertext that informs the receiver side about forking the corresponding decapsulation keys can be small—as we will see in Section 4.3.

Correctness. Informally, we call a UB-KEM construction correct iff decapsulated symmetric keys equal the symmetric keys computed at encapsulation, as long as all ciphertexts processed on the decapsulation side were output in the same order by the encapsulation side. The formal correctness definition for UB-KEM is specified via multi-instance game $\mathrm{CORR}_{\mathrm{BK}}$ in Figure 4. The multi-instance setting is particularly relevant to capture forks: once an encapsulation key is forked, the resulting new encapsulation key together with the corresponding decapsulation keys is required to behave like a new instance for which the same correctness guarantees must be fulfilled. Based on this, a UB-KEM construction is called correct iff $\Pr[\mathrm{CORR}_{\mathrm{BK}}(\mathcal{A}) = 0] = 1$.

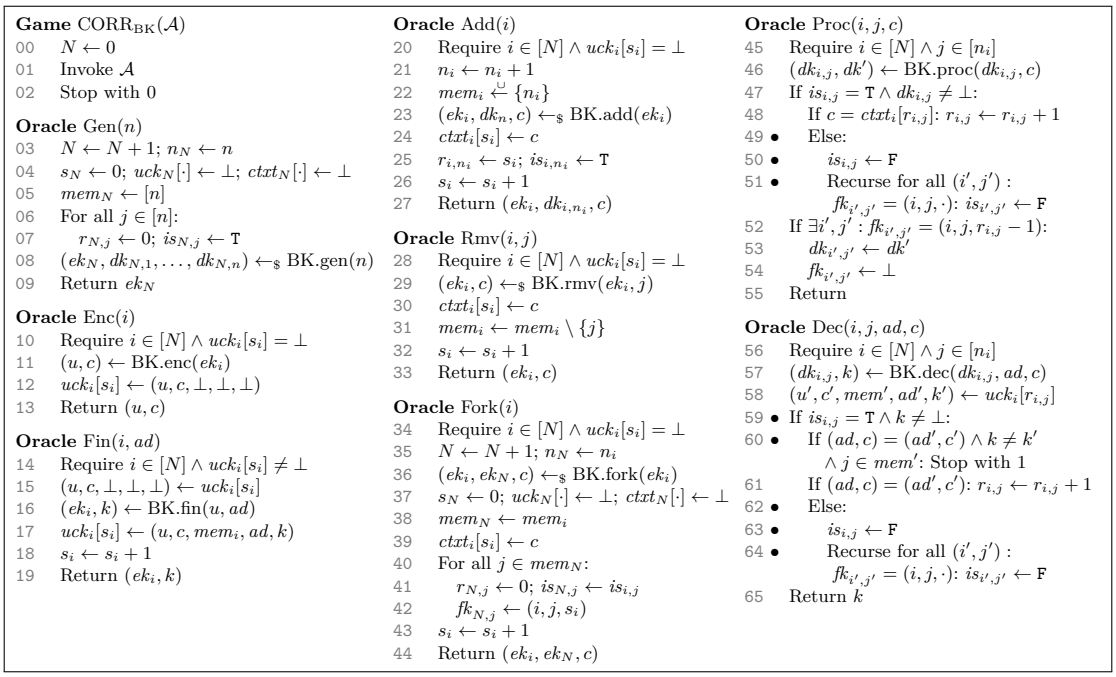


Fig. 4. Multi-instance game CORR for UB-KEM scheme BK. "Recurse" in lines 64 and 51 means that the forking dependencies via pointer fk are fully (i.e., recursively) resolved.

Security. We define security using multi-instance games $\mathrm{IND}^b_{\mathrm{BK}}, b \in \{0, 1\}$ from Figure 5 played by an adversary $\mathcal{A}$ who has access to the following oracles:

- $\mathrm{Gen}(n) \to ek$ Generates a new instance with n decapsulation keys.
- $\mathrm{Enc}(i) \to c$ Lets instance i prepare a ciphertext via algorithm BK.enc.
- $\mathrm{Fin}(i, ad, ch) \to (ek_i, k)$ Finalizes the encapsulation via algorithm BK.fin and either outputs the resulting real key or a random key k, depending on whether the adversary wants to be challenged (i.e., $ch = \mathtt{T}$) and depending on challenge bit b.
- $\mathrm{Dec}(i, j, ad, c) \to k/\epsilon$ Decapsulates ciphertext c with the jth decapsulation key of instance i via algorithm BK.dec and either outputs the real or no key, depending on the trivial attack strategies described below. Upon receipt, decapsulation key $dk_{i,j}$ and all its future fork-descendants are marked *out of sync* iff c was successfully processed but not encapsulated in that order.

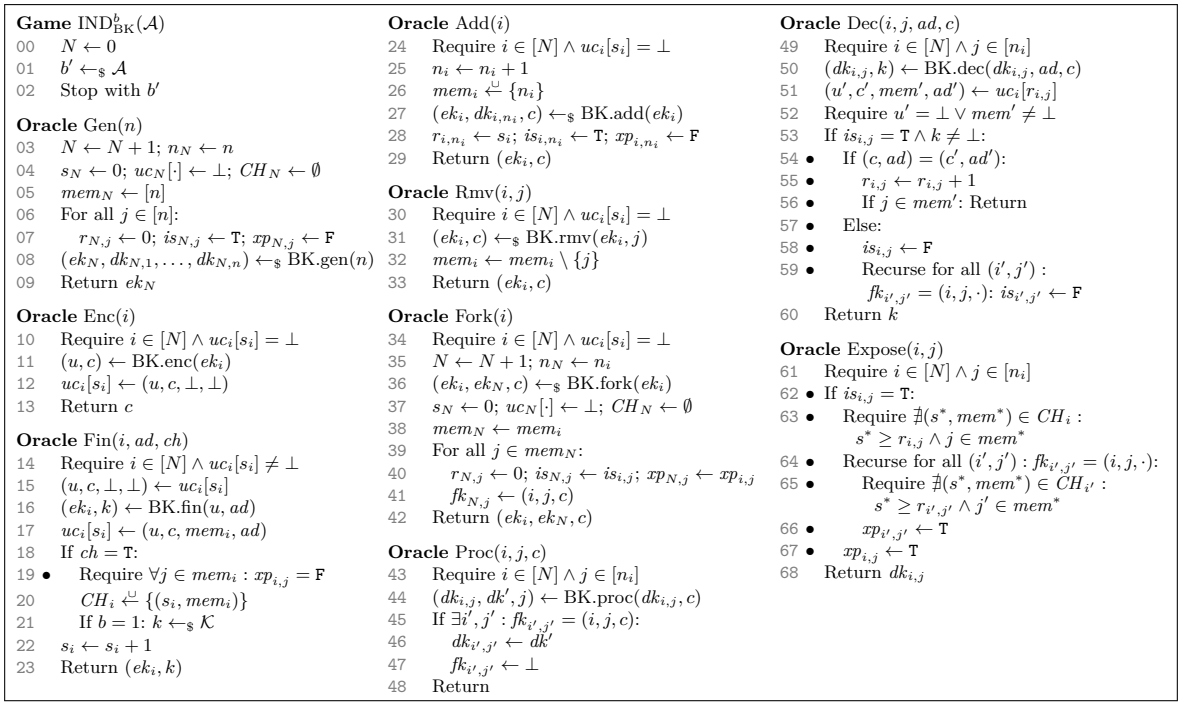


Fig. 5. Multi-instance games IND^b, $b \in \{0, 1\}$, for UB-KEM scheme BK. "Recurse" in lines 59 and 64 means that the forking dependencies via pointer fk are fully (i.e., recursively) resolved.

- $\mathrm{Add}(i)$ and $\mathrm{Rmv}(i, j)$ output (ek_i, c), $\mathrm{Fork}(i) \to (ek_i, ek, c)$ Perform the respective membership operations, resp., a fork with encapsulation key i.
- $\mathrm{Proc}(i, j, c)$ Processes membership operations and forks with key $dk_{i,j}$.
- $\mathrm{Expose}(i, j) \to dk_{i,j}$ Outputs decapsulation key j of instance i.

The adversary terminates by outputting guess b'. To forbid trivial attack strategies, we silence the oracles under the following conditions:

1. If one of an instance's member decapsulation keys was exposed while being *in sync* (i.e., everything it decapsulated was honestly transmitted from encapsulation), oracle Fin aborts if it was queried as a challenge (i.e., with $ch = \mathtt{T}$; line 19); otherwise, the challenge is trivially solved.
2. Oracle Dec will not output a decapsulated key if the decapsulation key stays *in sync* (lines 53-56); the corresponding (challenge) key was already output on the encapsulation side.
3. If decapsulation key (i, j) is *in sync*, oracle Expose will abort if not all corresponding encapsulation challenges were decapsulated yet (line 63)—this also applies to all future fork-descendants (lines 64-65); otherwise, these challenges are trivially solved.

The advantage of adversary $\mathcal{A}$ in winning games $\mathrm{IND}^b_{\mathrm{R}}$ is defined as

$$\mathrm{Adv}^{\mathrm{ind}}_{\mathrm{BK}}(\mathcal{A}) = |\Pr[\mathrm{IND}^0_{\mathrm{BK}}(\mathcal{A}) = 1] - \Pr[\mathrm{IND}^1_{\mathrm{BK}}(\mathcal{A}) = 1]|.$$

4.2 Static Group Construction

Our static group UB-KEM construction in Figure 6 is a direct generalization of the main design principle of two-party unidirectional ratcheted key exchange by Poettering and Rösler [PR18], using a standard KEM as well as a random oracle. The primary generalization step is that the single decapsulation key from the two-party setting is replicated such that all UB-KEM decapsulation keys in the group are initialized with the same KEM decapsulation key. For encapsulation, first a KEM encapsulation is executed; then, for finalization, the random oracle on input of the encapsulated key, the ciphertext, and the associated data outputs the final symmetric output key as well as a seed for a new KEM key pair. This seed is

Proc BK.gen(n)	**Proc** BK.enc(ek)	**Proc** BK.fin(u, ad)	**Proc** BK.dec(dk, ad, c)
00 $(ek, dk) \leftarrow_\$ \text{K.gen}$	03 $(k, c) \leftarrow_\$ \text{K.enc}(ek)$	06 $(k', c') \leftarrow u$	10 $k' \leftarrow \text{K.dec}(dk, c)$
01 For all $i \in [n]$: $dk_i \leftarrow dk$	04 $u \leftarrow (k, c)$	07 $(s, k) \leftarrow \text{H}(c', k', ad)$	11 $(s, k) \leftarrow \text{H}(c, k', ad)$
02 Return $(ek, dk_1, \ldots, dk_n)$	05 Return (u, c)	08 $(ek, _) \leftarrow \text{K.gen}(s)$	12 $(_, dk) \leftarrow \text{K.gen}(s)$
		09 Return (ek, k)	13 Return (dk, k)

Fig. 6. Construction of Static UB-KEM BK from KEM K and random oracle H.

used to update the encapsulation key. For decapsulation, the corresponding KEM decapsulation feeds the random oracle to obtain an updated decapsulation key as well as the symmetric output key.

By updating encapsulation and decapsulation keys via the random oracle, Forward Security is achieved, and diverging transcripts immediately lead to diverging key updates.

Theorem 1. *UB-KEM* BK *from Fig. 6 offers key indistinguishability. More precisely, if* H *is modeled as a random oracle, then for every adversary $\mathcal{A}$ attacking* BK *in games* IND^b_{BK} *from Fig. 5 that makes at most q_{H} queries to* H*, there exists adversary $\mathcal{B}$ against multi-instance exposable OW-CCA security of KEM* K *such that* $\text{Adv}^{\text{ind}}_{\text{BK}}(\mathcal{A}) \leq \text{Adv}^{\text{ow}}_{\text{K}}(\mathcal{B})$*, where the running time of $\mathcal{B}$ is about $q_{\text{Enc}} + q_{\text{Dec}}$ times as much as that of $\mathcal{A}$ and $q_{\text{Enc}}/q_{\text{Dec}}$ is the number of* Enc/Dec *queries $\mathcal{A}$ makes.*

We defer the proof to Appendix D.1.

4.3 Dynamic Group Construction

The primary challenge for the dynamic group setting is to make the decapsulation keys of all group members independent (enough) of each other. The reason for this is that, once a decapsulation key is removed from the group, this key should not leak any information about symmetric keys encapsulated to the remaining group thereafter. To do so, we make use of a tree structure, which is a common design principle. Each decapsulation key stores secrets along the path from its individual leaf to the joint root. Thus, when removing a decapsulation key, the group only needs to renew the secrets on this path. To do so, we use standard Non-Interactive Key Exchange, defined in Section A.4. While this idea ostensibly resembles the design of TreeKEM [BBR18], we elaborate on crucial differences below.

Tree Data Structure. Before describing our dynamic UB-KEM construction, we introduce a data structure that simplifies accessing and manipulating data in a tree. This data structure provides the following algorithms:

- $\text{T.init}(n) \rightarrow_\$ \tau$ Initializes the tree structure for n leaves.
- $\text{T.nodes}(\tau) \rightarrow (t_j)_{j \in [N]}$ Outputs the list of indexes of all nodes in the tree.
- $\text{T.setnodes}(\tau, (ek_j)_{j \in [N]}) \rightarrow ek$ Packs the list of all nodes' public encapsulation keys in the public tree description ek.
- $\text{T.getnodes}(ek) \rightarrow (\tau, (ek_j)_{j \in [N]})$ Outputs a variable τ that represents the tree structure and the list of all nodes' public encapsulation keys.
- $\text{T.path}(\tau, i) \rightarrow (p_l)_{l \in [L_i]}$ Outputs the list of indexes of all nodes on the path from leaf i to the root.
- $\text{T.setpath}(\tau, i, (dk_{p_l})_{l \in [L_i]}) \rightarrow dk^i$ Packs the list of all nodes' secret decapsulation keys on the input leaf's path in the secret path description dk^i.
- $\text{T.getpath}(dk) \rightarrow (\tau, i, (dk_l)_{l \in [L]})$ Outputs a variable τ that represents the tree structure, the input path's leaf index, and the list of all nodes' secret decapsulation keys on the input path.
- $\text{T.add}(ek) \rightarrow (\tau, (ek_j)_{j \in [N]}, (p_l)_{l \in [L]}, (cp_l)_{l \in [2,L]}, n)$, resp., $\text{T.add}(dk) \rightarrow (\tau, i, (dk_l)_{l \in [L]}, l^*)$ Adds a user to the tree and either outputs—on input of the public tree description ek—a variable τ that represents the tree structure, the list of all nodes' public encapsulation keys, the path from the added user's leaf to the root, the corresponding co-path[9], and the current number of leaves; or it outputs—on input of a secret path description dk—a variable τ that represents the tree structure, the input path's leaf index, the list of all nodes' secret decapsulation keys on the input path, and the intersection of that path with the added user's path.

[9] A *co-path* of a path is the sequence of siblings along this path.

– $\mathrm{T.rem}(ek, i) \to (\tau, (ek_j)_{j\in[N]}, (p_l)_{l\in[L]}, (cp_l)_{l\in[2,L]})$, resp.,
$\mathrm{T.rem}(dk, i) \to (\tau, i', (dk_l)_{l\in[L]}, l^*)$ Removes a user i from the tree structure and either outputs—on input of the public tree description ek—a variable τ that represents the tree structure, the list of all nodes' public encapsulation keys, the path that needs to be updated due to the removal, and the corresponding co-path; or it outputs—on input of a secret path description dk—a variable τ that represents the tree structure, the input path's leaf index i', the list of all nodes' secret decapsulation keys on the input path, and the intersection of that path with the path removed user i.

