WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs

David Balbás[1,2] *, Daniel Collins[3] **, and Phillip Gajland[4,5] ***

[1] IMDEA Software Institute, Spain
[2] Universidad Politécnica de Madrid, Spain
[3] EPFL, Switzerland
[4] Max Planck Institute for Security and Privacy, Germany
[5] Ruhr University Bochum, Germany

Abstract Developing end-to-end encrypted instant messaging solutions for group conversations is an ongoing challenge that has garnered significant attention from practitioners and the cryptographic community alike. Notably, industry-leading messaging apps such as WhatsApp and Signal Messenger have adopted the *Sender Keys* protocol, where each group member shares their own symmetric encryption key with others. Despite its widespread adoption, Sender Keys has never been formally modelled in the cryptographic literature, raising the following natural question:

What can be proven about the security of the Sender Keys protocol, and how can we practically mitigate its shortcomings?

In addressing these questions, we first introduce a novel security model to suit protocols like Sender Keys, deviating from conventional group key agreement-based abstractions. Our framework allows for a natural integration of two-party messaging within group messaging sessions that may be of independent interest. Leveraging this framework, we conduct the first formal analysis of the Sender Keys protocol, and prove it satisfies a weak notion of security. Towards improving security, we propose a series of efficient modifications to Sender Keys without imposing significant performance overhead. We combine these refinements into a new protocol that we call Sender Keys+, which may be of interest both in theory and practice.

Keywords: Secure Messaging, Group Messaging, WhatsApp, Signal, Sender Keys, Post-Compromise Security.

* This work was in part done while visiting Max Planck Institute for Security and Privacy and EPFL. This work is supported by the PICOCRYPT project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101001283), and partially supported by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU / PRTR. The author is partially funded by Ministerio de Universidades (FPU21/00600).

** This work was in part done while visiting Max Planck Institute for Security and Privacy.

*** This work was in part done while visiting EPFL. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Contents

1 Introduction

Messaging applications like WhatsApp, Facebook Messenger, Signal, and Telegram have witnessed remarkable global adoption, serving as essential communication tools for billions of users. All of these applications rely, to a varying degree, on cryptography to provide diverse forms of authenticity and secrecy. Among end-to-end encrypted messaging services (this excludes, among others, Telegram and Facebook Messenger by default), numerous cryptographic solutions have emerged, each with its own merits. Notably, for two-party messaging, Signal's Double Ratchet Protocol [MP16a] stands out as the dominant choice in practice. In the context of group messaging, Signal [M+16][1] and later WhatsApp [Wha20] have adopted the so-called *Sender Keys* protocol, which has enjoyed widespread adoption for numerous years. Besides, other popular solutions such as Matrix [ADJ24] and Session [Jef20] implement variants of the protocol. In Sender Keys, messages are encrypted using a user-specific symmetric key (which is then hashed forward) and then authenticated with a signature. Additionally, parties rely on secure two-party channels (instantiated in practice with the Double Ratchet) to share key material between them. Looking ahead, two-party channels will be central to determine the security attained by any instantiation of Sender Keys.

Concurrently, the recently completed standardisation effort of the IETF Messaging Layer Security (MLS) [BBR+23] working group resulted in a secure group messaging protocol with sub-linear complexity for group operations (adding/removing members and updating key material). Academic works have also explored so-called continuous group key agreement (CGKA) [BBR18, ACDT20, KPPW+21, ACJM20, ACDT21], although these are only a component of a fully-fledged group messaging protocol. Besides DCKGA [WKHB21], which is tailored to the decentralised setting, the only complete group messaging protocol that has been rigorously formalised to date is MLS [ACDT21]. In addition, CGKA protocols have some drawbacks. While some exhibit sub-linear performance in specific executions (and this class of executions is not well-characterized), their performance can degrade to linear in general, which is unavoidable at least when using off-the-shelf cryptographic primitives [BDG+22]. Moreover, they tend to be complex, increasing their attack surface and making them more susceptible to design and implementation bugs. Recent academic works and ongoing discussions in mailing lists have identified and addressed several security issues that emerged during the standardisation of MLS [ACDT20, AJM22, IETF23].

Hence, Sender Keys and similar approaches to group messaging, which diverge from the CGKA and MLS paradigms, remain an essential and practical alternative with different security / performance trade-offs. Surprisingly, despite being the most popular alternative to MLS (and moreover the protocol with the widest adoption), Sender Keys has not been formally studied in the literature, prompting the following natural question:

Can we formalise the Sender Keys protocol in a meaningful security model?

To answer this question we start by introducing a new cryptographic primitive along with a security model to capture a broad class of group messaging protocols that do not necessarily employ CGKA [ACDT20] at their core. Our framework provides native support for group messaging protocols that utilise secure two party communication channels under the hood, for

[1] Contrary to the folkore understanding that the Signal Messenger uses the pairwise channels approach for group messaging in small groups, Signal currently uses Sender Keys whenever possible.

which we introduce a clean level of abstraction. This novel framework proves instrumental in our analysis, as existing literature predominately focuses on CGKA-oriented models that do not suit Sender Keys and similar protocols. Subsequently, we present a detailed description of Sender Keys within our framework and provide a security proof validating the soundness of the protocol. In the process, we demonstrate that Sender Keys presents several security deficiencies that can be fixed without imposing substantial overhead. This addresses the following pertinent question:

What are the security deficiencies of Sender Keys, and how can they be addressed whilst preserving its practical efficiency?

In this regard we propose multiple modifications to Sender Keys, employing readily available primitives that have minimal impact on its efficiency. Our approach veers away from a theoretically systematic exploration to determine the "optimal" security for a Sender Keys-like protocol, as this would require non-standard primitives that considerably degrade performance [BRV20, ACJM20].

Although we do not identify any fundamental flaws in the Sender Keys protocol, we show that the protocol does not satisfy numerous standard and desirable security notions, such as forward security under message injections, resilience against injections impacting group membership changes[2], and fast PCS healing. These findings call into question the widespread use of the term "secure messaging" by commercial messaging solutions.

Nonetheless, Sender Keys and related protocols offer some advantages compared to alternative approaches like MLS. Firstly, Sender Keys stands out for its simplicity, benefiting form a long-standing open-source implementation of its core cryptographic operations [M+16]. This reduces the potential attack surface, making the protocol less susceptible to vulnerabilities in both its design and implementation. Secondly, Sender Keys offers good performance in small to moderate-sized groups, as evidenced by its successful adoption for groups of up to 1024 participants in WhatsApp and Signal [Wha20, M+16]. While the main group operations (adding and removing users) respectively have $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ communication complexity for groups of size n, concrete efficiency suffices in practice. Thirdly, Sender Keys offers forward-secure confidentiality and robust support for concurrent and out-of-order application message exchange. Thus, we believe that the formalisation and establishment of a provably secure variant of Sender Keys can serve as a valuable foundation for future implementations of the protocol.

1.1 Contributions

In summary, the main scientific contributions of our paper are the following:

- We introduce a new cryptographic primitive that we call *Group Messenger* (GM). We establish a modular security model for GM designed to capture messaging protocols like Sender Keys that are not necessarily based on group key agreement. It accounts for an active adversary capable of controlling the network and adaptively learning the states of different parties.
- We develop a general framework for composing two-party channels with group messaging protocols that use them. Our approach parameterises the security of the Group Messenger primitive based on the underlying two-party channels, presenting a novel perspective that, to the best of our knowledge, has not been explored previously.

[2] Note that Signal uses a dedicated private group management solution in practice [CPZ20] that we do not capture and is not affected by this attack vector.

– We formally describe Sender Keys, based on an analysis of Signal's source code [M+16], WhatsApp's security white paper [Wha20], and the `yowsup` library [Gal21].
– We prove the security of Sender Keys in our model and highlight shortcomings in the security of the protocol. To carry out the proof, we need to restrict the capabilities of the adversary substantially.
– We propose security fixes and improvements to the Sender Keys protocol, several of which result in a protocol that we call Sender Keys+. In particular, we secure group membership changes, improve the forward security of the protocol, and introduce an efficient key update mechanism. We also formalise the additional security guarantees in our model.

1.2 Paper Overview

Security in group messaging. Besides standard notions such as confidentiality, authenticity, and integrity of sent messages, two security properties are commonly considered in the messaging literature: forward security (FS) and post-compromise security (PCS) [CCG16]. Both properties require some form of key updating mechanism as well as erasures to achieve. Additionally, protocols must secure group membership updates, namely removed members must not be able to read messages sent after their removal, and newly added members must not (by default) be able to read past messages.

Most of the different formalisations of security in the literature model an adversarial Delivery Service (DS), the entity responsible for delivering messages between participants via the communication channel. The adversary (modelling the DS) can act as an eavesdropper with extended capabilities, e.g., that can schedule messages to be consistently delivered by users, as in [ACDT20], as a semi-active adversary that can schedule messages arbitrarily [KPPW+21], or as an active adversary that can inject messages [ACJM20, BCV23]. In many protocols, including Sender Keys and MLS, the DS relies mainly on some centralized infrastructure (the *central server* hereafter).

Sender Keys. In a Sender Keys group $\boldsymbol{G}$, every user $ID \in \boldsymbol{G}$ owns a so-called *sender key* which is shared with all group members. A sender key is a tuple $\mathsf{SK} = (\mathsf{spk}, \mathsf{ck})$, where spk is a public signature key (with a private counterpart ssk), and ck is a symmetric *chain key*. Every time ID sends a message m to the group, ID encrypts m using a *message key* mk that is deterministically derived (via a key derivation function H_1) from its chain key ck and erased immediately after being used. Upon message reception, group members derive mk to decrypt the corresponding ciphertext, which can also be delivered out-of-order. Messages are authenticated by appending the sender's signature to them. In Figure 1, we show a high-level abstraction of what occurs in a three-member group $\boldsymbol{G} = \{A, B, C\}$ when A sends a message that parties B and C receive.

Informally, forward security is provided by using a fresh message key for every message: every time a message is sent, the chain key is symmetrically ratcheted, i.e., hashed forward using a key derivation function H_2. The protocol, that we describe further in Section 4, also requires that there exist confidential and authenticated two-party communication channels between each pair of users. These are used for sharing sender keys when parties are added or removed from the group, or when some party updates their key material. For example, in the event that some ID leaves the group, members erase their own sender key and start over. This mechanism provides a form of PCS when a user is removed.

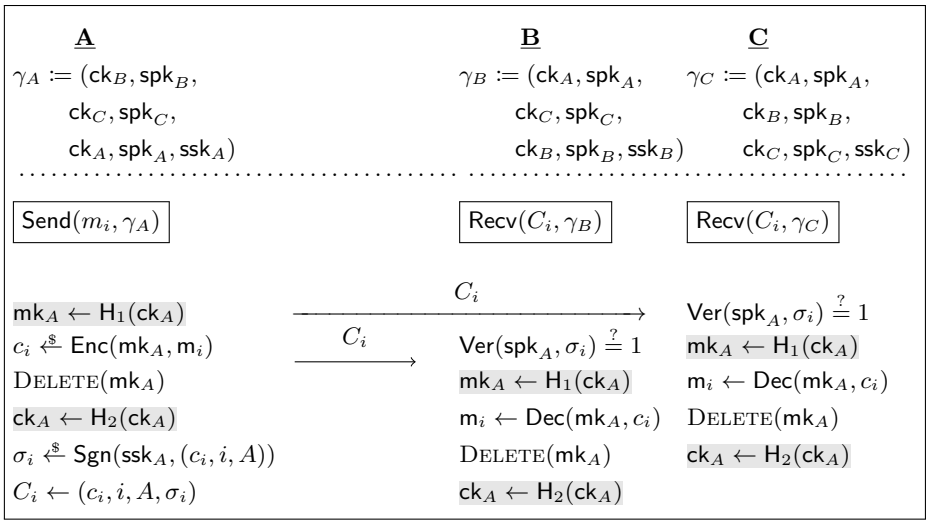

Figure 1. Simplified diagram for sending/receiving messages between three group members. For $ID \in \{A, B, C\}$, ID's initial sender key is $(\mathsf{ck}_{ID}, \mathsf{spk}_{ID})$. The state γ_{ID} of ID contains the sender keys of all group members.

