A Quantum-Safe Private Group System for Signal from Key Rerandomizable Signatures

Graeme Connell
Signal Messenger
USA
gram@signal.org

Sebastian Faller
IBM Research Europe - Zurich
ETH Zurich
Zurich, Switzerland
sebastian.faller@ibm.com

Felix Günther
IBM Research Europe - Zurich
Zurich, Switzerland
mail@felixguenther.info

Julia Hesse
IBM Research Europe - Zurich
Zurich, Switzerland
jhs@zurich.ibm.com

Vadim Lyubashevsky
IBM Research Europe - Zurich
Zurich, Switzerland
vad@zurich.ibm.com

Rolfe Schmidt
Signal Messenger
USA
rolfe@signal.org

Abstract

Instant messaging services are an integral part of today's communication and their privacy has wide societal implications. Major messengers deploy end-to-end encryption, hiding message contents from the service provider. Group messaging, however, creates the challenge of also keeping the group membership list private.

The Signal messenger currently implements private group management using techniques inspired by Chase, Perrin, and Zaverucha (CCS 2020). Transitioning this system to quantum-safe turns out to be challenging: While one-to-one messaging can often adopt the newly standardized KEMs and signatures in a relatively direct way, private group management is more complex. Signal's existing design heavily relies on the discrete-log structure to combine anonymous credentials, verifiable encryption, and oblivious PRFs for privacy and functionality. Quantum-safe versions of these components unfortunately are typically far less efficient, requiring heavy zero-knowledge proofs and large communication per group operation. As a result, simply "swapping in" quantum-safe primitives is unlikely to yield an optimal protocol.

This paper reconsiders the design of the entire group system from the ground-up. Our result is a scheme that possesses the same strong privacy guarantees, but it is built in a more modular way using simpler underlying cryptographic building blocks that permit a more efficient quantum-safe instantiation. The modularity of our protocol further allows for gradual migration to quantum-safe: we can immediately transition components vulnerable to harvest-now-decrypt-later attacks (such as classical public-key encryption, computationally hiding commitments, etc.) while deferring the transition of other building blocks, such as authentication. We prove our design secure in an extended security model that more comprehensively captures the rich feature set of Signal's group messaging system. In our experimental evaluation, the core operations of our new design turn out to be even more efficient than those of Signal's current deployment.

1 Introduction

Signal is an open-source messaging service, used by tens of millions of users [13], that is known for putting an emphasis on end-to-end encryption and user privacy. In addition to conventional one-to-one messaging, Signal allows users to create groups and exchange messages in these groups. In a typical implementation of group messaging, the server would centrally store important information such as a list of group members, the group's name, or the group's image. These data can clearly be sensitive, and they might even be equally attractive targets for adversaries as the chat messages themselves. In 2019, Signal introduced their private group system [7, 19] that hides group data from the server while still allowing the server to perform fine-grained access control and spam protection. A private group system (PGS) allows the realization of many complex operations on the group, differentiating between admins and normal users, mediated by a central server. Beyond core operations such as admins creating groups and adding/removing members, these also include allowing users to leave a group on their own and updating their user data, as well as (depending on policies) to join a group via an invite link, to add other users, or to change group attributes.

Together with the desired privacy guarantees against the server, such a private group system requires mechanisms outside of the scope of solutions like Message Layer Security (MLS). The main security goals of Signal's private group system are the following:

- *Privacy of group membership.* The server does not learn which users are members of which groups.
- *Consistency.* All honest users should have a consistent view of who is a member of the group (and who is not anymore).
- *Anonymous group management.* Only users with the necessary privileges can create groups, add or remove members, or change the group's name. Yet, when the server authorizes such users, their identity must remain hidden.
- *Confidential key material.* The system facilitates key material used to encrypt users' profile data or authorize sending group messages, accessible only to users that share a group.
- *Authorized addition.* To decrease the risk of spam groups, group administrators can only add users they know (and that did not block them).

Signal's current private group system. Signal's current system is based on the work by Chase, Perrin, and Zaverucha [7]. The system uses advanced cryptographic primitives to achieve the security goals described above: the list of group members is encrypted using *deterministic verifiable encryption*, and the identity of users is proven

J. Hesse was supported by the Swiss National Science Foundation (SNSF) under the Starting Grant *Low-Entropy Cryptography for the Quantum Era*. V. Lyubashevsky was supported by the ERC Consolidator Grant *PLAZA*.

with *key-verification anonymous credentials (KVACs)*. Simplifying, one can think of the server checking a user's privilege to, e.g., add another user as follows: Using a zero-knowledge proof (ZKP), the user proves to the server that 1) the user possesses a credential that their identity corresponds to one entry of the encrypted member list, 2) the user possesses a credential that proves that the user and the new group member were in contact before, and 3) the user provides a correct encryption that contains the newly added member's information. Signal recently added a layer of spam protection: The Sealed Sender Authorization Pass system [23] allows the server to recognize legitimate group messages, without revealing the sender's identity (as Signal is using sealed-sender sending) and without the server learning for which group the message is intended. To this end, Signal combines an oblivious pseudo-random function (OPRF) and their verifiable encryption scheme in a non–black-box manner. Concretely, the scheme exploits that the DH-based OPRF and the ElGamal-based verifiable encryption work over the same group.

The quantum-safe challenge. In Signal's current system, the group list and all additional information about the group, like the name or the image of the group, are encrypted with an ElGamal-like encryption scheme. This sensitive information is hence an attractive target for so-called harvest-now-decrypt-later (HNDL) attacks: An adversary that gets hold of the classically encrypted group information *today* could store this data until a cryptographically-relevant quantum computer is available to decrypt it later. The possibility of such an attack being mounted *now* mandates a timely transition to a system that is resistant to HNDL attacks as soon as possible, with a transition to a full quantum-resistant system in the near future.

Given the complexity of the current system and its non–black-box use of primitives, simply replacing all the primitives by quantum-safe candidates such as [1, 3, 4, 18] will almost certainly be too expensive. The quantum-safe version of the core anonymous credential primitive (e.g., [4]) has output sizes of around 30KB and its usage in every group operation would significantly increase Signal's operational cost of sending data between cloud providers. For the OPRF-based Sealed Sender Authorization Pass, it is even unclear if we can find a quantum-safe replacement. Current OPRFs [1, 3] do not seem to allow for the same strong coupling with the verifiable encryption, which could completely roadblock the transition.

Beyond the above issues, there is a further challenge: when transitioning their secure messaging protocols to quantum safety, Signal has set a requirement of providing *hybrid* security for any public-key primitives. This would make the quantum-safe part even less efficient as we would need to use ZK proofs which are statistically-hiding, rather than the more efficient computationally-hiding variants (e.g. [18]). In short, making the current Signal protocol even HNDL secure seems practically and economically prohibitive based on the current state of development of quantum-safe cryptography.

1.1 Our Work

We reconsider the design of the group management protocol from the ground up. The goal is to have a protocol that can be immediately secure against harvest-and-decrypt attacks using efficient quantum-safe primitives, while allowing other aspects of the scheme to, for now, remain sound only with respect to classical attacks. The rationale is that protocol parts involving, for example, authentication, which only need to be secure at the moment they are executed, can wait to be upgraded to their post-quantum versions once quantum computers capable of breaking them actually exist. The amount of data sent between the Signal servers (which run on multiple cloud providers) and clients directly affects Signal's expenses, and so one ideally would like to wait to upgrade to quantum-safe protocols, which are all more bandwidth-intensive, until necessary. On the other hand, it is also very important to be able to easily switch to quantum-safe without redesigning the protocol yet again. Our modular design makes it possible to swap out components for one part of the system without affecting the others. It also allows for easy testing of the fully quantum-safe protocol to determine its viability when the time to switch finally arrives.

The main conceptual difference between Signal's current group management structure, based on [7], and our proposal is that in the former, the server acts as a "gatekeeper" making sure that all changes to the group structure follow the prescribed policy. In our proposal, it is the group members themselves (or more specifically, their client software) that check that the list entries and performed operations are valid. The server's main role is simply to check that members of the group (either administrators or plain users) are the ones writing the entries in the positions of the list where they are allowed. This new approach is able to retain all the operability and privacy of the prior, with the main advantage being that most operations require only basic cryptographic primitives. In particular, we only use standard symmetric- and public-key encryption schemes, which can all be instantiated quantum-safely, making the scheme immediately secure against harvest-and-decrypt attacks.

Very importantly, the system is set up such that group members are able to trace any activity to the specific group member who initiated it, and so any misbehavior is immediately detected by everyone in the group. Any honest administrator detecting invalid operations (either due to malicious actors or, more likely, buggy software) can restore the group system to a valid state. A non-administrator can identify the misbehaving party and inform them that their software is possibly buggy. Crucially, such tracing must not interfere with privacy, and we prudently use a *key rerandomizable* signature scheme (akin to [10, 11]) to ensure group members can sign under pseudonym keys (allowing the server to perform certain authorization checks) while hiding their identity, and being untraceable between groups, from group outsiders. Practical key rerandomizable signature schemes follow from Schnorr signatures or a tweak of the new ML-DSA (Dilithium) [2] PQC NIST standard, which we adapt to our needs.

We formalize and analyze the desired security of a private group system through a Universal Composability (UC) [6] security model. Revisiting the model of [7], we ultimately chose to re-design the security model from the ground up, not least to capture more fine-grained security guarantees that our system can provide. These include the distinction between admins and regular users (and the effects of corrupting a regular, but non-admin user), the limited power a corrupt server should have, and the ability to refresh key material of both groups and users.

In Section 4.2, we compare the security properties of the new scheme to the one currently implemented by Signal. Our proposed scheme actually has stronger privacy guarantees in several areas, in particular against a malicious server. The trade-off is that in the

new scheme the server cannot be sure that the group members in the lists are actual registered Signal users. While this does not lead to any privacy concerns, it means that a group without any honest clients can contain invalid entries, which perhaps can be seen as an "inelegant" consequence of our design.

In Section 6, we discuss this aspect, and other deployment considerations, mapping our design to the advanced features of Signal's messaging architecture and discussing challenges and trade-offs. In that section we revisit the possibility for using some anonymous credential system for these tangential issues (such as proving that a member is still active or a registered Signal user, that a group entry is a registered member, etc.), but the key point is that these credentials are completely separate from the core group operations. They are used for "liveness" checks which occur rarely, and thus even if they are instantiated with less-efficient quantum-safe primitives, the efficiency cost to the whole system will be minimal. They are furthermore not connected to any verifiable encryption scheme (as they are in the current protocol) and so do not cause any HNDL security issues, and thus one can wait to update them to their quantum-safe versions until later.[1]

In Section 7 we compare the performances of the implementation of our scheme to the currently deployed system. Benchmarking our new system, we experimentally evaluated the core group operations for an EC version with HNDL security using elliptic-curve–based, rerandomizable signatures and a fully quantum-safe PQ version based on ML-DSA, as well as for Signal's deployed system. Notably, in terms of computational efficiency, our EC and PQ systems outperform the current deployment for all group sizes supported by Signal. And even in terms of transmission sizes, where quantum-safe schemes are notoriously known to be quite inefficient, our fully quantum-safe scheme is more compact than the current scheme for some operations.

1.2 Paper Outline

In Section 2 we provide details on key rerandomizable signatures and their instantiation. In Section 3, we introduce the functionality of private group systems in general and the specifics of Signal and the overall threat model, before providing an overview of our design, its rationale, and core components. In Section 4, we then present the technical details and protocol descriptions of our system, and compare their security to the currently deployed scheme. We introduce our security model for private group systems in Section 5 and relate it to the prior work of [7]. We cover deployment considerations in Section 6 and present our experimental evaluation in Section 7. The appendix contains further protocol specifications, our detailed PGS security model and analysis and a thorough comparison of our system with Signal's currently deployed PGS.

2 Key Rerandomizable Signatures

The (folklore) version of the rerandomizable signature scheme built on top of the Schnorr scheme, as described in [11], has its algorithms defined as follows: The Gen algorithm outputs $(x, y = g^x)$ as the secret / public values. The RandSK algorithm outputs $\rho + x$, while the RandVK algorithm outputs $y \cdot g^\rho$. Since $g^{\rho+x} = y \cdot g^\rho$, the outputs of RandSK and RandVK form the secret and public key of a Schnorr signature scheme.

In the quantum-safe scenario, the above can be analogously constructed with a modification of lattice constructions. One such construction was proposed in [10] where a randomizable scheme was based on the (non-standardized and less efficient) Dilithium-QROM scheme from [15]. The modification of the scheme involved changing several internal parameters, which at the end also involved modifying the verification algorithm. In our work, we instead build on top of the standardized Dilithium / ML-DSA without needing to change the verifier. [2]

For simplicity, we first provide a high level description which does not go into the details of the Dilithium optimizations. The fully-optimized scheme is then described in Figures 3 and 4. Let B be a matrix of the form $B = [\, A \mid I \,]$, where A is randomly generated and I is the identity matrix, that is fixed for the entire protocol. The KeyGen algorithm chooses a low-norm vector x and outputs $(x, y = Bx)$ as the secret / public parameters. The secret key x has exactly the same distribution as in Dilithium. The difference between y and the Dilithium public key is that the latter is rounded in order to make it smaller, whereas we will use the entire y. The RandSK algorithm chooses a random small ρ (from the same distribution as x) and outputs $x' = \rho + x$. The RandVK algorithm uses the random ρ to create $y' = y + B\rho$. Since $B \cdot x' = y'$, x' and y' are a secret key / public key pair for a simplified version of the Dilithium signature without rounding the public key(cf. [17, Figure 5]). At this point, the RandVK can round the public key in the same way as in Dilithium, which would make the resulting public key have exactly the same size.

The main effect that adding ρ to the secret key x has on the signing algorithm is that the coefficients of the new secret key are twice as large as before. If we would like to keep the norm of the outputted signature the same, then doubling the size of the coefficients will need to affect the signing time. Lattice-based signatures, such as Dilithium, use the Fiat-Shamir with aborts paradigm [16] where a signature attempt gets output as a signature only if the potential output vector has small-enough coefficients; and the attempt is repeated otherwise. Doubling the maximum size of the secret key coefficients ends up squaring the expected number of signing attempts (cf. [2, Equation 5 and Table 2]). This translates into an average of about a 5x slowdown versus the ML-DSA standard. But since the signing operation in ML-DSA is already very fast (less than a millisecond on a standard processor [2, Table 1]), and in our scheme these signatures will only be used when the group structure changes, this should not be a problem in practice.

By the Module-LWE assumption, since $(B, B\rho)$ is indistinguishable from uniform, the outputs of RandVK look uniform as well. Forging a signature implies finding a polynomial vector $\mathbf{z}$ with small coefficients and a polynomial c such that $Bz = y'c$. We can rewrite this as $Bz = (y + B\rho)c$, which implies that $B(z - \rho c) = yc$. The coefficients of $z - \rho c$ are small, and so this implies solving the

[1] One could even argue that until the cost of breaking classical cryptography using a quantum computer gets to below the cost of simply opening a valid Signal account, one can even continue using these classical schemes in the era of quantum computers.

[2] As a side note of independent interest, since key rerandomizable signatures immediately imply (cf. [9]) (non-hierarchical) deterministic wallets [5] used in the bitcoin BIP32 proposal [25], our construction gives a quantum-safe version of deterministic wallets with signature sizes being the same as those of ML-DSA and the same verification algorithm as ML-DSA.

same SIS problem as in Dilithium (cf. [17, Equation 81]). [3] The only thing that would change in the actual Dilithium scheme versus the above description is that we would apply a rounding step to y' as in Dilithium's key generation [2, Figure 4, Line 5] in order to reduce the public key size, and then apply the signing and verification algorithms exactly [2, Figure 4] as in the standard, with the value β being set to double the previous value to accommodate the larger secret key parameters.

In Section 2.1, we state the formal definition of a key-rerandomizable signature scheme. In Figures 3 and 4, we give an instantiation of the ML-DSA based re-randomized scheme and prove its security in Section 2.2.

2.1 Formal Definition

Definition 1 (Key-rerandomizable signature scheme [11]). *A* key-rerandomizable signature scheme *is a tuple of algorithms* $\Sigma = (\mathsf{Gen}, \mathsf{Sgn}, \mathsf{Vfy}, \mathsf{RandVK}, \mathsf{RandSK})$ *of the following algorithms:*

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda)$ *takes as input the security parameter and outputs a base key pair* $(\mathsf{sk}, \mathsf{vk})$.
- $\sigma \leftarrow \mathsf{Sgn}(\mathsf{sk}, M)$ *takes as input the secret key* sk *and a message* M *and outputs a signature* σ.
- $1/0 \leftarrow \mathsf{Vfy}(\mathsf{vk}, M, \sigma)$ *takes as input the public verification key* vk*, a message* M*, and a signature* σ *and outputs 1 if the signature verifies for this message and key and else 0.*
- $\mathsf{vk}' \leftarrow \mathsf{RandVK}(\mathsf{vk}, \rho)$ *takes as input a verification key* vk*, and a randomizer* ρ *and outputs a new verification key* vk'.
- $\mathsf{sk}' \leftarrow \mathsf{RandSK}(\mathsf{sk}, \rho)$ *takes as input a secret key* sk *and a randomizer* ρ *and outputs a new secret key* sk'.

Note that Fleischhacker et al. [11] defined signatures with *perfectly* rerandomizable keys, where perfect means that the rerandomized keys have the exact same distribution as the original keys. This property is not required in the Dlog setting and it is also not satisfied by the Dilithium-based construction of the scheme because the norm of the secret keys is bigger and the original public keys are uncompressed, whereas the rerandomized public keys are compressed. However, we require that the original public keys and the rerandomized keys are not linkable. To that end, we slightly adapt the definition of unlinkability for key-blinding signatures from [10]. We drop the epochs from their definition, as we do not need them.

Definition 2 (Unlinkability under chosen-message attacks). *A signature scheme with re- randomizable keys* $\Sigma = (\mathsf{Gen}, \mathsf{Sgn}, \mathsf{Vfy}, \mathsf{RandSK}, \mathsf{RandVK})$ *is* unlinkable under chosen-message attacks (UL-CMA) *if for all PPT adversaries* $\mathcal{A}$ *the probability that the experiment* $\mathrm{Exp}^{\text{ul-cma}}_{\Sigma,\mathcal{A}}(\lambda)$ *Figure 1 evaluates to 1 is negligible in* λ.

Similarly, we slightly adapt the definition of unforgeability of [10] by removing the epochs.

Definition 3 (Unforgeability under rerandomized keys). *A signature scheme with re- randomizable keys* $\Sigma = (\mathsf{Gen}, \mathsf{Sgn}, \mathsf{Vfy}, \mathsf{RandSK}, \mathsf{RandVK})$ *is* unforgeable under rerandomized keys (UFRK) *if for all PPT adversaries* $\mathcal{A}$ *the probability that the experiment* $\mathrm{Exp}^{\text{ufrk}}_{\Sigma,\mathcal{A}}(\lambda)$ *Figure 2 evaluates to 1 is negligible in* λ.

[3] The SIS bound of the extracted solution increases by an additive factor of $\|\rho c\|_\infty = \beta < 200$ (see Table 1) vs. the extracted SIS solution in Dilithium, which has size at least $2\gamma_1 > 2^{18}$. This small increase in the norm (i.e. 2^{18} vs. $2^{18} + 200$) has less than a 1 bit impact on the security of Dilithium, which is confirmed by running the SIS security estimator, which results in exactly the same security parameter.

$\underline{\texttt{Experiment}\ \mathrm{Exp}^{\text{ul-cma}}_{\Sigma,\mathcal{A}}(\lambda)\text{:}}$
$(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$b \xleftarrow{\$} \{0,1\}$
if $b = 0$: $\mathsf{vk}^* \xleftarrow{\$} \mathcal{V}$
else $\rho^* \xleftarrow{\$} \mathcal{RS}, \mathsf{vk}^* \leftarrow \mathsf{RandVK}(\mathsf{vk}, \rho^*)$
$b^* \leftarrow \mathcal{A}^{O(\mathsf{sk},\cdot,\cdot)}(\mathsf{vk}, \mathsf{vk}^*)$
if $b = b^*$: **return** 1
else return 0

$\underline{O(\mathsf{sk}, M, \rho)}$
$\sigma \leftarrow \mathsf{Sgn}(\mathsf{RandSK}(\mathsf{sk}, \rho), M)$
return σ

Figure 1: The experiment $\mathrm{Exp}^{\text{ul-cma}}_{\Sigma,\mathcal{A}}(\lambda)$**, where** $\mathcal{V}$ **is the set of all possible verification keys output by** Gen **and** $\mathcal{RS}$ **is the set of all possible randomizers taken as input by** RandVK **and** RandSK**.**

$\underline{\texttt{Experiment}\ \mathrm{Exp}^{\text{ufrk}}_{\Sigma,\mathcal{A}}(\lambda)\text{:}}$
$(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$Q \leftarrow \emptyset$
$(M^*, \sigma^*, \rho^*) \leftarrow \mathcal{A}^{O_1(\mathsf{sk},\cdot), O_2(\mathsf{sk},\cdot,\cdot)}(\mathsf{vk})$
if one of the following conditions holds:
1. $\mathsf{Vfy}(\mathsf{vk}, M^*, \sigma^*) = 1$ and $M^* \notin Q$
2. $\mathsf{Vfy}(\mathsf{RandVK}(\mathsf{vk}, \rho^*), M^*, \sigma^*) = 1$ and $M^* \notin Q$
return 1
else return 0

$\underline{O_1(\mathsf{sk}, M)}$
$Q \leftarrow Q \cup \{M\}$
$\sigma \leftarrow \mathsf{Sgn}(\mathsf{sk}, M)$
return σ

$\underline{O_2(\mathsf{sk}, M, \rho)}$
$Q \leftarrow Q \cup \{M\}$
$\sigma \leftarrow \mathsf{Sgn}(\mathsf{RandSK}(\mathsf{sk}, \rho), M)$
return σ

Figure 2: The experiment $\mathrm{Exp}^{\text{ufrk}}_{\Sigma,\mathcal{A}}(\lambda)$**.**

2.2 Security Analysis

We first state the MLWE problem and the SelfTargetMSIS problem that will underlie the security of the scheme.

Definition 4 (MLWE [2]). *For integers* m, k *and a probability distribution* $D : \mathcal{R}_q \to [0,1]$*, we say that the advantage of algorithm*

Table 1: Parameters of Key Rerandomizable Dilithium . The number of repetitions inside the signing procedure is the square of the value in standard Dilithium from [2, Table 2].