Proc BK.gen(n)
00 $\tau \leftarrow \mathrm{T.init}(n)$
01 $(t_j)_{j\in[N]} \leftarrow \mathrm{T.nodes}(\tau)$
02 For all $j \in [N]$:
03 $\quad (pk_j, sk_j) \leftarrow_\$ \mathrm{N.gen}$
04 $ek \leftarrow \mathrm{T.setnodes}(\tau, (pk_j)_{j\in[N]})$
05 For all $i \in [n]$:
06 $\quad (p_l)_{l\in[L_i]} \leftarrow \mathrm{T.path}(\tau, i)$
07 $\quad dk^i \leftarrow \mathrm{T.setpath}(\tau, i, (sk_{p_l})_{l\in[L_i]})$
08 Return $(ek, dk^1, \ldots, dk^n)$

Proc BK.enc(ek)
09 $(pk, sk) \leftarrow_\$ \mathrm{N.gen}$
10 $c \leftarrow (\mathtt{E}, pk)$
11 $u \leftarrow (ek, sk, c)$
12 Return (u, c)

Proc BK.fin(u, ad)
13 $(ek, sk, c) \leftarrow u$
14 $(\tau, (pk_j)_{j\in[N]}) \leftarrow \mathrm{T.getnodes}(ek)$
15 For all $j \in [N]$:
16 $\quad k' \leftarrow \mathrm{N.key}(sk, pk_j)$
17 $\quad (s, k_j) \leftarrow \mathrm{H}(k', c, ad)$
18 $\quad (pk_j, _) \leftarrow \mathrm{N.gen}(s)$
19 $ek \leftarrow \mathrm{T.setnodes}(\tau, (pk_j)_{j\in[N]})$
20 $k \leftarrow k_1$
21 Return (ek, k)

Proc BK.add(ek)
22 $(\tau, (pk_j)_{j\in[N]}, (p_l)_{l\in[L]}, (cp_l)_{l\in[2,L]}, n)$
$\quad \leftarrow \mathrm{T.add}(ek)$
23 $pk^* \leftarrow (pk_{cp_l})_{l\in[2,L]}$
24 $(pk_{p_L}, sk_{p_L}) \leftarrow_\$ \mathrm{N.gen}$
25 $s' \leftarrow_\$ \{0,1\}^\kappa$
26 For l from L to 2:
27 $\quad (pk'_l, sk'_l) \leftarrow \mathrm{N.gen}(s')$
28 $\quad k \leftarrow \mathrm{N.key}(sk'_l, pk_{cp_l})$
29 $\quad (s, s') \leftarrow \mathrm{H}(k, pk'_l)$
30 $\quad (pk_{p_{l-1}}, sk_{p_{l-1}}) \leftarrow \mathrm{N.gen}(s)$
31 $ek \leftarrow \mathrm{T.setnodes}(\tau, (pk_j)_{j\in[N]})$
32 $dk \leftarrow \mathrm{T.setpath}(\tau, n, (sk_{p_l})_{l\in[L]})$
33 $c \leftarrow (\mathtt{A}, pk^*, (pk'_l)_{l\in[2,L]})$
34 Return (ek, dk, c)

Proc BK.dec(dk, ad, c)
35 $(\tau, i, (sk_l)_{l\in[L]}) \leftarrow \mathrm{T.getpath}(dk)$
36 $(t, pk) \leftarrow c$
37 For all $l \in [L]$:
38 $\quad k' \leftarrow \mathrm{N.key}(sk_l, pk)$
39 $\quad (s, k_l) \leftarrow \mathrm{H}(k', c, ad)$
40 $\quad (_, sk_l) \leftarrow \mathrm{N.gen}(s)$
41 $dk \leftarrow \mathrm{T.setpath}(\tau, i, (sk_l)_{l\in[L]})$
42 $k \leftarrow k_1$
43 Return (dk, k)

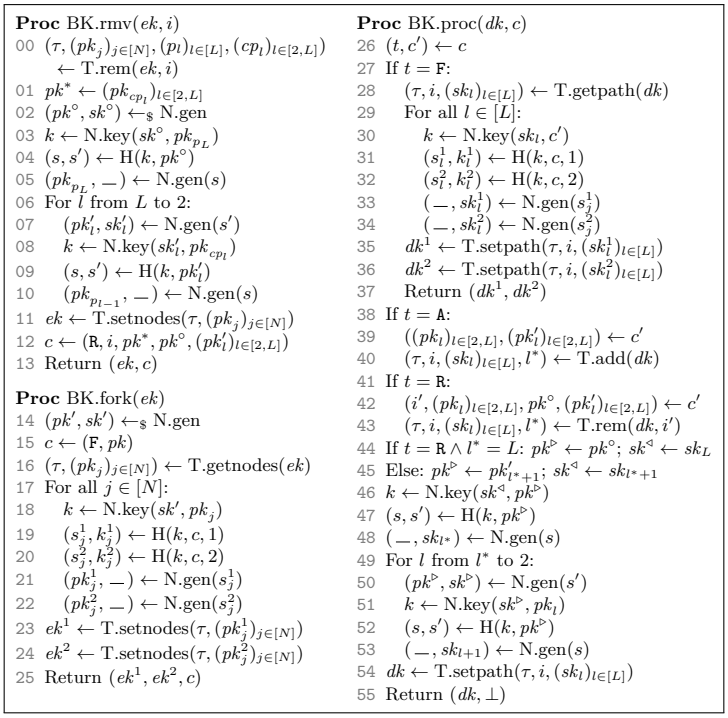
Fig. 7. Construction of Dynamic UB-KEM scheme BK from NIKE N and random oracle H, using tree operations of the form T.getnodes(ek). Algorithms BK.rmv, BK.fork, and BK.proc are in Figure 8.

Key Structure. We begin with presenting the core idea of our dynamic UB-KEM construction with algorithms BK.gen, BK.enc, BK.fin, BK.add, and BK.dec in Figure 7. The remaining algorithms, which do not add a deeper understanding, are in Figure 8. Algorithm BK.gen generates one NIKE key pair per node in the tree (Figure 7, lines 02-03; note that, when n is a power of 2, for example, there are $N = 2n$ nodes in the tree). All NIKE public keys are stored in the UB-KEM encapsulation key (line 04); each UB-KEM decapsulation key only stores the NIKE secret keys on the path from its leaf to the root (lines 05-07).

Encapsulation. To encapsulate a symmetric key to the group and update all components of the encapsulation and decapsulation keys, algorithm BK.enc starts with generating a new NIKE key pair (pk, sk) in line 09. The public key of this key pair becomes the ciphertext sent to all receivers (line 09); label $\mathtt{E}$ is added to ciphertext c here to differentiate it from ciphertexts from other operations, like forking ($\mathtt{F}$).

The corresponding secret key sk is then used with all NIKE public keys of the nodes in the tree to establish preliminary symmetric keys (line 16). Akin to the design principle from our static UB-KEM in Section 4.2, the random oracle is then queried on input of these symmetric keys together with the ciphertext and associated data to output shared keys k_j as well as seeds s; the seed of each node is then used to generate this node's updated NIKE key pair (lines 17-18). The final symmetric output key is the one established with the tree's root node k_1 (line 20).

Decapsulation. To derive the same symmetric key and update all NIKE secret keys on the path of a UB-KEM decapsulation key accordingly, algorithm BK.dec takes the sent NIKE public key pk and computes the shared symmetric key with every node's secret key sk_l on that path. The subsequent output of the



Proc BK.rmv(ek, i)
00 $(\tau, (pk_j)_{j\in[N]}, (p\imath)_{l\in[L]}, (cp_l)_{l\in[2,L]})$
$\quad \leftarrow$ T.rem(ek, i)
01 $pk^* \leftarrow (pk_{cp_l})_{l\in[2,L]}$
02 $(pk^\circ, sk^\circ) \leftarrow_\$$ N.gen
03 $k \leftarrow$ N.key(sk°, pk_{p_L})
04 $(s, s') \leftarrow$ H(k, pk°)
05 $(pk_{p_L}, _) \leftarrow$ N.gen(s)
06 For l from L to 2:
07 $\quad (pk'_l, sk'_l) \leftarrow$ N.gen(s')
08 $\quad k \leftarrow$ N.key(sk'_l, pk_{cp_l})
09 $\quad (s, s') \leftarrow$ H(k, pk'_l)
10 $\quad (pk_{p_{l-1}}, _) \leftarrow$ N.gen(s)
11 $ek \leftarrow$ T.setnodes($\tau, (pk_j)_{j\in[N]}$)
12 $c \leftarrow (\mathtt{R}, i, pk^*, pk^\circ, (pk'_l)_{l\in[2,L]})$
13 Return (ek, c)

Proc BK.fork(ek)
14 $(pk', sk') \leftarrow_\$$ N.gen
15 $c \leftarrow (\mathtt{F}, pk)$
16 $(\tau, (pk_j)_{j\in[N]}) \leftarrow$ T.getnodes(ek)
17 For all $j \in [N]$:
18 $\quad k \leftarrow$ N.key(sk', pk_j)
19 $\quad (s_j^1, k_j^1) \leftarrow$ H($k, c, 1$)
20 $\quad (s_j^2, k_j^2) \leftarrow$ H($k, c, 2$)
21 $\quad (pk_j^1, _) \leftarrow$ N.gen(s_j^1)
22 $\quad (pk_j^2, _) \leftarrow$ N.gen(s_j^2)
23 $ek^1 \leftarrow$ T.setnodes($\tau, (pk_j^1)_{j\in[N]}$)
24 $ek^2 \leftarrow$ T.setnodes($\tau, (pk_j^2)_{j\in[N]}$)
25 Return (ek^1, ek^2, c)

Proc BK.proc(dk, c)
26 $(t, c') \leftarrow c$
27 If $t = \mathtt{F}$:
28 $\quad (\tau, i, (sk_l)_{l\in[L]}) \leftarrow$ T.getpath(dk)
29 $\quad$ For all $l \in [L]$:
30 $\quad\quad k \leftarrow$ N.key(sk_l, c')
31 $\quad\quad (s_l^1, k_l^1) \leftarrow$ H($k, c, 1$)
32 $\quad\quad (s_l^2, k_l^2) \leftarrow$ H($k, c, 2$)
33 $\quad\quad (_, sk_l^1) \leftarrow$ N.gen(s_j^1)
34 $\quad\quad (_, sk_l^2) \leftarrow$ N.gen(s_j^2)
35 $\quad dk^1 \leftarrow$ T.setpath($\tau, i, (sk_l^1)_{l\in[L]}$)
36 $\quad dk^2 \leftarrow$ T.setpath($\tau, i, (sk_l^2)_{l\in[L]}$)
37 $\quad$ Return (dk^1, dk^2)
38 If $t = \mathtt{A}$:
39 $\quad ((pk_l)_{l\in[2,L]}, (pk'_l)_{l\in[2,L]}) \leftarrow c'$
40 $\quad (\tau, i, (sk_l)_{l\in[L]}, l^*) \leftarrow$ T.add(dk)
41 If $t = \mathtt{R}$:
42 $\quad (i', (pk_l)_{l\in[2,L]}, pk^\circ, (pk'_l)_{l\in[2,L]}) \leftarrow c'$
43 $\quad (\tau, i, (sk_l)_{l\in[L]}, l^*) \leftarrow$ T.rem(dk, i')
44 If $t = \mathtt{R} \wedge l^* = L$: $pk^\triangleright \leftarrow pk^\circ$; $sk^\triangleleft \leftarrow sk_L$
45 Else: $pk^\triangleright \leftarrow pk'_{l^*+1}$; $sk^\triangleleft \leftarrow sk_{l^*+1}$
46 $k \leftarrow$ N.key($sk^\triangleleft, pk^\triangleright$)
47 $(s, s') \leftarrow$ H($k, pk^\triangleright$)
48 $(_, sk_{l^*}) \leftarrow$ N.gen(s)
49 For l from l^* to 2:
50 $\quad (pk^\triangleright, sk^\triangleright) \leftarrow$ N.gen(s')
51 $\quad k \leftarrow$ N.key($sk^\triangleright, pk_l$)
52 $\quad (s, s') \leftarrow$ H($k, pk^\triangleright$)
53 $\quad (_, sk_{l+1}) \leftarrow$ N.gen(s)
54 $dk \leftarrow$ T.setpath($\tau, i, (sk_l)_{l\in[L]}$)
55 Return $(dk, \perp)$

Fig. 8. Remaining algorithms of Dynamic UB-KEM construction from Figure 7.

random oracle yields all symmetric keys as well as the seeds to generate the nodes' updated NIKE secret keys (lines 37-40).

Membership Operations. When adding a new decapsulation key, algorithm BK.add begins with generating a fresh leaf NIKE key pair (line 24). Then, with each existing NIKE public key on this new leaf's co-path to the root, an ephemeral non-interactive key exchange is performed deterministically to derive shared key k (lines 27-28). From this key k, two seeds are derived via the random oracle: one for updating the joint parent's key pair and one for generating an ephemeral key pair for the NIKE with next level's node on the co-path (lines 29-30, resp. 27-28). All generated ephemeral public keys are contained in the sent ciphertext. The final added decapsulation key dk contains all (non-ephemeral) new secret keys on the processed path (line 32).

Similarly, algorithm BK.rmv in Figure 8 updates the removed leaf's path such that only the nodes on the co-path but not the removed decapsulation key can reproduce key material on the updated path.

Forking. As we will see in our D-MSMR GURKE construction for adding senders, it is beneficial to fork a UB-KEM instance instead of generating a fresh one from scratch. The idea of algorithm BK.fork follows the principle of algorithms BK.enc and BK.fin: a single NIKE key pair is sampled to perform key exchanges with all existing key pairs in the tree; yet, instead of using the exchanged keys for updating each node in the tree *once*, the forking algorithm updates each node in the tree *twice* by (1) performing this key exchange with each node's NIKE public key, (2) feeding the random oracle twice, each with a different input tag, and (3) deriving two new key pairs via seeds from the random oracle output. This produces one tree for each of the two output encapsulation keys. On the receiver side, the same method is applied to each UB-KEM leaf's path of NIKE secret keys. This yields two paths for every receiver, of which the latter corresponds to the new forked instance.

Performance. The UB-KEM encapsulation key contains $2n$ NIKE public keys, where n is the group size. Each corresponding decapsulation key contains $\log n$ NIKE secret keys. Ciphertexts output by algorithms BK.enc and BK.fork contain a single NIKE public key, whereas ciphertexts for membership operations via algorithms BK.add and BK.rmv consist of $\log n$ NIKE public keys each. Using Hashed Diffie-Hellman, NIKE public and secret keys consist of a single group and field element, respectively. We

provide a proof of concept implementation of our construction in [Grü25] and we present a performance evaluation thereof in Appendix B.1 that demonstrates its practicality.

Differences to TreeKEM Constructions. The core novelty of our UB-KEM design is that the entire trees of NIKE public and secret keys in the UB-KEM encapsulation key and decapsulation key are continuously updated for FS and IA. For FS, all NIKE secret keys in the tree are replaced after each operation such that later corruptions cannot affect the secrecy of symmetric keys decapsulated earlier. This level of FS is not achieved by standard TreeKEM [BBR18] as standardized in MLS [BBR+23], which only refreshes single key-paths in the tree upon updates of the corresponding leafs' users; since other users' keys in the tree are not changed during these path updates, corrupting these other users' key material breaks the secrecy of past decapsulated symmetric keys. The strengthened TreeKEM variant by Alwen et al. [ACDT20] uses Updatable KEM [JMM19] to overcome this weakness by replacing both the keys on updating users' paths as well as keys on their co-paths[10]. This solution achieves a similar FS level compared with our UB-KEM at the cost of logarithmic communication overhead. (Note that encapsulations with our UB-KEM yield constant sized ciphertexts.)

Yet, neither construction fulfills IA: once a receiver in these TreeKEM variants [BBR18, ACDT20] processes a ciphertext that is carefully manipulated due to an active attack, their key material stays compatible and does not diverge from the sender's key material. Since the updates in our UB-KEM construction are influenced by the sent and received associated-data–ciphertext pairs, active attacks that manipulate this information immediately diverge encapsulation keys from the corresponding decapsulation keys. This level of IA has so far only been achieved with far heavier tools: in two even stronger TreeKEM variants that are also secure against bad randomness, Alwen et al. [ACJM20, AMT23] use inefficient Hierarchical Identity-Based Encryption (HIBE) for continuously updating the entire tree of keys. With our UB-KEM design, we demonstrate that practically efficient tools like NIKE suffice for strong FS and IA under good randomness with low overhead. (As noted earlier, HIBE-like tools are necessary under bad randomness [BRV20, RSS23], which we elaborate on in Appendix B.2.)

Theorem 2. *UB-KEM* BK *from Fig. 7 and 8 offers key indistinguishability. More precisely, if* H *is modeled as a random oracle, then for every adversary* $\mathcal{A}$ *attacking* BK *in games* $\mathrm{IND}^b_{\mathrm{BK}}$ *from Fig. 5, there exists adversary* $\mathcal{B}$ *in game* OW_{N} *of NIKE* N *from Fig. 17 such that* $\mathrm{Adv}^{\mathrm{ind}}_{\mathrm{BK}}(\mathcal{A}) \leq q_{\mathrm{Gen}} \cdot (q_{\mathrm{Add}} + q_{\mathrm{Rmv}}) \cdot 2(n_{init} + q_{\mathrm{Add}}) \cdot (q_{\mathrm{Fin}} + 2q_{\mathrm{Fork}}) \cdot \mathrm{Adv}^{\mathrm{ow}}_{\mathrm{N}}(\mathcal{B})$*, where* n_{init} *is the largest initial group size, and the running time of* $\mathcal{B}$ *is about that of* $\mathcal{A}$*.*

We defer the proof to Appendix D.2.

5 Static Group Ratcheting Constructions

We begin with building GURKE for static groups. All three sub-cases thereof, S-SSMR, S-MSSR, and S-MSMR, are based on the same design principle. Since the complexity overhead for the most powerful S-MSMR variant is manageable, we begin with this case and then simplify it for the S-MSSR and S-SSMR cases.

5.1 S-MSMR

Our construction in Figure 9 uses a static UB-KEM and a signature scheme.

State Structure. Initially, a UB-KEM instance with one decapsulation key per receiver is generated for each sender (line 01). Furthermore, each sender generates an individual signing key pair (line 02). This key material is distributed such that each sender holds the UB-KEM encapsulation key for their own instance as well as their signing key (line 03), and each receiver holds a UB-KEM decapsulation key and a verification key for every sender (lines 04-07).

[10] I.e., all nodes on the path from the updating user's leaf to the root as well as their siblings.