Modelling two-party channels. Our modelling starts in Section 2 with the introduction of a primitive 2PC for two-party channels. We define a two-party channel with standard initialisation (Init), send (Send) and receive (Recv) algorithms. Building on top of [ACD19] and [BBL+22], our research draws inspiration from their work in parameterising the PCS of the underlying channels. Our security model captures both forward security and post-compromise security. To model PCS, we introduce a crucial parameter, denoted as Δ and referred to as the *PCS bound*. This parameter serves as an upper bound on the number of communication steps or *channel epochs* required to restore security following a compromise.

Formally capturing the security of two-party channels is central to our analysis of Sender Keys. In particular, two-party channels that are not regularly used can undermine security. For example, if a group member ID's state is compromised, there is no guarantee that new, independent sender keys that are then sampled by other members are secure, since ID's two-party channels may not yet have healed yet. Moreover, two-party channels can take more than one round trip to heal when using the Double Ratchet, as is the case for WhatsApp and the Signal Messenger [ACD19].

Our primitive: Group Messenger. In Section 3, we define a new cryptographic primitive, Group Messenger (GM), which includes five stateful algorithms that: initialise a party's state (Init), send an application message (Send), receive an application message (Recv), execute a change proposal in the group (Exec), and process a change in the group (Proc). Supported group changes are: group creation, member addition, member removal, and sender key updates. Note this contrasts with the three-phase propose/commit/process flow for updates (the so-called propose-commit paradigm [ACJM20]) used in newer CGKA protocols.

We define a game-based security notion for GM that captures a partially active adversary with control over the Delivery Service, taking inspiration from previous CGKA modelling [ACJM20, BCV23]. However, CGKA-based modelling is not readily amenable to protocols where several keys (rather than one evolved key) are managed by users at a given point in time [ACDT21, AJM22]. In our model in Section 3.1, we capture the security of each protocol by parameterising the game with a *cleanness predicate*, which excludes trivial attacks and reflects security weaknesses. Cleanness

predicates can optionally be defined in our model as explicitly assuming the use of two-party channels, which is the approach we follow for modelling Sender Keys.

Formal analysis of Sender Keys. With this formalism established, in Section 5.1 we define cleanness predicates for Sender Keys that precisely captures its security. We define three sub-predicates that restrict the capabilities of the adversary for message *challenges*, capturing confidentiality; for message *injections*, capturing integrity and authenticity; and for re-orderings and forgeries of control messages (*concurrency*), capturing the message ordering provided by the central server.

Notably, we need to impose severe restrictions in challenges and injections due to the weak form of PCS achieved by the protocol through key updates. For concurrency, we require that all control messages are delivered in-order and generated honestly, since they are not properly authenticated. Moreover, whenever key material is sent over a two-party channel that has not been refreshed (parametrized by the PCS bound Δ), such key material does not have the desired healing effect, which results in an important security shortcoming. Our predicates, via integration with the two-party channel modelling, capture the impact of the concrete FS and PCS offered by these channels in the security of the group messages.

Shortcomings and proposed improvements. The limitations that we observe on the security of Sender Keys are notable, as elegantly captured by the cleanness predicates. Leaving aside the security limitations that are intrinsic to the design of the protocol, we find that one can improve security in several other aspects without adding a large efficiency overhead. Hence, in Section 6 we propose modifications to the protocol with the aim of securing group membership, strengthening the (weaker than expected) forward security for authentication, and integrating efficient post-compromise security updates. Notably, our novel PCS update mechanism improves both the trivial solution of erasing all keys and starting over (with $\mathcal{O}(n^2)$ complexity), and the naive key update mechanism implemented by the core protocol and performed in Signal. Moreover, as a result of our modular approach with respect to modelling two-party channels, we can capture the security improvement (or weakening) that results from replacing the Double Ratchet by an alternative two-party messaging protocol.

We extend the techniques used in the original proof to establish the security of our modified protocol, called Sender Keys+. The main technical step involves redefining the cleanness predicates, which are strictly less restrictive compared to those used for the original protocol. Notably, the adversary is now allowed to inject control messages (given the group has recovered from any state exposures). We also allow the adversary to mount more fine-grained attacks for application message forgeries, and allow challenges after *some* party has updated over a refreshed channel (as opposed to strictly the updater).

1.3 Additional Related Work

A notable direction of research revolves around the MLS protocol [BBR+23] and the CGKA abstraction [ACDT20]. This line of work started with asynchronous ratchet trees [CCG+18] and quickly led to TreeKEM [BBR18] and its variants [KPPW+21, ACJM20, AJM22]. Some approaches deviate from a tree-based approach [HKP+21, AHKM22]: although they have $\mathcal{O}(n)$-sized ciphertexts in all cases, their relative simplicity makes them attractive and for smaller groups can be competitive in performance.

The formal extension of CGKA to group messaging was explored in [ACDT21], while the key schedule of MLS was proven secure in [BCK21]. Concurrently, the work of [CEST22] shares broad similarities with ours as it also constructs group messaging from two-party channels. By contrast, beyond modelling Sender Keys, we delve into evaluating security and performance trade-offs within the core messaging protocol and formally capturing out-of-order message delivery and dynamic groups. Moreover, they do not model two-party channels as a standalone primitive, and their protocols require more interaction than ours (e.g. the initial group key agreement protocol can take several rounds).

Concurrency, a crucial aspect in CGKA-based protocols, has been a central topic in works such as [AAN+22b, AAN+22a, BDR20]. Secure administration in CGKAs was explored in [BCV23]. In [WKHB21] a Sender Keys-like approach is utilized to construct a decentralised CGKA protocol but they do not capture group messaging, and their security model does not support message injections (hence considering a passive adversary). Moreover, the theorems in their work assume a non-adaptive adversary where the adversary must announce all queries at the game's outset. This work extends the scope to decentralised networks without a central authority, diverging from existing approaches that target centralised networks with a trusted server. A simplified (notably lacking forward security) decentralised variant of Sender Keys is implemented by the Session app [Jef20].

Also relevant to our work are secure two-party messaging protocols that propose alternatives to the Double Ratchet [MP16a], such as [JS18, PR18, DV19, ACD19, BRV20, PP22]. Inspired by more practical-oriented endeavors, we acknowledge recent cryptographic audits conducted on Telegram [AMPS22], Matrix [ACDJ23], Threema [KGP23], and most recently WhatsApp's backup service [DFG+23].

Sender Keys. While some works on Sender Keys lack formalism and security proofs, they offer valuable insights. In [RMS18] the authors evaluate Sender Keys, provide a high-level description of the protocol, and examine practical vulnerabilities in WhatsApp group chats. Multi-group security and key update mechanisms for Sender Keys are informally discussed in [CHK21]. In [BCG22], a preliminary analysis of the security of Sender Keys is carried out. While the paper only includes informal discussions and no proofs, it serves as an initial exploration for the ideas in the present work. We remark that the scope of the paper is limited, as it does not formally develop a security model, and assumes that all two-party channels used by Sender Keys are *perfectly secure*, which is unrealistic and impossible to develop in practice.

Concurrent work by Albrecht, Dowling and Jones [ADJ24] develops a device-oriented security model and a proof for a recent specification of Matrix (i.e., for the updated protocol that mitigates the issues described in [ACDJ23]). For group messaging, Matrix implements the Megolm protocol, which is Sender Keys-inspired but still deviates significantly from our description in this work, particularly regading server interaction. Remarkably, [ADJ24] and our work arrive to similar conclusions in our analysis, such as the insecurity of group management and the challenges imposed by message ordering. Our works are complimentary and open new research directions. Examples include exploring whether the improvements behind Sender Keys+ can also be applied to Megolm, as well as extending our modelling to consider the (challenging) multi-device setting as they do.

2 Preliminaries

Notation. Unless otherwise stated, all algorithms are probabilistic, and $(x_1, \dots) \xleftarrow{\$} \mathcal{A}(y_1, \dots)$ is used to denote that $\mathcal{A}$ returns $(x_1, \dots)$ when run on input $(y_1, \dots)$. Blank values are represented by $\perp$, which we return in case of algorithm failure. We denote the security parameter by λ and its unary representation by 1^λ. We also define the state γ of a user ID as the data required by ID for protocol execution, including message records, group-related variables, and cryptographic material. We store such material in dictionaries $M[\cdot]$ and write $a \leftarrow M[b]$ to assign, to a, the value stored in M under key b. All dictionaries can optionally be indexed by an oracle query q to represent the state of a dictionary at the time q is made, e.g., $\mathcal{E}[ID; q]$ denotes the value of $\mathcal{E}[ID]$ at the beginning of query q. We define the clause **require** P on a logical predicate P that immediately returns $\perp$ if P is not satisfied (or **false** if the algorithm returns a boolean value). For two sets $\mathcal{S}$ and $\mathcal{T}$, let $\mathcal{S} \xleftarrow{\cup} \mathcal{T}$ denote the reassignment of $\mathcal{S}$ to the set $\{s : s \in \mathcal{S} \vee t : t \in \mathcal{T}\}$, and let $\mathcal{S} \xleftarrow{-} \mathcal{T}$ denote the reassignment of $\mathcal{S}$ to the set $\{s \in \mathcal{S} : s \notin \mathcal{T}\}$. To indicate that certain variable values are not crucial to the algorithm's logic, we use "$\cdot$" notation. For instance, $\texttt{Receive}(ID, C) = (\cdot, ID', e''_{\mathsf{2pc}}, i''_{\mathsf{2pc}})$ denotes that the first variable can take any value. We defer the definitions of standard cryptographic primitives used throughout this work to Appendix A.

2.1 Two-Party Channels

Towards defining our Group Messenger primitive with support for two-party channels, we define them below as a standalone primitive.

Definition 1 (Two-Party Channels). *A two-party channel scheme* $\mathsf{2PC} := (\mathsf{Init}, \mathsf{InitCh}, \mathsf{Send}, \mathsf{Recv})$ *is defined as the following tuple of* PPT *algorithms.*

- $\gamma \xleftarrow{\$} \mathsf{Init}(ID)$*: Given a user identity* ID*, the probabilistic initialisation algorithm returns an initial state* γ*.*
- $b \xleftarrow{\$} \mathsf{InitCh}(ID^*, \gamma)$*: Given a state* γ *and a user identity* ID^**, the probabilistic channel initialization algorithm returns an acceptance bit* $b \in \{0, 1\}$ *and updates the caller's state.*
- $(C, e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) \xleftarrow{\$} \mathsf{Send}(\mathsf{m}, ID^*, \gamma)$*: Given a message* m*, the intended message recipient* ID^* *and a state* γ*, the probabilistic sending algorithm returns a ciphertext* C *and a channel epoch-index pair* $(e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$ *corresponding to* m *(or* $\perp$ *upon failure), and updates the state.*
- $(\mathsf{m}, ID^*, e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) \xleftarrow{\$} \mathsf{Recv}(C, \gamma)$*: Given a ciphertext* C *and a state* γ*, the deterministic receiving algorithm returns a message* m*, a user identity* ID^* *corresponding to the sender of* m *and a channel epoch-index pair* $(e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$ *corresponding to* m *(or* $\perp$ *upon failure), and updates the state.*

Our $\mathsf{2PC} := (\mathsf{Init}, \mathsf{InitCh}, \mathsf{Send}, \mathsf{Recv})$ primitive captures two initialisation functions. The first function initialises the state of a party by taking its ID as input, while the second function is used to initialize a communication channel with a counterpart ID^*. Consider two parties, ID and ID^* who intend to communicate over a two-party channel. Both parties initialise their states, γ_{ID} and γ_{ID^*} using the Init function. Subsequently, ID (or ID^*) initiates the communication channel by invoking the $\mathsf{InitCh}(\gamma_{ID}, ID^*)$ (or $\mathsf{InitCh}(\gamma_{ID^*}, ID)$) function. It is worth noting that, similar to DCGKA [WKHB21], our two-party channel primitive assumes the presence of a public-key infrastructure, which is omitted here for simplicity.