NIST Security Level	2	3	5
Parameters			
q [modulus]	8380417	8380417	8380417
d [dropped bits from $\mathbf{t}$]	13	13	13
τ [# of ±1's in c]	39	49	60
challenge entropy [$\log\binom{256}{\tau} + \tau$]	192	225	257
γ_1 [$\mathbf{y}$ coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, ℓ) [dimensions of $\mathbf{A}$]	$(4, 4)$	$(6, 5)$	$(8, 7)$
η [(randomized) secret key and range]	2	4	2
β [$\tau \cdot \eta$]	78	196	120
ω [max. # of 1's in the hint $\mathbf{h}$]	80	55	75
Repetitions	$\boxed{4.25^2 \approx 18}$	$\boxed{5.1^2 \approx 26}$	$\boxed{3.85^2 \approx 15}$

$\mathcal{A}$ in solving the decisional $MLWE_{k,\ell,D}$ problem over the ring $\mathcal{R}_q$ is

$$\mathbf{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathcal{A}) \coloneqq |\Pr\left[b = 1 : \begin{matrix}\mathbf{A} \xleftarrow{\$} \mathcal{R}_q^{k\times\ell}; \mathbf{t} \xleftarrow{\$} \mathcal{R}_q^k \\ b \leftarrow \mathcal{A}(\mathbf{A}, \mathbf{t})\end{matrix}\right] - \Pr\left[b = 1 : \begin{matrix}\mathbf{A} \xleftarrow{\$} \mathcal{R}_q^{k\times\ell} \\ \mathbf{s}_1 \leftarrow D^\ell, \mathbf{s}_2 \leftarrow D^k \\ b \leftarrow \mathcal{A}(\mathbf{A}, \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2)\end{matrix}\right]|.$$

Definition 5 (SelfTargetMSIS [2]). *For integers k, ℓ, ζ and a hash function $H : \{0,1\}^* \to B_\tau$, we say that the advantage of algorithm $\mathcal{A}$ in solving the $SelfTargetMSIS_{H,k,\ell,\zeta}$ problem over the ring $\mathcal{R}_q$ is*

$$\mathbf{Adv}^{\mathsf{SelfTargetMSIS}}_{H,k,\ell,\zeta}(\mathcal{A}) \coloneqq \Pr\left[\begin{matrix}0 \le \|\mathbf{y}\|_\infty \le \zeta \\ \wedge H(\mu \| [\mathbf{I} \mid \mathbf{A}] \cdot \mathbf{y}) = c\end{matrix} : \left(\mu, \mathbf{y} \coloneqq \begin{bmatrix}\mathbf{r} \\ c\end{bmatrix}\right) \leftarrow \mathcal{A}^{H(\cdot)}(\mathbf{A})\right].$$

We would like to highlight that SelfTargetMSIS can be seen as a handy abstraction that allows us to not think of rewinding. Still, there is a classical (non-tight) reduction from MSIS to SelfTargetMSIS in the ROM, using the standard forking lemma [20], see [2, Sec. 6.1].

Theorem 1. *For every PPT adversary $\mathcal{A}$, there are PPT adversaries $\mathcal{B}$ and $\mathcal{D}$ such that the advantage of $\mathcal{A}$ breaking the unforgeability under rerandomized keys of the Dilithium-based key-rerandomizable signature scheme from Figure 4 and Figure 3 is*

$$\Pr\left[\mathrm{Exp}^{\mathsf{ufrk}}_{\Sigma,\mathcal{A}}(\lambda) = 1\right] \le \mathbf{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathcal{D}) + \mathbf{Adv}^{\mathsf{SelfTargetMSIS}}_{H,k,\ell+1,\zeta}(\mathcal{B}),$$

where D is the uniform distribution over S_η and $\zeta = \max\{\gamma_1, \beta + 2^{d-1} \cdot \tau + 2\gamma_2 + 1\}$.

Proof sketch. As usual for Fiat-Shamir-type signatures, we first show that a reduction can simulate signing oracle queries without using the secret key, i.e., we prove that the underlying identification scheme is zero-knowledge. In our case, we also have to simulate signatures under randomized keys. But as we will see, this does not make a big difference. Second, we replace the base public key $\mathbf{t}^b$ by a random value using MLWE. Finally, we show that a forgery of the adversary amounts to an attack against the SelfTargetMSIS assumption. The major difference to the original Dilithium proof is that the extracted solution might have slightly bigger norm ($+\beta$ bigger) due to the added blinding of the secret key. We note that one could avoid this small loss in security by instead changing the verification algorithm of the original scheme to only accept signatures with $\|\mathbf{z}\|_\infty \le \gamma_1 - 2\beta$. But we think that keeping the standard verification unchanged is worthwhile while the loss in security is negligibly small. The proof mainly follows the arguments from the Dilithium specification [2, Sec. 6, App. B] and we only discuss the changes to that.

Proof. Throughout the proof, we will denote as $\mathbf{s}_1^b, \mathbf{s}_2^b$ the polynomials of the base secret key sk^b and $\mathbf{t}^b$ and $\mathbf{t}_0^b, \mathbf{t}_1^b$ for the polynomials of the base public key vk^b output by Gen. Similarly, we will write $\mathbf{s}_1^r = \mathbf{s}_1^b + \mathbf{s}_1'$, $\mathbf{s}_2^r = \mathbf{s}_2^b + \mathbf{s}_2'$ for the polynomials of a randomized secret key sk^r and $\mathbf{t}^r = \mathbf{t}^b + \mathbf{A}\mathbf{s}_1' + \mathbf{s}_2'$ (and $\mathbf{t}_0^r, \mathbf{t}_1^r$, resp.) the polynomials of a randomized public key vk^r output by RandVK.

Assume there is an adversary $\mathcal{A}$ winning $\mathrm{Exp}^{\mathsf{ufrk}}_{\Sigma,\mathcal{A}}(\lambda)$ with non-negligible advantage. We construct a reduction $\mathcal{B}$ that internally runs $\mathcal{A}$ and breaks the Self Target MSIS assumption with non-negligible advantage.

In a first hybrid game, we change the way $\mathcal{B}$ responds to $\mathcal{A}$'s oracle queries. For the simulation, note that $\mathcal{B}$ can simulate oracle queries for the base key sk^b in a similar way as for a randomized key $\mathsf{RandSK}(\mathsf{sk}^b, \rho)$ just by setting $(\mathbf{s}_1', \mathbf{s}_2') = \mathbf{0}$. From then on, the simulator can act exactly as the zero-knowledge simulator for Dilithium, described in [2, App. B].

After simulating signing queries, $\mathcal{B}$ no longer uses the base secret key $\mathbf{s}_1^b$ and $\mathbf{s}_2^b$ to answer oracle queries. Therefore, in a next game hop, we can replace $\mathbf{t}^b = \mathbf{A}\mathbf{s}_1^b + \mathbf{s}_2^b$ in the computation of the base public key by a uniformly random $\mathbf{t}^b \in \mathcal{R}_q^k$. If this would change the success probability of $\mathcal{B}$, we could turn $\mathcal{B}$ into an adversary $\mathcal{D}$ against $\mathrm{MLWE}_{k,\ell,D}$, where D is the uniform distribution over S_η in the straightforward way.

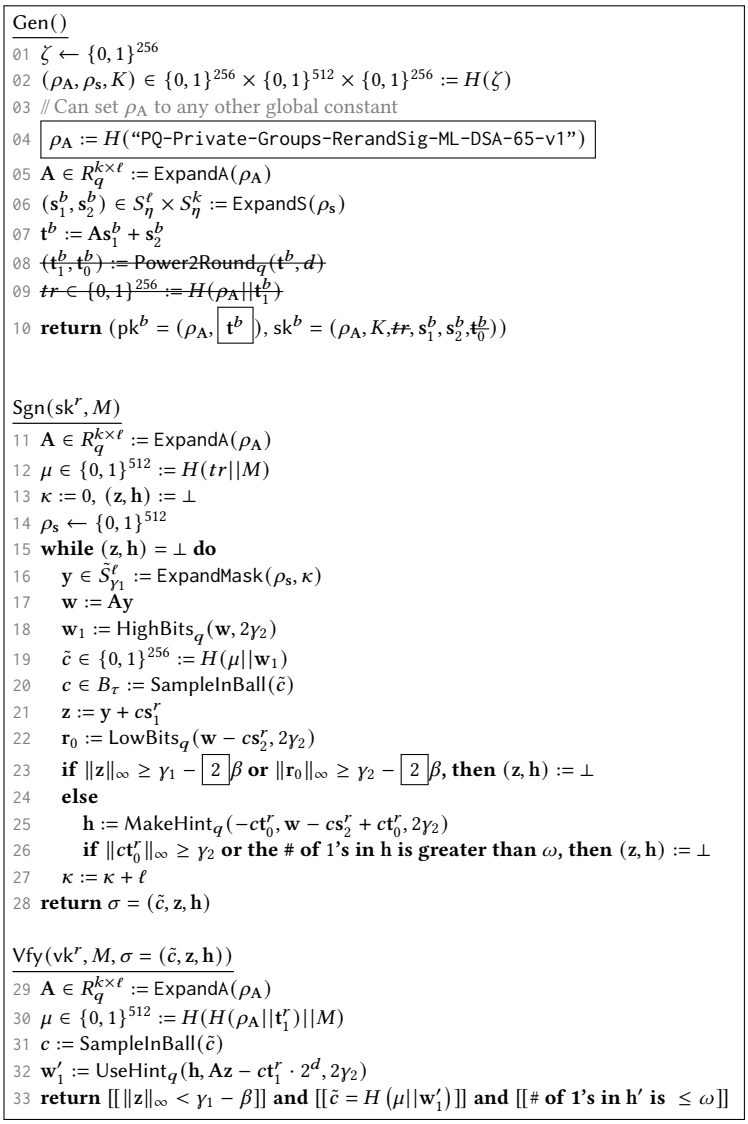

Figure 3: The base key generation and signing procedures. The framed and crossed out parts are the differences between these protocols and the standardized Dilithium ones from [2] (If desired, the crossed out parts could be kept, but their outputs will be over-written by the rerandomization algorithms). We denote as $\mathbf{s}_1^b, \mathbf{s}_2^b$ the polynomials of the base secret key sk^b and $\mathbf{t}^b$ as the polynomial of the base public key vk^b output by Gen. Similarly, we write $\mathbf{s}_1^r, \mathbf{s}_2^r$ for the polynomials of a randomized secret key sk^r and $\mathbf{t}^r$ (and $\mathbf{t}_0^r, \mathbf{t}_1^r$, resp.) the polynomials of a randomized public key vk^r output by RandVK. Sgn is denoted to use sk^r but works just the same using sk^b. The description of the supporting algorithms (e.g. SampleInBall, etc.) can be found in [2].

Finally, we're ready for a reduction from Self Target SIS. Assume that $\mathcal{A}$ submits a forgery $(M^*, (\mathbf{z}^*, c^*, \mathbf{h}^*))$ passing verification, i.e., $\mu^* = H(H(\rho_A\|\mathbf{t}_1^r)\|M^*)$, $\|\mathbf{z}^*\|_\infty < \gamma_1 - \beta$, there are at most ω many 1s in $\mathbf{h}^*$, and

$$c^* = H(\mu^*, \mathsf{UseHint}_q(\mathbf{h}^*, \mathbf{A}\mathbf{z}^* - c^*\mathbf{t}_1^r \cdot 2^d, 2\gamma_2)).$$

Like in the original proof, we can argue that there exists a $\mathbf{u} \in \mathcal{R}_q^k$ with $\|\mathbf{u}\| \leq 2\gamma_2 + 1$ such that

$$2\gamma_2 \cdot \mathsf{UseHint}_q(\mathbf{h}^*, \mathbf{A}\mathbf{z}^* - c^*\mathbf{t}_1^r \cdot 2^d, 2\gamma_2) = \mathbf{A}\mathbf{z}^* - c^*\mathbf{t}_1^r \cdot 2^d + \mathbf{u},$$

and only ω coefficients of $\mathbf{u}$ have magnitude greater than γ_2. We can rewrite the randomized public key $\mathbf{t}^r = \mathbf{t}_1^r \cdot 2^d + \mathbf{t}_0^r$ with $\|\mathbf{t}_0^r\| \leq 2^{d-1}$ and therefore

$$\begin{aligned}\mathbf{A}\mathbf{z}^* - c^*\mathbf{t}_1^r \cdot 2^d + \mathbf{u} &= \mathbf{A}\mathbf{z}^* - c^*(\mathbf{t}^r - \mathbf{t}_0^r) + \mathbf{u} \\ &= \mathbf{A}\mathbf{z}^* - c^*\mathbf{t}^r + \underbrace{(c\mathbf{t}_0^r + \mathbf{u})}_{:=\mathbf{u}'} \\ &= \mathbf{A}\mathbf{z}^* - c^*\mathbf{t}^r + \mathbf{u}'.\end{aligned}$$

$\underline{\mathsf{RandSK}(\mathsf{sk}^b, \rho)}$
$\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho_{\mathbf{A}})$
$(\mathbf{s}'_1, \mathbf{s}'_2) \in S_\eta^\ell \times S_\eta^k := \mathsf{ExpandS}(\rho)$
$\mathbf{t}^r := \mathbf{A}(\mathbf{s}_1 + \mathbf{s}'_1) + (\mathbf{s}_2 + \mathbf{s}'_2)$
$(\mathbf{t}_1^r, \mathbf{t}_0^r) := \mathsf{Power2Round}_q(\mathbf{t}^r, d)$
$tr \in \{0,1\}^{256} := H(\rho_{\mathbf{A}} || \mathbf{t}_1^r)$
$\mathsf{sk}^r = (\rho_{\mathbf{A}}, K, tr, \mathbf{s}_1^r := \mathbf{s}_1^b + \mathbf{s}'_1, \mathbf{s}_2^r := \mathbf{s}_2^b + \mathbf{s}'_2, \mathbf{t}_0^r)$

$\underline{\mathsf{RandVK}(\mathsf{pk}^b, \rho)}$
$\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho_{\mathbf{A}})$
$(\mathbf{s}'_1, \mathbf{s}'_2) \in S_\eta^\ell \times S_\eta^k := \mathsf{ExpandS}(\rho)$
$\mathbf{t}' := \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0$
$\mathbf{t}^r := \mathbf{t}^b + \mathbf{A}\mathbf{s}'_1 + \mathbf{s}'_2$
$(\mathbf{t}_1^r, \mathbf{t}_0^r) := \mathsf{Power2Round}_q(\mathbf{t}^r, d)$
$\mathsf{vk}^r = (\rho, \mathbf{t}_1^r)$

Figure 4: Randomizers functions RandVK **and** RandSK**. We denote as $\mathbf{s}_1^b, \mathbf{s}_2^b$ the polynomials of the base secret key sk^b and $\mathbf{t}^b$ and $\mathbf{t}_0^b, \mathbf{t}_1^b$ for the polynomials of the base public key vk^b output by** Gen**. Similarly, we write $\mathbf{s}_1^r, \mathbf{s}_2^r$ for the polynomials of a randomized secret key sk^r and $\mathbf{t}^r$ (and $\mathbf{t}_0^r, \mathbf{t}_1^r$, resp.) the polynomials of a randomized public key vk^r output by** RandVK**. The value $\rho_{\mathbf{A}}$ is a fixed public common reference string set to the value $H($`"PQ-Private-Groups-RerandSig-ML-DSA-65-v1"`$)$.**

Next, note that $\mathbf{t}^r = \mathbf{t}^b + \mathbf{A}\mathbf{s}'_1 + \mathbf{s}'_2$. That means we can further regroup to

$$\begin{aligned}\mathbf{A}\mathbf{z}^* - c^*\mathbf{t}^r + \mathbf{u}' &= \mathbf{A}\mathbf{z}^* - c^*(\mathbf{t}^b + \mathbf{A}\mathbf{s}'_1 + \mathbf{s}'_2) + \mathbf{u}' \\ &= \mathbf{A}\underbrace{(\mathbf{z}^* - c^*\mathbf{s}'_1)}_{:=\mathbf{z}'} - c^*\mathbf{t}^b - \underbrace{(c^*\mathbf{s}'_2 + \mathbf{u}')}_{:=\mathbf{u}''} \\ &= \mathbf{A}\mathbf{z}' - c^*\mathbf{t}^b + \mathbf{u}''.\end{aligned}$$

We know $\|\mathbf{z}'\|_\infty \le \|\mathbf{z}^*\| + \|c^*\mathbf{s}'_1\| \le \gamma_1 - \beta + \beta = \gamma_1$ and

$$\|\mathbf{u}''\| \le \|c^*\mathbf{s}'_2\| + \|c^*\mathbf{t}_0^r\| + \|\mathbf{u}\| \le \beta + \tau \cdot 2^{d-1} + 2\gamma_2 + 1.$$

Therefore, if $\mathcal{A}$ outputs a forgery, $\mathcal{B}$ found a solution to the $\mathrm{SelfTargetMSIS}_{H,k,\ell+1,\zeta}$ problem because

$$H\left(\mu^* \| \frac{1}{2\gamma_2} \cdot [\mathbf{A} \mid \mathbf{t}^b \mid \mathbf{I}_k] \cdot \begin{bmatrix}\mathbf{z}' \\ c^* \\ \mathbf{u}''\end{bmatrix}\right) = c^*.$$

One can get rid of the factor $1/(2\gamma_2)$ by defining $H(\mu\|x) := H'(\mu\|2\gamma_2 \cdot x)$. □

Note that the bound

$$\zeta = \max\{\gamma_1, \beta + 2^{d-1} \cdot \tau + 2\gamma_2 + 1\}$$

of the SelfTargetMSIS solution increased by β in comparison to the original Dilithium version. When picking the parameters as in [2, Table 2], then β is at most 196 while $\gamma_1 \ge 2^{17}$ and $\tau 2^{d-1} + 2\gamma_2 + 1 > 2^{17}$, so the additional β is small in comparison. When estimating the concrete hardness of the SelfTargetMSIS problem using the Dilithium parameters from [2], this change does not lead to different estimated running times for all known attacks.

Theorem 2. *For every PPT adversary $\mathcal{A}$, there is a PPT adversary $\mathcal{B}$ such that the advantage of $\mathcal{A}$ breaking the unlinkability under chosen message attacks of the Dilithium-based key-rerandomizable signature scheme from Figure 4 and Figure 3 is*

$$\Pr\left[\mathrm{Exp}^{\text{ul-cma}}_{\Sigma,\mathcal{A}}(\lambda) = 1\right] \le \mathbf{Adv}^{\mathsf{MLWE}}_{k,\ell,D}(\mathcal{B}),$$

where D is the uniform distribution over S_η.

Proof. The adversary $\mathcal{B}$ gets an MLWE challenge $(\mathbf{A}, \mathbf{t}^*)$ where either $\mathbf{t}^* = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ or $\mathbf{t}^* \xleftarrow{\$} \mathcal{R}_q^k$. $\mathcal{B}$ computes a base key pair $(\mathsf{pk}^b, \mathsf{sk}^b)$ using the normal Gen algorithm. In particular, it can compute $\mathbf{t}^b = \mathbf{t}_1^b \cdot 2^d + \mathbf{t}_0^b$. Next, it computes vk^* using $\mathbf{t}' = \mathbf{t}^b + \mathbf{t}^*$ and runs $\mathcal{A}$ on input $(\mathsf{vk}^b, \mathsf{vk}^*)$. $\mathcal{B}$ can answer all O queries using sk^b.

If $\mathbf{t}^* = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, then $\mathbf{t}'$ is computed like in the real code of Figure 4, i.e., $\mathbf{t}' = \mathbf{t}^b + \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, where $\mathbf{s}_1, \mathbf{s}_2 \xleftarrow{\$} S_\eta^\ell$. However, if $\mathbf{t}^* \mathcal{R}_q^k$, then $\mathbf{t}'$ is uniformly random, i.e., it is computed as in the second case of the experiment.

Thus, $\mathcal{B}$ can output 1 iff $\mathcal{A}$ outputs 1 and wins with the same probability. □

3 System Overview

A group system is deployed between a *central server* and arbitrarily many *users*. The *core functionality* of a group system is to maintain a consistent user list of the current members of each group, distinguishing between regular users and those with admin privileges. This list is managed through a set of *core operations* (cf. Table 2) which allow users to create a new group (Create) and fetch the current version of an existing group (Fetch), as well as admins to add and remove group members (AddMember / RemoveMember) and toggle admin privileges (SetAdminRights). As a prerequisite, users need to set up an account (RegisterUser) and establish "acquaintance" with each other (GrantAcquaintance), ensuring users can only add users they know to a group.

To support additional features and security properties, Signal's private group system further encompasses the following *extended functionality*. Users can refresh their key material (Refresh), used in particular to block other users, and leave a group on their own (Leave). The group settings can be modified by admins (ChangeSettings), to allow regular users to add other users (via UserAdd) or modify the group's attributes, such as name, picture, etc. (ChangeAttr). Admins can activate (or deactivate) a join link (ToggleJoinLink) which allows anyone with this link to join the group (JoinViaLink), possibly requiring approval (ApproveJoin). Finally, admins can invite users they are not acquainted with to a group (Invite) and revoke such an invitation (RevokeInvitation); the invited user can choose to accept or decline such an invitation (AcceptInvitation).

3.1 Threat Model

In a private group system, we deal with a potentially corrupted server, corrupted group members, and even corrupted group administrators. *Privacy of group membership* is required to hold against even a corrupt server, while no privacy can be guaranteed if any member of the group is corrupted and can leak the whole group state. We require *consistency* to hold in the presence of corrupt

Core Functionality	Extended Functionality
RegisterUser (7)	Refresh (12)
GrantAcquaintance (8)	Leave (12)
Create (9)	UserAdd (12)
Fetch (10)	ChangeSettings* (19)
AddMember* (11)	ChangeAttr(*) (12)
RemoveMember* (17)	ToggleJoinLink* (B.8)
SetAdminRights* (B.5)	JoinViaLink (12)
	Invite* (B.7)
	RevokeInvitation* (B.7)
	AcceptInvitation (B.7)
	RotateGroupKey (20)

Table 2: Core functionality of a private group system (left) and extended functionality (right), with numbers in parentheses referring to the figure or appendix defining them. Operations marked with * require admin privileges; those with (*) optionally so, depending on group settings.

group members. A corrupt server can always withhold updates from members, but crucially cannot edit the group state or roll back to previous versions of the group. Consistency is lost in case a corrupt server colludes with a group member, reflecting the fact that a server in possession of the group key material can possibly rewrite the group list arbitrarily. *Anonymity* in group management is required to hold against a corrupt server, but only as long as no corrupt group members leak any information. Admin privileges cannot be faked by corrupt members as long as they do not collude with the server. *Confidentiality of key material* and *authorized addition* are both guaranteed in the presence of a corrupt server.

3.2 Group Data Structure

At the heart of our private group system design is an evolving data structure, stored by the server and updated by group admins and regular group members through the operations described above; see Figure 5 for an illustration of that data structure. For privacy reasons, most of this data structure (e.g., user identifiers and key material) needs to be encrypted to protect it from the server. This data structure can be conceptually split into a *core* part and an *appendix*, as well as an *invite queue*. Only group admins can modify the core part, with each change constituting a new *major version* of the group, while users can add entries to the appendix, with each addition constituting a new *minor version*.

Key rerandomizable signature scheme. While in Signal's current protocol, anonymous credentials are centrally used to privately prove a user's membership in a list (needing to be anonymous so that the identity is not traced between lists), our protocol achieves this task using much simpler primitives. In particular, the only public-key primitive we need is (a relaxation of) a signature scheme with rerandomizable keys [10, 11]. Such schemes can be constructed from Schnorr signatures or ML-DSA / Dilithium (e.g. as proposed in [10], but we use a different tack which we will explain in Section 2) if one wishes to achieve quantum-safety; combining both further enables *hybrid* security if desired.

A rerandomizable signature scheme consists of algorithms

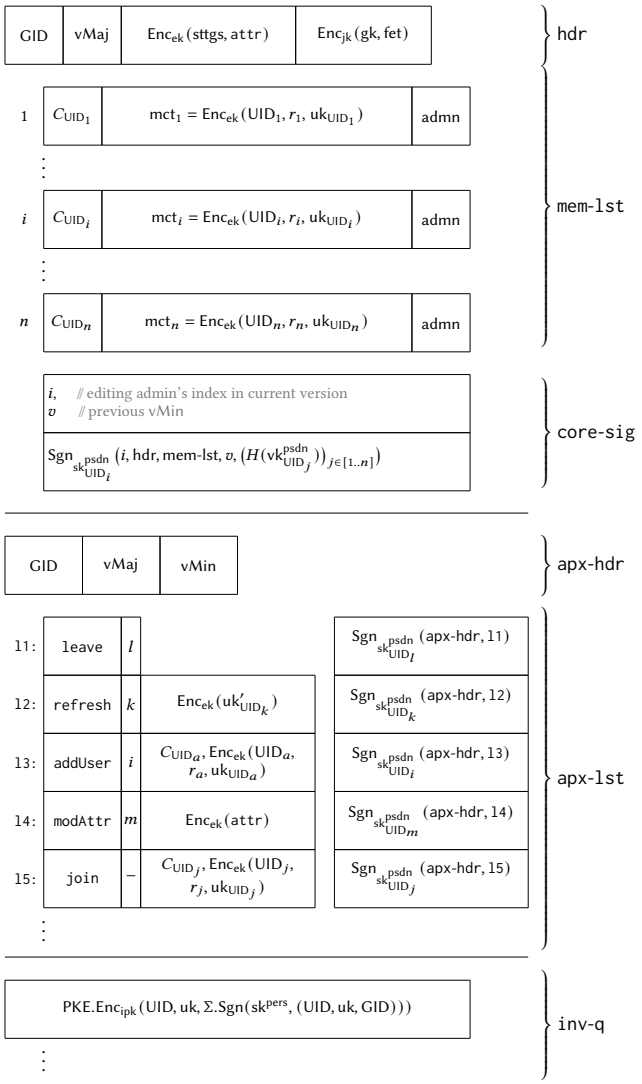

Figure 5: Illustration of the group data structure with core member data (top), appendix (middle), and invite queue (bottom). In the appendix, `1i` denotes the contents of appendix line i without the signature. The server additionally stores the pseudonym verification key $\mathsf{vk}^{\mathsf{psdn}}$ of each member, and a fetching token fet.