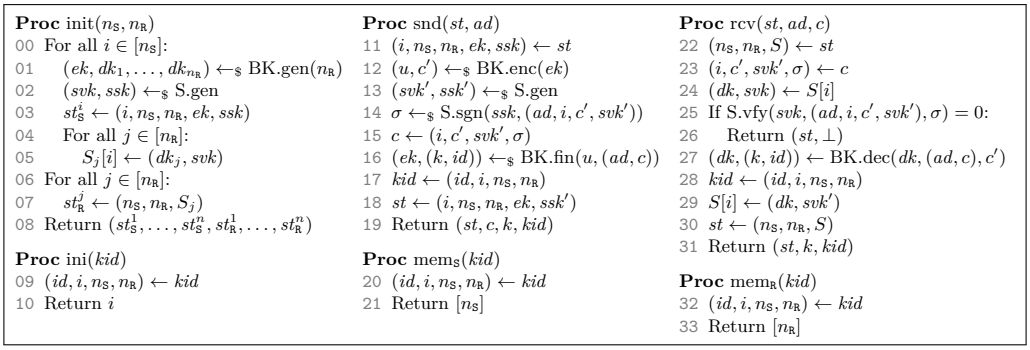


Fig. 9. S-MSMR Construction from static UB-KEM BK and signature scheme S.

Sending. To establish a new symmetric key with the receivers, a sender starts with encapsulating to the current UB-KEM key and generating a new signing key pair (ssk', svk') (lines 12-14). Using the prior signing key ssk, the sender signs the resulting UB-KEM ciphertext as well as the new verification key svk' (line 13). The composition of these two components as well as the computed signature σ and the sender's identifier i yield the final GURKE ciphertext c (line 14). Together with the input associated data, c is used to finalize the UB-KEM encapsulation that outputs the updated encapsulation key, the final symmetric output key, and a random string for the output key identifier (line 16).

Receiving. Algorithm rcv begins with verifying the signature (line 25) and, if this succeeds, decapsulating the UB-KEM ciphertext (line 27), which updates the decapsulation key and outputs the (same) symmetric output key and random string for the key identifier.

Security. Intuitively, confidentiality, including Forward Security, as well as the ability to diverge the receiver state from the sender state upon differing processed transcripts is based on UB-KEM. For this, note that UB-KEM encapsulation key and decapsulation key are updated during every send and receive operation, respectively, by processing the final ciphertext and the associated data. The primary guarantee enabled by employing signatures is that receivers only accept manipulated ciphertexts if the sender(s) are impersonated—which requires the exposure of the most recent sender state.

Performance. Each receiver state contains n_S UB-KEM decapsulation keys and verification keys. A receiver state size linear in the number of senders is asymptotically optimal, as we prove in Appendix E.1. We show this by building puncturable encryption from S-MSSR, then applying a recent result from Barooti et al. [BCC+23a].

Each sender state only contains a single UB-KEM encapsulation key as well as a signing key. Since GURKE trivially implies a one-time signature scheme as well as a KEM, this is optimal, too. Finally, the ciphertext contains a UB-KEM ciphertext, a verification key, and a signature. Since we do not require special properties of the signature scheme, and static UB-KEM can be built from standard KEMs (see Section 4.2), this is highly efficient and asymptotically optimal, too.

Theorem 3. *S-MSMR GURKE* R *from Fig. 9 offers key indistinguishability. More precisely, for every adversary $\mathcal{A}$ attacking* R *in games* IND^b_R *from Fig. 3, there exists adversary $\mathcal{B}_{BK}$ in game* IND_{BK} *from Fig. 5 and adversary $\mathcal{B}_S$ in game* SUF_S *from Fig. 15 such that* $\mathrm{Adv}^{\mathrm{ind}}_R(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{ind}}_{BK}(\mathcal{B}_{BK}) + \mathrm{Adv}^{\mathrm{suf}}_S(\mathcal{B}_S)$*, where the running times of $\mathcal{B}_{BK}$ and $\mathcal{B}_S$ are about that of $\mathcal{A}$.*

We defer the proof to Appendix D.3.

5.2 S-MSSR

Reducing the number of receivers for the S-MS*S*R to a single one primarily affects the initialization: since only one receiver state needs to be created, each sender just generates a UB-KEM instance with a single corresponding decapsulation key. Thus, intuitively, each sender has a two-party session with the receiver.

Nevertheless, the overall performance remains unchanged: each sender still holds a UB-KEM encapsulation key as well as a signing key, and the single receiver holds one UB-KEM decapsulation key and one verification per sender. Also the ciphertext structure remains the same.

Corollary 1. *Key indistinguishability of S-MSMR GURKE* R *from Figure 9 implies key indistinguishability of S-MSSR GURKE.*

Lower Bounds. The lower bound on the size of the receiver state, which we prove in Appendix E.1, applies to S-MSSR, too. This shows that our S-MSSR construction's state sizes are asymptotically optimal.

5.3 S-SSMR

For the single-sender case, only one UB-KEM instance and one signature key pair is generated. Thus, both the sender and the receivers only store one component per key pair each, which is clearly optimal.

Corollary 2. *Key indistinguishability of S-MSMR GURKE* R *from Figure 9 implies key indistinguishability of S-SSMR GURKE.*

5.4 Sender Key Mechanism

The Sender Key Mechanism [Mar14, BCG23], which Signal, WhatsApp, and other secure messengers use as the core of their group messaging protocol, can be considered a weakly secure instantiation of S-SSMR GURKE. Concretely, every group chat in these messengers consists of n Sender Key instances, one for each group member. For their own Sender Key instance, each group member is the single sender who sends their ciphertexts to the static group of remaining group members, who act as the corresponding receivers. The sent ciphertexts continuously establish and update key material in a long-term group conversation.

Treatment as S-SSMR. While the Sender Key Mechanism entangles establishing and updating keys with the encryption of payload messages under these keys, we extract its S-SSMR GURKE core in Figure 10. The main principle is that the sender and all its receivers store a so-called *chain key ck* in their state that is refreshed upon every send operation via a Pseudorandom Generator (PRG). Along with the refreshed chain key, a new message key is output by the PRG (lines 11). (In the actual Sender Key Mechanism, this message key is used for payload encryption.) To allow for out-of-order receipts, each ciphertext contains the current value of send-counter l as well as the sender's identity i (line 14). Upon receipt, the receiver performs chain key updates until the right message key is computed (lines 27-30). For authenticity, ciphertexts are signed (lines 13-25).

Proc $\text{init}(n_{\text{R}})$
00 $i \leftarrow_{\$} \mathcal{UID}$
01 $ck \leftarrow_{\$} \mathcal{K}$
02 $M[\cdot] \leftarrow \bot$
03 $(svk, ssk) \leftarrow_{\$} \text{S.gen}$
04 $st_{\text{S}} \leftarrow (i, n_{\text{R}}, 0, ck, ssk)$
05 For all $j \in [n_{\text{R}}]$:
06 $\quad st^j_{\text{R}} \leftarrow (i, n_{\text{R}}, 0, ck, M, ssk)$
07 Return $(st_{\text{S}}, st^1_{\text{R}}, \ldots, st^n_{\text{R}})$

Proc $\text{ini}(kid)$
08 $(i, n_{\text{R}}, l) \leftarrow kid$
09 Return i

Proc $\text{snd}(st, ad)$
10 $(i, n_{\text{R}}, l, ck, ssk) \leftarrow st$
11 $(ck, mk) \leftarrow \text{G}(ck)$
12 $kid \leftarrow (i, n_{\text{R}}, l)$
13 $\sigma \leftarrow_{\$} \text{S.sgn}(ssk, (i, l))$
14 $c \leftarrow (i, l, \sigma)$
15 $st \leftarrow (i, n_{\text{R}}, l+1, ck, ssk)$
16 Return (st, c, mk, kid)

Proc $\text{mem}_{\text{S}}(kid)$
17 $(i, n_{\text{R}}, l) \leftarrow kid$
18 Return $\{i\}$

Proc $\text{mem}_{\text{R}}(kid)$
19 $(i, n_{\text{R}}, l) \leftarrow kid$
20 Return $[n_{\text{R}}]$

Proc $\text{rcv}(st, ad, c)$
21 $(i, n_{\text{R}}, l, ck, M, svk) \leftarrow st$
22 $(i', l', \sigma) \leftarrow c$
23 If $i \neq i' \vee l' < l \wedge M[l'] = \bot$:
24 $\quad$ Return $(st, \bot, \bot)$
25 If $\text{S.vfy}(svk, (i', l')) = 0$:
26 $\quad$ Return $(st, \bot, \bot)$
27 While $l < l'$:
28 $\quad (ck, mk) \leftarrow \text{G}(ck)$
29 $\quad M[l] \leftarrow mk$
30 $\quad l \leftarrow l + 1$
31 $mk \leftarrow M[l']$
32 $M[l'] \leftarrow \bot$
33 $kid \leftarrow (i', n_{\text{R}}, l')$
34 $st \leftarrow (i, n_{\text{R}}, l, ck, M, svk)$
35 Return (st, mk, kid)

Fig. 10. Interpretation of the Sender Key Mechanism as an S-SSMR Construction.

Security Weaknesses. Balbás et al. [BCG23] formally analyzed the security of the Sender Key Mechanism. Based on their results and considering our security definition, we identify its following shortcomings: (i) since key updates are deterministic, once the sender state is exposed, all future keys are revealed, (ii) since the signing key pair is static, once the sender state is exposed, the sender can be impersonated forever, and (iii) since key updates are independent of the sent and received ciphertexts, after the sender is impersonated, the corresponding receiver's key material remains compatible with the one of the sender; thus, exposing this receiver will reveal keys established between the original sender and all receivers who were not affected by the impersonation.

Out-of-Order Tolerance. A special feature of the Sender Key Mechanism is that receivers can receive out of order: even if ciphertexts are dropped or reordered, the computation of message keys is unaffected by this. Out-of-order delivery in the two-party messaging context has been studied fundamentally by Barooti et al. [BCC+23a]; for the group setting, we consider this an interesting, yet orthogonal question for future work.

Direct Performance Comparison. The main differences between our S-SSMR GURKE construction from Figure 9 and the Sender Key Mechanism from Figure 10 are that our construction additionally (a) generates a fresh signing key pair and (b) encapsulates with the UB-KEM during every send operation. Since each UB-KEM encapsulation includes a Non-Interactive Key Exchange and a subsequent key generation, this is induces three additional public-key operations. State sizes and ciphertext sizes are comparable.

Performance in Dynamic Group Messaging. Although, the Sender Key Mechanism itself only works for a single sender communicating with a static group of receivers, in practice it is used to build communication in dynamic groups of users who are senders and receivers simultaneously. Concretely, WhatsApp, Signal, and other messengers [RMS18, BCG23] implement their dynamic group chats based on the Sender Key Mechanism by letting each user be a sender to the group as well as a receiver for every other user in the group; as mentioned above, for a group of n members, n Sender Key instances are deployed. In this application scenario, the main performance drawback of the Sender Key Mechanism takes effect: every time a new user is added to the group, this user distributes their chain key and their verification key to all other members via pairwise messaging channels; additionally, all existing members share their current chain key and verification key with the new user via the pairwise messaging channels. When removing a member from the group, all remaining members have to generate and share fresh chain keys via the pairwise channels. Thus, every add operation induces $2n$ pairwise messages, and every remove operation induces n^2 pairwise messages.

Using D-MSMR GURKE. Looking ahead, our D-MSMR GURKE construction in Section 6.3 can be used more efficiently than the Sender Key Mechanism to build full group messaging: each member of the group is simultaneously a GURKE sender and a GURKE receiver, but all these senders and receivers are part of the same, single D-MSMR instance. When a new user is added to the group chat, one GURKE *sender* and one GURKE *receiver* are added for them to this D-MSMR instance.

For adding a GURKE *sender*, our D-MSMR construction sends a constant sized ciphertext to all existing senders, another constant sized ciphertext to all receivers, and the new sender-state of size n via the secure pairwise channel to the newly added sender. For adding a GURKE *receiver*, our D-MSMR construction sends a ciphertext of size n to all senders, a ciphertext of size $n \log n$ to all receivers[11], and the new receiver-state of size n via the secure pairwise channel to the newly added receiver. For removing a GURKE *sender*, our D-MSMR construction sends a constant sized ciphertext to all remaining members. For removing a GURKE *receiver*, our D-MSMR construction sends a constant sized ciphertext to all senders and a ciphertext of size $n \log n$ to all remaining receivers[11].

In total, this yields a communication overhead of $\mathcal{O}(n \log n)$ for adding and removing with our D-MSMR GURKE construction compared to $\mathcal{O}(n)$ for adds and $\mathcal{O}(n^2)$ for removals with the Sender Key Mechanism. We could further improve our construction by cleverly splitting ciphertexts at the server and individually forwarding relevant information to each receiver. Finally, we want to remark that the main drawbacks of using black-box CGKA as an evident alternative for building group messaging would be

[11] More concretely, each existing sender sends an individual ciphertext of size $\log n$ to all receivers, which is queued and piggybacked to their next operation's ciphertext; thus, the total size of $n \log n$ is amortized.

(even further) increased complexity and relying on an active server that schedules and resolves concurrent operations.

6 Dynamic Group Ratcheting Constructions

We now turn to building dynamic GURKE constructions that are simplified due to introducing dynamic UB-KEM as an intermediate abstraction layer. As for the static case, the dynamic GURKE constructions primarily differ in bookkeeping tasks as well as additional coordination between users. Nevertheless, this is non-trivial, and for the dynamic multi-sender constructions, we use UB-KEM's fork mechanism for adding senders.

6.1 D-SSMR

At a high level, our GURKE construction D-SSMR in Figure 11 resembles S-SSMR in Figure 9. Its core is abstracted in helper encaps: the sender first calls BK.enc (line 38), all information sent to the receivers including a new signature verification key is jointly signed (line 40), and finally the sender calls BK.fin (also with respect to all information) (line 42). Yet, the crucial difference is that the underlying UB-KEM instantiations differ significantly, as we illustrate in Sections 4.2 and 4.3. In particular, the tree-based dynamic UB-KEM used in the D-SSMR GURKE construction allows for adding and removing decapsulation keys, which we use in algorithms add and rmv.

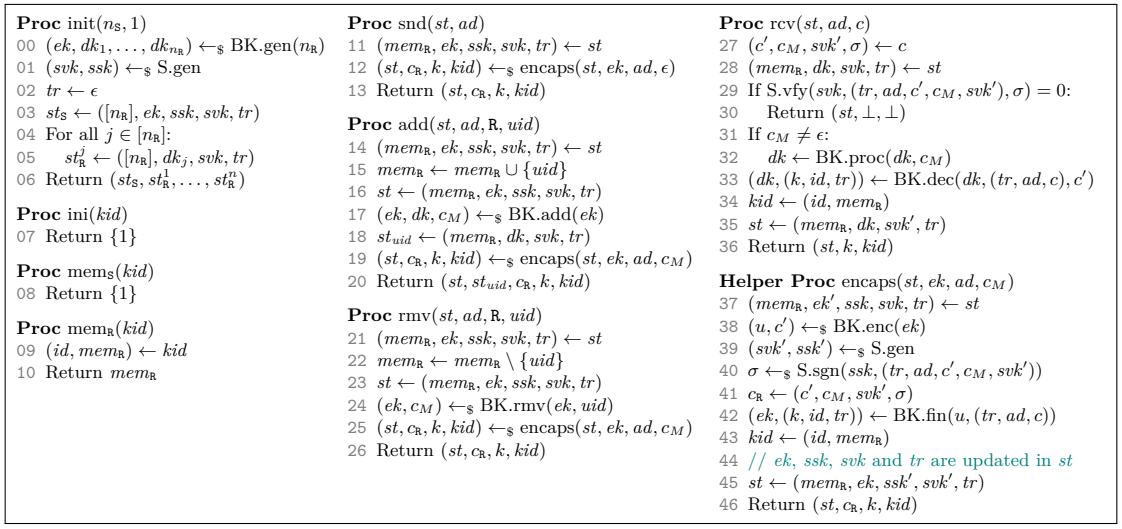


Fig. 11. D-SSMR Construction from UB-KEM BK and signature scheme S.

Adding Receivers. In order to add a new receiver, algorithm add uses UB-KEM algorithm BK.add that outputs a new decapsulation key (line 17). This new decapsulation key then becomes part of the new receiver's initial state, along with the sender's current verification key, set mem_R and variable tr (line 18). After this, the standard routine for establishing a symmetric key with the entire group of receivers (including the new one) is executed (line 19).

Removing Receivers. When removing a receiver, only their decapsulation key is removed via UB-KEM algorithm BK.rmv (line 24). Thereafter, the standard key establishing routine is executed (line 25).

In rcv, if the sender adds or removes receivers, the corresponding UB-KEM membership management ciphertexts are processed upon receipt (line 32); it otherwise proceeds in analogy to the static case.

Security. As for the static group case, security of D-SSMR GURKE relies on the UB-KEM that provides Forward Security and diverges receiver states from sender states as soon as the processed transcript differs. The employed signature scheme enforces that the adversary has to expose the sender in order to mount an impersonation attack.

Corollary 3. *Key indistinguishability of D-MSMR GURKE* R *from Figures 12 and 13 implies key indistinguishability of D-SSMR GURKE.*

6.2 D-MSSR

The core challenge for building D-MSSR is adding new senders.

Naive Sender Adding. The simplest approach for adding a sender would be generating a fresh UB-KEM instance and encrypting the resulting fresh decapsulation key to the receiver. For encryption, the adding sender could use the symmetric key established automatically when sending. However, this solution does not scale for larger sets of receivers—which is important for the upcoming D-MS*M*R case. In particular, since each receiver would need to obtain an individual UB-KEM decapsulation key for the new sender's instance, the communication overhead for adding a sender would be linear in the number of receivers.