We adopt the notion of channel epochs from [ACD19], such that in each two-party channel, ID and ID' are associated with a channel epoch e_{2pc}, indicating the number of times the direction of communication has changed (alongside a message index i_{2pc}). Appendix B provides a more detailed description of the semantics, including the correctness notion.

Security. The Double Ratchet protocol has been the subject of several academic works [ACD19, CCD+20, CJSV22, BFG+22] that analyse its security on a fine-grained level for two-party communication. When used by multiple parties in a group during the execution of Sender Keys, analysing the Double Ratchet protocol becomes complex, making it difficult to replace it with other protocols. To tame this complexity, we adopt a comparatively simpler notion of two-party communication in similar complexity to the formalism of Weidner et al. for their DCGKA protocol [WKHB21]. Our modelling captures forward security, is parameterised by the post-compromise security of the underlying channels and supports out-of-order message delivery.

Definition 2 (Security of $\mathsf{2PC}$). *Let $\mathsf{2PC}$ be a two-party channels scheme.* Two-party channel indistinguishability *or 2PC-IND security parameterised by $b \in \{0,1\}$, cleanness predicate C and PCS bound $\Delta > 0$ for $\mathsf{2PC}$ is defined via the game $\mathbf{2PC\text{-}IND}^{\mathcal{A}}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ depicted in Figure 20. We define the advantage of adversary $\mathcal{A}$ in $\mathbf{2PC\text{-}IND}^{\mathcal{A}}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ as*

$$\mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{A}) := \left|\Pr[\mathbf{2PC\text{-}IND}^{\mathcal{A}}_{\mathsf{2PC},1,\mathsf{C}_{\mathsf{2pc}},\Delta} \Rightarrow 1] - \Pr[\mathbf{2PC\text{-}IND}^{\mathcal{A}}_{\mathsf{2PC},0,\mathsf{C}_{\mathsf{2pc}},\Delta} \Rightarrow 1]\right|.$$

We say that $\mathsf{2PC}$ is (ϵ, q)-$\mathbf{2PC\text{-}IND}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}$ if for all PPT adversaries $\mathcal{A}$ who make at most q oracle queries, we have $\mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{A}) \leq \epsilon$.

The notion of security defined in the $\mathbf{2PC\text{-}IND}^{\mathcal{A}}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ game in Figure 20 is parameterised by a cleanness predicate C and a PCS bound $\Delta > 0$. Broadly, the cleanness predicates C prevent the adversary from winning the game by making a trivial attack, i.e., via a bit guess (resp. forgery) based on a challenge (resp. delivery) using exposed key material. PCS after an exposure is parameterised by Δ, which is the number of message round-trips (i.e. number of times that the sender-receiver roles alternate) that the channel requires for healing. We defer the precise description, including Figure 20 and the predicates, to Appendix B.

Instantiations. By previous work [ACD19], the Double Ratchet can be seen to achieve a PCS bound of $\Delta = 3$. However, by replacing the Diffie Hellman key exchange component in the Double Ratchet (referred to as continuous key agreement [ACD19]) with a KEM, the PCS bound can be improved to $\Delta = 2$. This is optimal since if a user is exposed in channel epoch e_{2pc} and acts as the sender, then they can decrypt a message from channel epoch $e_{\mathsf{2pc}} + 1$ based on correctness requirements. While larger values of Δ are possible, including $\Delta = \infty$ by never injecting new randomness in key derivation, protocols lacking forward security are considered insecure within our model.

For channel initialisation InitCh, an initial key exchange between the parties needs to be carried out. Typically this is done via the asynchronous X3DH protocol [MP16b] and by relying on a PKI, that we abstract away in this work.

3 Group Messenger

We introduce our main cryptographic primitive called *Group Messenger*. In contrast to abstractions such as Continuous Group Key Agreement (CGKA) [ACDT20], our primitive is not restricted to

only modelling key agreement based protocols, since we capture sending and receiving application messages.

Definition 3 (Group Messenger). *A Group Messenger* $\mathsf{GM} := (\mathsf{Init}, \mathsf{Send}, \mathsf{Recv}, \mathsf{Exec}, \mathsf{Proc})$ *is defined as the following tuple of* PPT *algorithms.*

$\gamma \xleftarrow{\$} \mathsf{Init}(ID)$**:** *Given a user identity ID, the probabilistic initialisation algorithm returns an initial state* γ*.*

$C \xleftarrow{\$} \mathsf{Send}(\mathsf{m}, \gamma)$**:** *Given a message* m *and a state* γ*, the probabilistic sending algorithm returns a ciphertext* C *(or* $\perp$ *upon failure) and updates the state.*

$(\mathsf{m}, ID^*, e, i) \xleftarrow{\$} \mathsf{Recv}(C, \gamma)$**:** *Given a ciphertext* C *and a state* γ*, the deterministic receiving algorithm returns a message* m*, an identity* ID^* *corresponding to the sender, a group epoch* e *and index* i *both corresponding to* m *(or* $\perp$ *upon failure), and updates the state.*

$T \xleftarrow{\$} \mathsf{Exec}(\mathtt{cmd}, \boldsymbol{IDs}, \gamma)$**:** *Given a command* $\mathtt{cmd} \in \{\mathtt{crt}, \mathtt{add}, \mathtt{rem}, \mathtt{upd}\}$*, a list of identities* $\boldsymbol{IDs}$ *and a state* γ*, the probabilistic execution algorithm returns a control message* T *(or* $\perp$ *upon failure) and updates the state.*

$b \xleftarrow{\$} \mathsf{Proc}(T, \gamma)$**:** *Given a control message* T *and a state* γ*, the deterministic processing algorithm outputs an acceptance bit* $b \in \{0, 1\}$ *and updates the state.*

In our syntax, a distinction is made between application messages and control messages. Specifically, distinct algorithms are employed for the transmission and reception of application messages, as well as for the execution and processing of group modifications. These modifications, executed via Exec, are parameterised by a command `cmd` that encompasses various operations such as user addition `add`, removal `rem`, group creation `crt`, or key material update for users `upd`. Moreover, in scenarios where two-party messaging protocols are necessitated for the implementation of the group primitive (although not applicable to CGKAs such as TreeKEM [ACDT20]), these protocols are assumed to be encapsulated within ciphertexts or control messages. Consequently, they are abstracted away from the core definition presented herein. Looking ahead, we will enforce that ciphertexts and control messages are sent alongside two-party channel ciphertexts (and can be received with a different ciphertext or control message) when we define security in Section 3.1.

Message epochs. We define a message epoch as a pair[3] of integers (e, i), internal to the state γ of a party ID, that captures time and synchronisation between parties. Message epochs are central in our description of Sender Keys and in our security model. Each application message sent by ID corresponds to a single message epoch (e, i), which is output by the Recv algorithm at the receiver's end. The epoch e advances whenever ID processes a new group change (i.e., a control message). The index i advances when ID sends a new message. If control messages are delivered to group members in lockstep, parties who have the same epoch e will have the same view of the group membership. We define a total ordering $(e, i) \leq (e', i')$ when $e < e'$, or $e = e'$ and $i < i'$. Nevertheless, we remark that 2PC channel epochs are independent from GM message epochs.

Oracles for Correctness and Security. We introduce game-based notions for GM correctness and security, where the adversary $\mathcal{A}$ will have access to various oracles that we outline below.

[3] One can alternatively use an auxiliary variable aux for metadata.

`Create`$(ID, \boldsymbol{IDs})$**:** creates a group by executing $\mathsf{Exec}(\mathtt{crt}, \boldsymbol{IDs}, \gamma)$ with ID as the initiator, generating a control message T.

`Challenge`$(ID, \mathsf{m}_0, \mathsf{m}_1)$**:** outputs a ciphertext C_b corresponding to the message m_b sent by ID, for $b \xleftarrow{\$} \{0,1\}$. Namely, $\mathcal{A}$ obtains $C_b \leftarrow \mathsf{Send}(\mathsf{m}_b, \gamma[ID])$.

`Send`(ID, m)**:** ID sends an application-level message m using the Send algorithm, producing a ciphertext C.

`Receive`(ID, C)**:** ID receives a ciphertext C by calling $\mathsf{Recv}(C, \gamma[ID])$. The sender ID' is inferred from the message as output by Recv. In the event of a successful forgery, $\mathcal{A}$ obtains the value of b.

`Add`(ID, ID')/`Remove`(ID, ID')/`Update`(ID)**:** ID adds ID' / removes ID' / refreshes ID's secrets by calling $\mathsf{Exec}(\mathtt{add}, ID', \gamma[ID])$ / $\mathsf{Exec}(\mathtt{rem}, ID', \gamma[ID])$ / $\mathsf{Exec}(\mathtt{upd}, ID, \gamma[ID])$, generating control message T.

`Deliver`(ID, T)**:** ID is delivered a control message T via $\mathsf{Proc}(T, \gamma[ID])$.

`Expose`(ID)**:** Leaks the γ of ID to $\mathcal{A}$.

Correctness. To ensure correctness, several properties must be satisfied given that all messages are generated honestly.

- *Message delivery*: Application messages (generated by `Send`) must be received correctly by all group members.
- *Group evolution*: Group operations, such as `crt` (group creation), `add` (user addition), `rem` (user removal), and `upd` (key update), must have their intended effects on the group when received and processed.
- *Group membership consistency*: The list of group members must be consistent among all group members, assuming they process the same sequence of control messages.
- *Out-of-order delivery*: Messages corresponding to past epochs must be decryptable if delivered out-of-order. Messages corresponding to future epochs must be rejected upon reception.

To formally capture correctness, a game-based correctness notion can be considered between a challenger and a computationally unbounded adversary. In the correctness game, the adversary initiates the protocol by calling the `Create` oracle once. The adversary can use the `Send` and `Receive` oracles as usual, and also has access to the `Add`, `Remove`, `Update`, `Deliver` and `Expose` oracles. However, the `Challenge` oracle may not be called, and `Send` and `Deliver` can only be called on honestly generated ciphertexts or control messages.

Looking ahead, the predicates used in the correctness analysis need to be modified from those used in security to suit the context. Notably, there is no need for a challenge predicate since the challenge oracle is not allowed. However, the concurrency predicate (defined later) is essential, addressing situations where members propose concurrent group changes or process group changes in different orders.