(1) $\mathsf{KeyGen}() \to (s, t)$, generating random secret / public "base" values;
(2) $\mathsf{RandSK}(s, \rho) \to sk$, deterministically deriving a signing key from s with randomness ρ;
(3) $\mathsf{RandVK}(t, \rho) \to vk$, deterministically deriving a verification key from t with randomness ρ;
(4) $\mathsf{Sgn}(sk, m) \to \sigma$, signing a message m using sk; and
(5) $\mathsf{Vfy}(vk, m, \sigma) \to 0/1$, verification using vk.

The correctness notion that we need is that (sk, vk) derived from a base (s, t) with the same randomness is a valid key pair for a signature scheme—i.e., $\mathsf{Vfy}(vk, m, \mathsf{Sgn}(sk, m)) = 1$. The privacy notion is that for any two t, t', given oracle access to RandVK and Sgn, when ρ is chosen at random each time, one cannot distinguish whether the transcript was produced when using t or t'. The unforgeability notion states that one cannot forge signatures under any produced vk, even if one knows (and even chooses) ρ.[4]

In a nutshell, we will have users rerandomize their base keys for each group, using them as pseudonym signing keys for which other group members can re-derive the public key. This allows all other members in the group to link operations to the acting user, while the server is merely able to check that a signature verifies under an (unlinkable) pseudonym public key (i.e., is done by a group member).

The core. The core part of our group data structure begins with a *header*, carrying the group identifier GID, current major version, the AEAD-encrypted group settings and attributes (such as group name, group picture, etc.), and an encryption of the group key for use in join links. The group key gk is refreshed with every major version, and communicated out-of-band to added members. A random fetching token fet is sampled for every major version, used as a bearer token towards the server to authorize fetching a group.

The primary component is then the *member list*, which contains a *member entry* per group member with their relevant information: the member's encrypted user identifier UID and user key uk, and a server-visible flag admn marking admin members. It also contains a statistically hiding commitment $C\{\mathsf{UID}\}$ to the user identifier, allowing optional linking to an anonymous credential to implement various policies (e.g., user key recovery or liveness checks; see Section 6 and below for further discussion and details).

With each line, we also associate the group member's pseudonym verification key $\mathsf{vk}^{\mathsf{psdn}}$, which is unique to each group that the user belongs to, and to which only that user knows the corresponding secret key $\mathsf{sk}^{\mathsf{psdn}}$. The pseudonym key can be re-derived by members; to the server, it is given by the user adding the respective member entry. All operations that the user performs in the group will be signed with $\mathsf{sk}^{\mathsf{psdn}}$, which proves to the server that the operation is being done by the group member in the particular line, and makes this user identifiable (only) by other group members.

These keys are generated using the rerandomizable signature scheme. In particular, every Signal user with a unique user identifier UID has a static secret / public pair $(\mathsf{sk}_{\mathsf{base}}, \mathsf{vk}_{\mathsf{base}})$ generated using the KeyGen algorithm. Each group also has a secret rerandomization seed rrs, which is derived from the group key. The $\mathsf{sk}^{\mathsf{psdn}}$ and $\mathsf{vk}^{\mathsf{psdn}}$ keys are *deterministically* generated as $\mathsf{RandSK}(\mathsf{sk}_{\mathsf{base}}, H(\mathsf{UID}, \mathsf{rrs}))$ and $\mathsf{RandVK}(\mathsf{vk}_{\mathsf{base}}, H(\mathsf{UID}, \mathsf{rrs}))$, respectively. In other words, all group members can compute each other's pseudonym public keys. The security properties of the rerandomizable signature imply that all users are unlinkable to the server across groups and that only the user with $\mathsf{sk}^{\mathsf{psdn}}$ can sign under $\mathsf{vk}^{\mathsf{psdn}}$.

The core part of the group data structure may only be edited by group admins, which is ensured by having the editing admin sign it with its pseudonym key in the *core signatures* section. This signature will be checked by the server against the claimed admin's member entry (esp. ensuring that the member indeed has admin privileges), as well as by all members to ensure knowledge of which admin performed the update. Beyond the header and member list, the signature also covers all members' pseudonym keys (ensuring server and users agree on the correct, unique keys), as well as the editing admin's index and the latest previous minor version (to ensure no gaps are introduced).

Jointly, this enables the system's core operations: admins can create groups (Create), add members (AddMember), remove members (RemoveMember), and assign admin privileges (SetAdminRights), and members can fetch the current group state (Fetch).

The appendix. So far, only admins can modify the group list. To enable the extended functionality described above (like allowing regular members to, e.g., leave a group, refresh their user keys, or add users to a group), we introduce an *appendix* to which any regular member can (only) append entries. The appendix has an *appendix header*, carrying the group id and major and minor version. The *appendix list* then consists of entries for one of the following operations, each carrying a signature of the performing user under their pseudonym key:

- `leave`: a member leaving the group.
- `refresh`: a member refreshing their user key, accompanied by the encrypted new key.
- `addUser`: a member adding a new user, accompanied by that user's encrypted member entry.
- `modAttr`: a member modifying the group attributes.
- `join`: a member joining via a join link, accompanied by that user's encrypted member entry.

The appendix should be seen as capturing an intermediate state between core member list updates. That is, once an admin comes online, they will reconcile the operations in the appendix to create a new version of the core member list. The admin who modifies the core list signs the entire list with their pseudonym key, and is thus "responsible" for the correctness of the whole list. Users who append lines to the appendix individually just sign that line, limiting their responsibility (and editing capabilities).

The intuition for splitting the group description into two parts is to prevent regular users from "destroying" the group[5] by continuously rewriting the group list to the point where the server, which perhaps stores several prior versions of the group structure, no longer has any valid backups. While everyone in the group will detect that the list is invalid and even know who made the invalid changes, their only recourse to reconstruct a valid group state is to recover it from their own state. This situation, however, becomes rather messy, and so we believe that avoiding it altogether is a much cleaner and simpler solution. The appendix, being append-only, can contain invalid lines, but they are simply ignored by everyone, and the originator of these lines can be traced back.

[4] The difference with [11] is that we do not necessarily require (s, t) to be a signature key-pair, and we do not need the produced sk to have the same distribution as s. But we do want the signatures not to be traced to the public base t.

[5] We do not consider an admin "destroying" the group to be an attack because an admin can legally simply delete the group.

The join queue. If join requests require approval, joining users enter themselves into a join queue by encrypting their user identifier and user key under a public-key encryption key ipk, and signing the request. Group members hold the corresponding decryption key, allowing them to decide join requests and add approved users.

Optimizing public-key storage. While in the classical setting, the pseudonym verification keys $\mathsf{vk}^{\mathsf{psdn}}$ can be as short as 32 bytes, when using the quantum-safe ML-DSA standard, verification keys can be on the order of several kilobytes. To minimize bandwidth and server storage space, an optimization is to only send the server hashes of $\mathsf{vk}^{\mathsf{psdn}}$ instead of $\mathsf{vk}^{\mathsf{psdn}}$. Then, when signing (for the first time) under this key, the signer will have to send the $\mathsf{vk}^{\mathsf{psdn}}$ together with the signature. The advantage of this approach is that, assuming many users in the list will not be performing many group operations during most epochs, it is wasteful to send the server many $\mathsf{vk}^{\mathsf{psdn}}$ that will probably never be used. This optimization has no disadvantages, and so we would recommend its use in the quantum-safe scenario. We present our scheme with send-and-stored $\mathsf{vk}^{\mathsf{psdn}}$ for simplicity, but we implemented and evaluated the optimized version and confirm the efficiency gains in Section 7.

3.3 Core Group Operations

To give a flavor of how the system works, we now sketch the main group operations for adding and removing members at a very high level. We describe all the group operations in Section 4.

Adding a user to the main list. To add a user, the admin adds a new line to the core list for that user and sends it to the server together with the new user's derived $\mathsf{vk}^{\mathsf{psdn}}$. The admin also signs the entire list under their own pseudonym key. The server checks the validity of the signature and checks that the line corresponding to the key under which it is signed is indeed an admin line allowed to perform this operation. The admin then sends the group identifier and group key material (incl. gk) to the added user over a secure one-to-one channel (in Signal, this is done within the core messaging protocol).

Note that at this point, all the server attests to is that an admin made a change and that they now know the $\mathsf{vk}^{\mathsf{psdn}}$ of the newly added user—nothing about the validity of the other information in the line is known to the server because it is encrypted under the group key gk. Thus, if the ciphertext is malformed, the blame falls on the admin who signed it. When the other users fetch the new line, they check that the encrypted information contains "correct" information about the new user and that the signature is valid.

Removing a user from the main list. Removing a user requires that the group key gk be changed, and the group data structure will need to be re-encrypted under the new key. The admin thus needs to completely rewrite the group list. They will do this by simply removing that user's line and re-encrypting all the others. The admin will also need to let the server know all the $\mathsf{vk}^{\mathsf{psdn}}$ of the users in the group, which have changed since the new group key implies a new rerandomization seed rrs. The admin sends this new group description, signed under their $\mathsf{sk}^{\mathsf{psdn}}$, to the server. When the group members fetch the new version, they verify that all the lines contain "correct" information about the users and that the signature verifies. Note that verifying the signature requires recomputing the $\mathsf{vk}^{\mathsf{psdn}}$ of all the users. While this operation is relatively fast (tens of milliseconds for a few hundred users using classical or ML-DSA signatures), one only needs to perform it after a member has been removed, thus it is also not very frequent.

Adding a user by a non-admin. If a non-admin group member wishes to add another group member (and this operation is permitted as per the group settings), the user performs the exact same action as what an admin does above, except the user's line is not stored in the member list, but in the appendix. This line, together with the $\mathsf{vk}^{\mathsf{psdn}}$ of the new user, is then individually signed.

Leaving the group. A group member wishing to leave the group publishes a leave entry to the appendix, pointing to their member entry in the main list (or in the appendix) and signing using that entry's pseudonym key. Messages are thus no longer sent to this user by others, and when an admin comes online, the user's entry in the main list is removed, and the group key is rotated.

User identifier commitments. The member entry also contains a commitment to the user's identifier. This commitment does not serve a specific purpose in our main protocol, but can be used to implement various mechanisms or policies. In Section 6, we already discussed how these commitments can be used in combination with anonymous credentials to allow users to recover from state/key loss. As another example, one could institute periodic spot checks to make sure the list members are active registered signal users. This could be done similarly to how it is currently implemented, using anonymous credentials that have an expiration date. One could also consider issuing tokens that limit the number of operations a user may perform on the group list if the server observes that some lists generate a lot of server overhead. The important point, however, is that these credentials or tokens are no longer tied to the rest of the system beyond users checking that they correctly commit to the group member's user identifier. This allows migrating to quantum-safe component-wise, a possibility that Signal's current private group system does not afford. When deploying quantum-safe commitments, these could be stored hashed in the group data structure to reduce the space overhead. Notably, the commitments have to be statistically hiding to never interfere with the confidentiality of user identifiers.

Possible optimizations for the quantum-safe setting. Another possible optimization involves a combination of using a key rerandomizable signature scheme whose signatures the server stores locally, and a more efficient (but non-rerandomizable) signature scheme with a static key whose outputs are stored in the group system. The Falcon signature [12], whose signature outputs are roughly 3x smaller than those of ML-DSA, is one such non-rerandomizable possibility. The output of the Falcon signature, of course, cannot be made public to the server (since it will then be able to trace users between groups), but it could be encrypted under the group key. The idea is to include the Falcon signature in the part that is encrypted under the group key, and sign the full update under the ML-DSA $\mathsf{vk}^{\mathsf{psdn}}$. The server would not transmit the ML-DSA signature to the other users, but the other users can still verify the correctness of the update using the Falcon signature. The ML-DSA signature under $\mathsf{vk}^{\mathsf{psdn}}$ would only get used in the case that the Falcon signature

does not properly verify, and the rest of the group members need to know who made this incorrect update. They would then request that the server tells them who it is by showing them the more expensive ML-DSA signature. In this scenario, the server would be locally storing (encryptions) of the Falcon signature in addition to all the ML-DSA signatures, but the ML-DSA signatures would only need to be transmitted in case of "disputes", which are hopefully rare.

On deniability. Compared to Signal's current system, users lose deniability of group membership. That is, another group member can prove to an outsider that this user was in the group (and that they were active).

In the current system, if a member of the group reveals the pseudonym signatures σ^{psdn} of a user together with $\mathsf{vk}_{\mathsf{base}}$ and the group key gk, to an outsider, $(\sigma^{\mathsf{psdn}}, \mathsf{vk}_{\mathsf{base}}, \mathsf{gk}, \texttt{grp-lst})$ proves to the outsider that the user was indeed a member of that group. For this, the outsider can check that σ^{psdn} verifies under $\mathsf{RandVK}(\mathsf{vk}_{\mathsf{base}}, H(\mathsf{UID}, \mathsf{rrs}))$, where rrs is decrypted using gk. In the old system, a group member can also reveal the group list to an outsider. However, there is no proof that the revealed list is correct and not modified. The verifiable encryption of the user's ID, as well as the profile key commitment, could have been computed by anybody knowing the user. This is because in the old system, only the server verifies the correctness of the group list by checking the anonymous credentials. Neither the model of Chase et al. [7] nor our model from Section 5 enforces deniability.

4 A New Private Group System

Let us now detail the protocols forming our new PGS.

Key hierarchies. Our protocols involve a range of user- and group-specific keys, which are derived using a key derivation function KDF in a hierarchical way, illustrated in Figure 6.

Users hold a main user key muk, from which they derive a (per-epoch) user key uk. A user key is shared with acquaintances and other members of a group, used to derive (a) an acquaintance tag acq that allows to validate another user's key, and (b) an exporter key expKey from which further application key material can be derived (e.g., a profile key for encrypting user profile data, or a sending token used to endorse message delivery). The user key (and hence all derived keys) can be refreshed to a new epoch, allowing, for example, acquaintance reset (i.e., blocking) of other users.

Group members share a group key gk, refreshed with every major version, which is used to derive an AEAD encryption key ek for encrypting the group data structure, a seed rrs for deriving rerandomized pseudonym keys for this group, and a PKE keypair (ipk, isk) used for encrypting and decrypting join requests.

4.1 Protocols

All protocols, except for user registration, are assumed to run over a server(-only)-authenticated secure channel, i.e., we assume the server is authenticated, the client is anonymous, and the channel is encrypted. For user registration, we assume a mutually authenticated channel.[6]

[6]Signal, for example, authenticates users via an SMS challenge upon registration.

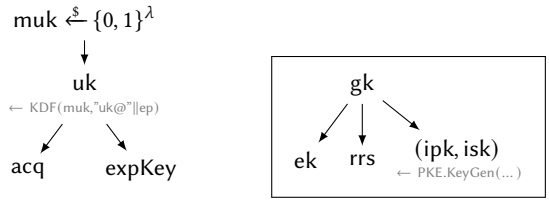

Figure 6: Key hierarchy for user and group keys (boxed)

Server S()		**User** U(UID)
		Ignore if already registered
		$(\mathsf{vk}^{\mathsf{pers}}, \mathsf{sk}^{\mathsf{pers}}) \leftarrow \Sigma.\mathsf{KeyGen}()$
		send $(\mathsf{UID}, \mathsf{vk}^{\mathsf{pers}})$ to $\mathcal{F}_{\mathsf{PKI}}$
		$\mathsf{muk} \xleftarrow{\$} \{0,1\}^\lambda$, $\mathsf{ep} \leftarrow 1$
		$(\mathsf{sk}_{\mathsf{base}}, \mathsf{vk}_{\mathsf{base}}) \leftarrow \Pi.\mathsf{KeyGen}()$
		$\mathsf{uk} \leftarrow \mathsf{KDF}(\mathsf{muk}, \text{"uk@"} \| \mathsf{ep})$
		$\mathsf{acq} \leftarrow \mathsf{KDF}(\mathsf{uk}, \text{"ACQ-Tag"})$
		$\mathsf{expKey} \leftarrow \mathsf{KDF}(\mathsf{uk}, \text{"EXP-Key"})$
	$\xleftarrow{\mathsf{UID}, \mathsf{vk}_{\mathsf{base}}, \mathsf{acq}, \sigma}$	$\sigma \leftarrow \Sigma.\mathsf{Sgn}(\mathsf{sk}^{\mathsf{pers}}, (\mathsf{vk}_{\mathsf{base}}, \mathsf{acq}))$
$\mathsf{ep} \leftarrow 1$	(UID-authenticated)	Store (KeyRec, UID, muk, ep,
Store (Acct, UID,		$\mathsf{vk}^{\mathsf{pers}}, \mathsf{sk}^{\mathsf{pers}}, \mathsf{vk}_{\mathsf{base}}, \mathsf{sk}_{\mathsf{base}}$)
$\mathsf{vk}_{\mathsf{base}}, \mathsf{acq}, \mathsf{ep}, \sigma$)		Output (expKey)

Figure 7: User registration protocol RegisterUser**.**

*User registration (*RegisterUser*).* This protocol, shown in Figure 7, registers a new user with the server. The user identifier UID can be thought of as the user's phone number, under which the user can authenticate to the server. To start registration, the user generates two pairs of signing keys: $(\mathsf{vk}^{\mathsf{pers}}, \mathsf{sk}^{\mathsf{pers}})$ for a regular signature scheme Σ and $(\mathsf{vk}_{\mathsf{base}}, \mathsf{sk}_{\mathsf{base}})$ for a key-rerandomizable signature scheme Π.[7] The user also samples a main user key muk, from which they derive their symmetric key material following the key hierarchy (see Figure 6) for their initial epoch (ep = 1). The acquaintance tag acq and $\mathsf{vk}_{\mathsf{base}}$ are signed with $\mathsf{sk}^{\mathsf{pers}}$. The user deposits their acquaintance tag acq and the signature with the server, transmitted over a UID-authenticated channel.

*Granting acquaintance (*GrantAcquaintance*).* A user U grants acquaintance (via the GrantAcquaintance protocol, Figure 8) to another user U′ by sending them their user identifier UID and key uk. Alternatively, U′ might obtain those from the data structure of a group where both are members. U′ can validate this information by (anonymously) sending UID to the server and obtaining the rerandomization base $\mathsf{vk}_{\mathsf{base}}$ and acquaintance tag acq for U from the server, which it locally checks before storing U as an acquaintance.

*Group creation (*Create*).* Any user can create a new group via the Create protocol shown in Figure 9, which initializes a group data structure (see Section 3.2) holding that user as sole member, with admin privileges. The protocol works as follows, for a user with identifier UID, group description attr, and group settings b_{attr} (whether or not a user can change the group description) and b_{add} (whether or not users can add more users):

- The server picks the next available group id GID.
- The group creator samples a group key gk and a fetching token fet. These keys will protect the privacy of the group.

[7]We assume $\mathsf{vk}^{\mathsf{pers}}$ is available to all users via some PKI ($\mathcal{F}_{\mathsf{PKI}}$); in practice Signal's user verification keys can be reused here.

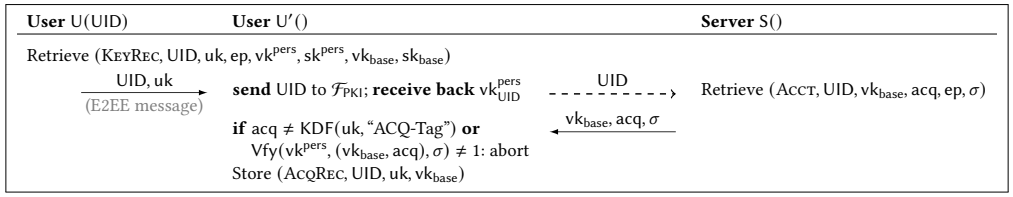

Figure 8: Protocol GrantAcquaintance **where user** U **with user identifier** UID **allows user** U′ **to add them to groups, and** U′ **validates the acquaintance through interaction with the server. Dashed arrows denote a user-anonymous communication.**

- The group creator composes the **header** $\mathtt{hdr} \leftarrow (\mathsf{GID}, \mathsf{vMaj}, c_{\mathsf{hdr}}, \mathsf{sttgs})$ as in Figure 9. The sensitive information in the header, the group description attr and settings $(b_{\mathsf{attr}}, b_{\mathsf{add}})$, is encrypted under the encryption key ek derived from gk.
- The group creator composes the **member list** $\mathtt{mem\text{-}lst} \coloneqq (C_{\mathsf{UID}}, \mathsf{mct}_1, \mathsf{admn} = \mathtt{true})$ as in Figure 9. While our PGS does not further use the commitments C_{UID}, with the commitment randomness stored in mct_1, they can be used for liveness checks (we discuss this in Section 6). The user-related private information, such as UID and the user's key material uk, is encrypted in mct_1, hiding it from the server.
- The group creator signs the group structure with their pseudonym signing key $\mathsf{sk}^{\mathsf{psdn}}$, rerandomized from the creator's base value $\mathsf{sk}_{\mathsf{base}}$ using randomness rrs derived from gk. This allows the server and group members to verify that the data structure was edited by the owner of pseudonym $\mathsf{vk}^{\mathsf{psdn}}$, which was tied to user UID upon registration.
- The group data structure (including the header, the member list, the pseudonym verification key and the core signature) as well as the fetching token are sent to the server. The server checks the pseudonym signature and, if it verifies, stores the group data structure for group id GID.

*Fetching a group (*Fetch*).* Shown in Figure 10 for group GID

- The user presents GID and the current fetching token fet to the server
- If the token matches the stored token for GID, the server sends the group description to the user.
- The user can check consistency of all entries in the member list, done via a subroutine UsrVfyUpdate (given in Figure 15 in Appendix B.1). Essentially, if any mal-formed entry is found, the user can detect this and act accordingly (which, in practice, could mean trying to recover from the error, or as a last resort deleting the whole group). As a result, all honest users have the same view of the group.
- Consistency checking includes:
  - Verifying the header (group id, version numbers progress correctly, flags decrypt).
  - Verifying the admin pseudonym signature, incl. all user's pseudonym verification keys.
  - Decrypting all member ciphertexts using ek without an error.
  - Verifying acquaintance tags of new members.
  - Recomputing and checking the user id commitment for all members.