Using UB-KEM Forks. Alternatively, the adding sender can use the UB-KEM BK.fork algorithm to derive a second UB-KEM instance that is given to the newly added sender. All receivers can then fork their individual UB-KEM decapsulation key. Thereby, the communication overhead for adding a sender is not (necessarily) linear in the number of receivers, but dependent on the efficiency of algorithm BK.fork. As we demonstrate in Section 4.3, forking an instance only induces a constant communication overhead.

The full D-MSSR GURKE construction can be extracted from the D-MSMR GURKE construction which we describe below.

Corollary 4. *Key indistinguishability of D-MSMR GURKE* R *from Figures 12 and 13 implies key indistinguishability of D-MSSR GURKE.*

6.3 D-MSMR

D-MSMR is the most general case we consider, in which a dynamic set of senders can add and remove receivers over time. The pseudocode for our construction is provided in Figures 12 and 13. Figure 12 specifies init, snd and add (separated between adding senders and receivers), as well as helpers encaps and enq-ops. Figure 13 specifies the remaining functions, namely rcv, rmv, proc, ini, $\text{mem}_{\mathtt{S}}$ and $\text{mem}_{\mathtt{R}}$, as well as additional helpers. We provide an overview of our protocol in the text below; if not stated otherwise, line numbers refer to Figure 12.

Sender State. A sender i keeps in st its view of the current senders and receivers ($mem_{\mathtt{S}}/mem_{\mathtt{R}}$), a UB-KEM (BK) public key ek used to encapsulate to its receivers (i.e., $mem_{\mathtt{R}}$), a signature key pair (svk, ssk), a trace variable tr, and a queue ops of operations from other senders. Note that each sender is associated with a different BK instance.

Receiver State. A receiver j stores, for *every* sender i, different membership sets $mem_{\mathtt{S}}$ and $mem_{\mathtt{R}}$, since senders may have differing views over time.[12] Receiver j further tracks for each sender i a BK decapsulation key dk, signature verification key svk and trace variable tr like our previous constructions.

Sending and Receiving. The core key encapsulation logic used in snd, add and rmv is as in D-SSMR, and is likewise abstracted in helper encaps. With multiple senders now though, sender i could have previously called proc one or more times since last sending (i.e., since last calling snd, add or rmv), in order to process adds or removes initiated by other senders. Therefore, inside of a snd call, helper enq-ops is first called to inform the sender's receivers that it has processed these adds/removes, and in some cases to send additional information, which in particular outputs a queue of tuples cq.

Receiving (rcv) is analogous to sending: j first processes queue cq, except when j was added by another sender and has not yet received a ciphertext from i, in which case it ignores all operations until its initiation message (detailed in Figure 13).

[12] To reduce communication, only membership updates are sent as needed to update these sets.

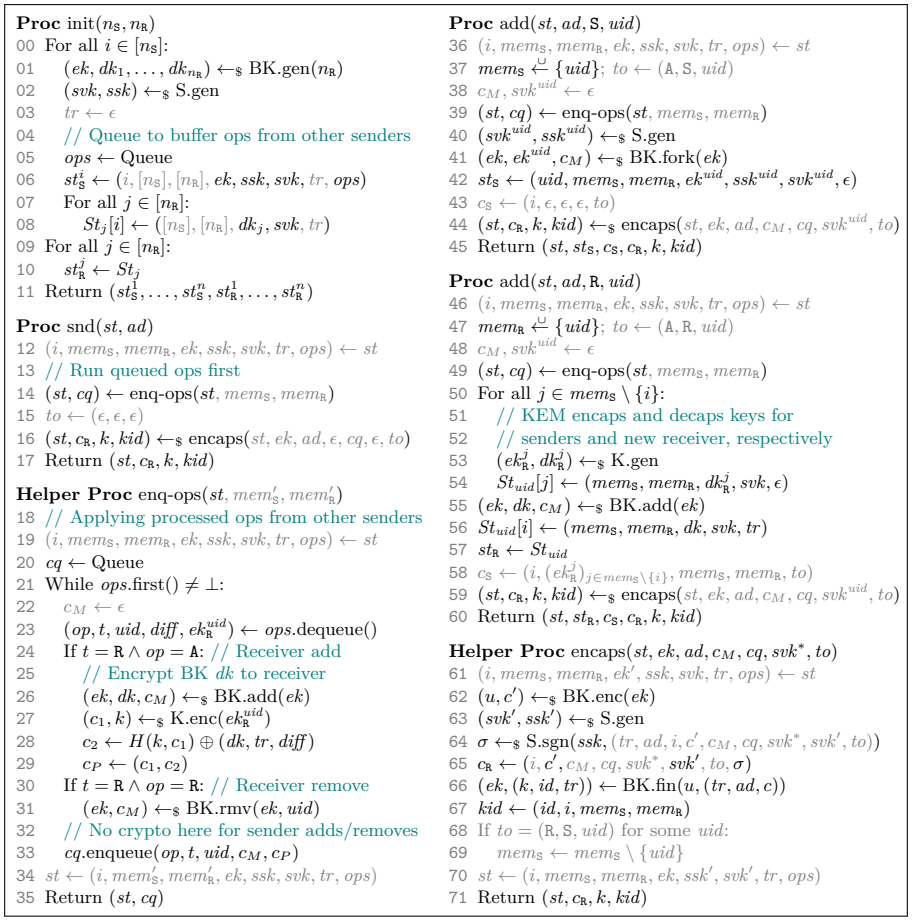


Fig. 12. D-MSMR Construction from UB-KEM BK and signature scheme S. Algorithms rmv, rcv, ini, mem_S and mem_R, as well as additional helpers, are in Figure 13. Gray text can safely be ignored on a first pass.

Adding/Removing Receivers. When adding a receiver r (via add($st, ad, \mathtt{R}, r$), Figure 12), sender i samples one one-time-use KEM encapsulation/decapsulation key pair for each sender $j \neq i$ in its view (line 53), includes all of these decapsulation keys as part of r's state (line 54), and will ultimately send each sender their one-time encapsulation key (via ciphertext c_S input to proc (line 58)). Sender i then calls BK.add to add r to its BK instance (line 55). The one-time encapsulation keys are used by each active sender, after they have called proc, to 'include' r in their BK instance when they next call snd, add or rmv (via enq-ops); concretely, they encrypt the decapsulation key output by BK.add to r (lines 26 to 28).[13]

Turning now to sender-to-sender communication, i sends their membership sets to all other senders (via ciphertext c_S, line 58)). Sender j, on processing c_S, communicates their view of the membership to the new receiver the next time they send, i.e., the difference between i and j's views. We do this to optimise communication to the receiver, in particular when senders are well-synchronised.

Removing a receiver is in some sense simpler since the sender i does not need to send cryptographic information to other senders, but here each sender removes the receiver from their BK instance via BK.rmv (i immediately and others when next sending after the corresponding proc call). The removed receiver itself does nothing; other senders will each remove the receiver from their BK instance the next time they send after the corresponding proc call.

Adding/Removing Senders. To add a sender s (via add($st, ad, \mathtt{S}, s$)), sender i calls BK.fork to derive a new BK encapsulation key ek^s for s (line 41). This key is in part output in s's initial state, along

[13] Note this logic is not necessary for sender i itself who directly includes the BK decapsulation key in r's state (line 56). To simplify presentation, sender ciphertexts are still sent to the sender who called add, in which case the queue is ignored.

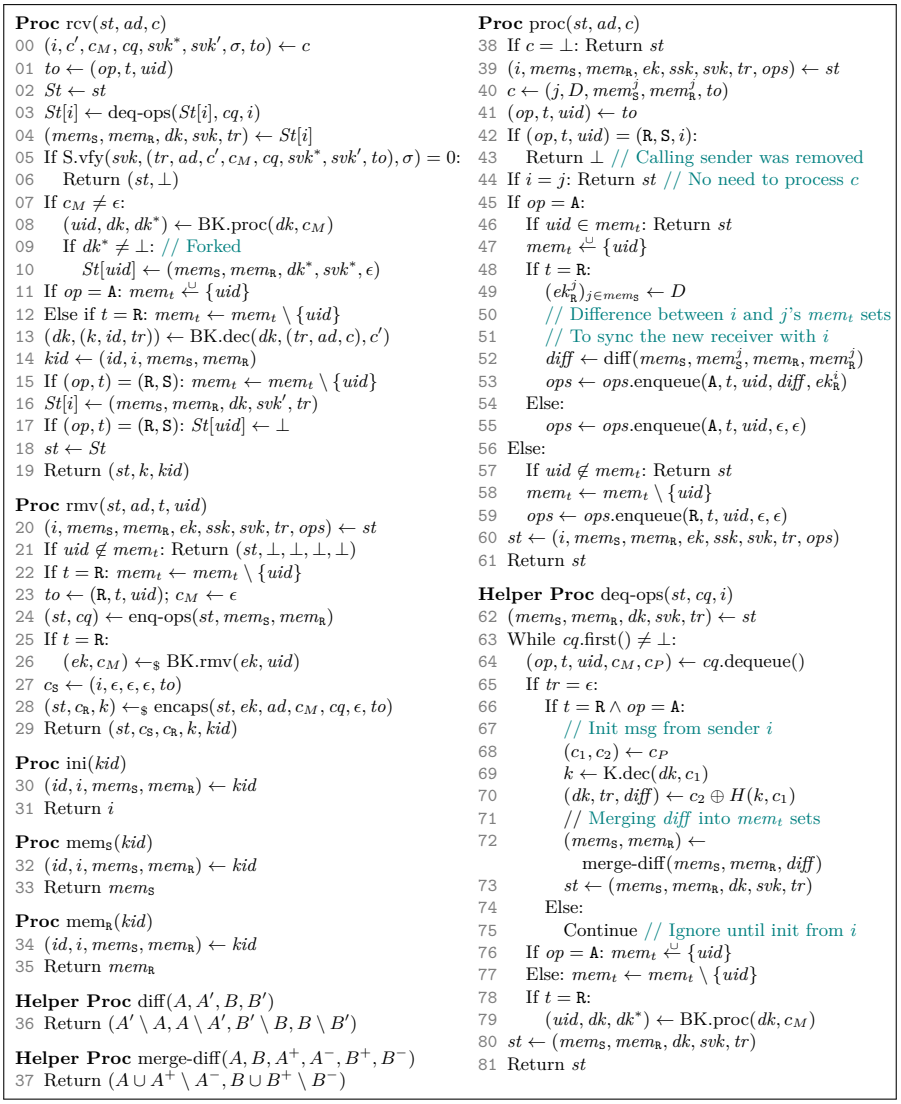


Fig. 13. Remaining algorithms of D-MSMR construction from Figure 12.

with a fresh signature key pair, sender i's membership sets and s's identifier (s) (line 42). The signature verification key and a ciphertext c_M are sent to all receivers, the latter enabling each receiver to derive its BK decapsulation key corresponding to ek^s. Sender-to-sender communication is only used here for updating the membership sets of other senders.

When removing a sender, note in our construction that sender s is included in $mem_{\mathtt{S}}(kid)$ one last time (unlike for removed receivers)—as senders receive their own sender ciphertexts, the sender deletes their state upon processing it (Figure 13, line 43).[14] Senders are otherwise removed from $mem_{\mathtt{S}}$ in proc by other senders (Figure 13, line 58), and state information is removed by receivers in rcv (Figure 13, line 17). Attempting to remove a sender (or receiver) that is not in $mem_{\mathtt{S}}$ (resp. $mem_{\mathtt{R}}$) immediately fails (Figure 13, line 21).[15] Note also if two senders concurrently remove a sender (or receiver), then on the sender side, the second operation processed in proc is ignored (Figure 13, lines 46 and 57), and on the receiver side, the encapsulated key is still processed.

[14] Logic is present in rmv (Figure 13, line 22), encaps (Figure 12, lines 68 and 69) and rcv (Figure 13, line 15) to handle this.

[15] We do not require comparable checks for adds since the correctness and security games make it impossible to add a user that already exists.

On Concurrent Adds and Removes. The above works as intuitively expected when all senders are strongly synchronised and in particular do not concurrently add/remove different senders or receivers. In proportion to concurrency here, our protocol does lead to divergence in the membership sets of both senders and receivers. The protocol still functions as ideally expected without a higher-level synchronisation mechanism.

For example suppose that sender i adds sender s, and sender j adds receiver r *concurrently*. Then, receiver r is not made aware of s and vice-versa, and so their membership sets will diverge from that of all other 'active' users. It is possible to circumvent this, for example if all other senders inform s and r of this information. We leave an exploration of different design choices and trade-offs to future work.

In any event, our protocol readily allows for keys to be concurrently established via calls to snd by several senders, highlighting that multi-sender ratcheted key exchange itself does not require strong synchronisation between senders (unlike what is assumed in typical continuous group key agreement protocols).

Theorem 4. *D-MSMR GURKE* R *from Fig. 12 and 13 offers key indistinguishability. More precisely, for every adversary $\mathcal{A}$ attacking* R *in games* $\mathrm{IND}^b_{\mathrm{R}}$ *from Fig. 3, there exists adversary $\mathcal{B}_{\mathrm{BK}}$ in game* $\mathrm{IND}_{\mathrm{BK}}$ *from Fig. 5, adversary $\mathcal{B}_{\mathrm{S}}$ in game* $\mathrm{SUF}_{\mathrm{S}}$ *from Fig. 15 and adversary $\mathcal{B}_{\mathrm{K}}$ in game* $\mathrm{IND}_{\mathrm{K}}$ *from Fig. 16 such that* $\mathrm{Adv}^{\mathrm{ind}}_{\mathrm{R}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{ind}}_{\mathrm{BK}}(\mathcal{B}_{\mathrm{BK}}) + \mathrm{Adv}^{\mathrm{ow}}_{\mathrm{K}}(\mathcal{B}_{\mathrm{K}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathrm{S}}(\mathcal{B}_{\mathrm{S}})$*, where the running times of $\mathcal{B}_{\mathrm{BK}}$ and $\mathcal{B}_{\mathrm{S}}$ are about that of $\mathcal{A}$ and $\mathcal{B}_{\mathrm{K}}$ is about q_{Enc} times that of $\mathcal{A}$, where q_{Enc} is the number of* Enc *queries $\mathcal{A}$ makes.*

We defer the proof to Appendix D.4.

Acknowledgements

We thank the anonymous reviewers for their thoughtful feedback as well as Julian Loss and Omar Renawi for discussions during the early stages of this project. Daniel Collins was supported in part by AnalytiXIN and Sunday Group, Inc., and completed part of this work while working at EPFL.

References

AAB+21. Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 222–253. Springer, Cham, November 2021.

AAN+22. 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, Cham, 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, Cham, 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, Cham, 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, Cham, November 2020.

ADJ24. Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. Device-oriented group messaging: A formal cryptographic analysis of matrix' core. In *2024 IEEE Symposium on Security and Privacy*, pages 2666–1685. IEEE Computer Society Press, May 2024.

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, Cham, August 2022.

AMT23. Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 396–429. Springer, Cham, August 2023.

ANPPP23. Benedikt Auerbach, Miguel Cueto Noval, Guillermo Pascual-Perez, and Krzysztof Pietrzak. On the cost of post-compromise security in concurrent continuous group-key agreement. Cryptology ePrint Archive, Report 2023/1123, 2023.

BBR18. Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018.

BBR+23. Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

BCC+23a. Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 362–395. Springer, Cham, August 2023.

BCC+23b. Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. Cryptology ePrint Archive, Report 2023/880, 2023.

BCG23. David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with sender keys? Analysis, improvements and security proofs. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 307–341. Springer, Singapore, December 2023.

BCV23. David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1253–1270. USENIX Association, August 2023.

BDG+22. Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 213–243. Springer, Cham, November 2022.

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, Cham, November 2020.

BDRW24. Alexander Bienstock, Yevgeniy Dodis, Paul Rösler, and Daniel Wichs. Interval key-encapsulation mechanism. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part II*, volume 15485 of *LNCS*, pages 361–393. Springer, Singapore, December 2024.

BDT22. Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. Multicast key agreement, revisited. In Steven D. Galbraith, editor, *CT-RSA 2022*, volume 13161 of *LNCS*, pages 1–25. Springer, Cham, March 2022.

Ber06. Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Berlin, Heidelberg, April 2006.

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.

BRV20. Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 621–650. Springer, Cham, December 2020.

BSJ+17. Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650. Springer, Cham, August 2017.

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.

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.

CDV21. Andrea Caforio, F. Betül Durak, and Serge Vaudenay. Beyond security and efficiency: On-demand ratcheting with security awareness. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 649–677. Springer, Cham, May 2021.

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.

CKS08. David Cash, Eike Kiltz, and Victor Shoup. The twin Diffie-Hellman problem and applications. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 127–145. Springer, Berlin, Heidelberg, April 2008.

CLM+18. Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 395–427. Springer, Cham, December 2018.

CR25a. Daniel Collins and Paul Rösler. GURKE: Group unidirectional ratcheted key exchange. In *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, 2025*, Lecture Notes in Computer Science. Springer, 2025.

CR25b. Daniel Collins and Paul Rösler. GURKE: Group unidirectional ratcheted key exchange. Cryptology ePrint Archive, 2025. https://eprint.iacr.org/2025/.

DHK+22. Julien Duman, Dominik Hartmann, Eike Kiltz, Sabrina Kunzweiler, Jonas Lehmann, and Doreen Riepel. Group action key encapsulation and non-interactive key exchange in the QROM. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 36–66. Springer, Cham, December 2022.