3.1 Security Model

We introduce a game-based model of security for our Group Messenger primitive that captures the main desirable security properties of a group messaging scheme. In brief, our game $\mathbf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}}$ captures a partially active adversary who can, in particular, expose the state of users and inject (possibly malformed) messages at any time. We consider the confidentiality of application messages with FS and PCS, and we also model the out-of-order delivery of application and control messages.

Definition 4 (Message indistinguishability of GM). Let $\mathsf{GM} := (\mathsf{Init}, \mathsf{Send}, \mathsf{Recv}, \mathsf{Exec}, \mathsf{Proc})$ be a group messenger. *Message indistinguishability* with $b \in \{0,1\}$ and cleanness predicate C for GM is defined via the game $\mathbf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}}$ depicted in Figure 2. We define the advantage of adversary $\mathcal{A}$ in $\mathbf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}}$ as

$$\mathsf{Adv}^{\mathbf{m\text{-}ind}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) := \left|\Pr[\mathbf{M\text{-}IND}^{\mathcal{A}}_{\mathsf{GM},1,\mathsf{C}} \Rightarrow 1] - \Pr[\mathbf{M\text{-}IND}^{\mathcal{A}}_{\mathsf{GM},0,\mathsf{C}} \Rightarrow 1]\right|.$$

We say that GM is (q,ϵ)-$\mathbf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}}$ if for all PPT adversaries $\mathcal{A}$ who make at most q oracle queries we have $\mathsf{Adv}^{\mathbf{m\text{-}ind}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \epsilon$.

Game $\mathbf{M\text{-}IND}^{\mathcal{A}}_{\mathsf{GM},b,\mathsf{C}}$

for all ID :
  $\gamma[ID] \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID)$
$\mathcal{T}[\cdot], \mathcal{M}[\cdot], \mathcal{CH}[\cdot], \mathcal{SM}[\cdot] \leftarrow \bot$
$\mathsf{ep} \leftarrow 0$
$\mathcal{E}[\cdot], \mathcal{I}[\cdot] \leftarrow 0$
$b' \leftarrow \mathcal{A}^{\texttt{Create},\ldots,\texttt{Receive}}$
require C
return b'

Oracle $\texttt{Create}(ID, \boldsymbol{IDs})$ // Called only once

require $ID \in \boldsymbol{IDs}$
require $\mathsf{ep} = 0$
$T \xleftarrow{\$} \mathsf{Exec}(\texttt{crt}, \boldsymbol{IDs}, \gamma[ID])$
$\mathcal{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep} + 1$
return T

Oracle $\texttt{Send}(ID, m)$

$C \xleftarrow{\$} \mathsf{Send}(m, \gamma[ID])$
require $C \neq \bot$
$\mathcal{I}[ID] \leftarrow \mathcal{I}[ID] + 1$
$\mathcal{M}[ID, \mathcal{E}[ID], \mathcal{I}[ID]] \leftarrow C$
return C

Oracle $\texttt{Challenge}(ID, \mathsf{m}_0, \mathsf{m}_1)$

require $|\mathsf{m}_0| = |\mathsf{m}_1|$
$C^* \xleftarrow{\$} \mathsf{Send}(m_b, \gamma[ID])$
require $C^* \neq \bot$
$\mathcal{I}[ID] \leftarrow \mathcal{I}[ID] + 1$
$\mathcal{CH}[ID, \mathcal{E}[ID], \mathcal{I}[ID]] \leftarrow C^*$
return C^*

Oracle $\texttt{Expose}(ID)$

return $\gamma[ID]$

Oracle $\texttt{Add}(ID, ID')$

$T \xleftarrow{\$} \mathsf{Exec}(\texttt{add}, \{ID'\}, \gamma[ID])$
require $T \neq \bot$
$\mathcal{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep} + 1$
return T

Oracle $\texttt{Remove}(ID, ID')$

$T \xleftarrow{\$} \mathsf{Exec}(\texttt{rem}, \{ID'\}, \gamma[ID])$
require $T \neq \bot$
$\mathcal{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep} + 1$
return T

Oracle $\texttt{Update}(ID)$

$T \xleftarrow{\$} \mathsf{Exec}(\texttt{upd}, \{ID\}, \gamma[ID])$
require $T \neq \bot$
$\mathcal{T}[\mathsf{ep}] \leftarrow T;\ \mathsf{ep} \leftarrow \mathsf{ep} + 1$
return T

Oracle $\texttt{Deliver}(ID, T)$

$\mathsf{acc} \leftarrow \mathsf{Proc}(T, \gamma[ID])$
require acc
$\mathcal{E}[ID] \leftarrow \mathcal{E}[ID] + 1$
$\mathcal{I}[ID] \leftarrow 0$
if $\mathsf{proc\text{-}forgery}(T)$:
  return b // $\mathcal{A}$ wins
return

Oracle $\texttt{Receive}(ID, C)$

$(m, ID', e, i) \leftarrow \mathsf{Recv}(C, \gamma[ID])$
if $m \neq \bot$:
  Update $\mathcal{SM}[ID, ID']$
  if $\mathsf{recv\text{-}forgery}(C)$:
    return b // $\mathcal{A}$ wins
return

Figure 2. Game defining $\mathbf{M\text{-}IND}^{\mathcal{A}}_{\mathsf{GM},b,\mathsf{C}}$ with adversary $\mathcal{A}$ and cleanness predicate C. Lines in teal correspond only to bookkeeping and state update operations. All oracles except for `Create` and `Expose` can only be called when $\mathsf{ep} > 0$.

Game Description. The $\textbf{M-IND}_{\mathsf{GM},\mathsf{C}}$ game that we introduce in Figure 2, is played between a PPT adversary $\mathcal{A}$ and a challenger. The game is parameterised by a bit b that has to be guessed by $\mathcal{A}$, as in a message indistinguishability game. The adversary wins the game if it guesses b correctly, or if it carries out a successful forgery. The game is further parameterised by a protocol-specific *cleanness* predicate (sometimes safety predicate [ACDT20]) that rules out trivial attacks and captures the exact security of the protocol.

Message epochs. We define a function $\mathsf{m\text{-}ep}(ID, ID', q)$ that indicates the highest message epoch (e, i), as output by the Recv algorithm, for which a user ID has received a message from ID' at the time of query q for $ID \neq ID'$. For $ID = ID'$, this indicates the local state $(\mathcal{E}[ID], \mathcal{I}[ID])$. The m-ep function reflects the view of user ID' by user ID.

Dictionaries. The challenger keeps a record of messages and game variables in several dictionaries. The state of each party is stored in $\gamma[\cdot]$ and updated when an algorithm is called on a given $\gamma[ID]$. Ciphertexts and challenged keys are stored in $\mathcal{M}$ and $\mathcal{CH}$, respectively, each of them indexed by an ID and a message epoch (e, i). The unique honest control message that starts a given epoch e is stored in $\mathcal{T}[e]$, and the most recent epoch of the group is stored in variable ep (note that we implicitly assume a total ordering of control messages). The current message epoch of ID is stored in $\mathcal{E}[ID], \mathcal{I}[ID]$. Even if each control message in $\mathcal{T}$ corresponds to a single epoch, different parties can be in different epochs. We say ID is in epoch e before query q if the last control message processed by ID before query q is $\mathcal{T}[e]$.

The message epochs corresponding to skipped messages from sender ID' stored by ID are kept in $\mathcal{SM}[ID, ID']$. We keep $\mathcal{SM}$ updated in the `Receive` oracle as follows: given a message epoch (e, i) and an ID' output by Recv, if $(e, i) \in \mathcal{SM}[ID, ID']$ then (e, i) is erased from $\mathcal{SM}[ID, ID']$. Otherwise, we add all pairs (e', i') such that $(e', i') < (e, i)$ and (e', i') corresponds to all messages sent by ID' not delivered to ID.

Outcome. After q oracle queries, $\mathcal{A}$ outputs a guess b' of b if the cleanness predicate C is satisfied (otherwise the game aborts). $\mathcal{A}$ can win the game in three different ways: by guessing a challenge correctly, by injecting a forged application message via `Receive` successfully, or by injecting a forged control message via `Deliver`. The cleanness predicate C parameterises the security of a given protocol by restricting the capabilities of the adversary to exclude a class of attacks. Additionally, we explicitly state predicates recv-forgery and proc-forgery in our game, which model the (general, not protocol-specific) conditions under which a `Receive` or `Deliver` call result in a successful forgery (leaking b to $\mathcal{A}$). We expand on these predicates in Section 3.2.

Related Security Notions. Our security model takes inspiration from the game-based modelling developed for MLS and for CGKA [ACDT20, KPPW+21]. Nevertheless, it is not possible to adopt these models as they consider a single group key, which is not compatible with a Sender Keys (or similar) approach to group messaging. The closest security model to ours in the literature comes from the DCGKA scheme [WKHB21], which however does not consider message injections nor adaptive security.

Limitations. Our security game allows a single successful injection to occur, since after this point the adversary is given the secret key for free. That is, we do not allow 'trivial' message forgeries that do not result in the adversary winning the game. Hence, full active security cannot be captured by

our modelling. Like several other models in the literature (e.g., [ACDT20, KPPW+21, WKHB21]), our security model considers a single group (see [CHK21] for an analysis of cross-group security) and ignores randomness exposure or manipulation [BRV20].

3.2 Modelling Two-Party Channel Ciphertexts

Given that the GM protocol uses two-party channels (as Sender Keys does), these need to be modelled accurately within the GM security game, particularly to describe forgeries via the recv-forgery and proc-forgery predicates. We introduce additional notation to define how two-party ciphertexts can be sent alongside GM messages; we opt for such modelling for convenience, as in this way the adversary gets access to all two-party channels explicitly. We remark that this subsection can be skipped for GM protocols that do not employ two-party channels.

Essentially, we want to capture the fact that an Exec or Send call can output *several* two-party channel ciphertexts, whereas Proc and Recv should only take as input a *single* two-party channel ciphertext (i.e., the one intended for the caller) for efficiency. We thus assume input/output ciphertexts and control messages for group messenger algorithms take the following form. Let C_{2pc} be a 2PC ciphertext and let T_{core} (resp. C_{core}) be the remaining part of a control (resp. application) message in the GM primitive. For output, we assume control messages output by Exec are of the form $(T_{\mathsf{core}}, C^1_{\mathsf{2pc}}, \ldots, C^k_{\mathsf{2pc}})$, and ciphertexts output by Send are of the form $(T_{\mathsf{core}}, C^1_{\mathsf{2pc}}, \ldots, C^k_{\mathsf{2pc}})$ for some k. For input, we assume control messages input to Exec (resp. to Recv) are of the form $(T_{\mathsf{core}}, C_{\mathsf{2pc}})$ (resp. $(C_{\mathsf{core}}, C_{\mathsf{2pc}})$).