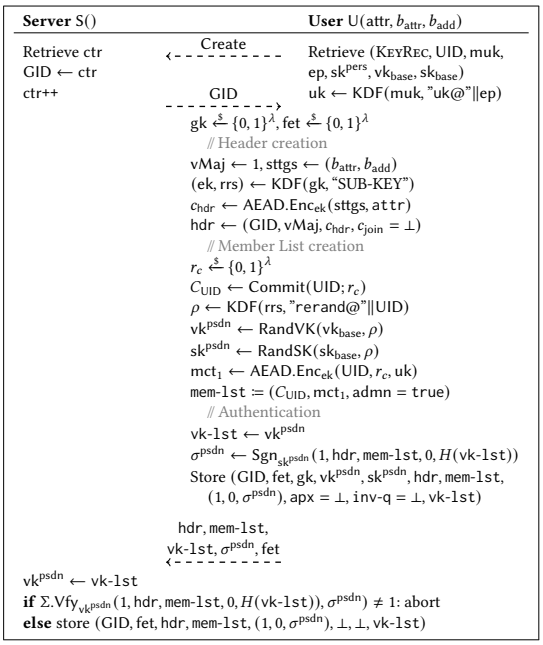

Figure 9: Our protocol Create **for creating a group. Dashed arrows denote user-anonymous communication.**

  - Checking the appendix header (group id, version).
  - Verifying all appendix entries' signatures and processing the operations.

Note that admins and users should fetch the protocol after any modification to the group list they performed. This is to ensure that the server indeed accepted their modifications.

*Adding someone to a group (*AddMember*).* Adding works very similarly to group creation, just that now an additional user entry is added. The server checks the executing user is an admin, verifies that admin's pseudonym signature, and adds the new member's pseudonym verification key to vk-lst. The group key gk and fet are shared with the new group member via 1:1 messages. The protocol is depicted in Figure 11.

*Removing users (*RemoveMember*).* The protocol is similar to AddMember; we hence defer the corresponding Figure 17 to Appendix B.3. The only difference is the way that the server composes the new member list mem-lst and pseudonym key list vk-lst, namely

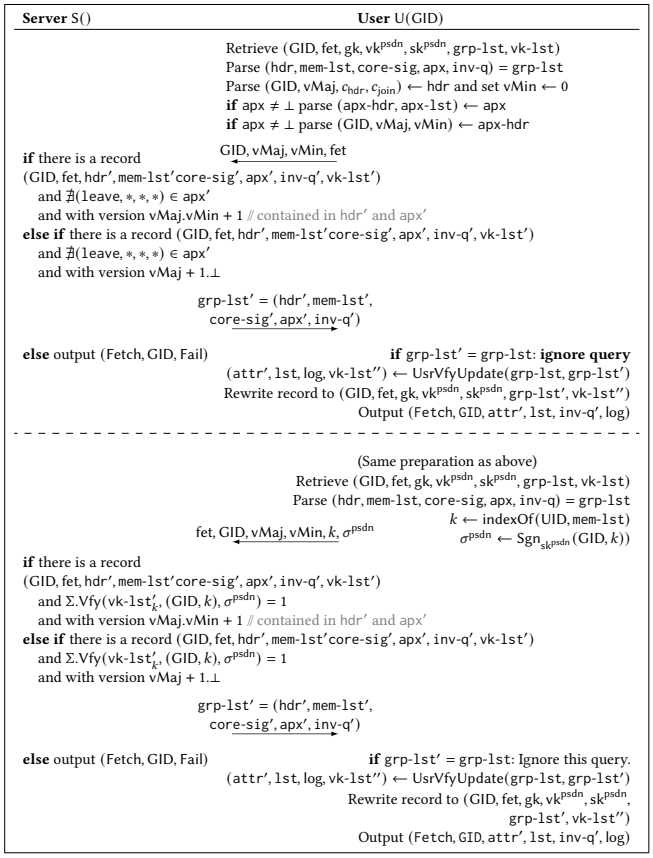

Figure 10: Fetch **for fetching the next update. The upper part shows fetching with a token (only possible before** leave**). The lower part shows fetching with a pseudonym signature (required after** leave**). After obtaining, the user verifies the update with Figure 15.**

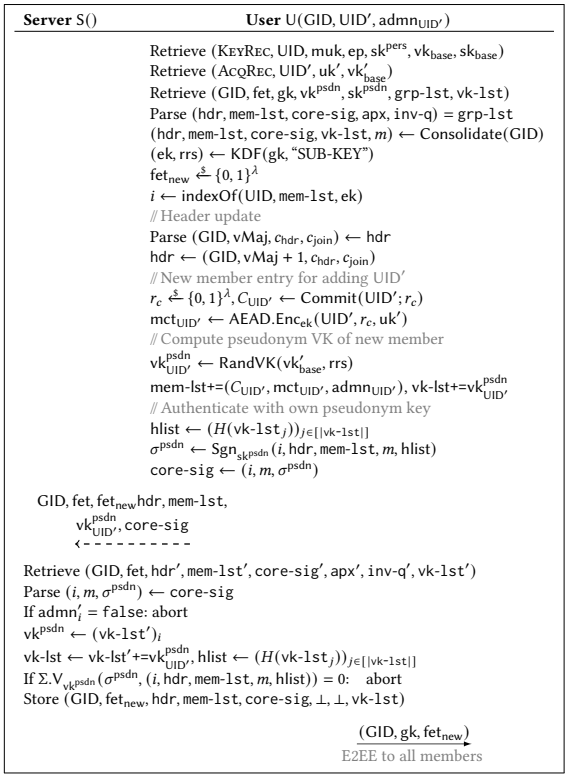

Figure 11: Protocol AddMember **for adding** UID′ **to a group.** indexOf(UID, mem-lst) **returns the row index of** UID **in** mem-lst**. Dashed arrows denote user-anonymous communication.**

by removing the j-th entry instead of appending a new one. Note that the protocol also works for an admin removing themselves.

Modifying the appendix. Non-admin members of the group cannot follow the same procedure as admins since they cannot provide an admin signature. To enable members to still perform non-admin group actions, the group structure contains an append-only appendix that collects group structure updates written by non-admins. There, each added entry is signed individually by the executing member's pseudonym key.

All non-admin initiated protocols work similarly, and hence we can present them all in Figure 12. The user composes the appendix line `line` consisting of the action label, its own index in the membership list, and possibly further encrypted details depending on the operation. When new users join the group, either on their own behalf via JoinViaLink or when users add other users via UserAdd, the corresponding UID commitment has to be added to the line, and the new joiner's pseudonym verification key has to be computed and communicated to the server. Note that for joining, knowledge of the group key is required, and we propose in Appendix B.8 a mechanism for allowing users to derive the group key from a join link. Finally, the user computes a signature of the update under their own pseudonym key. The server obtains the new line, verifies the signature (either against the verification key stored in the membership list, or against the received ones), and stores the updated appendix. We note that a malicious computation of pseudonym verification keys cannot be detected by the server but will be detected by honest users upon fetching the group state. Such invalid entries will simply be discarded by users (see Fetch in Figure 10).

Cleaning up the appendix. When an admin fetches a non-empty appendix, they consolidate all updates into a new major version of the group data structure. For that, the admin first evaluates validity of each appendix line, and writes every valid change to the main list. We give this subroutine Consolidate in Appendix B.4.

Further protocols. We defer the description of all further protocols (incl. setting admin rights, changing group settings, inviting users, handling join links, and key rotation) to Appendix B.

4.2 Comparison of our Private Group System with Signal's Current System

Signal currently deploys a private group system based on anonymous credentials and verifiable deterministic encryption [7]; see Appendix A for details. Our main goal is to propose a new PGS

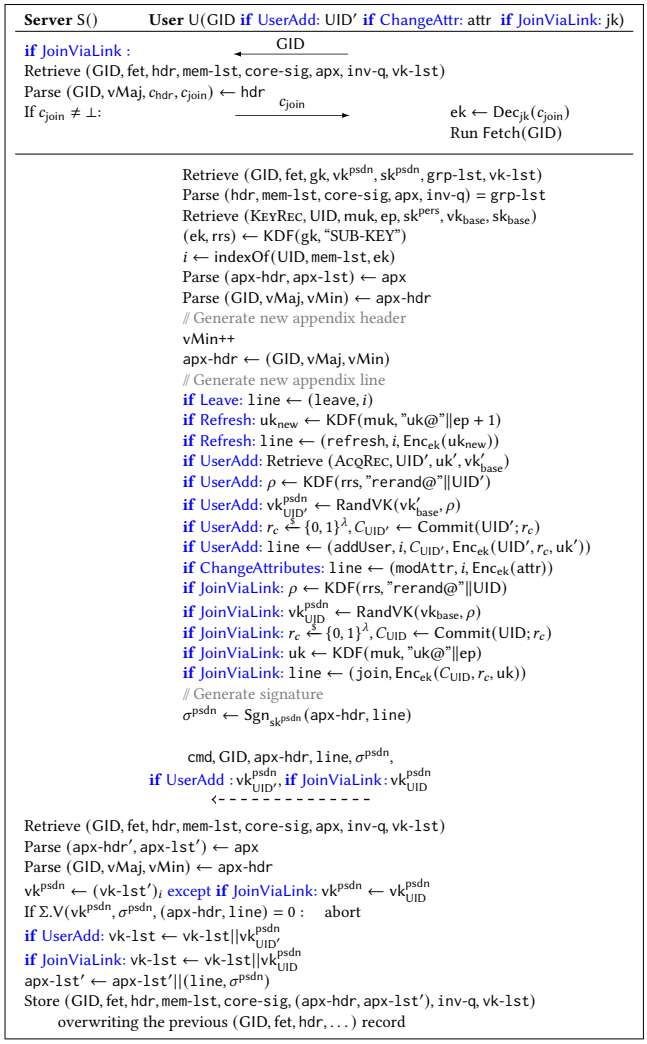

Figure 12: Protocols Leave, Refresh, UserAdd, ChangeAttr, JoinViaLink **for user** U **with identifier** UID **writing into the appendix. Blue lines mark code that is run only in specific protocols.** cmd **stands for the protocol that is run. The lower part shows the** JoinViaLink **protocol.**

that facilitates the transition to a quantum safe system, with immediate harvest-now-decrypt-later protection of all private user (meta-) data. In doing so, we are able to fix other shortcomings of the currently deployed system, but also have to enter some trade-offs. The main differences between Signal's system and our PGS can be summarized as follows:

- *Membership verification:* in Signal, users prove membership using anonymous credentials, such that proofs of membership can only be verified by the server. In our PGS, members use a (rerandomizable) signature key to attest membership to the server *and* to all other group members.
- *Integrity of the group structure:* in Signal, there are no integrity-protecting measures such as MACs or admin signatures. In our PGS, admins sign the whole group structure.
- *Group key rotation:* in Signal, group encryption keys cannot be rotated, while our PGS allows admins to rotate them.
- *Well-formedness of entries:* in Signal, the server checks validity of member entries through verifiable encryption. In our PGS, users need to decrypt to verify validity of an entry.

This results in the following major differences in security guarantees of both systems.

- *Our PGS provides stronger guarantees against a malicious server.* In Signal's current system, the lacking integrity protection and full trust in the server to verify membership allows a malicious server to, e.g., ignore admin privileges and roll back old membership entries. In our system, users would catch such actions when fetching a group update.
- *Our PGS provides stronger guarantees when a malicious server colludes with a former group member.* Lacking group key rotation, in Signal's current system previous group members still posess relevant group key material and can collude with the server in order to perform any group action. In our PGS, admins can prevent this by rotating the group key.
- *Signal's system provides stronger DoS guarantees.* With Signal's current system enforcing users to prove validity of entries through anonymous credentials, the server can prevent the addition of garbage entries, resulting in clean group lists. In our PGS, only users can detect garbage entries and only honest admins can clean up membership lists.

In Appendix D, we give an extended list of security properties of both our and Signal's currently deployed PGS, for a more fine-grained comparison and assessment of entered trade-offs.

5 Security of Private Group Systems

We here summarize our formal security model for a private group system and the security guarantees. As in previous work [7], we use the Universal Composability (UC) [6] framework for this formalization. Due to the differences in our and their system, we depart significantly from the security model of [7], esp. focusing strongly on end-to-end security. For readers familiar with the model of [7] we provide some justification and comparison in Appendix A.2.

On a very high level, we define an ideal functionality $\mathcal{F}_{\text{PGS}}$ that serves as a trusted storage for all of the private groups' state. This includes the group metadata (identifier, attributes, settings), the member list (incl. admin privileges), invites, export key material of all users, as well as a changelog of past group modifications. The core security properties targeted by a private group system are captured by $\mathcal{F}_{\text{PGS}}$ as follows:

- $\mathcal{F}_{\text{PGS}}$ allows access to this state only to group members (capturing *privacy of group memberships*),
- $\mathcal{F}_{\text{PGS}}$ displays the same group state to all members (capturing *consistency*),
- $\mathcal{F}_{\text{PGS}}$ allows group manipulations only according to the group settings, without leaking which user made the modification to the server (capturing *anonymous group management*),

- $\mathcal{F}_{\text{PGS}}$ hands export key material only to users that share a group (capturing *confidentiality of key material*), and
- $\mathcal{F}_{\text{PGS}}$ enforces that only acquainted users can be added to a group (capturing *authorized addition*).

Our functionality reflects what is currently implemented by Signal. For example, by default, only admins can add, invite, and remove users, and everybody can always leave the group. $\mathcal{F}_{\text{PGS}}$ also allows admins to configure groups such that, e.g., a group can be joined via a link, users can change the group attributes, or users are allowed to add other users. We now sketch the functionality's behavior for some example protocols, Create, AddMember, RemoveMember, and UserAdd; the full description is deferred to Appendix C.

Group creation. To create a group, any previously registered user calls a Create interface, providing group attributes attr and initial settings b_{attr} (whether users can change the group attributes) and b_{add} (whether users are allowed to add more users). $\mathcal{F}_{\text{PGS}}$ stores the new group in an object $T_{\text{G}}[\text{GID}][1]$, where 1 indicates the version of the group. The group gets assigned a unique identifier GID.

Adding and removing users. After the creation of a group, $\mathcal{F}_{\text{PGS}}$ allows admins to add and eventually remove group members via interfaces AddMember and RemoveMember. Every such modification of the group state increases the version number, and $\mathcal{F}_{\text{PGS}}$ will maintain all versions of a group. This is done by first copying the whole group state into $T_{\text{G}}[\text{GID}][\text{ver}+1]$ and then performing the action in the list of members $T_{\text{G}}[\text{GID}][\text{ver}+1].\text{members}$. $\mathcal{F}_{\text{PGS}}$ enforces that these actions can only be performed by admins by ensuring that the calling user's UID is in the admins list of the current group version $T_{\text{G}}[\text{GID}][\text{ver}]$.

Non-admin adding of group members is done via a UserAdd interface. This interface works very similarly to the AddMember interface but allows the request only if the group settings allow users to add members.

Leakage. The ideal functionality formalizes the leakage that a private group system has. While we demand strong privacy properties against non-members and the server storing the group data, we cannot expect to protect a group's privacy in the presence of corrupt group members. Hence, $\mathcal{F}_{\text{PGS}}$ has several levels of leakage depending on server and member corruption. In a nutshell:

- If the server and all group members are honest, only the type of query is leaked to the adversary (accounting for leakage through communication size and patterns).
- For a corrupt server but honest group members, $\mathcal{F}_{\text{PGS}}$ additionally leaks the touched group's identifier GID and the indices of the involved users to the adversary. The group id is a mere administrative handle. The indices of the involved users (e.g., "The 3rd user who joined removed the user who joined 8th") are an artifact of our (as well as Signal's current) protocol that stores an encrypted entry per user. Crucially, these indices do not reveal the identity of the involved users, nor do they allow for linking users across groups.
- When a group member is corrupt, $\mathcal{F}_{\text{PGS}}$ reveals the full query, including involved UIDs, to the adversary. Since group changes are visible to all members of the group, this leakage is unavoidable.

Any non-leakage remains protected. Crucially, this means $\mathcal{F}_{\text{PGS}}$ guarantees privacy of group membership (i.e., UIDs) towards a corrupt server.

Out of scope. With group messaging being a much more complex task than just maintaining membership lists, we have to carefully pick the boundaries of a private group system. In general, we do not consider the sending of messages as part of PGS, but let a PGS export symmetric key material tied to group membership to the application in order to allow the derivation of anonymized sending tokens. Some features related to group membership such as a user blocking a group are application-level features that can be handled at the client level. They do not seem to require any cryptography, and hence we do not consider them a part of PGS.

5.1 Security Analysis

We provide a security analysis of our private group system in Appendix E. In brief, we argue how our design realizes the ideal $\mathcal{F}_{\text{PGS}}$ functionality describing corresponding simulators. For this, we model hash functions and KDFs as random oracles (RO) and the authenticated encryption scheme as an ideal authenticated cipher (cf. [14]); further, we assume the commitment scheme to be equivocable and extractable, the signature to be unforgeable, and the key rerandomizable signature scheme to be unforgeable under rerandomized keys and unlinkable under chosen-message attacks (cf. Definition 3 and Definition 2).

6 Deployment Considerations

In the following, we discuss a range of deployment considerations and how our design maps to advanced features of Signal's architecture.

6.1 Migration

Migrating from one system to another in real time requires having both operational and making sure that changes are synchronized. With our design, the only actors that can write to the new system are users, so they will be required to do the synchronization. During the transition period, non-administrator users will attempt to write each update to both systems, and administrators will reconcile any differences within the new system. Signal administrators will be able to track when all users have migrated to the new system, and can then update code to only use the new system safely.

Future migration. While the initial migration from Signal's current system to our proposed system will be challenging, this system is designed to make future transitions toward full quantum safety straightforward. An initial deployment can use efficient primitives with HNDL confidentiality and classical authentication. When quantum-safe authentication becomes a more pressing concern, our system is designed so that each authentication component has a reasonably efficient quantum-safe alternative. Signatures can be replaced with standardized quantum-safe signatures. Commitments, credentials, and zero-knowledge proofs have feasible lattice-based analogues (unlike the verifiable encryption used by Signal today). The Discrete Log-based authorization used for adding pseudonym keys has a Dilithium-based replacement. In each case, there is a clear path forward that will not require redesign of the full system.

6.2 User Re-registration

While authentication through rerandomizable signatures will work for the large majority of cases, there is an important scenario where a different approach is needed: user re-registration. This can happen when, e.g., a user loses their phone and recovers their account on a new device. The user will not have any cryptographic keys from the old device and will register new keys with the service. At this point, that user will be unable to authenticate with our system. To handle this scenario, anonymous credentials can be used in combination with the user id commitments to allow a client to prove to the server that it holds a credential issued by the server corresponding to a line in the group roster, allowing it to perform a `refresh` operation. Further, it could encrypt a certificate from the server that its new long-term keys are correctly registered for its user ID in the `refresh` line it adds to the appendix.

6.3 Abuse Mitigation

Rate-limiting. Signal already has an anonymous credential-based system for rate-limiting resource creation. Clients connect over an authenticated connection to obtain a credential authorizing them to perform an action, such as creating a call link. Later, they connect over an unauthenticated connection, present the credential, and perform the action. We can use this mechanism to limit group creation and to limit non-administrator additions to the appendix.

Storage abuse. The appendix might be abused as free cloud storage. However, the storage capacity available per group is quite small (esp. when comparing to, e.g., message attachments), making this economically unattractive, particularly if gated by a rate-limiting process. A conceivable option to control this directly would be to place a strict cap on the size of an appendix. This would, however, have limited effect, since an attacker could simply create multiple groups. It would also have a clear negative effect on usability, since in groups with a full appendix, non-admin users would be unable to perform actions until an administrator cleans up the group state.

Denial of service and spam. Malicious group members could attempt to spam the appendix with nonsense entries or flood the system with requests. Our design ensures that users can reliably identify which group member is misbehaving (through authenticated appendix entries). This allows the development of product features to enable community members to take appropriate measures—such as notifying users that their clients are malfunctioning or removing problematic members from the group.

6.4 Message Delivery and Sending

This work focused on metadata protection for group management, but metadata can also be leaked by message delivery. Signal's Sealed Sender protocol protects the social graph by hiding sender identity from the server, but requires authorization tokens or anonymous credentials to prevent abuse. The credential system Signal uses for this today, Group Send Endorsements [23], does not have a feasible quantum-safe analogue, so we must rely on symmetric-key based tokens. In our system, these send tokens can be derived from the exported application key material (expKey), which in turn is derived from each user's key uk. Since our model guarantees that only group members learn other members' latest user keys, this provides a natural authorization mechanism.

However, Sender Keys—Signal's current approach for efficient group messaging—is fundamentally at odds with social graph protection, as it requires revealing group membership structure to the server. An ideal system would use existing 1:1 channels with Sealed Sender or Anonymity Wrapper [24] for message delivery, potentially with timing jitter to obscure message patterns. Today, for usability and scalability, Signal makes a compromise using Sender Keys with a variant of Sealed Sender, trusting the server not to log delivery patterns. Improvements to this trade-off would be an interesting direction for future work.

6.5 Handling Inactive Users

If all members of a group deregister their accounts, the group becomes unusable. Server administrators want the ability to delete such groups, but due to the metadata protection built into the private group system, they have no way of identifying them. This is a problem in Signal's system today.

We propose two broad strategies for mitigating this. The first strategy ensures that the group is populated by registered users of the service, but leaks some information to the server about which group members are active (information that is leaked by Signal's service today). The second leaks less information and is more lightweight, but offers no assurance that group members are still registered with the service. A mix of both approaches is possible and allows for a tunable trade-off.

Credential-based check-in. We can have clients regularly (e.g., every 30 days) check in by authenticating against their user identifier commitment and have the server record the time of check-in at low granularity. This ensures not only that the group member is active, but also that they have a valid credential and thus they are, at least recently, a registered user of the service. The server can store the last check-in time per user, per group, or store "last administrator check-in" and "last member check-in" separately. The service can then use this information to set a policy for group maintenance. For example, they could set a policy that any group with no check-in in the last 90 days will be deleted. Similarly, administrators may wish to delete groups where no administrators have been active for an extended time period, as this system requires active administrators to clean up the appendix. With more fine-grained information about last check-in time, informative warnings could be presented to group members.

Token- or signature-based check-in. More lightweight check-ins can be performed using the symmetric key primitives already used for most authorization in this system. In fact, each action, such as fetching a group, can be recorded as a check-in event. This would have many of the features of the credential-based check-in above, but provides no assurance that group participants are still registered users of the service. They could, e.g., build a third-party application using the service infrastructure without registered accounts.

Atomic operations vs batching. We chose to present our PGS in terms of protocols for *atomic* update operations. That is, adding two users works by adding one user first and then performing the same operation again for the second user. Moreover, these atomic

updates must be fetched individually. This choice is to optimize for cleaner protocol presentation and analysis, and Section 7 demonstrates that group operations and fetching are performant enough to execute them even multiple times. On the other hand, it is also relatively straightforward to batch our PGS, both for group updates and fetching, by changing multiple group list entries within one update, and by downloading several versions from the server and aggregating all changes locally.

6.6 Application-level Features

Some features are better implemented at the application level rather than within the group system.

Blocking a group. Signal's current implementation handles group blocking outside the cryptographic group system. When a user blocks a group, the client leaves the group, sets a local flag to ignore future messages for that group, and rotates the user's profile key. This general approach remains appropriate for our system, with users rotating their uk after leaving the group.

Verifying registration. Checking account existence (via credentials and the user identifier commitments) allows the server to verify that group members are registered with the service, and as described above, this can be used to identify inactive groups or groups with inactive administrators. While our protocol provides this affordance, it is an application level decision when and how this should be used.

7 Experimental Evaluation

We benchmark three implementations of the private group system: the deployed `zkgroup` [22] baseline, our EC construction using HNDL-secure rerandomizable signatures over Ristretto255, and our PQ construction using rerandomizable ML-DSA-65 signatures. We measure the four core group operations—Create, AddMember, RemoveMember, and Fetch— recording client request preparation time, server processing time, client response processing time, and wire size. Group sizes range over $n \in \{2, 5, 10, 50, 100, 500, 1000\}$, covering Signal's production group sizes. Relative to `zkgroup`, EC adds HNDL security, and PQ provides full post-quantum security.

We make one deviation from Signal's current deployment: Signal does not rotate the group key on member removal. Our RemoveMember benchmark therefore measures a stricter variant in which the remaining member entries are re-encrypted under a fresh group key.