DHRR22. Benjamin Dowling, Eduard Hauck, Doreen Riepel, and Paul Rösler. Strongly anonymous ratcheted key exchange. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part III*, volume 13793 of *LNCS*, pages 119–150. Springer, Cham, December 2022.

DV19. F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19*, volume 11689 of *LNCS*, pages 343–362. Springer, Cham, August 2019.

FHKP13. Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 254–271. Springer, Berlin, Heidelberg, February / March 2013.

GdKQ+24. Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. SWOOSH: Efficient lattice-based non-interactive key exchange. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

Grü25. Maxim Grüninger. Proof of concept code and performance evaluation for our updatable broadcast kem. GitHub, 2025. https://github.com/Maximtap/Gurke.

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.

HKP22. Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide MetaData in MLS-like secure group messaging: Simple, modular, and post-quantum. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1399–1412. ACM Press, November 2022.

JMM19. Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Cham, May 2019.

JS18. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Cham, August 2018.

KPPW+21. Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy*, pages 268–284. IEEE Computer Society Press, May 2021.

LGGR23. Julia Len, Esha Ghosh, Paul Grubbs, and Paul Rösler. Interoperability in end-to-end encrypted messaging. Cryptology ePrint Archive, Report 2023/386, 2023.

Mar14. Moxie Marlinspike. Private group messaging. https://signal.org/blog/private-groups/, 2014.

PM16. Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf, 11 2016.

PR18. Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Cham, August 2018.

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, Cham, May 2021.

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 EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 415–429. IEEE, 2018.

RSS23. Paul Rösler, Daniel Slamanig, and Christoph Striecks. Unique-path identity based encryption with applications to strongly secure messaging. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 3–34. Springer, Cham, April 2023.

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.

WPBB23. Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1217–1233. USENIX Association, August 2023.

A Standard Primitives

A.1 Pseudorandom Function (PRF)

Definition 1. *Consider the games* PRF^b *for* $b \in \{0,1\}$ *in Figure 14. We define the advantage of an adversary* $\mathcal{A}$ *against indistinguishability of a function* $\mathrm{F} : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ *as*

$$\mathrm{Adv}^{\mathrm{PRF}}_{\mathrm{F}}(\mathcal{A}) := \left|\Pr[\mathrm{PRF}^0_{\mathrm{F}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathrm{PRF}^1_{\mathrm{F}}(\mathcal{A}) \Rightarrow 1]\right| \ .$$

Game $\mathrm{PRF}^b_{\mathrm{F}}(\mathcal{A})$	**Oracle** $\mathrm{Eval}(d)$
00 $\mathrm{G} \leftarrow_{\$} \mathcal{F}$	04 Require $d \in \mathcal{D}$
01 $k \leftarrow_{\$} \mathcal{K}$	05 If $b = 1$:
02 $b' \leftarrow_{\$} \mathcal{A}(\omega)$	06 Return $\mathrm{G}(d)$
03 Stop with b'	07 Return $\mathrm{F}(k, d)$

Fig. 14. Games PRF^b, $b \in \{0,1\}$, for function $\mathrm{F} : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$, where $\mathcal{F}$ denotes the set of all functions G of the form $\mathrm{G} : \mathcal{D} \to \mathcal{R}$.

A.2 Signature Scheme

Syntax. A signature scheme S consists of three algorithms, S.gen, S.sgn and S.vfy with the following syntax:

- $\mathrm{S.gen} : \emptyset \to_{\$} \mathcal{VK} \times \mathcal{SK}$
- $\mathrm{S.sgn} : \mathcal{SK} \times \mathcal{M} \times \mathcal{R} \to \mathcal{S}$ resp. $\mathrm{S.sgn} : \mathcal{SK} \times \mathcal{M} \to_{\$} \mathcal{S}$
- $\mathrm{S.vfy} : \mathcal{VK} \times \mathcal{M} \times \mathcal{S} \to \{0,1\}$

Definition 2. *We say that a signature scheme* S *is* correct *if for all* $m \in \mathcal{M}$ *and sequences of calls 1.* $(svk, ssk) \leftarrow_{\$} \mathrm{S.gen}$*; 2.* $\sigma \leftarrow_{\$} \mathrm{S.sgn}(ssk, m)$*; it holds that* $\mathrm{S.vfy}(svk, m, \sigma) = 1$.

Definition 3. *Consider the game* SUF *in Figure 15. We define the advantage of an adversary* $\mathcal{A}$ *against the multi-instance and exposable strong unforgeability of a signature scheme* S *as*

$$\mathrm{Adv}^{\mathrm{SUF}}_{\mathrm{S}}(\mathcal{A}) := \Pr[\mathrm{SUF}_{\mathrm{S}}(\mathcal{A}) \Rightarrow 1].$$

A.3 Key Encapsulation Mechanism

A Key Encapsulation Mechanism (KEM) is a tuple of algorithms $\mathrm{K} = (\mathrm{K.gen}, \mathrm{K.enc}, \mathrm{K.dec})$ with encapsulation key space $\mathcal{EK}$, decapsulation key space $\mathcal{DK}$, ciphertext space $\mathcal{C}$, and key space $\mathcal{K}$. KEM K is *correct* if $\Pr[\mathrm{K.dec}(dk, c) = k \mid (ek, dk) \leftarrow_{\$} \mathrm{K.gen}, (k, c) \leftarrow_{\$} \mathrm{K.enc}(ek)] = 1$.

Definition 4. *Consider the game* OW *in Figure 16. We define the advantage of an adversary* $\mathcal{A}$ *against the multi-instance, exposable one-wayness of a KEM under chosen ciphertext attacks* K *as*

$$\mathrm{Adv}^{\mathrm{OW}}_{\mathrm{K}}(\mathcal{A}) := \Pr[\mathrm{OW}_{\mathrm{K}}(\mathcal{A}) \Rightarrow 1] \ .$$

Game SUF($\mathcal{A}$)
00 $n \leftarrow 0; XP \leftarrow \emptyset$
01 Invoke $\mathcal{A}$
02 Stop with 0

Oracle Gen
03 $n \leftarrow n + 1$
04 $sig_n \leftarrow \mathtt{T}$
05 $(svk_n, ssk_n) \leftarrow_{\$}$ S.gen
06 $MS_n[\cdot] \leftarrow \emptyset$
07 Return svk_n

Oracle Expose(i)
08 Require $i \in [n]$
09 $XP \xleftarrow{\cup} \{i\}$
10 Return ssk_i

Oracle Sgn(i, m)
11 Require $i \in [n]$
12 Require $sig_i = \mathtt{T}$
13 $sig_i \leftarrow \mathtt{F}$
14 $\sigma \leftarrow_{\$}$ S.sgn(ssk_i, m)
15 $MS_i[m] \xleftarrow{\cup} \{\sigma\}$
16 Return σ

Oracle Vfy(i, m, σ)
17 Require $i \in [n]$
18 $v \leftarrow_{\$}$ S.vfy(svk_i, m, σ)
19 If $i \notin XP \wedge \sigma \notin MS_i[m]$:
20 If $v = 1$: Stop with 1
21 Return v

Fig. 15. Game SUF for signature scheme S.

Game OW($\mathcal{A}$)
00 $n \leftarrow 0; XP \leftarrow \emptyset$
01 Invoke $\mathcal{A}$
02 Stop with 0

Oracle Gen
03 $n \leftarrow n + 1$
04 $(ek_n, dk_n) \leftarrow_{\$}$ K.gen
05 $CK_n[\cdot] \leftarrow \emptyset$
06 Return ek_n

Oracle Solve(i, c, k)
07 Require $1 \le i \le n$
08 Require $i \notin XP$
09 Require $CK_i[c] \neq \perp$
10 If $k = CK_i[c]$: Stop with 1
11 Return

Oracle Enc(i)
12 Require $1 \le i \le n$
13 $(k, c) \leftarrow_{\$}$ K.enc(ek_i)
14 $CK_i[c] \xleftarrow{\cup} k$
15 Return c

Oracle Dec(i, c)
16 Require $1 \le i \le n$
17 Require $CK_i[c] = \perp$
18 $k \leftarrow$ K.dec(dk_i, c)
19 Return k

Oracle Expose(i)
20 Require $1 \le i \le n$
21 $XP \xleftarrow{\cup} \{i\}$
22 Return dk_i

Fig. 16. Game OW for KEM scheme K.

Construction. Our notion be constructed from a OW-CCA or IND-CCA single-challenge KEM (with an exponentially-sized key space) with loss $n \cdot m$, where n is the number of users and m is the number of queries per user to Solve that the adversary makes (observe this holds even as we allow adaptive secret key exposures).

A.4 Non-Interactive Key Exchange

A Non-Interactive Key Exchange (NIKE) allows pairs of key pairs to compute a common symmetric key.

Syntax. A NIKE is a tuple of algorithms $\mathrm{N} = (\mathrm{N.gen}, \mathrm{N.key})$ with identity public key space $\mathcal{PK}$, samplable secret key space $\mathcal{SK}$, and symmetric key space $\mathcal{K}$:

- $\mathrm{N.gen} : \mathcal{SK} \to \mathcal{PK}$, resp. $\emptyset \to_{\$} \mathcal{PK} \times \mathcal{SK}$ Generates a public key from, resp. with, a secret key.
- $\mathrm{N.key} : \mathcal{SK} \times \mathcal{PK} \to \mathcal{K}$ Outputs a shared key, given a secret key and a secret key.

NIKE N is *correct* if $\Pr[\mathrm{N.key}(sk_1, pk_2) = \mathrm{N.key}(sk_2, pk_1) \mid (pk_1, sk_1) \leftarrow_{\$} \mathrm{N.gen}, (pk_2, sk_2) \leftarrow_{\$} \mathrm{N.gen}] = 1$.

Security. We consider one-way security where the adversary can generate key pairs (Gen), expose and learn corresponding secret keys (Corrupt), derive a session key between two users, one being a public key output by Gen and the other being potentially adversarially chosen, and solve a challenge, i.e., guess a key established between two honest users (Solve).

Definition 5. *Consider the game* OW *in Figure 17. We define the advantage of an adversary $\mathcal{A}$ against the one-wayness of a NIKE* N *as*

$$\mathrm{Adv}^{\mathrm{OW}}_{\mathrm{N}}(\mathcal{A}) := \Pr[\mathrm{OW}_{\mathrm{N}}(\mathcal{A}) \Rightarrow 1] \ .$$

Game $\mathrm{OW}_{\mathrm{N}}(\mathcal{A})$
00 $SK[\cdot] \leftarrow \bot$
01 $CR, KN \leftarrow \emptyset$
02 Invoke $\mathcal{A}$
03 Stop with 0

Oracle Gen
04 $sk \leftarrow_{\$} \mathcal{SK}$
05 $pk \leftarrow \mathrm{N.gen}(sk)$
06 $SK[pk] \leftarrow sk$
07 Return pk

Oracle Corrupt(pk)
08 Require $SK[pk] \neq \bot$
09 $CR \xleftarrow{\cup} pk$
10 Return $SK[pk]$

Oracle Key(pk_1, pk_2)
11 Require $SK[pk_1] \neq \bot$
12 $KN \xleftarrow{\cup} \{\{pk_1, pk_2\}\}$
13 Return $\mathrm{N.key}(SK[pk_1], pk_2)$

Oracle Solve(pk_1, pk_2, k)
14 Require $SK[pk_1] \neq \bot \neq SK[pk_2]$
15 Require $pk_1 \notin CR \wedge pk_2 \notin CR$
16 Require $\{pk_1, pk_2\} \notin KN$
17 If $\mathrm{N.key}(SK[pk_1], pk_2) = k$:
18 Stop with 1
19 Return

Fig. 17. Game OW for NIKE scheme N.

Constructions. Our security notion is a simplification of the 'm-CKS-heavy game' from [FHKP13] (CKS in the sense of [CKS08]) adapted to the one-way setting without identifiers.[16] Our notion is implied by m-CKS-heavy security given exponential $|\mathcal{K}|$. [FHKP13] and subsequent work provide several classically-secure NIKEs. As the security notion used to prove security of the lattice-based NIKE Swoosh [GdKQ+24] is polynomially equivalent to the m-CKS-heavy game (as shown in [FHKP13]), Swoosh is a suitable post-quantum construction. Variants of the post-quantum CSIDH [CLM+18] studied in [DHK+22] have also been formally proven secure.

B Further Details on UB-KEM

B.1 Performance

We provide in [Grü25] a proof of concept implementation of our dynamic UB-KEM construction from Figures 7 and 8. This implementation is written in Python and uses Diffie-Hellman key exchange based on Curve25519 [Ber06] for NIKE. As UB-KEM is the core building block of our GURKE constructions, we measure the running time of the most time-consuming UB-KEM algorithms BK.enc with BK.fin, BK.dec, and BK.fork. The results of this performance analysis are plotted in Figure 18.

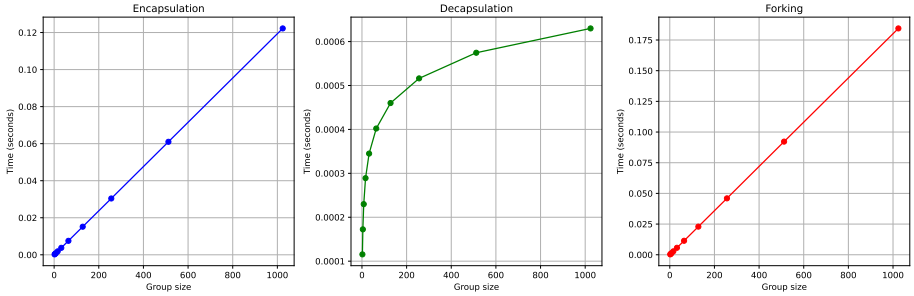


Fig. 18. Performance benchmarks of algorithms BK.enc with BK.fin (left), BK.dec (center), and BK.fork (right) of our Dynamic UB-KEM construction from figures 7 and 8 for groups with 2^{μ} members, where $1 \leq \mu \leq 10$.

We measure the running time of each algorithm for groups with 2^{μ} members, where $1 \leq \mu \leq 10$, since the maximum group sizes in WhatsApp and Signal are 1024 and 1000, respectively. Each measurement is the median of 200 repeated executions on a machine with a 13th Gen Intel Core i5-13 processor.

Although our implementation is written in Python and the code is not optimized for performance, the results demonstrate the practicality of our construction: encapsulations (using algorithms BK.enc and

[16] In this notion, the adversary can register keys both for honest and corrupt users (Reg), corrupt and learn the secret of honest users (Corrupt), derive a session key between an honest and corrupt user (Key), and make a one-way challenge.

BK.fin) take less than 10 milliseconds for typical group sizes (< 100 members). Decapsulations only take less than 1 millisecond for groups of 1000 members. Also the forking algorithm is highly efficient, as it does not even double the running time of the encapsulation. (Note that the forking algorithm resembles the encapsulation, except that the former generates and outputs two updated trees of public keys.)

B.2 Security under Bad Randomness

A shortcoming of our dynamic UB-KEM construction is that a malicious sender, who stores their own used random coins instead of forgetting them, can continuously trace their receivers' secret key material. The reason for this is that, with every encapsulation key update, the sender computes updated public keys *and* secret keys by deriving seeds from the random oracle and, then, deterministically generating the entire updated key pairs; only honest senders ignore the computed secret key components. For subsequent ciphertext, sent by such a sender to their receivers, this is unproblematic as the malicious sender only learns information that is anyway known to themselves. However, when forking the own encapsulation key, such a malicious sender learns secrets of their own receivers as well as the secrets of the forked instance.

In our dynamic GURKE constructions, UB-KEM forking is used to add new senders. As a result, the so-called *double-join* issue[17] arises: consider a malicious sender A, who adds honest sender B by forking their own UB-KEM instance and giving the public encapsulation key fork to B. Afterwards, A is removed by some honest sender. Although malicious sender A was removed, the secrets that belong to B's public key material can be traced by A indefinitely, since our UB-KEM forking algorithm implicitly revealed these secrets to A.

Countermeasures. The literature proposed several ideas and techniques to prevent this issue. The most practical one is to forgo our strong implicit authentication guarantees, which require that the key material of senders and receivers diverges upon active attacks against their communication. Hence, if key updates do not need to depend on the communication between senders and receivers, simpler, efficient update mechanisms, as in UPKE [JMM19], suffice. However, as long as such a strong form of implicit authentication is required and good randomness cannot be assumed, Balli et al. [BRV20] proved that a heavy tool, called Key-Updatable KEM, cannot be avoided. All constructions of Key-Updatable KEM [PR18, BRV20, RSS23], so far, rely on inefficient Hierarchical Identity-Based Encryption. Thus, it remains an open problem to find a sweet spot between resilience against bad randomness, strong (enough) notions of implicit authentication (under state exposures), as well as practically efficient constructions.

C Remaining Syntax Definitions

We list here the syntax of restricted variants of our full GURKE primitive D-MSMR.

Dynamic Multi-Sender Single-Receiver (D-MSSR).

- $\text{init} : \mathbb{N} \to_\$ (\mathcal{ST}_\mathtt{S})^+ \times \mathcal{ST}_\mathtt{R}$
- $\text{snd} : \mathcal{ST}_\mathtt{S} \times \mathcal{AD} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- $\text{add} : \mathcal{ST}_\mathtt{S} \times \mathcal{AD} \times \mathcal{UID} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{ST}_\mathtt{S} \times \mathcal{C}_\mathtt{S} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- $\text{rmv} : \mathcal{ST}_\mathtt{S} \times \mathcal{AD} \times \mathcal{UID} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{C}_\mathtt{S} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- $\text{rcv} : \mathcal{ST}_\mathtt{R} \times \mathcal{AD} \times \mathcal{C}_\mathtt{R} \to \mathcal{ST}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- $\text{proc} : \mathcal{ST}_\mathtt{S} \times \mathcal{AD} \times \mathcal{C}_\mathtt{S} \to \mathcal{ST}_\mathtt{S}$

Dynamic Single-Sender Multi-Receiver (D-SSMR).