Forgery predicates. We define the predicates proc-forgery and recv-forgery in Figure 3 used in Figure 2 using the input/output semantics introduced above. The purpose of these predicates is to handle ciphertext 'splitting' resulting from the use of two-party channels. Without accounting for this splitting, forgeries could be defined as usual, i.e. any ciphertext input to Proc (resp. Recv) that was not previously output by Exec (resp. Proc) would be considered a forgery. Essentially, we consider that a control message $T^* = (T^*_{\mathsf{core}}, C^*_{\mathsf{2pc}})$ is a forgery whenever either T^*_{core} or C^*_{2pc} are not part of an honestly generated message (i.e. in $\mathcal{T}[\cdot]$). Forgeries for Recv are defined analogously.

$$\boxed{\mathsf{proc\text{-}forgery}(T^* = (T^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}))} : \nexists \left\{(T, \vec{C}), (T', \vec{C}')\right\} \subseteq \mathcal{T}[\cdot] :$$
$$(T^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}) \in \left\{(T, C_i), (T', C_i), (T, C'_j)\right\} \wedge (C_i \in \vec{C}) \wedge (C'_j \in \vec{C}')$$
$$\boxed{\mathsf{recv\text{-}forgery}(C = (C^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}))} : \nexists \left\{(C_0, \vec{C}), (C'_0, \vec{C}')\right\} \subseteq \mathcal{CH}[\cdot] \cup \mathcal{M}[\cdot] :$$
$$(C^*_{\mathsf{core}}, C^*_{\mathsf{2pc}}) \in \left\{(C_0, C_i), (C'_0, C_i), (C_0, C'_j)\right\} \wedge (C_i \in \vec{C}) \wedge (C'_j \in \vec{C}')$$

Figure 3. Predicates that determine what is considered a forgery in Figure 2 for algorithms Proc and Recv.

The predicates imply that it is *not* considered a forgery if a two-party ciphertext is received with a different control message/ciphertext than it was sent with. That is, the adversary is allowed to mix-and-match ciphertexts, i.e., by replacing the C_i corresponding to some T (resp. C_0) by C'_i corresponding to some other T' (resp. C'_0).

4 Sender Keys

Two-Party Channels and the Server. The Sender Keys protocol assumes the existence of authenticated and secure two-party communication channels between each pair of users, which can be achieved through the use of Signal's Double Ratchet protocol [MP16a] as done in WhatsApp [Wha20]. Additionally, the protocol relies on a central server to distribute both control messages and application messages. We assume that the server provides a *total ordering for control messages*, ensuring that all parties process control messages in the correct order.[4] Total ordering is not required for application messages. User authentication is initially performed via the central server (modelled here with $\mathsf{2PC.InitCh}$), after which users authenticate other group members through the underlying two-party communication channel. We note that this deviates from other work in the literature such as [AJM22] where the *authentication service* is different to the *delivery service*.

4.1 Protocol

We describe the Sender Keys protocol in our GM syntax according to the details inferred from [Wha20] and [M+16], although we acknowledge that our interpretation may not precisely match the closed-source implementation of WhatsApp. In this section we present a detailed overview of the main algorithms depicted in Figure 4. For Exec and Proc, we only present the remove operation as it involves key refreshing and is considered the most complex, while the create, add, and update operations follow a similar approach. For the sake of clarity, we make minor simplifications, but the complete protocol logic can be found in Figures 23 to 25 (in Appendix E, where we also provide supplementary descriptions of the protocol logic).

Primitives. The protocol relies on standardised primitives including a symmetric encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, a signature scheme $\mathsf{Sig} = (\mathsf{Gen}, \mathsf{Sgn}, \mathsf{Ver})$, and two different key derivation functions $\mathsf{H}_1, \mathsf{H}_2$, (our improvements also make use of message authentication codes). We include formal definitions in Appendix A.

State initialisation. Each user is assumed to maintain a state γ containing: a secret key used for signing ssk, a list of group members $\boldsymbol{G}$, the current epoch ep, the current index of their chain key $\mathsf{i_{ck}}$, a list of key counters kc (variables that indicate the number of times that a sender key has been refreshed since ID initialised their state), a dictionary of sender keys $\mathsf{SK}[\cdot] := (\mathsf{spk}_{ID}, \mathsf{ck}_{ID}, \mathsf{i_{ck}})$ indexed by a user ID and a key counter, and a list of message keys $\mathcal{MK}$. The Init algorithm initialises the state variable of users; in practice this is done by a user when they install the messaging application.

Group creation. This occurs via $\mathsf{Exec}(\mathtt{crt}, \boldsymbol{IDs}, \gamma)$, which takes a list of users $\boldsymbol{G} := \{ID_1, \ldots, ID_{|\boldsymbol{G}|}\}$ as input; two-party channels are initialised by users upon processing the control message via $\mathsf{2PC.InitCh}$.

[4] We remark that total ordering is a standard assumption in the CGKA line of work [ACDT20, KPPW+21, ACJM20, AJM22, ACDT21]

$\underline{\mathsf{Init}(ID, 1^\lambda)}$

$\gamma.ME \leftarrow ID$
$\gamma.(\mathsf{ssk}, \boldsymbol{G}, \mathsf{ep}, \mathsf{i}_{ME}) \leftarrow \bot$
$\gamma.(\mathsf{SK}[\cdot], \mathcal{MK}[\cdot], \mathsf{kc}[\cdot]) \leftarrow \bot$
return γ

$\underline{\mathsf{Send}(\mathsf{m}, \gamma)}$

require $ME \in \boldsymbol{G}$
if $\mathsf{SK}[ME, \mathsf{kc}[ME]] = \bot$:
// Sample fresh sender key
  $\vec{C} \leftarrow$ PreSendFirst()
if $\mathsf{i}_{ME} = 0$:
  $\vec{C} \leftarrow (\vec{C}, \text{SendToMissing}())$
$\mathsf{mk} \leftarrow \mathsf{H}_1(\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{ck})$
$c \xleftarrow{\$} \mathsf{Enc}(\mathsf{mk}, \mathsf{m})$
UpdateCK($ME, \mathsf{kc}[ME]$)
$M \leftarrow (c, (\mathsf{ep}, \mathsf{i}_{ME}), \mathsf{kc}[ME], \mathsf{i}_{\mathsf{ck}}, ME)$
$\sigma \xleftarrow{\$} \mathsf{Sig.Sgn}(\mathsf{ssk}, M)$
return $C := ((M, \sigma), \vec{C})$

$\underline{\mathsf{Exec}(\mathtt{cmd} = \mathtt{rem}, ID, \gamma)}$

require $ID \in \boldsymbol{G}$
$\vec{C}[\cdot] \leftarrow \bot$
$T \leftarrow (\mathtt{rem}, ME, ID, \mathsf{ep} + 1)$
return $(T, \vec{C})$

$\underline{\mathsf{Recv}(C = ((M, \sigma), C_{\mathsf{2pc}}), \gamma)}$

parse M **as** $(c, (e, i), \mathsf{kc}', \mathsf{i}_{\mathsf{ck}}', ID)$
require $ID \in \boldsymbol{G}$
if $\mathsf{SK}[ID, \mathsf{kc}'] = \bot$:
  $(\mathsf{SK}[ID, \mathsf{kc}'], \mathsf{kc}^*, \mathsf{ep}', \mathsf{aux}, ID^*) \leftarrow \mathsf{2PC.Recv}(C_{\mathsf{2pc}}, \gamma)$
  require $(ID, e, \mathsf{kc}') = (ID^*, \mathsf{ep}', \mathsf{kc}^*)$
  DeleteOldCK(ID, aux)
else require $C_{\mathsf{2pc}} = \bot$
require $e \leq \mathsf{ep}$
require $\mathsf{Sig.Ver}(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{spk}, \sigma, M)$
$\mathsf{mk} \leftarrow$ UpdateKeysRecv()
$\mathsf{m} \leftarrow \mathsf{Dec}(\mathsf{mk}, c)$
return (m, ID, e, i)

$\underline{\mathsf{Proc}((T = (\mathtt{rem}, ID, ID', \mathsf{ep}'), C_{\mathsf{2pc}}), \gamma)}$

require $ID \in \boldsymbol{G} \wedge C_{\mathsf{2pc}} = \bot$
require $\mathsf{ep}' = \mathsf{ep} + 1$
$\boldsymbol{G} \xleftarrow{-} \{ID'\}$
$\mathsf{ep} \leftarrow \mathsf{ep} + 1$; $\mathsf{i}_{ME} \leftarrow 0$
for all $ID' \in \boldsymbol{G}$:
  $\mathsf{kc}[ID'] \leftarrow \mathsf{kc}[ID'] + 1$
$\mathsf{SK}[ID', \cdot] \leftarrow \bot$
if $ID = ME$: $\mathsf{Init}(ME, 1^\lambda)$ // Del. γ
return true

Figure 4. Sender Keys protocol description (main operations). For conditions of the form "**require** T" when T is false, the function outputs $\bot$ and all computation is reverted. The full protocol is available in Appendix E.

Message sending. To send an application message m to the group, every $ID \in \boldsymbol{G}$ must have the caller's (ME) sender key. The process is as follows:

- If ME does not have a sender key, ME generates a fresh sender key ($(\gamma.\mathsf{ssk}, \mathsf{spk}) \xleftarrow{\$} \mathsf{Sig.Gen}(1^\lambda)$ and $\mathsf{ck} \xleftarrow{\$} \{0,1\}^\lambda$). The sender key is then set as $\mathsf{SK}[ME, \mathsf{kc}[ME]] \leftarrow (\mathsf{spk}, \mathsf{ck}, \mathsf{i}_{\mathsf{ck}} = 0)$. ME shares this key with each $ID \in \boldsymbol{G}$ using $\mathsf{2PC.Send}$, resulting in a vector of ciphertexts $\vec{C}$.
- If ME has a non-empty sender key but not all parties have it, ME shares the key with them via $\mathsf{2PC.Send}$ and updates $\vec{C}$.

Then ME generates a new message key mk from their chain key $\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{ck}$, encrypts m using mk, and ratchets its chain key forward by setting $\mathsf{ck} \leftarrow \mathsf{H}_2(\mathsf{SK}[ME, \mathsf{kc}'].\mathsf{ck})$. Finally, ME signs the ciphertext and sends it together with $\vec{C}$.

Message receiving. To receive a message from ID, ME follows these steps:

- ME checks if they have ID's sender key $\mathsf{SK}[ID, \mathsf{kc}']$ corresponding to the key counter kc' indicated in the received message. If ME does not have it, they retrieve it from the two-party ciphertext C_{2pc} using $\mathsf{2PC.Recv}$.
- ME performs epoch consistency checks and verifies the signature on the ciphertext using the signature public key $\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{spk}$.

– The message key mk required to decrypt the message is computed from the chain keys as $\mathsf{mk} \leftarrow \mathsf{H}_1(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck})$, and is deleted after use.

Out-of-order messages. In the scenario of out-of-order message delivery, the following cases arise (we let $\mathsf{i}_{\mathsf{ck}} := \mathsf{SK}[ID, \mathsf{kc}'].\mathsf{i}_{\mathsf{ck}}$):

– If the received message comes from a past epoch $(e, i) < (\mathsf{ep}, \mathsf{i}_{\mathsf{ck}})$, *ME* searches for a skipped message key in $\mathcal{MK}$.
– If $e = \mathsf{ep}$ and $i > \mathsf{i}_{\mathsf{ck}}$, *ME* ratchets *ID*'s chain key $i - \mathsf{i}_{\mathsf{ck}}$ times, and stores the skipped message keys in $\mathcal{MK}$.
– If $e > e$, the message reception fails since *ME* is not synchronised with the latest group epoch and cannot (even) determine whether the sender is still a member of the group.

Handling out-of-order message delivery constitutes a significant portion of the protocol's logic. For instance, parties must keep track of (and announce) the highest i_{ck} associated with a given kc. Failing to do so can result in correctness and security issues, as parties may overlook the need to store keys in $\mathcal{MK}$.

Key updates. In certain implementations of Sender Keys (although not specified in [Wha20]) a simple (but weak) on-demand key update mechanism is supported. A party *ME* can update its key material via $\mathsf{Exec}(\mathtt{crt}, ME, \gamma)$. This operation samples a fresh sender key $(\mathsf{spk}, \mathsf{ck}, 0)$ and distributes it over the two-party channels. All users sample a fresh key after processing a removal.