Methodology. All measurements were collected on an Intel(R) Core(TM) i9-14900K (32 threads) running Rust 1.88. The EC scheme uses `curve25519-dalek` [8] for Ristretto255 arithmetic. The PQ scheme builds on the unoptimized PQClean [21] ML-DSA-65 implementation; an optimized implementation would improve PQ performance but would not change the qualitative conclusions. Each benchmark phase is measured with Linux `perf_event` `CPU_CYCLES`, excluding kernel and hypervisor activity, and we report per-phase cycle counts averaged over repeated runs. The harness also logs elapsed time with Rust's `std::time::Instant` as a sanity check, but all tables and figures below report cycles rather than wall-clock latency. The process is pinned to a single core while leaving the default scheduler, turbo setting (enabled (unchanged)), and CPU governor (powersave on cpu2 (unchanged)) unchanged. Each phase is averaged over 400 iterations for $n \leq 50$, 200 for $n = 100$, 120 for $n = 500$, and 80 for $n = 1{,}000$.

Table 3: Per-operation performance at $n = 10$. Computation is in Mcycles; wire is total request-response bytes.

Operation	System	Client prep (Mcycles)	Server (Mcycles)	Client resp (Mcycles)	Total (Mcycles)	Wire (bytes)
Create	EC	3.64	0.21	—	3.85	1,978
	PQ	15.59	0.52	—	16.11	7,143
	zkgroup	64.28	34.25	—	98.53	7,843
Add Member	EC	0.53	0.21	—	0.73	469
	PQ	7.99	0.53	—	8.52	7,554
	zkgroup	12.44	6.41	—	18.86	1,354
Remove Member	EC	6.46	0.41	—	6.86	1,953
	PQ	25.46	1.08	—	26.54	12,283
	zkgroup	58.46	31.15	—	89.61	7,122
Fetch	EC	—	—	3.80	3.80	1,658
	PQ	—	—	9.11	9.11	4,903
	zkgroup	5.89	5.85	21.21	32.95	2,342

Table 4: Per-operation performance at $n = 50$.

Operation	System	Client prep (Mcycles)	Server (Mcycles)	Client resp (Mcycles)	Total (Mcycles)	Wire (bytes)
Create	EC	17.91	0.25	—	18.15	8,898
	PQ	49.83	0.60	—	50.43	14,063
	zkgroup	302.11	157.99	—	460.10	36,683
Add Member	EC	0.57	0.25	—	0.82	469
	PQ	8.85	0.61	—	9.46	7,554
	zkgroup	12.45	6.41	—	18.86	1,354
Remove Member	EC	32.96	0.49	—	33.45	8,873
	PQ	73.50	1.31	—	74.82	19,203
	zkgroup	290.15	155.00	—	445.15	35,962
Fetch	EC	—	—	18.00	18.00	7,298
	PQ	—	—	42.56	42.56	10,543
	zkgroup	5.89	12.39	103.26	121.54	8,822

7.1 Results

Table 3 and Table 4 show per-operation performance for groups of 10 and 50 members, respectively, representative of typical Signal groups. We report the four client/server computation phases and total request-response wire size. Mutating operations perform no client work after receiving the server response.

At both $n = 10$ and $n = 50$, the two new constructions use fewer cycles than `zkgroup` on all four operations. At $n = 50$, for example, Create consumes 18.15 Mcycles (EC), 50.43 Mcycles (PQ), and 460.10 Mcycles (`zkgroup`); RemoveMember consumes 33.45 Mcycles, 74.82 Mcycles, and 445.15 Mcycles; and Fetch consumes 18.00 Mcycles, 42.56 Mcycles, and 121.54 Mcycles. Across these sizes, EC is 7–26× faster than `zkgroup`, and PQ is 2–9× faster.

Server cost. The main computational difference is where validation happens. In `zkgroup`, the server verifies zero-knowledge proofs, accounting for about 35% of total cycles in mutating operations (for example, 157.99 Mcycles of 460.10 Mcycles for Create at $n = 50$).

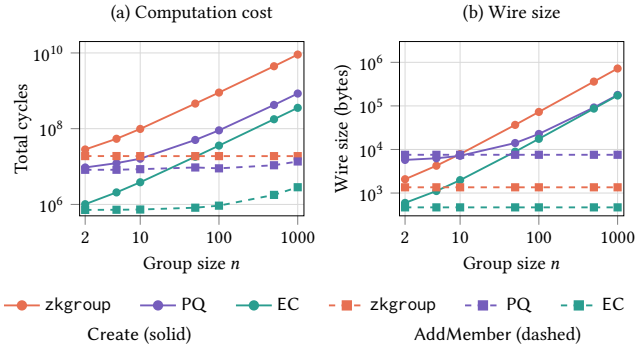

Figure 13: Scaling of Create **(solid) and** AddMember **(dashed) from $n = 2$ to 1,000. (a) Total computation in cycles. (b) Total wire size in bytes. Both axes are log-scaled.** RemoveMember **closely tracks** Create**; Fetch is also linear.**

In the new schemes, the server mainly verifies a signature and stores the update: at $n = 50$, server cost for Create cryptographic operations is 0.25 Mcycles for EC and 0.60 Mcycles for PQ. The corresponding client-side validation cost appears during fetching.

Wire size. EC reduces mutation wire size by about 75% relative to zkgroup. With the optimized representation, PQ Create and RemoveMember are also smaller than zkgroup (for example, 13.73 KiB vs. 35.82 KiB for Create at $n = 50$), while AddMember carries a fixed-size ML-DSA signature. The largest message we measure is zkgroup Create at $n = 1{,}000$, approximately 704.72 KiB; the largest message in the new schemes is PQ RemoveMember at 179.25 KiB. For Fetch, the stored per-member representation is comparable across the three systems.

7.2 Scaling

Figure 13 shows scaling behavior across group sizes from $n = 2$ to $n = 1{,}000$ for Create and AddMember, which capture the two distinct scaling patterns in the system. At Signal's 1,000-member group-size limit, Create takes 9.06 billion cycles in zkgroup, 355.55 million cycles in EC, and 847.07 million cycles in PQ. The server-side component is 3.10 billion cycles in zkgroup, compared with 1.23 million cycles in EC and 2.51 million cycles in PQ.

Create, RemoveMember, and Fetch all scale linearly in group size across the three schemes. For Create and RemoveMember, the EC advantage is stable over the measured range, at about 25× and 12–14×, respectively. The PQ advantage grows with group size, reaching about 10× for Create and 7× for RemoveMember near Signal's 1,000-member limit. For Fetch, the fixed cost of the zkgroup credential presentation is visible at small groups, but the large-group behavior settles near 6× for EC and 3× for PQ.

Fetch. Fetch reflects the same shift in validation more directly. In zkgroup, the server verifies an authorization-credential presentation and issues n Sealed Sender authorization passes, after which the client decrypts the passes and verifies the issuance proof. In our system, the server stores and returns the group state, while the client recomputes n pseudonym verification keys, checks n commitments, decrypts n AEAD ciphertexts, and verifies one signature. This makes fetch validation more client-heavy, but removes the server from the validation path.

AddMember. AddMember behaves differently from the above operations. The zkgroup implementation is $O(1)$ in group size (~18.87 million cycles regardless of n), as it only validates two credentials and proves the new member's ciphertexts are valid. Both new schemes exhibit $O(n)$ scaling because the covering signature serializes the full member list and hashes all $n+1$ pseudonym verification keys. The constants are low enough that EC and PQ remain below zkgroup throughout Signal's supported range. The PQ cost is dominated by a single signing operation in our unoptimized ML-DSA implementation, so optimized ML-DSA code should further reduce this gap.

7.3 Limitations

The PQ construction uses the PQClean reference implementation of ML-DSA-65 rather than an optimized AVX2 or AArch64 implementation, so its absolute numbers are conservative. The reported server costs include only cryptographic work and exclude database I/O, serialization, and replication costs, which all three systems would incur in deployment. We benchmark only the four core group-management operations: in our design, anonymous credentials are reserved for liveness checks and edge cases rather than the hot path, and we do not measure those auxiliary flows, non-admin operations, or appendix consolidation.

References

[1] Martin R. Albrecht and Kamil Doruk Gür. 2024. Verifiable Oblivious Pseudorandom Functions from Lattices: Practical-Ish and Thresholdisable. In *ASIACRYPT 2024, Part IV (LNCS, Vol. 15487)*, Kai-Min Chung and Yu Sasaki (Eds.). Springer, Singapore, 205–237. doi:10.1007/978-981-96-0894-2_7

[2] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2021. CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1). https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf

[3] Ward Beullens, Lucas Dodgson, Sebastian H. Faller, and Julia Hesse. 2025. The 2Hash OPRF Framework and Efficient Post-quantum Instantiations. In *EUROCRYPT 2025, Part VIII (LNCS, Vol. 15608)*, Serge Fehr and Pierre-Alain Fouque (Eds.). Springer, Cham, 332–362. doi:10.1007/978-3-031-91101-9_12

[4] Jonathan Bootle, Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Alessandro Sorniotti. 2023. A Framework for Practical Anonymous Credentials from Lattices. In *CRYPTO 2023, Part II (LNCS, Vol. 14082)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, Cham, 384–417. doi:10.1007/978-3-031-38545-2_13

[5] Vitalik Buterin. 2013. Deterministic Wallets, Their Advantages and their Understated Flaws. https://bitcoinmagazine.com/technical/deterministic-wallets-advantages-flaw-1385450276. https://bitcoinmagazine.com/technical/deterministic-wallets-advantages-flaw-1385450276

[6] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*. IEEE Computer Society Press, 136–145. doi:10.1109/SFCS.2001.959888

[7] Melissa Chase, Trevor Perrin, and Greg Zaverucha. 2020. The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 1445–1459. doi:10.1145/3372297.3417887

[8] dalek cryptography. 2026. curve25519-dalek. https://github.com/dalek-cryptography/curve25519-dalek. https://github.com/dalek-cryptography/curve25519-dalek

[9] Poulami Das, Sebastian Faust, and Julian Loss. 2019. A Formal Treatment of Deterministic Wallets. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 651–668. doi:10.1145/3319535.3354236

[10] Edward Eaton, Douglas Stebila, and Roy Stracovsky. 2021. Post-quantum Key-Blinding for Authentication in Anonymity Networks. In *LATINCRYPT 2021 (LNCS, Vol. 12912)*, Patrick Longa and Carla Ràfols (Eds.). Springer, Cham, 67–87. doi:10.1007/978-3-030-88238-9_4

[11] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. 2016. Efficient Unlinkable Sanitizable Signatures from Signatures with Re-randomizable Keys. In *PKC 2016, Part I (LNCS, Vol. 9614)*, Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang (Eds.). Springer, Berlin, Heidelberg, 301–330. doi:10.1007/978-3-662-49384-7_12

[12] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2020. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. https://falcon-sign.info/falcon.pdf.

[13] Camille François and Thomas Rid. 2024. At Signal, A Revolution in Messaging. Lawfare, https://www.lawfaremedia.org/article/at-signal-a-revolution-in-messaging. Accessed 29 April 2026..

[14] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks. In *EUROCRYPT 2018, Part III (LNCS, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Cham, 456–486. doi:10.1007/978-3-319-78372-7_15

[15] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. 2018. A Concrete Treatment of Fiat-Shamir Signatures in the Quantum Random-Oracle Model. In *EUROCRYPT 2018, Part III (LNCS, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Cham, 552–586. doi:10.1007/978-3-319-78372-7_18

[16] Vadim Lyubashevsky. 2009. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In *ASIACRYPT 2009 (LNCS, Vol. 5912)*, Mitsuru Matsui (Ed.). Springer, Berlin, Heidelberg, 598–616. doi:10.1007/978-3-642-10366-7_35

[17] Vadim Lyubashevsky. 2024. Basic Lattice Cryptography: The concepts behind Kyber (ML-KEM) and Dilithium (ML-DSA). https://eprint.iacr.org/2024/1287.pdf. https://eprint.iacr.org/2024/1287.pdf

[18] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. 2022. Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General. In *CRYPTO 2022, Part II (LNCS, Vol. 13508)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, Cham, 71–101. doi:10.1007/978-3-031-15979-4_3

[19] Jim Oleary. 2019. Technology Preview: Signal Private Group System. https://signal.org/blog/signal-private-group-system/. https://signal.org/blog/signal-private-group-system/ Accessed 05 October 2025..

[20] David Pointcheval and Jacques Stern. 1996. Security Proofs for Signature Schemes. In *EUROCRYPT'96 (LNCS, Vol. 1070)*, Ueli M. Maurer (Ed.). Springer, Berlin, Heidelberg, 387–398. doi:10.1007/3-540-68339-9_33

[21] PQClean Project. 2026. PQClean: Clean, portable and tested implementations of post-quantum cryptography. https://github.com/PQClean/PQClean. https://github.com/PQClean/PQClean

[22] Signal Messenger, LLC. 2026. libsignal. https://github.com/signalapp/libsignal.

[23] Signal Messenger, LLC. 2026. libsignal: Zero-Knowledge Credential Endorsements Implementation. https://github.com/signalapp/libsignal/blob/main/rust/zkcredential/src/endorsements.rs.

[24] Lea Thiemt, Paul Rösler, Alexander Bienstock, Rolfe Schmidt, and Yevgeniy Dodis. 2025. Generic Anonymity Wrapper for Messaging Protocols. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier (Eds.). ACM, 350–364. doi:10.1145/3719027.3765186

[25] Bitcoin Wiki. 2013. BIP 0032. https://en.bitcoin.it/wiki/BIP_0032. https://en.bitcoin.it/wiki/BIP_0032

A Signal's Currently Deployed System

Signal currently deploys a group system with strong privacy guarantees. The core of their protocol is the private group system of Chase, Perrin, and Zaverucha [7], which uses specialized anonymous credentials and verifiable encryption. Signal further augmented their protocol, which we refer to as the CPZ protocol herein, with features like join links and displaying the changelog.

A.1 The Private Group System of Chase et al. [7]

Chase et al. provide a private group system with the main security goals being that a malicious server cannot learn group membership, and cannot add new users to a group, despite serving as an availability layer storing membership lists of all Signal groups. Membership lists contain encrypted user IDs UID under a symmetric group key ek as well as profile information encrypted under a user's profile key prK, which is also stored encrypted in the list. More formally, a user entry is of the format

$$\mathsf{Enc}_{\mathsf{ek}}(\mathsf{UID}), \mathsf{Enc}_{\mathsf{ek}}(\mathsf{prK}), \mathsf{Enc}_{\mathsf{prK}}(\mathsf{info}), \mathsf{admn}$$

To modify the group state, a user claims "ownership" of an entry by showing an anonymous credential for UID to the server. If successful, the server then allows actions such as adding or removing entries, depending on the role admn of the user. Writing new entries is enforced to happen in a consistent way, namely that (a) ciphertexts were computed correctly, and (b) the admin knows the profile key prK of the added user. The latter requires the admin to show a second anonymous credential for UID, prK to the server. This latter credential is issued blindly on prK by the server in order to keep user profile information private.

Our terminology	Terminology of [7]
RegisterUser	GetAuthCred & CommitToProfileKey
GrantAcquaintance	GetProfileCredential
Create	CreateGroup
– (integrated below)	AuthAsMember
Fetch	FetchGroupMembers
AddMember	AddGroupMember
RemoveMember & Leave	DeleteGroupMember
SetAdminRights	-
Refresh	UpdateProfileKey
UserAdd	AddGroupMember
ChangeSettings	-
ChangeAttr	-
ToggleJoinLink	-
JoinViaLink	-
Invite & AcceptInvitation	InviteToGroup
RevokeInvitation	-
RotateGroupKey	-
-	DeleteGroup

Table 5: Mapping between [7] terminology and ours

To ease the linking between our work and the CPZ system, we match the terminology of both in Table 5. The registration and acquaintance phase of the current system consists of setting up the user's key material and having the server issue the anonymous credentials. The server then acts as a policy enforcer by verifying group membership before allowing any group action (sub-protocol AuthAsMember). Adding or removing members then works by creating or deleting list entries. The server allows users to refresh their profile key by letting them edit the corresponding parts of the entry they proved UID-ownership of. Inviting users works by writing only a UID ciphertext into an entry, and having the invited user complete the entry with their profile key and information after proving UID-ownership to the server.

A.2 Comparison with and Remarks on the Model of Chase et al. [7]

Chase et al. provide the first formal security model for private group systems. Their functionality, however, deviates significantly from ours. The most notable syntactical difference is that [7] provide two sets of interfaces, one set for the case of an honest server, and one set for the case of a corrupt server. Our $\mathcal{F}_{\mathrm{PGS}}$ only has one

set of interfaces whose behavior changes depending on the server corruption state. For $\mathcal{F}_{\text{PGS}}$, this works well because the difference between what $\mathcal{F}_{\text{PGS}}$ guarantees in the presence of a corrupt and an honest server is relatively small, while in [7] the gap is larger.

Another reason why we significantly deviate from their model is that a strong focus of our work is end-to-end security, i.e., providing guarantees in the presence of a corrupt server. If the server is corrupt, the functionality of Chase et al. does not store any group information anymore, but instead lets the adversary determine the output towards the user. This is a weaker model than our $\mathcal{F}_{\text{PGS}}$, where a corrupt server alone cannot alter the group state. The only power a corrupt server has in our model is to withhold the latest updates from users who fetch the group. This strengthening is important as one would not want a corrupt server to, e.g., add itself to arbitrary groups, as this would immediately void all privacy guarantees of the PGS. We note that also the *protocol* proposed by Chase et al. has certain guarantees against a malicious server in isolation (e.g., full privacy and some integrity, see Appendix A for details), so the gap between their protocol and their security definition is quite substantial.

A.3 Signal's Current Private Group System

We now describe how Signal augments the CPZ protocol suite to yield a functionality comparable to the system described in this work, for example the protocols in Table 5. Changing an admin flag, changing group settings, and changing group attributes are implemented much like adding users in the CPZ system: a user anonymously authenticates to the server by proving to hold a credential for a UID encrypted in a specific membership list entry. The server then allows the action depending on the group's access-control policy (stored in sttgs of the group) and the admn flag in that list entry. Unlike CPZ, Signal's deployed system derives separate verifiable encryption key pairs for encrypting UID and prK; we denote these ekid and ekpr respectively.

Symmetric attribute encryption. On top of the CPZ verifiable encryption keypairs ekid and ekpr, Signal derives a separate symmetric AEAD key sk from the group master key gk and uses it (with AES-GCM-SIV) to encrypt all group-level *attributes*: the group title, description, disappearing-message timer, the off-server avatar payload, and per-member labels. We write $\mathsf{Enc}_{\mathsf{sk}}(\cdot)$ for these symmetric ciphertexts; individual attributes are encrypted independently, with fresh nonces, and concatenated in the header.

Profile information lives elsewhere. Signal does not store user profile information in the group state. Each user's display name, avatar, and similar metadata are stored on a separate *profile* server, encrypted under that user's profile key prK; group members fetch and decrypt these profiles independently after recovering prK from the group's verifiable ciphertext $\mathsf{Enc}_{\mathsf{ekpr}}(\mathsf{prK})$. Signal does, however, store one piece of per-member information in the group entry itself: an optional "label" pair (a short emoji and a free-form string visible only inside the group), encrypted under sk.

Join links. Each group optionally carries a server-stored random byte string inviteLinkPwd. An invite link is an HTTPS URL whose URL fragment is a base64 encoding of (gk, inviteLinkPwd); anyone holding the link can re-derive the group keys and present inviteLinkPwd to the server to fetch a public "join info" digest (group title, current member count, and current join policy). The server then lets them either join directly or insert themselves into a server-side queue of pending join requests. The add-from-invite-link access-control field governs which mode is in force: Any, Admin, or Disabled. Admins can rotate inviteLinkPwd to invalidate outstanding links or set the field to Disabled to turn join-by-link off entirely.

Changelog. Every group-state mutation is encoded as a typed list of action atoms together with the encrypted UID of the editor and the new monotonic version number vMaj+1. On accepting a request, the server fills in the editor's encrypted UID and the group ID, signs the serialized action list with its long-term signing key, and stores the resulting tuple $(\text{actions}, \sigma_{\text{server}}, \text{epoch})$ in an append-only per-group log indexed by version. Members fetch any contiguous range of past changes, verify each σ_{server}, and replay the actions to render a per-action history (e.g., "Alice promoted Bob"). Each member's row carries a joinedAt field so clients suppress changelog entries from before that member's admission. The epoch field advertises the maximum action variant present, letting old clients reject changes they cannot interpret rather than silently misrender history.

Pending invitations and their revocation. A pending invitation is a row in a separate side-list $\texttt{mem-lst}_{\mathsf{pendingPrK}}$ alongside the core member list. Each row holds the invitee's encrypted UID, the encrypted UID of the inviting admin, and a server-supplied timestamp; the invitee's profile-key ciphertext is added later when they accept. Either the original inviter or any administrator may revoke a pending invitation by deleting the corresponding row (action DeleteMemberPendingProfileKey); the accept-invite path (action PromoteMemberPendingProfileKey) operates on the same row, so a race between revocation and acceptance is resolved by whichever request the server processes first.

Pending join-by-link requests. When a group's add-from-invite-link policy is Admin, a user following an invite link does not join directly. Instead they insert a row into the pending-approval side-list, which holds their encrypted UID, encrypted profile key, and a server-supplied timestamp. Any administrator may then approve the request, promoting the row to a full member entry, or refuse it and optionally ban the requester in the same atomic action.

Banned-user list. A group carries a side-list $\texttt{mem-lst}_{\mathsf{banned}}$ of pairs $(\mathsf{Enc}_{\mathsf{ekid}}(\mathsf{UID}), \text{timestamp})$. On any add-by-link or admin-approval-of-join-request operation, the server checks that the candidate's encrypted UID is not present in $\texttt{mem-lst}_{\mathsf{banned}}$ and rejects the change otherwise. Admins manage the list with AddMemberBanned and DeleteMemberBanned actions; the same admin call that removes a member or refuses a join request can atomically ban them. Server confidentiality of the ban list is partial: the encrypted UIDs hide *who* is banned from the server, but the cardinality and timestamps of the list are visible to the server, and the full ban history appears in the signed changelog visible to all current members.

No rotation of group key material. Signal currently has no protocol for rotating gk. This is not an artifact of the cryptography but of how the server indexes group state: GID is derived from gk

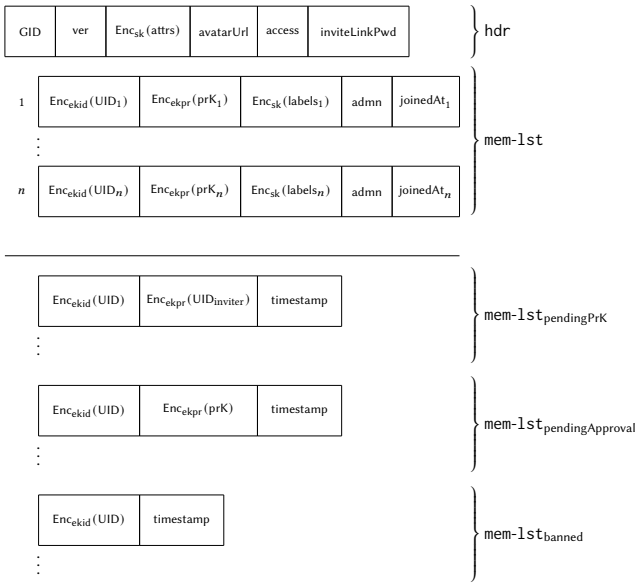

Figure 14: Group state currently stored on Signal's group server. (ekid, ekpr) **are the verifiable-encryption keypairs;** sk **is a separate symmetric AES-GCM-SIV key for attribute blobs. All are derived from the group master key** gk**. The header, member list, and three side-lists (pending invitations, pending join requests, and banned UIDs) are all stored on the server. User profile information (name, avatar, etc.) is *not* stored in the group record; it lives on a separate profile server, encrypted under each user's profile key, and is fetched independently by group members. The avatar payload at** avatarUrl **is itself encrypted under** sk**. Per-action signatures by the server's signing key on each log entry are not shown.**

and is the stable storage key for the entire group record, so any change to gk would force a migration of the entire server state into a new record. In practice this blocks Signal from ever rotating, even when rotation would be desirable (e.g., after a member leaves under suspicion of compromise, or to evict a joined-and-now-removed member's view of past group attributes).