- $\text{init} : \mathbb{N} \to_\$ \mathcal{ST}_\mathtt{S} \times (\mathcal{ST}_\mathtt{R})^+$
- $\text{snd} : \mathcal{ST}_\mathtt{S} \times \mathcal{AD} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- $\text{add} : \mathcal{ST}_\mathtt{S} \times \mathcal{AD} \times \mathcal{UID} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{ST}_\mathtt{R} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- $\text{rmv} : \mathcal{ST}_\mathtt{S} \times \mathcal{AD} \times \mathcal{UID} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- $\text{rcv} : \mathcal{ST}_\mathtt{R} \times \mathcal{AD} \times \mathcal{C}_\mathtt{R} \to \mathcal{ST}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$

[17] See https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/.

Static Multi-Sender Multi-Receiver (S-MSMR).

- init : $\mathbb{N} \to_\$ (\mathcal{ST}_\mathtt{S})^+ \times \mathcal{ST}_\mathtt{R}$
- snd : $\mathcal{ST}_\mathtt{S} \times \mathcal{AD} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- rcv : $\mathcal{ST}_\mathtt{R} \times \mathcal{AD} \times \mathcal{C}_\mathtt{R} \to \mathcal{ST}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$

Static Multi-Sender Single-Receiver (S-MSSR).

- init : $\mathbb{N} \to_\$ (\mathcal{ST}_\mathtt{S})^+ \times \mathcal{ST}_\mathtt{R}$
- snd : $\mathcal{ST}_\mathtt{S} \times \mathcal{AD} \to_\$ \mathcal{ST}_\mathtt{S} \times \mathcal{C}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$
- rcv : $\mathcal{ST}_\mathtt{R} \times \mathcal{AD} \times \mathcal{C}_\mathtt{R} \to \mathcal{ST}_\mathtt{R} \times \mathcal{K} \times \mathcal{KID}$

D Deferred Proofs

D.1 Static UB-KEM (Figure 6)

Correctness follows from the correctness of KEM K and the fact that consistent BK.enc/BK.fin and BK.dec calls lead to the same calls to H in BK.fin and BK.dec.

Proof. We proceed to the security proof that is based on game hopping. Note that since the UB-KEM is *static*, the oracles Add, Rmv, Fork and Proc are disabled.

Game $\mathbf{G_1}$. Consider a pair of consistent $(u, c) \leftarrow$ Enc and $(ek, k) \leftarrow \text{Fin}(ad)$ oracle queries (where call $(ek, k) \leftarrow \text{BK.fin}(u, ad)$ is made inside of the Fin call). Suppose, for any i, that that a corresponding call $\text{Dec}(i, c, ad)$ is made (considering only in-order pairs of consistent Enc/Fin queries here). Then consider the call $(dk_i, k') \leftarrow \text{BK.dec}(dk_i, c, ad)$ made in the Dec query. Then, $\mathbf{G_1}$ differs from IND in that all such outputs k' are replaced by the key k output by BK.fin in Fin.

Claim. $\text{Adv}^{\text{ind}}_{\text{BK}}(\mathcal{A}) = \text{Adv}^{\mathbf{G_1}}_{\text{BK}}(\mathcal{A})$.

Proof. This follows from the correctness of KEM K. □

We define an KEM OW adversary $\mathcal{B}$ below for KEM K that we claim perfectly simulates $\mathbf{G_1}$ for some $\mathcal{A}$.

Claim. $\text{Adv}^{\mathbf{G_1}}_{\text{BK}}(\mathcal{A}) \leq \text{Adv}^{\text{ow}}_{\text{K}}(\mathcal{B})$.

Proof. We say that a query $\text{Dec}(i, j, c, ad)$ is *in sync* if $is_{i,j} = \mathtt{T}$ after the query has returned. Otherwise, it is *out of sync*. $\mathcal{B}$ simulates as follows, noting that we describe $\mathcal{B}$'s simulation of H iteratively. We assume for simplicity of exposition that queries that result in "Require S" for some false statement S being executed are handled separately and can be easily identified by $\mathcal{B}$.

- $\mathcal{B}$ simulates call Gen locally by calling its own oracle Gen, outputting ek which is given to $\mathcal{A}$.
- Enc(i): $\mathcal{B}$ simulates using Enc(j) for the corresponding j which outputs c that $\mathcal{B}$ outputs to $\mathcal{A}$.
- Fin(i, ad, ch): $\mathcal{B}$ simulates $(s, k) \leftarrow \text{H}(u, ad)$ by programming random oracle H, namely with unknown input u, known input ad, unknown output k and simulated output s, and a call to Gen. In our simulation of H described below, we ensure that any call the adversary makes with 'correct' $u = (k, c)$ output by BK.enc in the snd call immediately ends the simulation, and thus we do not need to 'program' u here in the future (since the adversary must know k corresponding to c).
- Dec(i, j, c, ad):
  - If this is the first out of sync Dec query for a given sequence of (c, ad) values and a given (i, j), $\mathcal{B}$ uses Dec to simulate BK.dec and otherwise simulates locally, simulating $(s, k) \leftarrow \text{H}(c, k', ad)$ with known inputs and outputs (in particular simulating K.gen locally with s as seed input). Subsequent Dec queries in this (c, ad) sequence are then simulated locally.
  - Otherwise, Dec is in sync, and there must have therefore been an Enc query that output c and a corresponding Fin query. Simulate using the previously computed simulation of H in this Fin query.

– Expose(i, j): If the simulation of secret key j for instance i is local at this point (due to the simulation of Dec above), simply return it to $\mathcal{A}$. Otherwise, it must be that $is_{i,j} = \mathtt{T}$. In this case, first call Expose(j') for the corresponding j, and simulate $dk_{i,j}$ to be returned to $\mathcal{B}$. Then, iteratively program the input/outputs of H not previously set in Fin calls by simulating Dec calls using the output of Expose(j') and additional queries to Expose$(\cdot)$ where needed.
– H(c, k, ad): Call Solve(i, c, k) for each i, ending the simulation if the query succeeds. Otherwise, H(c, k, ad) has not been previously been called by the challenger unless it corresponds to a Fin or Dec call that can be trivially decrypted by the simulator. Thus, $\mathcal{B}$ simulates H locally with known inputs/outputs.

Note that BK.gen(s) calls simulated using Gen are distributed correctly since is correct since $u = (k, c)$ and k is uniform unless $\mathcal{A}$ queries H(c, k, ad), but this case is dealt with since $\mathcal{B}$'s call to Solve would have succeeded. Thus, the simulation is perfect. The run time of $\mathcal{B}$ is $O(q_{\text{Enc}} + q_{\text{Dec}})$ times that of $\mathcal{A}$, where $q_{\text{Enc}}/q_{\text{Dec}}$ is the number of Enc/Dec calls $\mathcal{A}$ makes, since in particular for each H call it makes, $\mathcal{B}$ calls Solve $O(q_{\text{Enc}} + q_{\text{Dec}})$ times, once per Gen call it makes (the simulation is otherwise efficient). □

The theorem follows by combining the results above. □

D.2 Dynamic UB-KEM (Figures 7 & 8)

The proof proceeds in a hybrid argument of three dimensions that follow the dependency of key pairs on other key pairs with which their generation seed was exchanged. At the end of each dimension of the chained hybrid argument, we reprogram the random oracle to break these dependencies. An adversary that still succeeds after this reprogramming step can be reduced to the security of the employed NIKE.

Hybrid Dimension 1: New Leaves (factor $q_{\text{Add}} + q_{\text{Rmv}}$)
We begin our proof with three nested dimension of a hybrid argument: The first dimension iterates chronologically over all queries to oracles Add and Rmv, such that $q_{\text{H1}} = q_{\text{Add}} + q_{\text{Rmv}}$ is the total number of these queries, q_{Add} is the number of queries to oracle Add, and q_{Rmv} is the number of queries to oracle Rmv.

The basic idea of this first dimension is to consider all sources of fresh randomness that establish a post-compromise secure secret at the leaves of the tree.

Hybrid Dimension 2: Leaves to Root (factor $2(n_{init} + q_{\text{Add}})$)
At step $i_{\text{H1}} \in [q_{\text{H1}}]$ of the first dimension, we consider a particular query to oracle Add resp. Rmv. At every such step, the second dimension iterates over all nodes $i_{\text{H2}} \in [q_{\text{H2},i_{\text{H1}}}]$ on the path from the added resp. removed user's leaf to the root of the tree, where $q_{\text{H2},i_{\text{H1}}} \in [n_{init} + q_{\text{Add}}]$ is the respective (possibly degenerated[18]) path length at every step i_{H1}, n_{init} is the initial group size, and q_{Add} is the number of queries to oracle Add. Based on this, the steps in this second dimension correspond to the steps of the for-loops in lines 26-30 in Figure 7 resp. 06-10 in Figure 8. For algorithm BK.rmv, we add an extra step at the leaf level due to the additional NIKE in line 03 in Figure 8.

The basic idea of this second dimension is to extend the primary source of secrecy established by fresh randomness at each leaf along its path to the root.

Hybrid Dimension 3: Node Updates (factor $q_{\text{Fin}} + 2q_{\text{Fork}}$)
At every step $(i_{\text{H1}}, i_{\text{H2}}) \in [q_{\text{H1}}] \times [q_{\text{H2},i_{\text{H1}}}]$ of the second dimension, we consider a particular node in the tree of secrets. Based on that, we start a third dimension that iterates over all queries to oracles Fin and Fork between the Add resp. Rmv oracle query at which that node's secret was derived from a leaf's secret and a later query to oracle Add resp. Rmv that replaces that node's secret. For example, the root's secret is replaced by every query to oracles Add and Rmv, whereas a leaf's secret is only replaced when the corresponding user is removed again. Thus, the third dimension iterates over $i_{\text{H3}} \in [q_{\text{H3},i_{\text{H1}},i_{\text{H2}}}]$ queries to oracles Fin and Fork, where $q_{\text{H3},i_{\text{H1}},i_{\text{H2}}} \in [q_{\text{Fin}} + 2q_{\text{Fork}}]$ is the number of subsequent queries to oracles Fin and Fork that refresh a particular node's secret before it is regenerated by a later query to oracle Add resp. Rmv. Note that a fork consists of two steps: one for the original instance and one for the forked instance.

[18] Using balancing mechanisms, the tree depth is reduced from a linear to a logarithmic factor. Studying tree balancing mechanisms is outside the scope of our cryptographic work, but our solutions are compatible with them.

The basic idea of this dimension is to extend and maintain the secrecy of each node in the tree along their updates at each invocation of BK.fin and each invocation of BK.fork.

Game 1: Split Seeds (reduced to OW_N)

Each step $(i_{\mathrm{H1}}, i_{\mathrm{H2}}, i_{\mathrm{H3}}) \in [q_{\mathrm{H1}}] \times [q_{\mathrm{H2},i_{\mathrm{H1}}}] \times [q_{\mathrm{H3},i_{\mathrm{H1}},i_{\mathrm{H2}}}]$ of the third dimension considers a node's key pair (pk, sk) that is derived via a seed from the random oracle on input of key k. This key k is established by an NIKE with this node's prior version public key pk' or with one of its child node's public key pk', where sk' is the corresponding secret key, respectively (see lines 16-17 and 28-29 in Figure 7, and lines 03-04, 08-09 in Figure 8). In this game hop, we abort the game if the adversary (explicitly) queries the random oracle for the established key k unless the secret key sk' was corrupted via oracle Expose before the random oracle query was issued.

Concretely, for a considered key pair generated in line 18/30 in Figure 7 or line 05/10 in Figure 8 that is derived via the random oracle on inputs (k, c, ad) or (k, pk^*) in line 17/29 in Figure 7 or line 04/09 in Figure 8 from the NIKE with key pair (pk', sk') in line 16/28 in Figure 7 or line 03/08 in Figure 8, we abort if the random oracle is queried by the adversary with inputs (k, ad, c) or (k, pk^*) unless sk' was corrupted before that query.

To detect this modification, the adversary has to query the random oracle on the respective established key k. We reduce this event to the OW_N security of the used NIKE scheme N (see Figure 17).

Reduction:

Intuitively, the reduction for Game 1 in step $(i_{\mathrm{H1}}, i_{\mathrm{H2}}, i_{\mathrm{H3}})$ simulates all key generations and key exchanges for key pairs generated until the current step by querying oracles of the underlying game OW_N. A successful distinguisher detecting the modification introduced in the current step then breaks this underlying game by providing the target exchanged NIKE key as input to the random oracle.

Concretely, the reduction simulates oracle queries in game $\mathrm{IND}_{\mathrm{BK}}$ as follows:

- All invocations of algorithm N.gen in algorithm BK.gen during queries to oracle Gen of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Gen of game OW_N.
- All invocations of algorithm N.gen in algorithm BK.enc during queries to oracle Enc of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Gen of game OW_N.
- All invocations of algorithm N.gen in algorithm BK.fin during queries to oracle Fin of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Gen of game OW_N. If both input keys of algorithm N.key are uncorrupted, this algorithm is only simulated implicitly via the random oracle (see below). Otherwise, N.key is simulated via oracle Key of game OW_N.
- All invocations of algorithm N.key in algorithm BK.dec during queries to oracle Dec of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Key of game OW_N for public keys that were not sent in ciphertexts by queries to oracle Enc with the corresponding node's public key pk.
  - The decryption of such ciphertexts that were synchronously output by oracle Enc is silently omitted until the corresponding decapsulation key is corrupted; however, as soon as this decapsulation key is corrupted, the random oracle is programmed (see below).
- All invocations of algorithm N.gen in algorithm BK.add during queries to oracle Add of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Gen of game OW_N. If both input keys of algorithm N.key are uncorrupted, this algorithm is only simulated implicitly via the random oracle (see below). Otherwise, N.key is simulated via oracle Key of game OW_N.
- All invocations of algorithm N.gen in algorithm BK.rmv during queries to oracle Rmv of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Gen of game OW_N. If both input keys of algorithm N.key are uncorrupted, this algorithm is only simulated implicitly via the random oracle (see below). Otherwise, N.key is simulated via oracle Key of game OW_N.
- All invocations of algorithm N.gen in algorithm BK.fork during queries to oracle Fork of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Gen of game OW_N. If both input keys of algorithm N.key are uncorrupted, this algorithm is only simulated implicitly via the random oracle (see below). Otherwise, N.key is simulated via oracle Key of game OW_N.
- All invocations of algorithm N.key in algorithm BK.proc during queries to oracle Proc of game $\mathrm{IND}_{\mathrm{BK}}$ are simulated via oracle Key of game $\mathrm{OW}_{\mathrm{AK}}$ for public key in ciphertexts that were not sent by queries to oracles Add, Rmv, or Fork with the corresponding public key pk.
  - The decryption of such ciphertexts that were synchronously output by oracles Add, Rmv, and Fork is silently omitted until the corresponding decapsulation key is corrupted; however, as soon as this decapsulation key is corrupted, the random oracle is programmed (see below).

– Exposing keys *sk* due to queries to oracle Expose of game IND_{BK} is simulated via queries to oracle Corrupt of game OW_{N}.
  – Upon such an exposure, the random oracle is programmed with outputs for queries that do not abort the game (anymore). In particular, random oracle queries with input keys exchanged with these corrupted secret keys are programmed with the matching output. This output can be the seed (i.e., secret key) for another generation of a node's NIKE key pair or the seed for another ephemeral NIKE key pair generation, of which the public key is sent as a ciphertext. Thus, programming the random oracle upon an exposure is a nested, iterative process. In either case, embedding (further) seeds of secret keys uses oracle Corrupt of game OW_{N}.

– Every adversarial random oracle query is checked via oracle Solve of game OW_{N} to observe whether its input matches the key of the current or a prior uncorrupted NIKE.

Based on this description, we observe: Simulation of oracles Enc, Fin, Add, Rmv, and Fork is straight forward; their operations are simply executed by the underlying game OW_{N}.

Also oracles Dec and Proc are simulated by either using the underlying game OW_{N} or by relying on the matching simulation of the sender (via oracles Enc, Fin, Add, Fork, and Rmv).

Exposures are simulated using the corruption oracle of the underlying game OW_{N}. Furthermore, upon exposures, the reduction programs random oracle outputs for queries that do not abort the game (anymore). Based on this, the reduction perfectly simulates exposures and their effect on the random oracle behavior.

The random oracle behavior is only changed once per hybrid step for the query that seeds the current key pair generation. Based on the hybrid argument, all queries to seeds of prior generated key pairs are either blocked, or, if the secret keys for the NIKE input key are corrupted, the random oracle is perfectly simulated by embedding the seeds of NIKE key generations in the outputs of the random oracle. Random oracle queries that seed key generations of later hybrid steps remain unchanged.

We finally observe that by checking each adversarial random oracle query, the reduction extracts the solution of a successful distinguisher between Game 0 and Game 1 in each hybrid step. This extracted solution immediately wins the underlying game OW_{N} such that the reduction is successful.