Membership changes. The protocol allows individual group members to be added or removed from the group via $\mathsf{Exec}(\mathtt{add}, ID, \gamma)$ and $\mathsf{Exec}(\mathtt{rem}, ID, \gamma)$. These operations result in the distribution of a control message T to the group sent in clear. Newly added members are also sent a welcome $\mathsf{2PC}$ ciphertext containing group information. Note that we model single adds/removes for simplicity but this can be extended in a straightforward manner to handle batched group changes.

Upon processing a control message via $\mathsf{Proc}(T, \gamma)$, *ME* proceeds as follows:

– If some ID^* is being removed at epoch e, *ME* erases all sender keys corresponding to ID^*.[5] For other users, old sender keys are replaced with new ones when receiving messages from epoch $e' \geq e$, ensuring messages sent concurrently with the removal can be received.
– If some ID^* is being added, *ME* initialises its $\mathsf{2PC}$ with ID^* via $\mathsf{2PC.InitCh}$.
– If *ME* is itself removed, it erases its state (re-initialising it). If it is added to some group (or if it processes a create message), it initialises the two-party channels with every $ID \in \boldsymbol{G}$.

Note that after adding or removing a user, new sender keys are only distributed once a party sends his first message.

5 Security

In this section, we argue that Sender Keys as described in Section 4 is secure with respect to our security model in Section 3.1. However, the security captured by our cleanness predicate is far from theoretically optimal since Sender Keys is relatively weak in security, and so in Section 6 we strengthen it by modifying the protocol in different ways. Our predicates are parameterised by the security of the underlying two-party channels.

[5] A different deletion schedule may be applied as long as these keys are clearly marked as no longer valid, e.g., if ID^* announces its maximum i_{ck} value over two-party channels when it processes its own removal.

Theorem 1. *Let* $\mathsf{SymEnc} := (\mathsf{Enc}, \mathsf{Dec})$ *be a* $(q, \epsilon_{\mathsf{sym}})$-$\mathbf{IND\text{-}CPA}_{\mathsf{SymEnc},b}$ *symmetric encryption scheme,* $\mathsf{Sig} := (\mathsf{Gen}, \mathsf{Sgn}, \mathsf{Ver})$ *a* $(q, \epsilon_{\mathsf{sig}})$-$\mathbf{SUF\text{-}CMA}_{\mathsf{Sig}}$ *signature scheme,* $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^\lambda \times \{0,1\}^\lambda$ *a* $(q, \epsilon_{\mathsf{prg}})$-$\mathbf{PRG}_{\mathsf{H}}$ *function and* 2PC *a* $(q, \epsilon_{\mathsf{2pc}})$-$\mathbf{2PC\text{-}IND}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}$ *two-party channels scheme for PCS bound* $\Delta > 0$*. Then Sender Keys (Figure 4) is*

$$(q, 2 \cdot \epsilon_{\mathsf{2pc}} + q^2 \cdot (\epsilon_{\mathsf{2pc}} + \epsilon_{\mathsf{sym}} + q \cdot \epsilon_{\mathsf{prg}}) + q \cdot \epsilon_{\mathsf{sig}})\text{-}\mathbf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}}$$

with respect to cleanness predicate $\mathsf{C} = \mathsf{C}_{\mathsf{sk}}^{\Delta}$ *(Figure 9), where two-party channels cleanness predicate* $\mathsf{C}_{\mathsf{2pc}}$ *is defined in Figures 21 and 22.*

We define C_{sk} in Section 5.1 and prove the theorem in Appendix C. A proof sketch is provided in Section 5.2. Our security notion is adaptive as users can adaptively call oracles and compromise users. Security is tighter when we restrict the game to consider non-adaptive adversaries as described below.

Corollary 1. *Under the same conditions of Theorem 1, and considering a non-adaptive security game, Sender Keys (Figures 23 to 25) is* $(q, 2 \cdot \epsilon_{\mathsf{2pc}} + q \cdot (\epsilon_{\mathsf{sym}} + q \cdot \epsilon_{\mathsf{prg}}) + q \cdot \epsilon_{\mathsf{sig}})$-$\mathbf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}}$ *with respect to cleanness predicate* $\mathsf{C} = \mathsf{C}_{\mathsf{sk}}$ *(Figure 9).*

Sender Keys and Two-Party Channels. To illustrate how the cleanness predicates for Sender Keys must depend on the underlying two-party channels, consider a strongly secure two-party channel 2PC that provides optimal FS and PCS. Now, consider an execution of Sender Keys where all parties share the same view of the group $G = \{ID_1, ID_2, ID_3\}$, in which

1. ID_1 generates a control message $(T_{\mathsf{core}}, \vec{C})$ to remove party ID_3 $(q_1 = \mathtt{Remove}(ID_1, ID_3))$,
2. ID_1 and ID_2 process T $(q_{2,1} = \mathtt{Deliver}(ID_1, (T_{\mathsf{core}}, \vec{C}[ID_1])), q_{2,2} = \mathtt{Deliver}(ID_2, (T_{\mathsf{core}}, \vec{C}[ID_2])))$,
3. $\mathcal{A}$ exposes ID_2 $(q_3 = \mathtt{Expose}(ID_2))$;
4. ID_1 sends an application message $(q_4 = \mathtt{Send}(ID_1, m))$.

Recall that in step 4, ID_1 samples a new sender key that it sends to ID_2 over 2PC, since processing remove messages results in the sender keys of all parties being refreshed. Even with optimally-secure 2PC, the adversary will be able to decrypt the key sent over 2PC (by the correctness of the channel) and thus decrypt the ciphertext output in query q_4.

5.1 Cleanness

Our goal is to describe a suitable cleanness predicate C_{sk} for Sender Keys. The intuition behind this cleanness predicate is based on the following observations about the protocol:

- The exposure of a group member compromises the security of subsequent chain and message keys[6] until a secure key refresh takes place. This enables the adversary to forge messages since they also gain access to the exposed signature keys.
- Control messages can be trivially forged and injected by a network adversary as they are not authenticated.
- Forward-secure confidentiality holds modulo out-of-order message delivery.

[6] Although it is not captured in our model, note that the exposure of a message key alone only compromises the message it refers to and does not (computationally) leak information about the chain key or other message keys.

– All parties recover from state exposure (via $\texttt{Expose}(ID)$) after security on the two-party channels is restored (considering the PCS bound Δ) and then either a) a removal is made effective, or b) all parties update their keys successfully. If messages arrive out-of-order, recovery is not guaranteed.

To formalize the security predicate we introduce conventions for tracking the channel epochs of each user's two party channels. We assume the game $\textbf{M-IND}_{\mathsf{GM},\mathsf{C}}$ maintains the largest channel epoch-index for each user's two-party channels over time. The game obtains this information by observing the channel epoch-index pairs generated by the 2PC.Send and 2PC.Recv operations within the group messenger. Specifically, we use a variable of the form $\texttt{EI}[ID, ID']$, where $\texttt{EI}[ID, ID']$ represents the largest channel epoch-index pair from ID's perspective for the channel between them and user ID', as in Figure 20.

The $\mathsf{refresh}_{\Delta}$ Predicate. We define the predicate $\mathsf{refresh}_{\Delta}(ID, ID', q_i, e)$, parameterised by the PCS bound $\Delta > 0$ of the underlying two-party channels. Informally, given that ID' is exposed in query q_i ($q_i = \texttt{Expose}(ID')$), $\mathsf{refresh}_{\Delta}(ID, ID', q_i, e)$ is true if the (ID, ID') channel has healed and *then* ID has sampled a fresh sender key in or by epoch e (or will do so upon their next Send call). If the predicate is true, the key material of ID has recovered from the exposure in q_i.

More formally, let $(e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) = \max\{\texttt{EI}[ID', ID; q_i], \texttt{EI}[ID, ID'; q_i]\}$. Then $\mathsf{refresh}_{\Delta}(ID, ID', q_i, e)$ is true if a) for $(e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}) = \texttt{EI}[ID', ID; q_j]$ for some $j > i$, $e'_{\mathsf{2pc}} \geq e_{\mathsf{2pc}} + \Delta$ holds; and b) during query q_k with $k \geq j$, member ID processes one of the following control messages corresponding to epoch e:

1. a removal of some member ID^*,
2. an addition of ID itself,
3. a group creation message,
4. or an update from ID itself.

In particular, if ID executes (and processes) an update that involves sending new key material over a refreshed two-party channel, this key material should be safe. We also define a simpler predicate $\mathsf{refresh\text{-}s}(ID, e)$ which is true if member ID processes one of the aforementioned control messages corresponding to epoch e. Observe that both $\mathsf{refresh}_{\Delta}$ and refresh-s events may only happen when ID moves to a new group epoch e.

Cleanness for Sender Keys. We divide our cleanness predicate into three components (challenge, injection, concurrency) that we specify below. The final predicate is defined in Figure 9.

Challenge (Figure 5). The effect of this predicate is to prevent challenges on exposed users (i.e., due to Expose calls). After exposing (with query q_i) *any* user ID', adversarial queries to Challenge are disallowed for every ID in the group until $\mathsf{refresh}_{\Delta}(ID, ID', q_i, e)$ occurs for some later epoch $e > \mathcal{E}[ID'; q_i]$. Note that this only restricts challenge queries q_j where $i < j$. To capture forward security precisely, some challenges made before an exposure ($i > j$) are also forbidden. These affect messages sent by some ID in epochs $(e, i) \geq \mathsf{m\text{-}ep}(ID', ID, q_i)$, which correspond to keys that ID' still stores (including skipped message keys stored at exposure time) or can derive due to being in a previous message epoch (for example if the user is offline).

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}chall:}}}\ \forall(i,j,ID,ID') : q_i = \texttt{Expose}(ID') \wedge q_j = \texttt{Challenge}(ID,\cdot,\cdot),$$
$$\big(i > j \wedge \mathsf{m\text{-}ep}(ID', ID, q_i) > (\mathcal{E}[ID;q_j], \mathcal{I}[ID;q_j]) \wedge$$
$$(\mathcal{E}[ID;q_j], \mathcal{I}[ID;q_j]) \notin \mathcal{SM}[ID', ID; q_i]\big)$$
$$\vee \big(i < j \wedge \exists e : \mathcal{E}[ID';q_i] < e \leq \mathcal{E}[ID;q_j] \wedge \mathsf{refresh}_{\Delta}(ID, ID', q_i, e)\big)$$

Figure 5. Challenge cleanness predicate for Sender Keys where the adversary makes oracle queries $q_1, \ldots, q_q$.

Injection (Figure 7). Firstly, let us recall the two-party ciphertext splitting semantics defined in Section 3.2. Namely, a GM ciphertext C naturally splits into $C = (C_{\mathsf{core}}, C_{\mathsf{2pc}})$ where C_{2pc} is processed by the two-party channels. An injection is said to have occurred when a message with a forged C_{core} and/or C_{2pc} was successfully processed.