The current group structure. We depict the (relevant parts of the) group structure as currently deployed by Signal in Figure 14. The actual stored structure carries a handful of additional flags whose comparison to our system is uneventful and which we omit from Figure 14: an announcements_only bit (only admins may post messages), a PNI/ACI-aware variant of the accept-invitation action introduced when Signal split user identifiers into account and phone identifiers, and several smaller per-action fields (e.g., changeEpoch byte ranges for forward compatibility) used to coordinate clients across protocol versions.

B Protocol Descriptions, Continued

B.1 Verification Subroutine

The subroutine for verifying the group data structure executed by a user upon fetching a new protocol version is given in Figure 15.

UsrVfyUpdate(grp-lst, grp-lst′)
Parse (hdr′, mem-lst′, core-sig′, apx′, inv-q′) ← grp-lst′
Parse (hdr, mem-lst, core-sig, apx, inv-q) ← grp-lst
(ek, rrs) ← KDF(gk, "SUB-KEY")
Parse (GID, ∗, vMin) ← apx-hdr
Set members = ⊥
// Check header
Parse (GID, vMaj′, c'_{hdr}, c'_{join}) ← hdr′, (GID, vMaj, c_{hdr}, c_{join}) ← hdr
if vMaj′ ≤ vMaj : call HandleError// attempted rollback
if vMaj′ > vMaj + 1 : call HandleError// version skipped
(sttgs, attr) ← $\mathrm{Dec}_{ek}(c_{hdr})$, (sttgs′, attr′) ← $\mathrm{Dec}_{ek}(c'_{hdr})$
// Check core sig
(i, v, σ^{psdn}) ← core-sig′
if vMaj′ > vMaj ∧ $v \neq$ vMin: call HandleError// min version skipped
for $j \in [1 \ldots n]$
  $\mathrm{UID}_j \leftarrow \mathrm{GetUID}(j, \text{mem-lst}, \text{ek})$
  $\mathrm{vk}_{base} \leftarrow \mathrm{GetPseudoVkBase}(\mathrm{UID}_j)$
  $\rho \leftarrow \mathrm{KDF}(\text{rrs}, \text{"rerand@"} \| \mathrm{UID}_j)$
  $\mathrm{vk}^{psdn}_j \leftarrow \mathrm{RandVK}(\mathrm{vk}_{base}, \rho)$
vk-lst ← $\left(H(\mathrm{vk}^{psdn}_{\mathrm{UID}_j})\right)_{j\in[1\ldots n]}$
$m \leftarrow (i^*, \text{hdr}, \text{mem-lst}, v, \text{vk-lst})$
if $\mathrm{Sgn.Vfy}(\mathrm{vk}^{psdn}_{i^*}, m, \sigma^{psdn}) \neq 1$: call HandleError
if (mem-lst′[i^*].admn
  ∧¬mem-lst[indexOf(GetUID(i^*, mem-lst′, ek′), mem-lst, ek)].admn):
  call HandleError// User made himself admin
// Check member list
Parse $\{C_{\mathrm{UID}_j}, \mathrm{mct}_j, \mathrm{admn}_j\}_{j\in[1\ldots n]}$ ← mem-lst
For all $j \in [1, \ldots, n]$:
  $(\mathrm{UID}_j, r_j, \mathrm{uk}_j, \mathrm{gk}') \leftarrow \mathrm{Dec}(\text{ek}, \mathrm{mct}_j)$// gk′ ≠ ⊥ only for invitees
  if $C_{\mathrm{UID}_j} \neq \mathrm{Commit}(\mathrm{UID}_j; r_j)$: call HandleError
  if $\mathrm{uk}_j = \perp$:
    // Check if UID_j added himslef to inv-q
    $\mathrm{uk}_j \leftarrow \mathrm{AcceptedInvite}(\text{GID}, \mathrm{UID}_j, \text{inv-q}', \text{gk}')$
    if $\mathrm{uk}_j \neq \perp$: append $(\mathrm{UID}_j, \mathrm{uk}_j)$ to members
  else append $(\mathrm{UID}_j, \mathrm{uk}_j)$ to members
// Check appendix
Parse (apx-hdr, apx-lst) ← apx′
Parse $\text{line}_j := (\text{cmd}_j, \text{ind}_j, *, *, *)_{j\in[1\ldots|\text{apx-lst}|]}$ ← apx-lst
for all $j \in [1 \ldots |\text{apx-lst}|]$ with $\text{cmd}_j \in \{\text{join}, \text{addUser}\}$:
  if false ≠ (UID, uk, vk^{psdn}) ← $\mathrm{CheckAppendixLine}(\text{gk}, \text{mem-lst}', \text{apx-hdr}, \text{line}_j, \text{sttgs}, c'_{join})$
    append $(\mathrm{UID}_j, \mathrm{uk}_j)$ to members
    vk-lst+=vk^{psdn}
for all $j \in [1 \ldots |\text{apx-lst}|]$ with cmd_j = refresh:
  if false ≠ (UID, uk) ← $\mathrm{CheckAppendixLine}(\text{gk}, \text{mem-lst}', \text{apx-hdr}, \text{line}_j, \text{apx}, c'_{join})$
    replace the j-th entry in members it by (UID, uk)
for all $j \in [1 \ldots |\text{apx-lst}|]$ with cmd_j = modAttr:
  if false ≠ (attr″) ← $\mathrm{CheckAppendixLine}(\text{gk}, \text{mem-lst}', \text{apx-hdr}, \text{line}_j, \text{sttgs}, c'_{join})$
    replace attr′ by attr″
for all $j \in [1 \ldots |\text{apx-lst}|]$ with cmd_j = leave:
  if false ≠ (UID) ← $\mathrm{CheckAppendixLine}(\text{gk}, \text{mem-lst}', \text{apx-hdr}, \text{line}_j, \text{sttgs}, c'_{join})$
  else remove (UID, ∗) from members
// Check new acquaintances
for all (UID′, uk′) ∈ members s.t. no record ⟨AcqRec, UID′, uk′⟩ exists:
  if ¬IsCorrectUserKey(UID′, uk′): HandleError
// Write changelog
if vMin = 0: changedBy ← i^*
else // ∃ appendix entries:
  (∗, k, ∗, ∗) ← apx-lst[|apx-lst| − 1]
  changedBy ← k
$\delta \leftarrow \Delta((\text{attr}, \text{sttgs}, \text{members}), (\text{attr}', \text{sttgs}', \text{members}'))$
if $\delta = \perp$: call HandleError
log ← (changedBy, δ)// line changedBy did δ.
// Derive group structure
Parse members = $(\mathrm{UID}_1, \mathrm{uk}_1)$, ..., $(\mathrm{UID}_n, \mathrm{uk}_n)$ for some n
Set $\mathrm{expKey}_i \leftarrow \mathrm{KDF}(\mathrm{uk}_i, \text{"EXP-Key"})$ for $i \in [n]$
return (attr′, $(\mathrm{expKey}_i, \mathrm{UID}_i)_{i\in[1\ldots n]}$, log, vk-lst)

Procedure HandleError:
  ignore this version and keep the old state.

Procedure AcceptedInvite(GID, UID, inv-q, gk′):
(ipk, isk) ← PKE.KeyGen(KDF(gk′, "INV-Key"))
  for all $c \in$ inv-q
    $(\sigma, (\mathrm{UID}', \mathrm{uk}', \mathrm{GID}')) \leftarrow \mathrm{Dec}_{\mathrm{isk}}(c)$
    if UID′ = UID ∧ GID = GID′:
      Send UID′ to $\mathcal{F}_{\mathrm{PKI}}$ and obtain $\mathrm{vk}^{pers}_{\mathrm{UID}'}$
      if $\mathrm{Sgn.Vfy}(\mathrm{vk}^{pers}_{\mathrm{UID}'}, (\mathrm{UID}', \mathrm{uk}', \mathrm{GID}'))$: **return** uk′
  return ⊥

Procedure IsCorrectUserKey(UID, uk):
  else send UID to S and $\mathcal{F}_{\mathrm{PKI}}$
  receive back $(\mathrm{vk}_{base}, \mathrm{acq}, \sigma)$ and vk^{pers}
  if $\mathrm{Vfy}(\mathrm{vk}^{pers}, (\mathrm{vk}_{base}, \mathrm{acq}), \sigma) \neq 1 \vee H(\mathrm{uk}) \neq \mathrm{acq}$: **return** false**else return** true

Figure 15: The verification procedure UsrVfyUpdate **executed by a user upon fetching a new protocol version with** Fetch**, cf. Figure 10. We write** $\Delta((\textbf{attr}, \text{sttgs}, \text{members}), (\textbf{attr}', \text{sttgs}', \text{members}'))$ **for the atomic difference between the two member list versions, i.e., a new** (UID, uk)**, changed** (UID, uk)**, a removed** (UID, uk)**, or changed** **attr** **or** sttgs**.**

Subroutine CheckAppendixLine(gk, mem-lst, apx-hdr, line, sttgs, c_{join})

Parse $(b_{\text{add}}, b_{\text{attr}}) \leftarrow$ sttgs
Parse (cmd, ind, $[C]$, ct, σ^{psdn}) $\leftarrow$ line
(ek, rrs) $\leftarrow$ KDF(gk, "SUB-KEY")
// Get pseudonym verification key, verify signature:
  if ind $\neq \perp$: UID $\leftarrow$ GetUID(ind, mem-lst, ek)
  else (UID, r_c, uk) $\leftarrow \text{Dec}_{\text{ek}}(ct)$
  $\text{vk}_{\text{base}} \leftarrow$ GetPseudoVkBase(UID)
  $\rho \leftarrow$ KDF(rrs, "rerand@"||UID)
  $\text{vk}^{\text{psdn}} \leftarrow \text{RandVK}(\text{vk}_{\text{base}}, \rho)$
  if $\text{Sgn.Vfy}(\text{vk}^{\text{psdn}}, (\text{apx-hdr}, \text{line}), \sigma^{\text{psdn}}) \neq 1$: **return** false
// Checks for a leave line:
if cmd = leave:
  return (UID)
// Checks for a refresh line:
if cmd = refresh:
  uk $\leftarrow \text{Dec}_{\text{ek}}(ct)$
  return (UID, uk)
// Checks for lines with new members:
if cmd $\in$ {addUser, JoinViaLink}:
  if cmd = addUser $\wedge \neg b_{\text{add}}$: **return** false
  if cmd = join $\wedge\, c_{\text{join}} = \perp$: **return** false
  (UID, r_c, uk) $\leftarrow \text{Dec}_{\text{ek}}(ct)$
  if $C \neq \text{Commit}(\text{UID}, r_c)$: **return** false
  else return (UID, uk, vk^{psdn})
// Checks for a modAttr line:
if cmd = modAttr:
  if $\neg b_{\text{attr}}$: **return** false
  attr $\leftarrow \text{Dec}_{\text{ek}}(ct)$
  return (UID, attr)

Subroutine GetPseudoVkBase(UID)

if $\exists$ record (AcqRec, UID, uk, vk_{base}): return vk_{base}
else send UID to S and $\mathcal{F}_{\text{PKI}}$
receive back (vk_{base}, acq, σ) and vk^{pers}
if $\text{Vfy}(\text{vk}^{\text{pers}}, (\text{vk}_{\text{base}}, \text{acq}), \sigma) \neq 1$: **return** Fail
else return vk_{base}

Subroutine GetUID(i, mem-lst, ek)

if $i \leq |\text{mem-lst}|$ **return** false
parse $\{C_{\text{UID}_j}, \text{mct}_j, \text{admn}_j\}_{j \in [1 \ldots n]} = \text{mem-lst}$
$(\text{UID}, *, *) \leftarrow \text{Dec}_{\text{ek}}(\text{mct}_i)$
return UID

Subroutine indexOf(UID, mem-lst, ek)

parse $\{C_{\text{UID}_j}, \text{mct}_j, \text{admn}_j\}_{j \in [1 \ldots n]} = \text{mem-lst}$
find $i \in [n]$ s.t. $(\text{UID}, *, *) \leftarrow \text{Dec}_{\text{ek}}(\text{mct}_i)$
return i

Figure 16: Subroutines called by our protocols: CheckAppendixLine(gk, mem-lst, apx-hdr, line) **for checking validity of an appendix line** line **against** mem-lst. GetPseudoVkBase(UID) **returns the base pseudonym verification key of** UID. GetUID(i, mem-lst, ek) **returns the user identifier encrypted under** ek **in the** i**-th entry of** mem-lst. indexOf(UID, mem-lst, ek) **returns the index of** UID **in the** ek**-encrypted** mem-lst**.**

B.2 Checking Appendix Lines

The subroutine for checking lines in the appendix is given in Figure 15.

B.3 Removing a Group Member

Figure 17 depicts the RemoveMember protocol.

Server S()	**User** U(GID, UID′)
	Retrieve (KeyRec, UID, muk, ep, sk^{pers}, vk_{base}, sk_{base})
	Retrieve (AcqRec, UID′, uk′, vk'_{base})
	Retrieve (GID, fet, gk, vk^{psdn}, sk^{psdn}, grp-lst, vk-lst)
	Parse (hdr, mem-lst, core-sig, apx, inv-q) = grp-lst
	(hdr, mem-lst, core-sig, vk-lst, m) $\leftarrow$ Consolidate(GID)
	(ek, rrs) $\leftarrow$ KDF(gk, "SUB-KEY")
	$\text{fet}_{\text{new}} \xleftarrow{\$} \{0,1\}^{\lambda}$
	// Admin at i removes user at j
	Let $i \leftarrow$ indexOf(UID, mem-lst, ek)
	Let $j \leftarrow$ indexOf(UID′, mem-lst, ek)
	mem-lst $\leftarrow$ mem-lst $\setminus$ {line j}
	vk-lst $\leftarrow$ vk-lst $\setminus$ {line j}
	// Header update
	(as in Figure 11)
	// Authentication
	(as in Figure 11)
GID, fet, fet_{new}, hdr, mem-lst, $\text{vk}^{\text{psdn}}_{\text{UID}'}$, core-sig $\dashleftarrow$	
(as in Figure 11 but vk-lst $\leftarrow$ vk-lst′ $\setminus$ $\{\text{vk}^{\text{psdn}}_{\text{UID}'}\}$)	
	Run subroutine RotateGroupKey

Figure 17: Protocol RemoveMember **where user** U **with identifier** UID **removes the user with identifier** UID′**.** indexOf(UID, mem-lst) **returns the index (row) of** UID **in** mem-lst**. Dashed arrows denote user-anonymous communication.**

B.4 Admins cleaning up the appendix: Subroutine (Consolidate)

Figure 18 shows the subroutine through which an admin consolidates all updates from a non-empty appendix into a new major version of the group data structure.

B.5 Setting Admin Rights (SetAdminRights)

Any admin can grant or revoke admin privileges to other group members. For this, the admin simply toggles the admin flag admn in the corresponding row of the member list mem-lst, signs the new list and uploads it to the server, who checks the admin's signature and priviledges. We do not depict this protocol as it is straightforward to derive from other admin protocols such as Figure 11.

B.6 Changing the Group Settings (ChangeSettings)

The protocol is depicted in Figure 19. It works similarly to other admin actions, but this time only the header is updated to encrypt the new group settings. The admin authenticates with their pseudonym signature and uploads the update to the server, which checks the admin flag and the signature before storing the update.

B.7 Inviting Users to the Group

Users can get into a group without being acquainted with one of the current group members in two ways: Via a join link and via an invitation. We start by describing the invitation procedure.

An admin who only knows the UID but not the user key uk of another user (i.e., only knows their phone number but never

Subroutine Consolidate(GID)

Retrieve (GID, fet, gk, $\mathsf{vk}^{\text{psdn}}$, $\mathsf{sk}^{\text{psdn}}$, grp-lst, vk-lst)
Parse (hdr, mem-lst, core-sig, apx, inv-q) = grp-lst
(ek, rrs) ← KDF(gk, "SUB-KEY")
Ignore if apx = ⊥
Parse (apx-hdr, apx-lst) ← apx
Parse (GID, vMaj, vMin) ← apx-hdr
Set m ← |apx-lst|
Retrieve (KeyRec, UID, muk, ep, $\mathsf{sk}^{\text{pers}}$, $\mathsf{vk}_{\text{base}}$, $\mathsf{sk}_{\text{base}}$)
Parse (GID, vMaj, c_{hdr}) ← hdr
(sttgs, attr) ← $\text{Dec}_{\text{ek}}(c_{\text{hdr}})$
Parse $\text{line}_i := (\text{cmd}_i, \text{ind}_i, (C_i, ct_i), \sigma_i)_{i\in[m]}$ ← apx-lst
For $i = 1, \ldots, m$ do:
  out ← CheckAppendixLine(gk, mem-lst, apx-hdr, line_i)
  Skip line **if** ¬out // Invalid appendix line
  // Adopt line into mem-lst and vk-lst
  if cmd_i = leave:
    i ← indexOf(out, mem-lst, ek)
    remove line i from both mem-lst and vk-lst
  if cmd_i = refresh:
    Parse (UID, uk) = out
    i ← indexOf(UID, mem-lst, ek)
    $(\text{UID}, r, *) \leftarrow \text{Dec}_{\text{ek}}(ct_i)$
    Set $ct_i' \leftarrow \text{Enc}_{\text{ek}}(\text{UID}, r, \text{uk})$
    Parse $\{*, *, \text{admn}_l'\}_{l\in[|\text{mem-lst}|]}$ = mem-lst
    overwrite line i in mem-lst with $(C_i, ct_i', \text{admn}_i)$
  if $\text{cmd}_i \in$ {addUser, JoinViaLink}:
    Parse (UID, uk, $\mathsf{vk}^{\text{psdn}}$) = out
    mem-lst+=(C_i, ct_i, σ_i)
    vk-lst+=$\mathsf{vk}^{\text{psdn}}$
  if cmd_i = modAttr: rewrite atts
    Set $c_{\text{hdr}} \leftarrow \text{Enc}_{\text{ek}}(\text{sttgs}, \text{Dec}_{\text{ek}}(ct_i))$
// Consolidate invite queue
for $(C, \text{mct}, \text{admn}) \in$ mem-lst s.t. $(\text{UID}', r, \perp, \text{gk}') = \text{Dec}(\text{ek}, \text{mct})$
  (ipk, isk) ← PKE.KeyGen(KDF(gk′, "INV-Key"))
  for $c \in$ inv-q s.t. $\perp \neq (\text{UID}', \text{uk}', \sigma) \leftarrow \text{Dec}_{\text{isk}}(c)$
    Send UID′ to $\mathcal{F}_{\text{PKI}}$ and obtain $\mathsf{vk}^{\text{pers}}_{\text{UID}'}$
    if $\text{Sgn.Vfy}(\mathsf{vk}^{\text{pers}}_{\text{UID}'}, (\text{UID}', \text{uk}', \text{GID})) = 1$
      $\text{mct}' \leftarrow \text{Enc}_{\text{ek}}(\text{UID}', r, \text{uk}')$
      Overwrite mct in mem-lst with mct′
Set inv-q ← ∅
// Compute new group structure
vMaj+=1, apx ← ⊥
hdr ← (GID, vMaj, c_{hdr})
i ← indexOf(UID, mem-lst, ek) // Admin's new index
$\sigma^{\text{psdn}} \leftarrow \text{Sgn}_{\mathsf{sk}^{\text{psdn}}}(i, \text{hdr}, \text{mem-lst}, m, H(\text{vk-lst}_j))_{j\in[|\text{vk-lst}|]})$
core-sig ← $(i, m, \sigma^{\text{psdn}})$
return (hdr, mem-lst, core-sig, vk-lst, m)

Figure 18: Subroutine Consolidate(GID)**, run by admins upon fetching** GID **with a non-empty appendix. The subroutine consolidates the appendix of a stored** GID **record into its main list** mem-lst**, proceeding from the oldest to the newest appendix line, and ignoring all invalid lines. The** CheckAppendixLine **subroutine is defined in Figure 16 in Appendix B.2.**

received a message from them) can invite the latter to the group. This user can decide whether they accept or decline the invitation.

To issue an invitation, an admin generates a key pair (ipk, isk) ← PKE.KeyGen(KDF(gk, "INV-Key")). It then performs the code of protocol AddMember (cf. Figure 11) but adding line $(C_{\text{UID}}, \text{Enc}(\text{ek}, \text{UID}, r, \perp, \text{gk}), 0)$ to the main list mem-lst and deriving $\mathsf{vk}^{\text{psdn}}_{\text{UID}}$ not from a stored record but from the server via GetPseudoVkBase(UID).

Server S() | **User** U(GID, sttgs)

Retrieve (KeyRec, UID, muk, ep, $\mathsf{sk}^{\text{pers}}$, $\mathsf{vk}_{\text{base}}$, $\mathsf{sk}_{\text{base}}$)
Retrieve (GID, fet, gk, $\mathsf{vk}^{\text{psdn}}$, $\mathsf{sk}^{\text{psdn}}$, grp-lst, vk-lst)
Parse (hdr, mem-lst, core-sig, apx, inv-q) = grp-lst
(hdr, mem-lst, core-sig, vk-lst, m) ← Consolidate(GID)
(ek, rrs) ← KDF(gk, "SUB-KEY")
$\text{fet}_{\text{new}} \xleftarrow{\$} \{0,1\}^\lambda$
i ← indexOf(UID, mem-lst, ek)
// Header update
Parse (GID, vMaj, c_{hdr}, c_{join}) ← hdr
$(*, \text{attr}) \leftarrow \text{Dec}_{\text{ek}}(c_{\text{hdr}})$
$\text{hdr} \leftarrow \text{Enc}_{\text{ek}}(\text{sttgs}, \text{attr})$
hdr ← (GID, vMaj + 1, c_{hdr}, c_{join})
// Authenticate with own pseudonym key
(As in Figure 11)

GID, fet, fet_{new}, hdr, mem-lst, core-sig
←----------

Retrieve (GID, fet, hdr′, mem-lst′, core-sig′, apx′, inv-q′, vk-lst′)
Parse $(i, m, \sigma^{\text{psdn}})$ ← core-sig
If admn_i' = false: abort
$\mathsf{vk}^{\text{psdn}}$ ← $(\text{vk-lst}')_i$
If $\Sigma.\text{V}_{\mathsf{vk}^{\text{psdn}}_{\text{UID}'}}(\sigma^{\text{psdn}}, (i, \text{hdr}, \text{mem-lst}, m, (H(\text{vk-lst}_j))_{j\in[|\text{vk-lst}|]})) = 0$: abort
Store (GID, fet_{new}, hdr, mem-lst, core-sig, ⊥, ⊥, vk-lst)

(GID, fet_{new}) →
E2EE to all members

Figure 19: Protocol ChangeSettings **for an admin to change the group settings that are encrypted in the header** hdr **of the group data structure.**

The admin then signs and propagates this new group list to the server.

The next step of the invitation process is that the user accepts the invitation. The user first signs its uk as $\sigma^{\text{join}} \leftarrow \Sigma.\text{Sgn}(\mathsf{sk}^{\text{pers}}, (\text{UID}, \text{uk}, \text{GID}))$ and then encrypts it together with the signature $c \leftarrow \text{PKE.Enc}_{\text{ipk}}(\text{UID}, \text{uk}, \sigma)$ under the received ipk. Ciphertext c is then uploaded to the server, which adds it to the invite queue inv-q. Not accepting an invitation works by simply not acting on the invite. The moment a user key uk for an invited UID is found in the invite queue, fetching users consider the invited user as a member (cf. Fetch, Figure 10).