Summary:
The modification of Game 1 in each hybrid step is reduced to the OW_{N} security of the used NIKE N. After all hybrid steps, the game aborts on all random oracle queries that output a (valid) challenge key for oracle Fin of game IND_{BK}. Thus, the adversary has no advantage in winning the game such that

$$\text{Adv}^{\text{ind}}_{\text{BK}}(\mathcal{A}) \leq q_{\text{Gen}} \cdot (q_{\text{Add}} + q_{\text{Rmv}}) \cdot 2(n_{init} + q_{\text{Add}}) \cdot (q_{\text{Fin}} + 2q_{\text{Fork}}) \cdot \text{Adv}^{\text{ow}}_{\text{N}}(\mathcal{B})$$

D.3 S-MSMR (Figure 9)

The correctness of our S-MSMR construction follows from the correctness of the underlying building blocks, namely the UB-KEM and signature scheme, from which we can conclude that if a sender is synchronised with a receiver, they will derive the same sequence of keys, and output the same *kid*'s by construction of the scheme.

Theorem 5. *For any adversary* IND $\mathcal{A}$ *there exists adversaries* $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *s.t.*

$$\text{Adv}^{\text{ind}}_{\text{R}}(\mathcal{A}) \leq \text{Adv}^{\text{suf}}_{\text{S}}(\mathcal{B}_1) + \text{Adv}^{\text{ind}}_{\text{BK}}(\mathcal{B}_2) \ .$$

Proof. We proceed by game hopping.

Game $\mathbf{G_1}$. This differs from IND in that, for all queries made to $\text{Rcv}(i, ad, c)$ that finish with $j \in [n_{\text{S}}]$ such that $IS[j, i] = \texttt{T}$ where the call $(st^i_{\text{R}}, k, kid) \leftarrow \text{rcv}(st^i_{\text{R}}, ad, c)$ inside with $kid = j$ outputs $k \neq \bot$, 1) the output (k, id) of the BK.dec call is replaced with the output (k, id) from the corresponding $\text{Snd}(j, ad, \cdot)$ call, and 2) the S.vfy call is removed.

Claim. $\text{Adv}^{\text{ind}}_{\text{R}}(\mathcal{A}) = \text{Adv}^{\mathbf{G_1}}_{\text{R}}(\mathcal{A})$.

Proof. This follows from the correctness of BK and S. □

We next construct a SUF adversary $\mathcal{B}$.

Game $\mathbf{G_2}$. This differs from $\mathbf{G_1}$ in that it aborts if at any point in execution, $\exists(i,j) \in [n_{\mathtt{R}}] \times [n_{\mathtt{S}}]$ such that $IS[j,i] = \mathtt{F}$.

Claim. $\mathrm{Adv}^{\mathbf{G_1}}_{\mathrm{R}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathbf{G_2}}_{\mathrm{R}}(\mathcal{A}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathrm{S}}(\mathcal{B})$.

Proof. Observe that $IS[j,i]$, for some j,i, is set to $\mathtt{F}$ during a query $\mathrm{Rcv}(i,ad,c)$ given that 1) $IS[j,i] = \mathtt{T}$; 2) (ad,c) is inconsistent with that from the corresponding Snd call, if it exists; 3) a key $k \neq \bot$ is output by rcv; and 4) at the time the Rcv query was made, $XP_{\mathtt{S}}[j,i].\mathrm{first}() \neq (\mathtt{T},\cdot)$. In particular, this implies that a $\mathbf{G_1}$ adversary was able to successfully query Rcv without having exposed a sender (captured by variable $XP_{\mathtt{S}}$), and, by construction, mount a signature forgery. With this observation in mind, we construct SUF adversary $\mathcal{B}$ that simulates for $\mathbf{G_1}/\mathbf{G_2}$ adversary $\mathcal{A}$ and wins if and only if such a forgery is made. $\mathcal{B}$ simulates as follows:

- $\mathcal{B}$ initially simulates locally the challenger's code and receives $(\omega, n_{\mathtt{S}}, n_{\mathtt{R}})$ from $\mathcal{A}$. $\mathcal{B}$ calls Gen $n_{\mathtt{S}}$ times to obtain $(svk_i)_{i\in[n_{\mathtt{S}}]}$, embeds svk_i in state $st^i_{\mathtt{S}}$ for $i \in [n_{\mathtt{S}}]$, sets $ssk \leftarrow \bot$ in each $st^i_{\mathtt{S}}$ and otherwise simulates locally and gives ω to $\mathcal{A}$.
- $\mathrm{Snd}(i,ad,ch)$: $\mathcal{B}$ simulates the S.gen call in snd by calling its oracle Gen to obtain key svk_j for some index j, simulates the S.sgn call via $\mathrm{Sgn}(j',m)$ with $m = (t,ad,c',svk')$, where j' corresponds to the previous key associated with sender i (sampled either in init or the previous snd call), and otherwise simulates locally. Observe that only a single Sgn query is made per Gen call.
- $\mathrm{Expose}_{\mathtt{S}}(i)$: $\mathcal{B}$ queries $\mathrm{Expose}(j)$ with index j that corresponds to the SUF secret key currently stored in $st^i_{\mathtt{S}}$ to obtain ssk_j (which it replaces in $st^i_{\mathtt{S}}$), and returns the resulting state to $\mathcal{A}$.
- $\mathrm{Rcv}(i,ad,c)$: $\mathcal{B}$ simulates the rcv call locally except it simulates signature verification using oracle Vfy. It follows that verification succeeds and $\mathcal{B}$ wins the game if Rcv is called on diverging input plus the corresponding signature secret key is not exposed (at which point the simulation ends).
- $\mathrm{Expose}_{\mathtt{R}}(i)$: These calls are simulated locally.
- If $\mathcal{A}$ returns bit b', $\mathcal{B}$ aborts.

Let E be the event that $\mathcal{B}$ wins. Observe $\mathbf{G_1}$ and $\mathbf{G_2}$ are identical given $\neg E$, and $\mathcal{B}$ simulates perfectly in this case. The claim then follows from standard bad event analysis. □

Finally, we construct an IND adversary $\mathcal{B}$ playing with BK who directly simulates for $\mathbf{G_2}$ adversary $\mathcal{A}$.

Claim. $\mathrm{Adv}^{\mathbf{G_2}}_{\mathrm{R}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{ind}}_{\mathrm{BK}}(\mathcal{B})$.

Proof. $\mathcal{B}$ is defined as follows.

- When $\mathcal{A}$ first outputs $(\omega', n_{\mathtt{S}}, n_{\mathtt{R}})$ to its challenger (i.e., $\mathcal{B}$), $\mathcal{B}$ calls $\mathrm{Gen}(n_{\mathtt{R}})$ $n_{\mathtt{S}}$ times, and simulates $\mathrm{init}(n_{\mathtt{S}}, n_{\mathtt{R}})$ locally except that it uses the keys ek_i from Gen. Namely, $\mathcal{B}$ embeds ek_i into the i-th sender's state, and marks each dk_j in $S_j[i]$ as blank (simulated below as needed), and outputs ω to $\mathcal{A}$.
- $\mathrm{Snd}(i,ad,ch)$: $\mathcal{B}$ simulates locally, except it replaces the call $(u,c') \leftarrow_{\$} \mathrm{BK.enc}(ek)$ in snd with the output of a call to $\mathrm{Enc}(i)$, and the call $(ek,(k,id)) \leftarrow_{\$} \mathrm{BK.fin}(u,(ad,c))$ with the output of a call $(ek,(k,id)) \leftarrow \mathrm{Fin}(i,ad,ch)$ (recall u is only used in BK.fin), and ultimately outputs $(ek,k,(id,i,n_{\mathtt{S}},n_{\mathtt{R}}))$ to $\mathcal{A}$.
- $\mathrm{Rcv}(i,ad,c)$: By construction of $\mathbf{G_2}$, variable $IS[j,i]$ for any i,j will never be set to $\mathtt{F}$. Therefore we only need to simulate in sync calls or injections (impersonation). Parsing $(j,c',svk',\sigma) \leftarrow c$, $\mathcal{B}$ simulates locally, except that in the rcv call, it replaces the call $(dk,(k,id)) \leftarrow \mathrm{BK.dec}(dk,(ad,c),c')$ with the call $(k,id) \leftarrow \mathrm{Dec}(j,i,c,ad)$.

Now, consider a call $\mathrm{Rcv}(i,ad,c)$ that is made that results in $IS[j,i]$ being set to $\mathtt{imp}$. We argue that this must result in the call $\mathrm{Dec}(j,i,c,ad)$ setting $is_{j,i} \leftarrow \mathtt{F}$. Given $IS[j,i]$ is set to $\mathtt{imp}$, we must have $(ad,c,kid) \neq adc_{\mathtt{R}}[j,i].\mathrm{first}()$, where the right hand side refers to the last sent (ad',c',kid'), if it exists. If it does not exist, then we are done. Otherwise, since (ad,c) is input as associated data to BK.dec, any divergence in (ad,c) in Rcv will result in divergent input to Dec. Then if (ad,c) is fixed, so too will kid by correctness of BK and construction of Figure 9. We thus continue:

- $\mathrm{Expose}_{\mathtt{S}}(i)$: $\mathcal{B}$ simulates locally.

– $\text{Expose}_{\mathtt{R}}(i)$: Since Rcv calls simulated with Dec are both either in sync or out of sync at the same time as argued above, and the conditions for state exposure are analogous between the two games, $\mathcal{B}$ can safely simulate using $\text{Expose}(j, i)$ to simulate dk_j in $S_j[i]$; $\mathcal{B}$ otherwise simulates locally.
– When $\mathcal{A}$ outputs b', $\mathcal{B}$ outputs b' to its challenger.

The simulation is perfect and the result follows. □

The theorem follows by combining the results above. □

D.4 D-MSMR (Figures 12 and 13)

Correctness. Correctness is more delicate than in the static case, and especially due to concurrent operations from senders. We argue that our construction is correct, implicitly assuming the correctness of the underlying building blocks as in the static case. Note that receivers keep track of a different view of sets $mem_{\mathtt{S}}$ and $mem_{\mathtt{R}}$ for each sender, and assuming no group changes have taken place that by construction, senders and receivers are synchronised in their *mem* sets and therefore *kid* values derived.

Assume a static set of senders. Then when a sender i calls $\text{add}(st, ad, \mathtt{R}, uid)$, the newly added sender's view will be consistent with the sender's. Further, i communicates via $c_{\mathtt{S}}$ this information to all other senders $j \neq i$. Concurrent adds by different senders may be processed by different senders in different orders, but when a sender j next encapsulates a key, it communicates its view via cq, so receivers keep track of this evolution of updates for each sender and can therefore agree on *kid*. Observe also that when a receiver is removed from the perspective of a given sender, they are removed from their $mem_{\mathtt{R}}$ set at the same time as from BK. By induction on the add/rmv operations all (active) receivers will have the same view as a given sender. Assume now a static set of receivers. Then when a sender i calls $\text{add}(st, ad, \mathtt{S}, uid)$, this new sender is communicated to all receivers via cq (so receivers stay synchronised with i), and as before to other senders (who stay synchronised with the receivers as above); the same holds for removal operations.

The above argument also holds if neither the set of senders nor receivers is fixed. If sender i adds a new sender s and sender j adds a new receiver r concurrently, for example, then although in our construction sender s is not made aware of receiver r, that this is reflected in the membership sets of all receivers corresponding to sender s.

Theorem 6. *Consider D-MSMR* R *in Figures 12 and 13. For any adversary* IND $\mathcal{A}$ *there exists adversaries* $\mathcal{B}_1$, $\mathcal{B}_2$ *and* $\mathcal{B}_3$ *s.t.*

$$\text{Adv}^{\text{ind}}_{\text{R}}(\mathcal{A}) \leq \text{Adv}^{\text{suf}}_{\text{S}}(\mathcal{B}_1) + \text{Adv}^{\text{ow}}_{\text{K}}(\mathcal{B}_2) + \text{Adv}^{\text{ind}}_{\text{BK}}(\mathcal{B}_3) \ .$$

Game $\mathbf{G_1}$. This differs from IND in that, for all queries made to $\text{Rcv}(i, ad, c)$ that finish with $j \in [n_{\mathtt{S}}]$ such that $IS[j, i] = \mathtt{T}$ where the call $(st^i_{\mathtt{R}}, k, kid) \leftarrow \text{rcv}(st^i_{\mathtt{R}}, ad, c)$ inside with $kid = j$ outputs $k \neq \bot$, 1) the output (k, id, tr) of the BK.dec call is replaced with the output (k, id, tr) from the corresponding BK.fin call made in a previous $\text{Snd}(j, ad, \cdot)$, $\text{Add}(j, ad, \cdot)$ or $\text{Rmv}(j, ad, \cdot)$ call; 2) the S.vfy call is removed; and 3) the call $k \leftarrow_{\$} \text{K.dec}(dk, c_1)$, if it exists, is replaced by the output of K.enc in the corresponding Add call.

Claim. $\text{Adv}^{\text{IND}}_{\text{R}}(\mathcal{A}) = \text{Adv}^{\mathbf{G_1}}_{\text{R}}(\mathcal{A})$.

Proof. This follows from the correctness of BK, S and K. □

Game $\mathbf{G_2}$. This differs from $\mathbf{G_1}$ in that it aborts if at any point in execution, $\exists (i, j) \in [n_{\mathtt{R}}] \times [n_{\mathtt{S}}]$ such that $IS[j, i] = \mathtt{F}$.

Claim. $\text{Adv}^{\mathbf{G_1}}_{\text{R}}(\mathcal{A}) \leq \text{Adv}^{\mathbf{G_2}}_{\text{R}}(\mathcal{A}) + \text{Adv}^{\text{SUF}}_{\text{S}}(\mathcal{B})$.

Proof. The proof largely follows the template of the corresponding game hop in the proof of Theorem 5. Thus, we mainly describe the differences in the proof here. In D-MSMR, S.gen queries are now made in snd, add and rmv calls, for which $\mathcal{B}$, simulating for $\mathcal{A}$, embeds signature keys into using Gen. In the case of the signature key sampled in encaps, $\mathcal{B}$ simulates signing with the previous signature key using Sgn. The simulation strategy still works even though two signature key pairs are sampled in add; here for the first key pair, $\mathcal{B}$ simulates Sgn queries for the added user *uid* rather than the sender as above. Noting that successful signature forgeries are still only allowed for R sender states that have not been exposed, the claim follows. □

Game $\mathbf{G_3}$. This differs from $\mathbf{G_2}$ in that the game aborts if the adversary ever queries random oracle $\mathrm{H}(k, c_1)$ with input (k, c_1), where $(k, c_1) \leftarrow_\$ \mathrm{K.enc}$ was executed in some encaps or add call made by the challenger, at which point the adversary had not exposed the corresponding newly added receiver's state or a state that allows it to derive it.

Claim. $\mathrm{Adv}_{\mathrm{R}}^{\mathbf{G_2}}(\mathcal{A}) \leq \mathrm{Adv}_{\mathrm{R}}^{\mathbf{G_3}}(\mathcal{A}) + \mathrm{Adv}_{\mathrm{K}}^{\mathrm{ow}}(\mathcal{B})$.

Proof. Let $\mathcal{B}$ be an OW adversary for KEM K that simulates $\mathbf{G_2}$ and $\mathbf{G_3}$. For a given b, $\mathcal{B}$ simulates as follows:

- Initialization: $\mathcal{B}$ simulates locally.
- Snd and Rmv: $\mathcal{B}$ replaces the K.enc call in enq-ops, if it exists, with a call to $\mathrm{Enc}(i)$ for the corresponding index i, and programs the output of $H(k, c_1)$ for unknown k. For each previous call $H(k, c_1)$, $\mathcal{B}$ calls $\mathrm{Solve}(i, k, c_1)$.
- $\mathrm{Add}(i, ad, t, ch)$: For $t = \mathtt{S}$, the simulation is as in the case of Snd/Rmv. For $t = \mathtt{R}$, $\mathcal{B}$ additionally embeds the output of $(|mem_{\mathtt{S}}| - 1)$ calls to Gen in its simulation.
- $\mathrm{Expose}_{\mathtt{S}}(i)$ and $\mathrm{Proc}(j, i)$: These oracles are simulated locally (for $\mathrm{Expose}_{\mathtt{S}}$, some encapsulation keys output by Gen calls may be embedded here).
- $\mathrm{Expose}_{\mathtt{R}}(i)$: $\mathcal{B}$ calls $\mathrm{Expose}(i')$ for all indices i' corresponding to KEM decapsulation keys in receiver i's state. For encapsulation keys i' for which a corresponding Enc call has been made, $\mathcal{B}$ decapsulates the corresponding ciphertext c_1, outputting k, and programs the corresponding input to the H call made in Enc with first argument k.
- $\mathrm{Rcv}(i, ad, c)$: For in sync calls and more generally calls where K.dec is called with dk and the matching ciphertext c_1, $\mathcal{B}$ simulates using the programmed $H(k, c_1)$ for unknown k. Otherwise (when c_1 is different), $\mathcal{B}$ simulates K.dec via $\mathrm{Dec}(i', c_1)$ for relevant i'.
- $H(k, c_1)$: For all i such that c_1 is a challenge ciphertext, call $\mathrm{Solve}(i, k, c_1)$.[19] Otherwise, lazily sample.