We define the injection predicate to prevent injections of *application messages* coming from a user that has been exposed and has not refreshed its keys. We start with the definition for C_{core}. After exposing *a specific* user ID' with query q_i, $\mathcal{A}$ cannot make a query $q_j = \texttt{Receive}(ID, C)$ to impersonate ID' with a forgery ciphertext C corresponding to some epoch e^* (i.e., such that (ID', e^*) are output by $\mathsf{Recv}(C, \gamma[ID])$ in the game) in the following situations:

1. $e^* \geq \mathcal{E}[ID'; q_i]$ and there hasn't been a $\mathsf{refresh}_{\Delta}(ID', ID, q_i, e')$ event for the sender ID' at some epoch e' such that $\mathcal{E}[ID'; q_i] < e' \leq e^*$, where the receiver ID has also processed the key update from ID''s message at injection time, i.e., $\mathcal{E}[ID; q_j] \geq e'$.
2. $e^* < \mathcal{E}[ID'; q_i]$ but the signature key of ID' at epoch e^* was the same key as in the exposure epoch $\mathcal{E}[ID'; q_i]$. Formally, this is expressed by the condition that there has not been any event $\mathsf{refresh\text{-}s}(ID', e')$ for an epoch $e^* < e' \leq \mathcal{E}[ID'; q_i]$.

For C_{2pc}, we directly adopt the injection cleanness predicate $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$ used to define two-party channel security (Figure 22). For additional clarity, we parametrize the predicates by the ciphertexts $C_{\mathsf{core}}, C_{\mathsf{2pc}}$. We define the auxiliary predicate $\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}})$ in Figure 6.

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}}) :}\ \forall(i,j,ID,ID') :$$
$$(C_{\mathsf{core}}, \cdot) \notin \mathcal{M}[ID', \cdot; q_j] \wedge (i < j) \wedge q_i = \texttt{Expose}(ID') \wedge$$
$$q_j = \texttt{Receive}(ID, (C_{\mathsf{core}}, \cdot)) \wedge (\cdot, e^*, \cdot, ID') \leftarrow \mathsf{Recv}((C_{\mathsf{core}}, \cdot), \gamma[ID]) \text{ in } q_j,$$
$$\exists e' : \big[\ (\mathcal{E}[ID'; q_i] < e' \leq e^* \wedge \mathsf{refresh}_{\Delta}(ID', ID, q_i, e'))$$
$$\vee\ (e^* < e' \leq \mathcal{E}[ID'; q_i] \wedge \mathsf{refresh\text{-}s}(ID', e'))\ \big]$$

Figure 6. Auxiliary core injection cleanness predicate for Sender Keys where the adversary makes oracle queries $q_1, \ldots, q_q$.

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj:}}}\ \forall\ C_{\mathsf{core}}, C_{\mathsf{2pc}}, \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}}) \wedge \mathsf{C}_{\mathsf{2pc\text{-}inj}}(C_{\mathsf{2pc}})$$

Figure 7. Injection cleanness predicate for Sender Keys which uses $\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}$ (Figure 6) and $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$ (Figure 22), where the adversary makes oracle queries $q_1, \ldots, q_q$.

Concurrency (Figure 8). This predicate ensures several properties in the protocol. Firstly, it enforces that users process control message in the same order, although they may process them at different rates. Additionally, it prevents the injection of *all* control messages. It is important to note that control messages are not signed in the protocol, making injections trivial. Furthermore, the predicate guarantees that every user proposing a group change (via the Exec, `Add`, `Remove` or `Update` oracles) is in the most recent epoch. In practice, this predicate ensures that there is a unique honest control message in each epoch of the game.

The concurrency predicate ensures both security and correctness by addressing scenarios where members propose concurrent group changes or process group changes in different orders. Without enforcing this predicate, the protocol's behavior becomes ill-defined.

$$\boxed{\mathsf{C}_{\text{sk-con}}:}\ \forall(i, ID) : q_i = \texttt{Deliver}(ID, (T_{\text{core}}, C_{2\text{pc}})), \exists j < i :$$
$$q_j = (\texttt{Add} \text{ or } \texttt{Remove} \text{ or } \texttt{Update} \text{ or } \texttt{Create})(ID, \cdot) \wedge \exists e' :$$
$$(T, \vec{C}) = \mathcal{T}[e'] \wedge (C_{2\text{pc}} \in \vec{C}) \wedge (\mathcal{E}[ID; q_i] = e' - 1 = \mathcal{E}[ID; q_j])$$

Figure 8. Concurrency cleanness predicate in the ideal case where the adversary makes oracle queries $q_1, \ldots, q_q$.

$$\boxed{\mathsf{C}^{\Delta}_{\text{sk}}:}\ \mathsf{C}^{\Delta}_{\text{sk-chall}} \wedge \mathsf{C}^{\Delta}_{\text{sk-inj}} \wedge \mathsf{C}_{\text{sk-con}}$$

Figure 9. Sender Keys cleanness predicate which makes use of sub-predicates defined in Figures 5, 7 and 8.

Limitations and Extensions. Our cleanness predicate enforces a total ordering on control messages, in contrast to considering causal ordering such as in [WKHB21] or no ordering at all. This assumption is consistent with real-world protocols (as in WhatsApp) where a central server is trusted to provide such an ordering, but makes our model unsuitable for decentralized protocols. If our security model allowed for it, one could modify our cleanness predicates to allow for 'trivial' injections that are non-winning, by not giving the adversary the challenger's bit b given that the forgery is trivial (i.e., it violates the injection predicate). Our concurrency predicate and security model could be strengthened to allow several Exec calls in an epoch, from which the network chooses one that is processed to all parties, which has been modelled for TreeKEM in the past [ACDT20].

5.2 Proof Sketch for Theorem 1

Towards proving the theorem (full proof in Appendix C), we construct a series of hybrids. We first transition to a game where injections on the two-party channels are disallowed, following from their underlying security. After that, we transition to a game where oracle `Receive` never outputs challenge bit b, reducing the transition to SUF-CMA signature security, while still excluding trivial injections due to cleanness. Then, we move to a game where the adversary is limited to a single `Challenge` query, losing a factor of q in the resulting reduction. Subsequently, we transition to a game where the message key used in the `Challenge` query (if it exists) is replaced by a uniformly random key that remains unknown to the adversary due to cleanness, and the two-party ciphertexts

that send the key's ancestor chain key are replaced by dummy ciphertexts, which follows from the 2PC security and the PRG security of (H_1, H_2). Finally, we directly reduce to the IND-CPA security of the symmetric encryption scheme.

6 Analysis and Improvements

For the proof of security of Sender Keys (Theorem 1) to go through, we need to impose severe restrictions on the adversarial behaviour through the cleanness predicate C_{sk}. Hence, even if we manage to prove Sender Keys secure, we do so under a weak model that reveals important security shortcomings of the protocol. In this section, we elaborate on these limitations and propose changes to enhance security while maintaining efficiency. We note that our preliminary findings were presented in an earlier version of our work [BCG22].

6.1 Security Analysis and Limitations

Injection of Control Messages. Our first observation is that control messages lack user authentication, necessitating a high level of trust in the server to prevent the crafting of its own messages. To address this, in predicate $\mathsf{C}_{\mathsf{sk\text{-}con}}$ we need to enforce that every delivered control message has been honestly generated. A server deviating from standard behavior could mount a host of attacks. Here are three examples.

Censorship attack: The server can remove any member(s) ID from $\boldsymbol{G}$ such that all remaining members assume ID left the group by himself, whilst ID believes a different user ID' removed him.

- The server delivers a control message $T := (\texttt{rem}, ID, ID, \cdot) \leftarrow \mathsf{Exec}(\texttt{rem}, ID, \bot)$ to every $ID' \in \boldsymbol{G} \setminus \{ID\}$.
- The server delivers a control message $T' := (\texttt{rem}, ID', ID, \cdot) \leftarrow \mathsf{Exec}(\texttt{rem}, ID, \bot)$ to $ID \in \boldsymbol{G}$.

Burgle into the group attack: This attack, observed in [RMS18], allows the server to add any member(s) ID to $\boldsymbol{G}$. For this, the server just delivers a control message $T := (\texttt{add}, \cdot, ID, \cdot) \leftarrow \mathsf{Exec}(\texttt{add}, ID, \bot)$ to every $ID' \in \boldsymbol{G}$.

Unsafe group administration: In general, administration cannot be enforced or trusted due to the lack of authentication of control messages, similarly to what has been observed for CGKA-based protocols in [BCV23].

Weak Post-Compromise Security. Sender Keys offers a very limited form of PCS. Essentially, a $\mathsf{refresh}_{\Delta}$ event is the only possibility for ID to recover from a state compromise. This event only occurs whenever another user is removed or whenever ID triggers an on-demand update (or trivially when ID is new to the group). On-demand updates are supported by our primitive syntax and protocol description, but it is not clear whether they are implemented in practice (for instance, there is no mention to them in [Wha20]).

Moreover, the update mechanism is not satisfactory. Since only the updater ID refreshes its sender key, this allows a passive adversary to eavesdrop on messages sent by any other group member due to the adversary's knowledge of the chain keys corresponding to those members. Extending the update mechanism to the entire group in a naive manner would result in a total communication complexity of $\mathcal{O}(n^2)$.

PCS and two-party channels. PCS guarantees are even weaker due to the reliance of Sender Keys on two-party channels. As parametrized by $\mathsf{refresh}_{\Delta}$, if ID sends new key material over a two-party channel with ID' that has not been healed (after Δ round-trip messages) since the last exposure of either ID or ID', then such key material is still compromised. In practice, if the state of ID is compromised, both the group and the two-party sessions will be exposed. Therefore, unless parties refresh their individual two-party channels consciously (by sending each other messages), executing updates or removals in the group session will not have the desired healing effect.

In the real world, usually not all pairs of members of a group exchange private messages regularly, hence not refreshing their two-party channels. The fact that even manually triggering a key update does not necessarily heal the group from a state compromise conveys an important security limitation.

Lack of Forward Security on Authentication. Beyond PCS limitations, we observe that the forward security guarantees for authentication provided by Sender Keys are sub-optimal. Consider a simple group $\boldsymbol{G} = \{ID_1, ID_2\}$ and the attack described in Figure 10. Note that q_3 is a forbidden query by $\mathsf{C}_{\mathsf{sk\text{-}inj}}$. q_3 attempts to inject a message that corresponds to key material used *before* the state exposure, hence one can envision stronger FS where queries like q_3 are allowed. This attack can occur naturally if ID_2 is offline when m is first sent.

01 $q_1 = \mathtt{Send}(ID_1, \mathsf{m})$ generates ciphertext C encrypted under mk and signed under ssk_1.
02 $q_2 = \mathtt{Expose}(ID_1)$, where $\mathcal{A}$ obtains ssk_1, but not mk.
03 $\mathcal{A}$ modifies C and signs it again under ssk_1 to create a forgery C' corresponding to the same message epoch as C.
04 $q_3 = \mathtt{Receive}(ID_2, ID_1, C')$, which is a successful injection.

Figure 10. Attack on authentication forward security in Sender Keys.

An attack of a similar nature can also occur in a messaging scheme where the same signature keys are re-used across groups, and are refreshed at different times, as pointed out in [CHK21].

Additional Remarks. In a Sender Keys group, each user is associated with a distinct symmetric key, resulting in a state that contains $\mathcal{O}(n)$ secret material at all times. However, the exposure of a single member compromises the keys of all group members, rendering the use of multiple keys ineffective for enhancing security. The primary advantage of employing multiple keys is for concurrency reasons. Nevertheless, in large groups, this approach can pose solubility challenges. As a result, it is possible to explore trade-offs between the level of concurrency supported and the amount of secret material that needs to be stored at any given time. At the opposite end of the spectrum, all users could maintain a single symmetric chain that is the *same* for everyone. This approach, suitable for scenarios where concurrent message transmission is unlikely, reduces the state size to $\mathcal{O}(1)$ and requires $\mathcal{O}(n)$ PCS updates. Moreover, this would improve security by reducing PCS updates to the sending and processing of a single constant-sized message, which represents an optimal solution.