The admin can also revoke the invitation if it was not accepted yet by removing the corresponding $(C_{\text{UID}}, \text{mct}, 0)$ line, similar as in RemoveMember (cf. Figure 17).

When an admin consolidates the group, they empty the invite queue. That is, they take all entries in the group list that do not contain a userkey, i.e., uk = ⊥, and check if there is a corresponding ciphertext $c \in$ inv-q such that c decrypts to the same UID and the signature in c verifies under UID's public key. In that case, the admin writes the uk found in c into the main list. If there are ciphertexts in inv-q without a matching entry in the main list, the admin can discard them. Note that the admin can derive all required gks that are used to derive isk from the group list entries.

B.8 Public Groups: Joining via a Link

Admins can create a link that allows users to join the group without being acquainted with a member of the group. For usability, this link must not change even when the group key is rotated. When a user clicked the link, still either an admin or a user (depending on

the group setting) has to complete the join request. Joining via the link can further be configured to require admin approval.

We can realize such links by allowing users to write `join` entries in the appendix. This was described already in Figure 12. We now propose a mechanism to let users learn ek from a link. Note that we cannot simply encode ek and fet in the link, since otherwise every rotation of the group key (executed at least every time a user is removed from the group) would invalidate the link. Hence, we propose to encode a "join key" jk in the link instead, and let the server store the current group key encrypted under this jk in the group header. When a user wants to join via a link, it sends GID to the server. If the server finds a non-empty join ciphertext in the header, it sends it back to the user, who decrypts the group key and fetching token, fetches the group structure with the token, and then joins the group via the appendix.

The admin who creates the join link also needs to specify via a boolean b_{adm} whether admins need or need not approve joiners from links. This configuration is uploaded to the server by ChangeSettings (cf. Figure 19). Users act on it upon fetching as follows: If the header contains a join ciphertext and $\neq b_{\mathsf{adm}}$, they interpret all correctly, decrypting and verifying join entries in the appendix as members.

Upon group key rotation, admins encrypt the new key material under the static join key jk. For this, the admin who generated the join link has to communicate jk to all other admins (cf. Figure 20).

B.9 Changing the Group Key: Subroutine (RotateGroupKey)

To rotate the group key material, an admin chooses a new group key gk and encrypts the whole group data structure under this new key. The protocol is depicted in Figure 20.

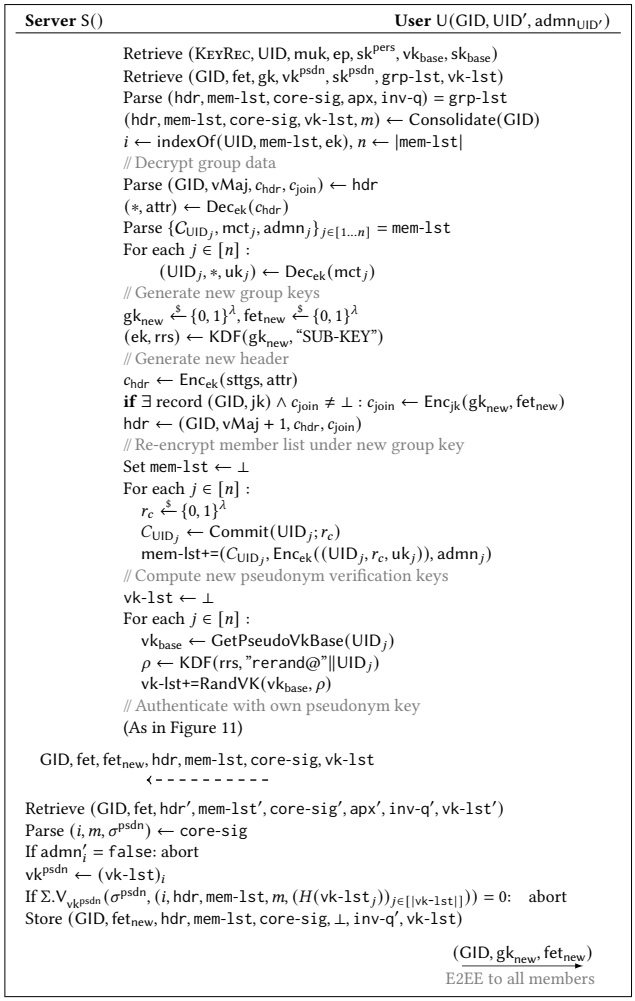

Figure 20: Subroutine RotateGroupKey **for an admin to rotate the group key.**

C Security Model for PGS

In this section, we formally define a private group system and its security guarantees. As in previous works [7], we use the Universal Composability (UC) [6] framework. Due to the differences in our and their system, we depart significantly from their security model. For the reader familiar with the model of [7] we provide some justification and comparison in Appendix A.2. Our formal model is underpinned by a systematization of system properties and guarantees, which we summarize in Appendix D.

On a very high level, functionality $\mathcal{F}_{\mathsf{PGS}}$ serves as a trusted storage for all of the private groups' state. A group's state consists of

- group identifier, attributes, and settings,
- current group members,
- current group admins,
- invited users who have not yet accepted the invite,
- export keys of all members, and
- changelog of all past group modifications.

$\mathcal{F}_{\mathsf{PGS}}$ allows access to this state only to group members, and it allows manipulation of the state according to the group settings. Here, the functions offered by $\mathcal{F}_{\mathsf{PGS}}$ reflect what is currently implemented by Signal. For example, by default, only admins can add, invite, and remove users, and everybody can always leave the group. But $\mathcal{F}_{\mathsf{PGS}}$ also allows admins to configure groups such that, e.g., a group can be joined via a link, users can change the group attributes, or users are allowed to add other users.

User registration. The functionality requires users to register (via interface RegisterUser, Figure 21) before being allowed to join or create a group. This registration requires setting a user identifier UID, a unique identifier under which the user is referred to in the system from now on, and a (system-wide) key expKey. The latter is not required for the PGS but is exported to the application, where it can be used, e.g., to encrypt a user's profile image. $\mathcal{F}_{\mathsf{PGS}}$ ensures that a user's expKey is only known among its groups, and it allows for rotating this key via the Refresh interface (Figure 22). In Signal, to protect users from being added to spam groups, one can only add users to a group from whom one has previously received a message, or with whom one shares a group. $\mathcal{F}_{\mathsf{PGS}}$ reflects the message approval mechanism by letting Bob call (GrantAcquaintance, Alice). Afterwards, Alice can add Bob to any group. $\mathcal{F}_{\mathsf{PGS}}$ stores Alice's acquaintances in list T_{Acq}[Alice] (Figure 21). While the server engages in the approval procedure, it only learns Alice's identity,

while Bob remains anonymous (GA.3). In other words, the PGS leaks to the server how "famous" users are, but not who already talked to whom.

Group creation and adding/removing users. To create a group, any registered user calls the Create interface, providing group attributes attr and initial settings b_{attr} (whether users can change the group attributes) and b_{add} (whether users are allowed to add more users). $\mathcal{F}_{\text{PGS}}$ stores the new group in an object $T_{\text{G}}[\text{GID}][1]$, where 1 indicates the version of the group. The group gets assigned a unique identifier GID (CG.2). After the creation of a group, $\mathcal{F}_{\text{PGS}}$ allows admins to add and eventually remove group members via interfaces AddMember and RemoveMember. Every such modification of the group state increases the version number, and $\mathcal{F}_{\text{PGS}}$ will maintain all versions of a group. This is done by first copying the whole group state into $T_{\text{G}}[\text{GID}][\text{ver} + 1]$ (see AM.3 (a) and RM.3 (a)) and then performing the action in the list of members $T_{\text{G}}[\text{GID}][\text{ver} + 1].\text{members}$. $\mathcal{F}_{\text{PGS}}$ enforces that these actions can only be performed by admins by ensuring that the calling user's UID is in the admins list of the current group version $T_{\text{G}}[\text{GID}][\text{ver}]$ (AM.3 , RM.3).

Fetching the group state. Any group member can fetch the group state by calling the Fetch interface (Figure 21) with the corresponding group identifier GID and the version the user last fetched. $\mathcal{F}_{\text{PGS}}$ ensures that users fetch all updates sequentially (FG.2 , FG.3). It also ensures that non-members cannot fetch even when they know the group ID (FG.6). Whenever a member fetches, $\mathcal{F}_{\text{PGS}}$ stores the version presented to the user in table $T_{\text{Ver}}[\text{GID}][\text{UID}]$ (FG.8 (a)), adds the user as acquaintance to all group members (FG.8 (b), meaning the user can from now on add all of them to more groups), and finally outputs the group state. This state consists of the identifiers and export keys of all group members, the fetched version number, the group attributes, the list of invited users, and the changelog from the previous update (FG.8). If the server is honest, $\mathcal{F}_{\text{PGS}}$ ensures that users always fetch the next version.

Advanced group functions. $\mathcal{F}_{\text{PGS}}$ offers various interfaces for advanced manipulation of a group state (Figure 22). Since all interfaces follow a similar structure but merely modify different parts of the group state object $T_{\text{G}}[\text{GID}]$, we only provide a brief description of them below. All interfaces have in common that they write successful executions into the group's changelog.

- Invite a user to a group (Invite): for each group, $\mathcal{F}_{\text{PGS}}$ maintains an invite queue $T_{\text{G}}[\text{GID}][\text{ver}].\texttt{inv-q}$ to which admins can add users of whom they only know the UID, i.e., to whom they are not acquainted. This models inviting users by just typing their phone number. Users in the invite queue are not considered members yet, but their identifiers are already visible to other group members ("Admin x invited user y to the group").
- Revoke a user invitation (RevokeInvitation): Similarly, admins can remove users from the invite queue.
- Respond to an invitation (AcceptInvitation): Invited users only become members if they accept an invite via this interface. $\mathcal{F}_{\text{PGS}}$ then writes them into the member list members.
- Activate or deactivate a group link (ToggleJoinLink): For "public" groups that can be joined by users without requiring an invite/add operation from a group member, this interface can be called. To add at least some control over who is joining, admin approval can be activated for users who join on their own behalf. $\mathcal{F}_{\text{PGS}}$ simply stores these settings in $T_{\text{G}}[\text{GID}][\text{ver}].\text{activeLink}$ and $T_{\text{G}}[\text{GID}][\text{ver}].\text{adminApproval}$.
- Join a group via a link (JoinViaLink): If the above interface was used to open up a group, this interface allows users who only know the GID to join the group. In case admin approval is required, $\mathcal{F}_{\text{PGS}}$ will only create a record of a user wanting to join (JvL.4).
- Admin approval of joining via a link (ApproveJoin): This interface allows an admin to approve a user who joins a link-activated group. $\mathcal{F}_{\text{PGS}}$ will look for a matching record (AJ.3) and write the user into the members list. We note that, while the names of this and the previous interfaces contain the word "link" as this is how we know them from Signal, $\mathcal{F}_{\text{PGS}}$ never exports or requires such a link: how the "open joining" feature is realized, e.g., how a link concretely looks like is left completely up to the protocol and hence not part of the definition.
- Changing admin privileges (SetAdminRights): Any admin can set or unset another member's admin permissions. $\mathcal{F}_{\text{PGS}}$ reflects this by adding or removing the member in the admins list.
- Leaving a group (Leave): Any member can leave the group. $\mathcal{F}_{\text{PGS}}$ denotes the intent of the user to leave by adding the user to a list leaveQueue (LG.3 (c)) and removing them from the members list. The effect of the user remaining in the leave queue is that it is still able to fetch group updates *if* a corrupt server allows them to fetch (FG.6). This accounts for the fact that group key rotation can only happen when another group member initiates it, and hence, the gone user is still able to decrypt data stored at the server. The leave only becomes effective when an admin comes online. $\mathcal{F}_{\text{PGS}}$ reflects this by flushing the leaveQueue in every interface that requires admin privileges (e.g., AM.3 (e)).
- Non-admin adding of group members (UserAdd): If the group is configured to allow users to add members, this interface can be used for this operation. It works very similarly to the AddMember interface but allows the request only if $T_{\text{G}}[\text{GID}][\text{ver}].\text{allowUserAdd}$ is set (UA.3).
- Changing the group attributes (ChangeAttr): This interface can be called by any group member. $\mathcal{F}_{\text{PGS}}$ allows to set the group attributes attr if the calling user is either an admin, or a group member and the group setting $T_{\text{G}}[\text{GID}][\text{ver}].\text{allowUserAttr}$ is set to `true`.
- Changing the group settings (ChangeSettings): This interface allows admins to change the group settings, namely whether users can add other users, and whether they can modify the group attributes. $\mathcal{F}_{\text{PGS}}$ simply checks whether the calling user is an admin in the group (GS.3) and writes the settings in the group object $T_{\text{G}}[\text{GID}]$.

This concludes the description of the interfaces of $\mathcal{F}_{\text{PGS}}$. For convenience, we give an overview and short description of all user-

Object	Type	Description
$T_{\mathsf{U}}[\mathsf{UID}]$	object	user data
.owner	pid	UC pid of user who registered UID
.expKey	bits	user's export key
$T_{\mathsf{Ver}}[\mathsf{GID}]$	int	current version of the group
$T_{\mathsf{admVer}}[\mathsf{GID}]$	int	last administrated group version
$T_{\mathsf{Ver}}[\mathsf{GID}][\mathsf{UID}]$	int	version of GID that UID last fetched
$T_{\mathsf{Acq}}[\mathsf{UID}]$	list	users who can add UID to a group
$T_{\mathsf{G}}[\mathsf{GID}][i]$	object	i-th group state
.admins	list	UIDs of all group admins
.members	list	UIDs & exp. keys $\forall$ users in group
.log	list	group change since version $i-1$
.invited	list	UIDs of invited users
.activeLink	bool	invite link activated?
.adminApproval	bool	join via link req. admin approval?
.allowUserAttr	bool	users can change group attributes?
.allowUserAdd	bool	users can add group members?

Table 6: Objects maintained by functionality $\mathcal{F}_{\mathsf{PGS}}$.

and group-specific objects and variables that $\mathcal{F}_{\mathsf{PGS}}$ stores in Table 6.

Leakage: We now turn to describing the leakage that $\mathcal{F}_{\mathsf{PGS}}$ admits. While we demand strong privacy properties against non-members and the Signal server storing the group data, we cannot expect to protect a group's privacy in the presence of corrupt group members. Hence, $\mathcal{F}_{\mathsf{PGS}}$ has several levels of leakage depending on server and member corruption. To simplify the definition, we opt for the same leakage in all group state manipulating interfaces, even though a concrete protocol might provide stronger guarantees for some of them. These leakage levels are (see Figure 21):

- **Honest server & all group members honest (PQ.4):** Only the type of query is leaked to the adversary. This accounts for the fact that communication size and pattern might already leak which operation is executed, and we do not expect a protocol to hide this information.
- **Corrupt server & all group members honest (PQ.5):** $\mathcal{F}_{\mathsf{PGS}}$ leaks the type of query, the group identifier GID of the group to be modified, and the indices of the involved users to the adversary. A server must know which stored group to update, and group IDs do not contain any private information as they are anyway chosen by the adversary (CG.1). The indices of the involved users (e.g., "The 3rd user who joined removed the user who joined 8th") are an artifact of our protocol that stores an encrypted entry per user. Crucially, these indices do not reveal the identity of the involved users, nor do they allow linking users across groups.
- **At least one group member is corrupt (PQ.6):** $\mathcal{F}_{\mathsf{PGS}}$ reveals the full query, including involved UIDs, to the adversary. Since group changes are visible to all members of the group, this leakage is unavoidable.

This completes the list of information leakage to the adversary, i.e., everything that is not mentioned remains protected. Crucially, $\mathcal{F}_{\mathsf{PGS}}$ guarantees privacy of group membership (i.e., UIDs) towards even a corrupt server.

Attacks: Depending on the corruption state, $\mathcal{F}_{\mathsf{PGS}}$ allows a number of attacks. $\mathcal{F}_{\mathsf{PGS}}$ does not contain interfaces that are unlocked if a group has **corrupt members** or even **corrupt admins**. By this, $\mathcal{F}_{\mathsf{PGS}}$ mandates that all such corrupt users can do amounts to *what honest members and admins can do*. In particular, such corrupt users gain no insights or powers over groups they are not a member of. A corrupt admin can upload an invalid group state, in which case $\mathcal{F}_{\mathsf{PGS}}$ ensures that honest users learn who to blame (FG.6 (b)). Next, we consider a **network adversary** who is present even if all users and the server are honest. A PGS protocol is always prone to Denial-of-Service (DoS) attacks by the network attacker dropping or tampering with messages from or to the server. $\mathcal{F}_{\mathsf{PGS}}$ accounts for this by allowing the adversary to DENY all operations (e.g., AM.2). To completely suppress the interface call (e.g., dropped first message), the adversary may infinitely delay the Fail output to the server. In any such network attack, the group state in $\mathcal{F}_{\mathsf{PGS}}$ will remain the same, i.e., the network adversary is not given the power of impacting the group state beyond preventing updates via the DENY flag. Finally, we detail which attacks $\mathcal{F}_{\mathsf{PGS}}$ admits to a **corrupt server**. First, a corrupt server can decide to acquaint arbitrary users, i.e., to always make acquaintance checks pass when a user is added (GA.1). A corrupt server can also arbitrarily chose the group identifier GID (CG.2), which however serves only as a pointer. A corrupt server can always "destroy" the group list by tampering with it; however, $\mathcal{F}_{\mathsf{PGS}}$ demands that in such a case, users learn that the server is to blame (FG.7 (a)). A corrupt server can also *withhold group updates* from individual users fetching the group. While this breaks consistency of a group's state presented to all members, it also seems unavoidable that a server can simply opt to present outdated versions of the group to a fetching user. $\mathcal{F}_{\mathsf{PGS}}$ hence allows the adversary to prevent fetching an update via the DENY flag (FG.6). However, $\mathcal{F}_{\mathsf{PGS}}$ protects against *rollback attacks* by demanding the fetched version to be *more recent* than what the user previously fetched ($T_{\mathsf{Ver}}[\mathsf{GID}][\mathsf{UID}]$ stores this information). Additionally, $\mathcal{F}_{\mathsf{PGS}}$ ensures that users do not miss updates (FG.2), i.e., it guarantees that if the user outputs group data, it is the update that happened after the user last fetched. Finally, in the case of a corrupt server colluding with a corrupt group member, $\mathcal{F}_{\mathsf{PGS}}$ does not provide consistency anymore and lets the adversary determine the output of Fetch (FG.7 (c)). This reflects the fact that, with all group key material exposed, the server and user can completely rewrite the whole group structure. This concludes the description of the attacks admitted by our model, and hence, we do not expect a PGS protocol to protect against them. Crucially, all attacks not listed in $\mathcal{F}_{\mathsf{PGS}}$ need to be protected against by a PGS protocol. To give some examples, these include a corrupt server manipulating the group state, or a colluding user and server bypassing the group settings.

D Systematization of System Properties and Guarantees

We provide an extended list of security properties of our PGS protocol (white rows) as well as Signal's currently deployed solution (gray rows) and detail in which corruption settings they hold and how they are guaranteed by our protocol. While the formal model

The functionality keeps initially empty tables T_U for user-specific data, $T_G[GID][ver]$ for the data of group GID in version ver, $T_{Ver}[GID]$ for storing the current version of each group, $T_{Ver}[GID][UID]$ for information which version a user last fetched, and T_{Acq} for acquaintance data, where $T_{Acq}[UID]$ stores the set of users who are acquainted with UID, i.e., who can add UID to a group. We implicitly assume all outputs are *delayed outputs*. The function indexOf(UID, $T_G[GID]$.members) outputs the index of (UID, $*$) in the list $T_G[GID]$.members (starting at 1). We assume implicit session ids *sid* in all messages. Initialize ctr $\leftarrow$ 1.

On input (RegisterUser, UID, expKey*) from U:

- R.1 **if** U already registered: **ignore query**
- R.2 **send** (RegisterUser, UID) to $\mathcal{A}$.
- R.3 **if** $T_U[UID]$ is already defined **output** ("bad request") to U and (Error, "user already exists") to S
- R.4 **else** initialize $T_U[UID]$
- R.5 Set $T_U[UID]$.owner $\leftarrow$ U
- R.6 Sample $T_U[UID]$.expKey $\xleftarrow{\$} \{0,1\}^\lambda$
- R.7 **if** U is corrupt, set expKey $\leftarrow$ expKey*
- R.8 **else output** expKey to U

On input (GrantAcquaintance, UID′) from U or $\mathcal{A}$:

- GA.1 **if** called by $\mathcal{A}$ and S is honest **ignore query**
- GA.2 Find UID s.t. $T_U[UID]$.owner = U, **else ignore query**
- GA.3 **if** S is corrupt, **send** (GrantAcquaintance, UID′) to $\mathcal{A}$, **else send** (GrantAcquaintance) to $\mathcal{A}$; **receive back** m
- GA.4 **if** $T_U[UID']$ is not defined **or** m = DENY **output** (GrantAcquaintance, UID′, Fail) to S
- GA.5 **else** add U to $T_{Acq}[UID']$

On input (Create, attr, b_{attr}, b_{add}) from U:

- CG.1 **send** (Create) to $\mathcal{A}$; **receive back** (GID, m)
- CG.2 **if** GID already exist, **ignore query**
- CG.3 **if** m = DENY, **output** (Create, GID, Fail) to S
- CG.4 **if** m = ALLOW:
  - (a) find UID s.t. $T_U[UID]$.owner = U: **else ignore query**
  - (b) set $T_{Ver}[GID] \leftarrow 1$, $T_{Ver}[GID][UID] \leftarrow 0$ and initialize $T_G[GID][1]$
  - (c) set $T_{admVer}[GID] \leftarrow 1$ // last administrated group version
  - (d) set $T_G[GID][1]$.attr $\leftarrow$ attr
  - (e) append UID to $T_G[GID][1]$.admins
  - (f) append (UID, $T_U[UID]$.expKey) to $T_G[GID][1]$.members
  - (g) set $T_G[GID][1]$.allowUserAttr $\leftarrow b_{attr}$
  - (h) set $T_G[GID][1]$.allowUserAdd $\leftarrow b_{add}$
  - (i) set $T_G[GID][1]$.log $\leftarrow$ (UID, created, b_{attr}, b_{add})

On input (Fetch, GID) from U:

- FG.1 Find UID s.t. $T_U[UID]$.owner = U, **else ignore query**
- FG.2 Set ver $\leftarrow T_{Ver}[GID][UID] + 1$ // Fetching the next update
- FG.3 Ignore the query if ver $\geq T_{Ver}[GID]$ // User already up to date
- FG.4 **if** S honest **then send** (Fetch) to $\mathcal{A}$ **else send** (Fetch, GID) to $\mathcal{A}$
- FG.5 **receive back** m from $\mathcal{A}$
- FG.6 **if** m = DENY **or** $T_G[GID]$ undefined **or** $((UID,*) \notin T_G[GID][ver]$.members $\wedge \neg(UID \in T_G[GID][ver]$.leaveQueue $\wedge$ S corrupt$))$, **output** (Fetch, GID, Fail) to S // Left user can still fetch if corrupt Server helps
- FG.7 **if** m = (ATTACK, P, resp*):
  - (a) **if** $(\neg$isCorrupt(GID) $\vee$ P = S) and S is corrupt, **output** (Fetch, Fail, S) to U // Server to blame
  - (b) **if** $T_U[P]$.owner is corrupt and P $\in T_G[GID]$.admins, **output** (Fetch, Fail, P) to U // Admin P to blame
  - (c) **if** isCorrupt(GID) **and** S is corrupt, output resp* to U // Corrupt member & server can present any group data
- FG.8 **if** m = ALLOW:
  - (a) set $T_{Ver}[GID][UID] \leftarrow$ ver // Remember which version the user last fetched
  - (b) for each (UID′, $*$) in $T_G[GID][ver]$.members, add U to $T_{Acq}[UID']$ // U can now add members to other groups
  - (c) **output** (Fetch, GID, $T_G[GID][ver]$.attr, $T_G[GID][ver]$.members, $T_G[GID][ver]$.invited, $T_G[GID][ver]$.log) to UID