Now, if $\mathcal{A}$ ever calls $H(k, c_1)$ for a challengeable c_1 and the correct key k (event E), by construction, $\mathcal{B}$ will make a corresponding Solve call and win. Thus, $\mathcal{B}$ wins at least as often as E. Moreover, given $\neg E$, the simulation is perfect. At most one Enc call is made for each instance of KEM K, and (very pessimistically) at most q_{Enc} calls are made each query to Solve. The claim follows. □

Claim. $\mathrm{Adv}_{\mathrm{R}}^{\mathbf{G_3}}(\mathcal{A}) = \mathrm{Adv}_{\mathrm{BK}}^{\mathrm{ind}}(\mathcal{B})$

Proof. Note that by the previous hop, all ciphertexts c_2 that $\mathcal{A}$ cannot trivially derive are uniform from the perspective of $\mathcal{A}$. Then, the proof follows the same high-level strategy as the corresponding hop in Theorem 5. Let $\mathcal{B}$ be a IND adversary with respect to broadcast KEM BK that simulates for $\mathbf{G_3}$ adversary $\mathcal{A}$.

As for Theorem 5, $\mathcal{B}$ embeds UB-KEM keys ek into each sender's state. $\mathcal{B}$ simulates similarly to before for Snd queries, replacing BK.enc and BK.fin calls with Enc and Fin queries, as well as any BK.add or BK.rmv calls in enq-ops with Add and Rmv queries, respectively; $\mathcal{B}$ proceed similarly for Add and Rmv calls. As before, whenever a successful impersonation query Rcv is made, it must be that a corresponding UB-KEM variable variable $is_{i,j}$ is set to F. Then, for similar reasons to the argument for Theorem 5, $\mathcal{B}$ can simulate $\mathrm{Expose}_{\mathtt{S}}$ locally and $\mathrm{Expose}_{\mathtt{R}}$ using its oracle Expose. Finally, $\mathcal{B}$ outputs the same bit as $\mathcal{A}$ to its challenger. □

E Lower Bounds

Our GURKE constructions offer high practical efficiency. We here discuss how far the efficiency is even partially optimal.

Optimality of S-SSMR. Sender and receiver states as well as ciphertexts in the S-SSMR case are of constant size. This is clearly optimal as S-SSMR GURKE generically implies an IND-1-CCA secure KEM and a 1-SUF-CMA secure signature scheme.

[19] If the encapsulation key ek is also hashed or if there are no collisions on c_1, then this will be constant.

Optimality of S-MSSR. In Appendix E.1 below, we prove that the state size for a receiver in S-MSSR must be *linear* in the number of senders n_{S}. To prove this, we show that S-MSSR can be used to build puncturable encryption with a tag space of cardinality n_{S}, where the secret key consists of the receiver's key and, for technical reasons (to derandomize the puncture operation), a PRF key also. One can then apply a recent result from Barooti et al. [BCC+23b] who showed a linear lower bound on the secret size in the number of punctures.

Corollary 5. *Let* R *be a S-MSSR scheme associated with receiver state space* $\mathcal{ST}_B = \{0,1\}^s$ *that supports up to* n *senders. Consider randomness space* $\mathcal{R} = \{0,1\}^r$ *and integer* $n_p \leq n$*. Fix the security parameter and let* $\mathcal{A}$ *be an efficient adversary. Suppose that*

$$\mathrm{Adv}^{\mathrm{IND}}_{\mathrm{R}}(\mathcal{A}) \leq 2^{-\gamma}$$

where γ *is maximal. Then*

$$s \geq (\log n - \log n_p)n_p - \frac{2}{1 - \log(n_p)2^{-\gamma}} - r \ .$$

Setting $n_p = n/c$ *for constant* c*, we have*

$$s \geq (\log n - \log n/c)n/c - \frac{2}{1 - \log(n/c) \cdot 2^{-\gamma}} - r$$

which is $\approx O(n - r)$ *for* $n_p \ll 2^{\gamma}$*.*

Note that the r term arises due to the PRF key we use in Figure 20 due to the fact that punc is defined as a deterministic function in [BCC+23b] and we need to derandomise snd to fit their syntax. If a bound is shown to hold for randomized punc, then our result is immediately tightened.

Alternatively, one could essentially use the same encoder/decoder as [BCC+23b] for their puncturable encryption bound to remove the loss in 'tightness' of our lower bound. To get a bound that is linear in the security parameter, it seems likely that one could adapt the encoder/decoders in the main body of their paper.

Our compiler at a high level resembles that of Durak and Vaudenay [DV19]. Namely, they show that their notion of two-party ratcheting implies public-key cryptography by constructing IND-CPA KEM from it in a black-box fashion.

Optimality of S-MSMR. Since the S-MSMR case of GURKE implies the S-MSSR case, the former offers optimal sender state size, ciphertext size, receiver state size based on the results from the last two paragraphs.

Optimality of Dynamic Sender Constructions. Since the dynamic sender cases of GURKE imply the S-MSSR case, the lower bound on receiver state sizes are equally implied.

E.1 S-SSMR Lower Bound

In this subsection, we prove that in static multi-sender single-receiver GURKE, i.e. S-MSSR, the state size of the receiver is *linear* in the number of senders. This of course implies the same if the group of senders is dynamic and/or there are also multiple receivers. To this end, we show that S-MSSR implies *public-key puncturable encryption*, which was shown to have a linear size secret key lower bound in [BCC+23a]. To this end, we first recall the relevant definitions of puncturable encryption (PE), before constructing a PE scheme that we prove is secure, from which the lower bound on S-MSSR directly follows.

Puncturable Encryption (PE)

Syntax. A PE scheme PE consists of four algorithms, gen, enc, punc and dec with the following syntax:

- gen : $\emptyset \to_{\$} \mathcal{PK} \times \mathcal{SK}$ Generates a public key and a secret key
- enc : $\mathcal{PK} \times \mathcal{M} \times \mathcal{T} \times \mathcal{R} \to \mathcal{C}$ resp. enc : $\mathcal{PK} \times \mathcal{M} \times \mathcal{T} \to_{\$} \mathcal{C}$ Encrypts a message with respect to a tag
- punc : $\mathcal{PK} \times \mathcal{SK} \times \mathcal{T} \to \mathcal{SK}$ Punctures the secret key on a tag
- dec : $\mathcal{SK} \times \mathcal{C} \times \mathcal{T} \to \mathcal{M}$ Decrypts a ciphertext encrypted with respect to a tag

Correctness. We state our required correctness notion below.

Definition 6. *We say that a PE scheme* PE *is* correct *if, for all* n, $m \in \mathcal{M}$, $(t_1, \ldots, t_n) \in \mathcal{T}$, $t^* \in \mathcal{T} \setminus \{t_1, \ldots, t_n\}$, *the sequence 1.* $(pk, sk_0) \leftarrow_\$ \text{gen}$*; 2. for all* $i \in [n]$, $sk_i \leftarrow \text{punc}(pk, sk_{i-1}, t_i)$*; 3.* $c \leftarrow_\$ \text{enc}(pk, m, t^*)$*; is such that* $\text{dec}(sk_n, c, t^*) = m$.

Security. Our result requires a OW-CPA game for puncturable encryption which captures Forward Security. The game is provided in Figure 19.

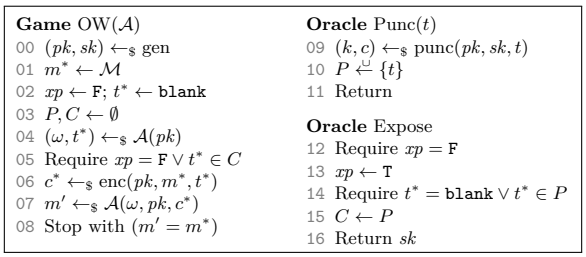


Fig. 19. Game OW for PE scheme PE.

Definition 7. *Consider the game* OW *in Figure 19. We define the advantage of an adversary* $\mathcal{A}$ *against the one-wayness of a PE* PE *as*

$$\text{Adv}^{\text{OW}}_{\text{PE}}(\mathcal{A}) := \Pr[\text{OW}_{\text{PE}}(\mathcal{A}) \Rightarrow 1] \ .$$

Construction. Our construction is provided in Figure 20. For tag set $[n]$, key generation calls $\text{init}(n, 1)$, sets the public key to the n states st^i_{S}, $i \in [n]$ output. The secret key consists of the receiver's state (the target of our lower bound) and a PRF key. Encryption with respect to the i-th tag is simply performed by calling snd with the state of the i-th sender, after which the message and the output key are XORed; decryption follows analogously. Puncturing on tag $t \in [n]$ comprises of sending via the t-th sender's state, then receiving the ciphertext and updating the receiver's state. Since punc is deterministic, we derandomize snd with a key rk that is ratcheted forward after punc invocation. Intuitively, the Forward Security of R means that after a message from sender t has been received, further keys encapsulated under the original state of t can no longer be decapsulated.

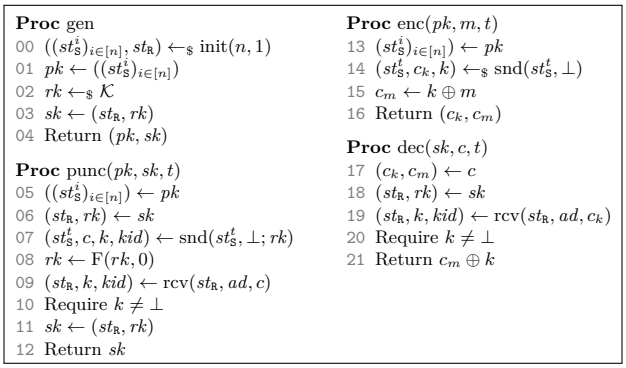


Fig. 20. Construction of PE scheme PE with tag space $\mathcal{T} = [n]$ from an IND S-MSSR R and PRF $\text{F} : \mathcal{R} \times \{0\} \to \mathcal{R}$.

Correctness and Security Proofs. We assume for simplicity that the length of all elements in PE message space $\mathcal{M}$ and R key space $\mathcal{K}$ are the same (it is easy to see that our construction/proofs can be generalised). We first prove correctness.

Claim. PE (Figure 20) is a correct PE scheme.

Proof. Firstly, since the correctness of R holds with probability 1, derandomizing snd in punc with rk, regardless of the distribution of rk, does not affect correctness. By the correctness of R, the output of the rcv call made by the correctness definition's call $\mathrm{dec}(sk_n, c, t^*)$ is k_m, in particular since punc must not have previously been called on tag t^*. Finally, by construction of SE, the output of SE.dec by the correctness challenger is m^*. □

We next prove security. We proceed by a hybrid argument.

Game $\mathbf{G_1}$. Consider all queries to Punc that the adversary makes before posibly calling Expose. $\mathbf{G_1}$ differs from OW in that the snd calls inside each punc call inside each such query is replaced by the same call with uniform randomness.

We construct a sequence of $q' \leq q$ PRF adversaries $\mathcal{B}$ to hop between OW and $\mathbf{G_1}$.

Claim. $\mathrm{Adv}^{\mathrm{OW}}_{\mathrm{PE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathbf{G_1}}_{\mathrm{PE}}(\mathcal{A}) + q \cdot \mathrm{Adv}^{\mathrm{PRF}}_{\mathrm{F}}(\mathcal{B})$.

Proof. Let $\mathbf{H_1} = \mathrm{OW}$, and consider an adversary that makes $q' \leq q$ queries to Punc before calling Expose. Then let $\mathbf{H_i}$ for $1 < i \leq q'$ be $\mathbf{H_{i-1}}$ except that in the i-th Punc query, the call $(st_t, c, k) \leftarrow \mathrm{snd}(st^t_A, \bot; rk)$ is replaced by the call $(st_t, c, k) \leftarrow_{\$} \mathrm{snd}(st^t_A, \bot)$, i.e., snd now uses uniform randomness. By construction of $\mathbf{H_i}$, clearly $\mathbf{H_{q'}} = \mathbf{G_1}$.

We define adversary $\mathcal{B}$ playing PRF with respect to F, and simulates $\mathbf{H_{i-1}}$ if it is playing PRF^0 and $\mathbf{H_i}$ if it is playing PRF^1. $\mathcal{B}$ simulates all queries locally, except that when $\mathcal{A}$ queries Punc for the i-th time without having called Expose, $\mathcal{B}$ simulates the call $rk \leftarrow F(rk, 0)$ using its oracle Eval(0). Finally, $\mathcal{B}$ returns the same bit as $\mathcal{A}$. The simulation is perfect, and the claim follows by combining the sequence of hybrids. □

Game $\mathbf{G_2}$. This differs from $\mathbf{G_1}$ in that output key k in the snd call inside of the challenger's call $\mathrm{Enc}(pk, m^*, t^*)$ is replaced by a uniformly random key.

For this hop, we consider an IND R adversary $\mathcal{B}$ for S-MSSR R.

Claim. $\mathrm{Adv}^{\mathbf{G_1}}_{\mathrm{PE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathbf{G_2}}_{\mathrm{PE}}(\mathcal{A}) + \mathrm{Adv}^{\mathrm{IND}}_{\mathrm{R}}(\mathcal{B})$.

Proof. We construct S-MSSR adversary $\mathcal{B}$ that simulates $\mathbf{G_1}$ given it is playing IND^0 and $\mathbf{G_2}$ given it is playing IND^1. Given PE has tag space $[n]$, $\mathcal{B}$ first outputs (ω, n) to its challenger, and then receives ω as input. $\mathcal{B}$ then starts simulating for $\mathcal{A}$ as follows:

- $\mathcal{B}$ simulates the challenger's gen call by calling $\mathrm{Expose}_A(i)$ for $i \in [n]$, sets $pk = (st^0_A, \ldots, st^{n-1}_A)$ and gives pk to $\mathcal{A}$. Note that calling Expose_A does not affect the simulation below (i.e., does not prevent any query from succeeding).
- To simulate $\mathrm{Punc}(t)$ calls made before Expose is possibly called, $\mathcal{B}$ calls $\mathrm{Snd}(ad, t, \mathtt{T})$, which outputs (c, k), and then $\mathrm{Rcv}(1, ad, c)$. To simulate $\mathrm{Punc}(t)$ calls made after Expose is called, $\mathcal{B}$ simulates locally.
- To simulate Expose, $\mathcal{B}$ calls $st_B \leftarrow \mathrm{Expose}_B(1)$, samples $rk \in \mathcal{R}$ and returns (st_B, rk) to $\mathcal{B}$. Observe by construction of $\mathbf{G_1}$ that rk is correctly distributed, and that the query $\mathrm{Expose}_B(1)$ is allowed since there are no active challenge ciphertexts in the IND game except in the middle of oracle query simulations.
- When $\mathcal{A}$ outputs (ω, t^*) to $\mathcal{B}$, $\mathcal{B}$ simulates locally except for the following. In the $c^* \leftarrow \mathrm{enc}(pk, m^*, t^*)$ call, $\mathcal{B}$ replaces the call $(st^t_A, c_k, k) \leftarrow_{\$} \mathrm{snd}(st^t_A, \bot)$ with the output of the previous call $(c_k, k) \leftarrow \mathrm{Snd}(t^*, \bot, \mathtt{T})$ (note st^t_A is not used in enc thereafter) which much exist unless $t^* \notin C$ for $\mathbf{G_1}/\mathbf{G_2}$ game variable C, in which case $\mathcal{B}$ aborts anyway. Since Snd was called with third argument $ch = \mathtt{T}$, k is uniform given $\mathcal{B}$ is playing IND^1 and real given it is playing $k = 0$.
- Finally, when $\mathcal{A}$ outputs m', $\mathcal{B}$ outputs 1 if $m' = m^*$ and 0 otherwise.

To complete the proof, note that the simulation is perfect and $\mathcal{B}$ has the same advantage as $\mathcal{A}$. □

Claim. $\mathrm{Adv}^{\mathbf{G_2}}_{\mathrm{PE}}(\mathcal{A}) \leq \frac{1}{|\mathcal{M}|}$.

Proof. Recall in $\mathbf{G_2}$ that the adversary is given challenge ciphertext $c^* = (c_k, c_m)$ where $c_m = k \oplus m^*$, k is uniformly sampled and all other information revealed to $\mathcal{A}$ is independent of m^* (in particular the challenger's call to snd). Therefore, the adversary learns no information about m^* and thus its best strategy is randomly guessing it. □

Lower Bound. We recall the statement of the lower bound on the secret key size of a PE scheme from Barooti et al. [BCC+23a, BCC+23b].

Theorem 7. *[BCC+23b, Theorem 11] Let* PE *be a correct puncturable encryption scheme with respect to tag space $\mathcal{T} = \{0,1\}^n$ and secret key space $\mathcal{SK} = \{0,1\}^k$ (without loss of generality) that supports at most n_p punctures. For a fixed security parameter, suppose that*

$$\mathrm{Adv}^{\mathrm{OW}}_{\mathrm{PE}}(\mathcal{A}') \leq 2^{-\gamma}$$

where γ is maximal and $\mathcal{A}'$ is the efficient adversary defined in [BCC+23b, Figure 24]. Then

$$k \geq \log \binom{2^n}{n_p} - \frac{2}{1 - n_p \cdot 2^{-\gamma}} \geq n n_p - n_p \log(n_p) - \frac{2}{1 - n_p \cdot 2^{-\gamma}} \ ,$$

where the rightmost expression is $\approx (n - \log(n_p))n_p - 2$ when $n_p \ll 2^\gamma$.

Observe that we have constructed puncturable encryption from S-MSSR in Figure 20, where the secret key consists of the receiver state st_B and randomness $rk \in \mathcal{R}$. We therefore obtain the corollary stated above.