On input (AddMember, GID, UID′, admn′) from U:

- AM.1 (UID, ver, $*$, m) $\leftarrow$ prepQuery(U, UID′, GID, AddMember)
- AM.2 **if** m = DENY or $T_G[GID]$ undefined, **output** (AddMember, GID, Fail) to S
- AM.3 **if** m = ALLOW **and** U $\in T_{Acq}[UID']$ **and** UID $\in T_G[GID][ver]$.admins:
  - (a) copy $T_G[GID][ver+1] \leftarrow T_G[GID][ver]$ and set $T_{Ver}[GID] \leftarrow T_{Ver}[GID] + 1$
  - (b) append (UID′, $T_U[UID']$.expKey) to list $T_G[GID][ver]$.members
  - (c) set $T_G[GID][ver+1]$.log $\leftarrow$ (UID, added, UID′, admn′)
  - (d) **if** admn′, add UID′ to $T_G[GID][ver]$.admins
  - (e) set $T_{admVer}[GID] \leftarrow$ ver + 1 // last administrated group version
  - (f) set $T_{Ver}[GID][UID] \leftarrow$ ver // UID can fetch from ver + 1 onwards
  - (g) flush $T_G[GID][ver+1]$.leaveQueue

On input (RemoveMember, GID, UID′) from U:

- RM.1 (UID, ver, $*$, m) $\leftarrow$ prepQuery(U, UID′, GID, RemoveMember)
- RM.2 **if** m = DENY **or** $T_G[GID]$ undefined, **output** (RemoveMember, GID, Fail) to S
- RM.3 **if** m = ALLOW **and** UID $\in T_G[GID][ver]$.admins **and** (UID′, $*$) $\in T_G[GID][ver]$.members:
  - copy $T_G[GID][ver+1] \leftarrow T_G[GID][ver]$ and set $T_{Ver}[GID] \leftarrow T_{Ver}[GID] + 1$
  - set $T_G[GID][ver+1]$.log $\leftarrow$ (UID, removed, UID′)
  - set $T_{admVer}[GID] \leftarrow$ ver + 1 // last administrated group version
  - remove (UID′, $*$) from $T_G[GID][ver+1]$.members
  - remove UID′ from $T_G[GID][ver+1]$.admins (if in it)
  - flush $T_G[GID][ver+1]$.leaveQueue

On input (SetAdminRights, UID′, b_{adm}) from U:

- TA.1 (UID, ver, $\perp$, m) $\leftarrow$ prepQuery(U, $\perp$, GID, SetAdminRights)
- TA.2 **if** m = DENY or $T_G[GID]$ undefined, **output** (SetAdminRights, GID, Fail) to S
- TA.3 **if** m = ALLOW **and** UID $\in T_G[GID][ver]$.admins **and** (UID′, $*$) $\in T_G[GID][ver]$.members **and** (b_{adm} **or** $|T_G[GID][ver]$.admins$| > 1$) :
  - (a) copy $T_G[GID][ver+1] \leftarrow T_G[GID][ver]$ and set $T_{Ver}[GID] \leftarrow T_{Ver}[GID] + 1$
  - (b) **if** b_{adm} **then** add UID′ to $T_G[GID][ver+1]$.admins
  - (c) **else** remove UID′ from $T_G[GID][ver+1]$.admins
  - (d) set $T_G[GID][ver+1]$.log $\leftarrow$ (UID, toggleAdmin, UID′, b_{adm}) item set $T_{admVer}[GID] \leftarrow$ ver + 1 // last administrated group version
  - (e) flush $T_G[GID][ver+1]$.leaveQueue

Procedure isPrivate(GID, ver): // Is there a corrupt user in the group?

- PR.1 Set $b \leftarrow$ false
- PR.2 **if** $\exists$ UID $\in T_G[GID][ver]$.members s.t. $T_U[UID]$.owner is corrupt **then** $b \leftarrow$ true
- PR.3 **return** b

Procedure isCorrupt(GID, ver): // Is there a corrupt admin in the group?

- CO.1 Set $b \leftarrow$ false
- CO.2 **if** $\exists$ UID $\in T_G[GID][ver]$.admins s.t. $T_U[UID]$.owner is corrupt **then** set $b \leftarrow$ true
- CO.3 **return** b

Procedure prepQuery(U, UID′, GID, CMD): // Common preparation step

- PQ.1 Find UID s.t. $T_U[UID]$.owner = U, **else ignore query** // U never registered
- PQ.2 **if** UID′ $\neq \perp$ **then** set U′ $\leftarrow T_U[UID']$.owner **and ignore query** if undefined // UID′ unknown
- PQ.3 Set ver $\leftarrow T_{Ver}[GID]$ // Leakage, depending on Server and group member corruption:
- PQ.4 **if** isPrivate(GID, ver) and S honest **then send** (CMD) to $\mathcal{A}$
- PQ.5 **if** isPrivate(GID, ver) and S corrupt **then** set $i \leftarrow$ indexOf(UID, $T_G[GID][ver]$.members) and $j \leftarrow$ indexOf(UID′, $T_G[GID][ver]$.members) and **send** (CMD, GID, i, j) to $\mathcal{A}$
- PQ.6 **if** $\neg$isPrivate(GID, ver), send (CMD, GID, UID, UID′) to $\mathcal{A}$
- PQ.7 **receive back** m from $\mathcal{A}$
- PQ.8 **return** (UID, ver, U′, m)

Figure 21: Ideal private group system $\mathcal{F}_{PGS}$, basic functions.

in Figures 21 and 22 can be used to derive all these, the list helps the reader interpret the model and understand how our PGS protocol satisfies it. We use the terminology *user* for actual humans using the PGS, and *client* for the user device.

We mark with

- ✓$^{\{P\}}$ *positive* properties that are satisfied even when party P behaves maliciously. We assume all other parties to be semi-honest. If we write ✓$^{\{U \wedge S\}}$, both U and S may maliciously collude, while ✓$^{\{U \vee S\}}$ means that the property holds when facing either a malicious S, or a malicious U, but not both.
- ✗$^{\{P\}}$ *negative* properties (i.e., ones that are *not* achieved against a malicious P, leading to *weaker* security or features)

Accepted risk. The following risks are accepted in the system:



Figure 22: Private group system $\mathcal{F}_{\text{PGS}}$, extended functions.

- Rogue acquaintances of Alex can give Alex's uk to an adversary who can then claim to know Alex, send Alex messages, and read their profile information.
- Rogue group members know the group secret and can share it with anyone.
- Rogue group members know the group list in the clear and can share it with anyone.

E Security Analysis

We sketch how to simulate each individual interface. The case where both parties are corrupted is trivial to simulate, as the simulator just executes all instructions it receives from the environment.

For the sake of this discussion, we will assume that all hash functions and KDFs are modeled as random oralces (RO), the AEAD encryption scheme is modeled as ideal authenticated cipher (cf. [14]), the commitment scheme is assumed to be equivocable and extractable, signatures are assumed to be unforgeable, and the key rerandomizable signature scheme has unforgeability under rerandomized keys (cf. [11, Def. 8]).

Privacy properties	Satisfied?	Why?
Group membership is hidden from S	✓$^{\{S\}}$	encrypted under group key
	✓$^{\{S\}}$	(same)
Group information (e.g., image and name) is hidden from S.	✓$^{\{S\}}$	encrypted under group key
	✓$^{\{S\}}$	(same)
After user U is removed by an admin, they are no longer able to decrypt updates to the group list	✓$^{\{U\wedge S\}}$	group key rotation (RotateGroupKey)
	✓$^{\{U\}}$, ✗$^{\{U\wedge S\}}$	S controls fetching, requires membership. No group key rotation.
A user leaving the group is no longer able to decrypt the group list…		
– immediately	✓$^{\{U\}}$, ✗$^{\{U\wedge S\}}$	Fetch via σ^{psdn} required after leave. Mal. S could still deliver group list and key is not rotated yet.
– after the first group member went online	✓$^{\{U\}}$, ✗$^{\{U\wedge S\}}$	(same)
– after the first admin went online	✓$^{\{U\wedge S\}}$	admin rotates group key
	✓$^{\{U\}}$, ✗$^{\{U\wedge S\}}$	S controls fetching, requires membership. No group key rotation.
Group sizes are hidden from S	✗$^{\{S\}}$	List length leaks
	✗$^{\{S\}}$	(same)
A user can revoke the access of some users (U) to their (future) profile information.	✓$^{\{U\wedge S\}}$	uk rotation (Refresh)
	✓$^{\{U\wedge S\}}$	profile key rotation (UpdateProfileKey)
Profile information of a user is hidden from S	✓$^{\{S\}}$	encrypted under expKey
	✓$^{\{S\}}$	encrypted under profile key
Future profile information of users who once shared a group with user U is hidden	✗$^{\{U\wedge S\}}$/ ✓$^{\{U\wedge S\}}$	before refreshing uk, the old group members still have uk (✗$^{\{U\wedge S\}}$); afterwards, neither server nor former group colleagues can decrypt U's profile information anymore (✓$^{\{U\wedge S\}}$)
	✗/ ✓$^{\{U\wedge S\}}$	(same with profile key instead of uk)
User registration cannot be attributed to a UID	✗$^{\{S\}}$	S chooses UID
	✗$^{\{S\}}$	(same)
User privacy: Fetching of a group		
– is unlinkable across and within groups	✓$^{\{U\wedge S\}}$	per-group fetching token
	✗$^{\{S\}}$	users prove ownership of list entries
– cannot be attributed to a UID	✓$^{\{U\wedge S\}}$	per-group fetching token
	✓$^{\{U\wedge S\}}$	the proof of ownership is anonymous (using anonymous credentials)
User privacy: Being removed, added or invited by an admin		
– is unlinkable across groups	✓$^{\{S\}}$	UID encrypted with per-group key
	✓$^{\{S\}}$	(same)
– is unlinkable within the same group	✓$^{\{S\}}$/✗$^{\{S\}}$	repeated addition/invitation is unlinkable because commitments are randomized and encryption involves nonces. Adding and removing (the same line) is linkable.
	✗$^{\{S\}}$	UID encryption is deterministic and group encryption keys are persistent
– cannot be attributed to a UID	✓$^{\{S\}}$	encrypted UID
	✓$^{\{S\}}$	(same)
User privacy: Leaving a group and joining via invite link		
– is unlinkable across groups	✓$^{\{S\}}$	per-group UID encryption key and vk^{psdn}
	✓$^{\{S\}}$	per-group UID encryption key
– is unlinkable within the same group	✗$^{\{S\}}$	re-joining is linkable because same rrs leads to same vk^{psdn}. Joining-then-leaving is linkable (same line).
	✗$^{\{S\}}$	UID encryption is deterministic
– cannot be attributed to a UID	✓$^{\{S\}}$	UID encrypted
	✓$^{\{S\}}$	(same)
Admin privacy: actions (Create, AddMember, RemoveMember, DeleteGroup) of an admin are		
– unlinkable across groups	✓$^{\{S\}}$	different vk^{psdn} per group
	✓$^{\{S\}}$	Anonymity of the membership proof
– unlinkable within the same group	✗$^{\{S\}}$	claimed row index leaks this
	✗$^{\{S\}}$	(same)
– cannot be attributed to a UID	✓$^{\{S\}}$	UID encrypted
	✓$^{\{S\}}$	(same)
Admin privacy: creating invitation links		
– is unlinkable across groups	✓$^{\{S\}}$	link does not contain admin info
	✓$^{\{S\}}$	(same)
– cannot be attributed to a UID	✓$^{\{S\}}$	link does not contain admin info
	✓$^{\{S\}}$	(same)

Figure 23: Private group system properties (Part I). Uncolored rows detail the properties of our PGS, while gray rows specify the properties of Signal's currently deployed private group system.

Consistency properties	Satisfied?	Why?
Only admins can add, invite, or remove users	✓$^{\{U \wedge S\}}$	σ^{psdn} required for changes to main list. Fetch requires to check admin flag (even against old version). σ^{psdn} allows to trace changes of admn.
	✓$^{\{S\}}$, ✗$^{\{U \wedge S\}}$	S checks admin policy. U knowing group key material can collude with S to perform any group operation.
Only admins can grant admin privileges	✓$^{\{U \wedge S\}}$	Fetch requires to check admin flag (even against old version). σ^{psdn} allows to trace changes of admn.
	✗$^{\{S\}}$	admn not protected, can be overwritten by S
Only acquaintances of a user can be added to a group by that user (note: this is a direct add, not an invite)	✓$^{\{U \wedge S\}}$	For writing a valid `mem-lst` entry, user must know the added user's uk. Colluding with the server to obtain an acquaintance signature does not help in writing a valid entry.
	✓$^{\{U \wedge S\}}$	For writing a valid `mem-lst` entry, user must know the added user's profile key. Bypassing server's verification of the anonymous profile credential does not help in writing a valid entry

Blaming	Satisfied?	Why?
Users can detect whether S modifies the group structure	✓$^{\{S\}}$	Server cannot forge σ^{psdn} of any admin
	✗$^{\{S\}}$	S can do several modifications to the group structure that go unnoticed, like changing admin flags, rolling back old entries, etc.
Given a semi-honest S, the users know which admin U made a flawed update	✓$^{\{U\}}$	Admin signs own index, checked by server
	✓$^{\{U\}}$	not applicable: the server verifies consistency of entries and rejects flawed ones
Even a malicious admin U and a malicious S colluding cannot frame another admin to have performed some update	✓$^{\{U \wedge S\}}$	Cannot forge σ^{psdn} under that admins vk^{psdn}
	✓$^{\{U \wedge S\}}$	`mem-lst` entries do not contain any information on who created them, and the anonymous authentication to the server does not allow framing

Rollback attacks	Satisfied?	Why?
Users/clients cannot be rolled back to an older group version than the last one they have seen	✓$^{\{S\}}$	version number signed by σ^{psdn}
	✗$^{\{S\}}$	no integrity of group structure: S can take an old membership list, add new version number to it, and present it to users
The server cannot violate consistency, i.e., present different member lists to different users	✗$^{\{S\}}$	S can withhold updates from individual users
	✗$^{\{S\}}$	(same)
The server cannot re-introduce previous members into the group	✓$^{\{S\}}$	prevented by admin signature on the whole group structure
	✗$^{\{S\}}$	no integrity of group structure. Old entries remain valid unless a user changes their profile key, and a malicious S can add them back to `mem-lst`. Same holds for pending and banned entries.

DoS Properties	Satisfied?	Why?
The server can identify a legitimate request to fetch a group description	✓	fetching only with fetching token or σ^{psdn}
	✓	for fetching, users need to claim membership via anonymous credentials
Admins U cannot add garbage entries to the group structure	✗$^{\{U\}}$	S can't verify correct encryption
	✓$^{\{U\}}$	S verifies correct encryption
Non-admins cannot add invalid member entries to the group structure	✓$^{\{S\}}$	need vk^{psdn} for a line with admn set to change. (Note: this does not hold for the appendix)
	✓$^{\{U\}}$	S verifies correctness of ciphertexts

Post-quantumness	Satisfied?	Why?
Group member lists are protected from harvest-now-decrypt-later attacks	✓$^{\{S\}}$	symmetric group encryption and statistically hiding commitments
	✗$^{\{S\}}$	encryption of UID and profile key are DDH-based and hence not quantum safe
Performing group actions is protected against quantum attackers	configurable	every group action requires an admin signature, so ✓$^{\{U \wedge S\}}$ when using Dilithium-based key rerandomizable signatures, and ✗$^{\{U \wedge S\}}$ when DDH-based
	✗$^{\{U \vee S\}}$	neither encryption nor anonymous credentials are quantum safe, so a quantum attacker could potentially forge valid entries, or/and fake membership proofs

Figure 24: Private group system properties (Part II). Uncolored rows detail the properties of our PGS, while gray rows specify the properties of Signal's currently deployed private group system.

Honest server, honest user: For all protocols, simulation is facilitated by our use of secure channels. The simulator merely has to simulate that messages are sent over the network. Because the environment calls the functionality interfaces of the honest parties, the functionality creates all records correctly.

Simulation of RegisterUser. *Honest server, honest user:* Additionally to the secure channel, the simulator picks a uniformly random

uk and computes acq like in the real protocol as KDF(uk, "ACQ-Tag"). Note that the environment receive a uniformly random expKey picked by the functionality and yet unknown to the simulator. If the user gets corrupted at some point later, the simulator can program the RO KDF, such that expKey is the output of KDF(uk, "EXP-Key"). The simulator also chooses $(\mathsf{sk}_{\mathsf{base}}, \mathsf{vk}_{\mathsf{base}})$ for this user and stores them.

Honest server, corrupt user: The simulator obtains UID directly from the adversarial message. The simulator observes the RO KDF. If there is a query such that acq = KDF(uk*, "ACQ-Tag"), then the simulator computes expKey* ← KDF(uk*, "EXP-Key") and gives input (RegisterUser, UID, expKey*) to $\mathcal{F}_{\mathsf{PGS}}$ on behalf of the corrupt user.

Corrupt server, honest user: The simulator samples $(\mathsf{sk}_{\mathsf{base}}, \mathsf{vk}_{\mathsf{base}})$, $(\mathsf{sk}^{\mathsf{pers}}, \mathsf{vk}^{\mathsf{pers}})$ like in the real protocol and uk, acq as above. With these values, the simulator computes σ like in the real protocol.

Simulation of Create. *Honest server, corrupt user:* In this case, the simulator must extract the secret inputs b_{attr}, b_{add}, and attr. The simulator can do this by observing the ideal cipher. For a given ciphertext c, the simulator can look for a corresponding Enc query leading to c. Collisions happen only with negligible probability. In fact, the ideal cipher outputs have to be authenticated to allow the simulator distinguishing properly encrypting ciphertexts from erroneous ones to check that the obtained ciphertext does encrypt to attributes and settings. The simulator gives the extracted values to $\mathcal{F}_{\mathsf{PGS}}$ via the Create interface on behalf of the corrupt user. Finally, the simulator checks σ^{psdn} using the provided $\mathsf{vk}^{\mathsf{psdn}}$. If the signature is valid, the simulator gives ALLOW to the functionality, else DENY. Also, the simulator stores the encrypted group list like a real server, if the signature verifies.

Corrupt server, honest user: The simulator uses that the encryption scheme is modeled as an ideal cipher (IC), and the commitment is equivocable. That is, it can produce a group header and a member list without knowing attr and UID. If the group gets corrupted later, the simulator learns the attributes and the user secrets stored in this group and can program the IC, and equivocate the commitment accordingly. The fetching token can be a random string. To simulate the pseudonym signature, the simulator retrieves $(\mathsf{sk}_{\mathsf{base}}, \mathsf{vk}_{\mathsf{base}})$ that it created for this user when simulating registration. Next, the simulator picks a group key gk and derives $\mathsf{vk}^{\mathsf{psdn}}$ and σ^{psdn} like in the protocol description using $\mathsf{sk}_{\mathsf{base}}$ and gk.

Simulation of AddMember. *Honest server, corrupt user:* Like in Create, the simulator can extract UID′ from the new member ciphertext mct, using the IC. The GID and the new admin flag admn′ are provided in plain. This allows the simulator to give input (AddMember, GID, UID′, admn′) to $\mathcal{F}_{\mathsf{PGS}}$ on behalf of the corrupt user. Extracting UID′ from the IC also yields the ek that is used by the corrupt user. The simulator, thus, can check if there was a KDF(gk, "SUB-KEY") query that evaluated to (ek, rrs). If not such query exists, the other group members reject the update with overwhelming probability because either they derive a different rrs and the pseudonym signature does not verify or they derive a different ek and decryption of mct fails. Therefore, the simulator can ignore this query. If such a record exists, the simulator first uses ek to decrypt mct_i, where i is the index of the changing admin, i.e., the value specified in core-sig. With that, the simulator can look up the $\mathsf{vk}_{\mathsf{base}}$ that was stored at registration for UID and derive $\mathsf{vk}^{\mathsf{psdn}} \leftarrow \mathsf{RandVK}(\mathsf{vk}_{\mathsf{base}}, \mathsf{rrs})$. Again, like in Create, the simulator verifies σ^{psdn} and if verification succeeds, it returns ALLOW to $\mathcal{F}_{\mathsf{PGS}}$ and else DENY to ensure that the server outputs a fail.

Corrupt server, honest user: Like in Create, the simulator uses the equivocable commitment scheme and the IC to produce a new entry without knowing UID′. Then, the simulator retrieves the gk that it simulated at creation to derive $\mathsf{vk}^{\mathsf{psdn}}$ and σ^{psdn} like in the real protocol.

Simulation of SetAdminRights, RemoveMember. These protocols work similarly to AddMember. We only describe the differences in the simulation:

- In RemoveMember, the user cannot decrypt a new entry to extract UID′ but it has to compare the new member list with the old one and decrypt the one mct that is missing in the new version.
- After RemoveMember, the simulator must also simulate a RotateGroupKey, which works exactly like simulating Create.
- To simulate SetAdminRights, the simulator can simply read the changed admin flag from the new group list. It extracts UID′ like in AddMember.

Simulation of GrantAcquaintance. If the user U′ who receives the uk from U is corrupted, but U is not, the simulator must now program KDF(uk, "ACQ-Tag") ← acq and KDF(uk, "EXP-Key") ← expKey, as uk is leaked to the environment. For simulating a honest server, note that the server holds no particular secrets and thus, the simulator can just run the real server's code. Finally, to simulate an honest user receiving $(\mathsf{vk}_{\mathsf{base}}, \mathsf{acq}, \sigma)$, the simulator can also receive $\mathsf{vk}^{\mathsf{pers}}_{\mathsf{UID}}$ from (the internally simulated $\mathcal{F}_{\mathsf{PKI}}$) and verify the signature. Depending on the result, the simulator either gives DENY or ALLOW to $\mathcal{F}_{\mathsf{PGS}}$.

Simulation of Fetch. *Honest server, corrupt user:* The simulator also checks internally, if it stored a record corresponding to the required GID. If that is the case, the entry corresponds to a corrupt group, i.e., a group where at least one member is corrupt. The simulator just returns the created group list like a real server also does. As this group is either created by the adversary or was already equicocated when handing gk to $\mathcal{A}$, the simulator can just behave like an honest server. If instead the corrupt user is requesting a GID is an honest group, the adversary does not know fet and also cannot forge a σ^{psdn}, by the security of the key rerandomizable signature scheme. If no corresponding fet is stored for GID, the simulator can give input DENY to $\mathcal{F}_{\mathsf{PGS}}$ to trigger a fail output of the server.

Corrupt server, honest user: The simulator must perform all consistency checks detailed in UsrVfyUpdate. First, the simulator can again use the IC to extract a group key gk that was used to create the received group list. Using this, the simulator is able to compute $\mathsf{vk}_i^{\mathsf{psdn}}$, where i is the admin specified in core-sig. If σ^{psdn} does not verify under this key, the server is to blame, i.e., the simulator can give input (ATTACK, S, r^*) to $\mathcal{F}_{\mathsf{PGS}}$, where r^* is the view of the group that the simulator decrypted. If the signature verifies, but any of the other checks fail, the simulator can give

(ATTACK, GetUID(i, mem-lst)) to $\mathcal{F}_{\text{PGS}}$, indicating that the i-th admin is to blame for the failure. By the unforgeability of the key rerandomizable signature scheme, the adversary can not frame any other admin than the one extracted by the simulator.

Simulation of adaptive user corruption. When a user U gets corrupted, the simulator must equivocate the commitments and all mcts for all groups where U is a member. The functionality gives the simulator all the required UIDs and the simulator can also use the uk that it picked for this honest user upon registration.