Sender Keys, as described in Section 4, is susceptible to randomness exposure and randomness manipulation attacks. Namely, the adversary does not need to leak a member's state, but simply control the randomness used by the device, inhibiting any form of PCS. Protection against this

family of attacks can be attained at small cost if freshly generated keys are hashed with the state as in [JS18] and the classic NAXOS trick used in authenticated key exchange [LLM07].

Other attacks are possible, but unavoidable unless symmetric encryption for application messages is replaced by some form of public-key encryption. Suppose that the adversary $\mathcal{A}$ exposes user ID ($\gamma \leftarrow \texttt{Expose}(ID)$) and then calls $C \leftarrow \texttt{Challenge}(ID, m_0, m_1)$. $\mathcal{A}$ can trivially win since it can derive mk used in the $\texttt{Challenge}$ query trivially from γ. Some ratcheting protocols provide strong security in that, if a user is impersonated towards, their state should 'diverge' and no longer be useful for decrypting messages from honest parties [PR18, JS18, BRV20]. This is not possible to achieve in a Sender Keys-like protocol (nor is achieved by the Double Ratchet or MLS for messages) since the key schedule for message encryption is deterministic and independent of previously received messages.

6.2 Proposed Improvements: Sender Keys+

We propose several improvements to Sender Keys below. Our improvements are constrained by the desire to retain the performance characteristics and structure of Sender Keys. In particular, we retain $\mathcal{O}(1)$-sized ciphertexts, do not increase key sizes, and utilize only standard cryptographic primitives. Our improved version of Sender Keys, which we call Sender Keys+, is presented fully in Figures 23 to 25. We formalise security by introducing several modifications to our cleanness predicate that we describe at the end of this section.

Secure Control Messages. A simple way of resolving the attacks in Section 6.1 would be for users to sign their own control messages and verify signatures before processing control messages. Additional protocol logic for correctness is required, namely that users who craft a control message but have not shared their sender key yet (because they have not spoken in the group) generate a signature key pair and share their public key over the two-party channels.

By introducing this tweak, we can weaken the cleanness predicate such that it no longer enforces honest control message delivery ($\mathsf{C}_{\mathsf{sk}^+\text{-}\mathsf{con}}$). On the other hand, we need to introduce the restriction that no secret signature key ssk can be known to the adversary at delivery time, similar as in the injection predicate. We do so by introducing a new control predicate $\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-}\mathsf{ctr}}$ that follows the blueprints of the injection predicate (Figure 7), and that we show in Figure 11.

Improved Forward-Secure Authentication. We propose two possible improvements that address the attack in Section 6.1 to varying extents.

MACing from the chain key. The first improvement, which has minimal overhead, is to MAC the application messages with a MAC key $\tau\mathsf{k}$ that we derive via an additional $\mathsf{H}_3(\mathsf{ck})$. The modification is done in the Send algorithm as follows: given an unsigned ciphertext $\tilde{C} = (c, (e, i), ME)$, we obtain the MAC tag $\tau \leftarrow \mathsf{MAC.Tag}(\tau\mathsf{k}, \tilde{C})$. Then, we sign the ciphertext with the appended tag $\sigma \leftarrow \mathsf{Sig.Sgn}(\mathsf{ssk}, (\tilde{C}, \tau))$. The verification of the MAC tag is easily carried out at the receiver's end. We include this simple tweak in the protocol in Figures 23 to 25. Naturally, symmetric encryption can alternatively be replaced with an AEAD to achieve the same effect.

The main security improvement that results from this upgrade is that, in the attack in Section 6.1, the adversary additionally needs knowledge of $\tau\mathsf{k}$ to forge the MAC tag. Hence, one of the following situations must occur before delivery:

– The sender ID is exposed before the message is sent. Then, both $\tau\mathsf{k}$ and ssk are compromised.
– The sender ID is exposed after the message is sent (leaking ssk), and another group member ID' is exposed before the message is delivered (leaking $\tau\mathsf{k}$).

In particular, the attack of Figure 10 no longer results in a successful message delivery. The MAC key can be stored together with the message key for out-of-order messages, such that the MAC can always be verified in a correct execution of the protocol. We note that insider attacks (forgeries from other group members) cannot be prevented by MACing, but we do not model these.

The modified injection predicate that results from this improvement is shown in Figure 13. Essentially, we define an auxiliary predicate $\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-}\mathsf{inj}\text{-}\mathsf{extra}}$ that considers the security given by the message/MAC keys (similarly as in Figure 5). Then, the modified $\mathsf{C}_{\mathsf{sk}^+\text{-}\mathsf{inj}}$ is the logical disjunction of the former injection predicate with $\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-}\mathsf{inj}\text{-}\mathsf{extra}}$, and hence strictly weaker.

Ratcheting signature keys. An alternative mitigation strategy for the attack of Figure 10 is to ratchet signature keys. Let $(\mathsf{ssk}, \mathsf{spk})$ be ID's signature key pair, where spk is part of its sender key. Before sending a new message m to the group, ID can generate a new key pair $(\mathsf{ssk}', \mathsf{spk}') \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$. Then, ID can attach the new spk' to the ciphertext corresponding to encrypting m, and sign the package using ssk. This (by now standard) countermeasure not only provides strong forward security but also post-compromise security for the authentication of messages. Nevertheless, it involves larger overhead, so it may not be desirable in all scenarios and we refrain from including it in Sender Keys+.

Efficient PCS Updates. We propose an asynchronous update mechanism to refresh all chain keys at once, recovering PCS on-demand for the whole group with a single update (and $\mathcal{O}(n)$ complexity for a group of n users). Recall that our Group Messenger primitive supports updates via $\mathsf{Exec}(\mathsf{upd}, \{ID\})$.

A naive solution. Let ID be the updating party. ID generates a new sender key for himself as in the case of a remove operation; namely samples a fresh ck and a fresh $(\mathsf{ssk}, \mathsf{spk}) \xleftarrow{\$} \mathsf{Sig.Gen}(1^\lambda)$. Additionally, ID samples randomness $r \xleftarrow{\$} \{0,1\}^\lambda$. Then, it distributes $(\mathsf{ck}, \mathsf{spk}, r)$ over the two-party channels. Upon reception, every group member (including ID itself) sets $\mathsf{SK}[ID] \leftarrow (\mathsf{ck}, \mathsf{spk})$; and then for every $ID' \in \boldsymbol{G}$, set $\mathsf{SK}[ID'].\mathsf{ck} \leftarrow \mathsf{H}_r(\mathsf{SK}[ID'].\mathsf{ck}, r)$, where $\mathsf{H}_r : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda$ is a secure key derivation function. Since r is freshly sampled and distributed securely, all chain keys recover from exposure. Note that r must be used and erased immediately, as all updated chain keys are exposed if r is leaked at any future time.

Our solution. The previous solution fails in out-of-sync scenarios such as the following. Suppose that ID' is in message epoch $(1,1)$ when ID sends an update message T. Then, ID' speaks in the group before receiving T (for example, while being offline), ratcheting its key to $(1,2)$. All group members will update the chain key $\mathsf{ck}^{1,1}_{ID'}$ (i.e. corresponding to the message epoch $(e,i) = (1,1)$) in $\mathsf{SK}[ID']$, but ID' will be in message epoch $(1,2)$ (and therefore will have erased $\mathsf{ck}^{1,1}_{ID'}$). In general, if there are application messages in transit concurrently with the update, users will be out-of-sync.

To support asynchronicity, we propose a modification in which all parties ratchet their chain key N times forward, where N is a fixed constant that we call the *concurrency bound* (for example $N = 100$; in practice the cost of executing 100 hash function calls sequentially is negligible). In the event that ℓ messages have been sent out-of-sync, then the chain key is ratcheted $N - \ell$ times

instead. Then, parties update the ratcheted chain keys with the sent randomness r. To synchronise between them and with the update initiator ID, the latter sends a list with his view of the key indices of each group member (in the control message). We describe the update protocol as part of Figures 23 to 25. Note that this mechanism requires the assumption of total ordering of control messages to avoid overlapping updates.

The security improvement is reflected in the challenge cleanness predicate in Figure 12. The predicate is as the challenge predicate for Sender Keys (Figure 5), except that now it also suffices that *some* arbitrary member ID^* that has a healed channel with ID' updates after the exposure, and that ID processes such update before the challenge.

Efficient Remove Operations. The previous update mechanism can be extended to improve the efficiency of group removals from $\mathcal{O}(n^2)$ (everyone needs to generate and distribute a new key) to $\mathcal{O}(n)$ in terms of communication complexity. Note a removal can be made effective if the party that sends the remove message T distributes update material among all group members except for the removed party ID'. If ID' leaves, the next member that speaks in the group must also trigger an update. This tweak, like our solution above, has the drawback that the signature keys are not refreshed. Thus, we do not include this tweak in Sender Keys+. Furthermore, considering the minimal overhead of updates, they could potentially become the preferred method for sharing sender keys in the group under all circumstances. This approach allows the group to achieve PCS almost for free.

Cleanness Predicates for Sender Keys+ The cleanness predicates corresponding to our improvements and described informally above are detailed in Figures 11 to 13, and the joint predicate $\mathsf{C}^{\Delta}_{\mathsf{sk}^+}$ in Figure 14.

$$\boxed{\mathsf{C}_{\mathsf{sk}^+\text{-con}}:}\ \forall (i, ID) : q_i = \texttt{Deliver}(ID, T), \exists j < i :$$
$$q_j = (\texttt{Add} \text{ or } \texttt{Remove} \text{ or } \texttt{Update} \text{ or } \texttt{Create})(ID, \cdot) \wedge$$
$$\exists e' : \mathcal{E}[ID; q_i] = e' = \mathcal{E}[ID; q_j]$$

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-ctr-core}}(T_{\mathsf{core}}):}\ \forall (i, j, ID, ID') :$$
$$(T_{\mathsf{core}}, \cdot) \notin \mathcal{T}[\cdot] \wedge (i < j) \wedge q_i = \texttt{Expose}(ID') \ \wedge q_j = \texttt{Deliver}(ID, (T_{\mathsf{core}}, \cdot)),$$
$$\exists e' : \big[\ (\mathcal{E}[ID'; q_i] < e' \leq e^* \wedge \mathsf{refresh}_{\Delta}(ID', ID, q_i, e'))$$
$$\vee\ (e^* < e' \leq \mathcal{E}[ID'; q_i] \wedge \mathsf{refresh\text{-}s}(ID', e'))\ \big]$$

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-ctr:}}}\ \forall\ T_{\mathsf{core}}, C_{\mathsf{2pc}} : \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}ctr\text{-}core}}(T_{\mathsf{core}}) \wedge \mathsf{C}_{\mathsf{2pc\text{-}inj}}(C_{\mathsf{2pc}})$$

Figure 11. Modified concurrency predicate, additional auxiliary core control predicate, and additional control predicate for Sender Keys+.

$$\boxed{\mathsf{C}^{\Delta}_{\mathsf{sk}^+\text{-chall}}:}\ \mathsf{C}^{\Delta}_{\mathsf{sk\text{-}chall}} \vee \big[i < j \wedge \exists e, e', k, ID^* : \mathcal{E}[ID'; q_i] < e' < e \leq \mathcal{E}[ID; q_j] \wedge$$
$$q_k = \texttt{Update}(ID^*) \wedge \mathsf{ep}[q_k] = e - 1 \wedge \mathsf{refresh}_{\Delta}(ID^*, ID', e', q_k)\big]$$

Figure 12. Modified challenge cleanness predicate for Sender Keys+.

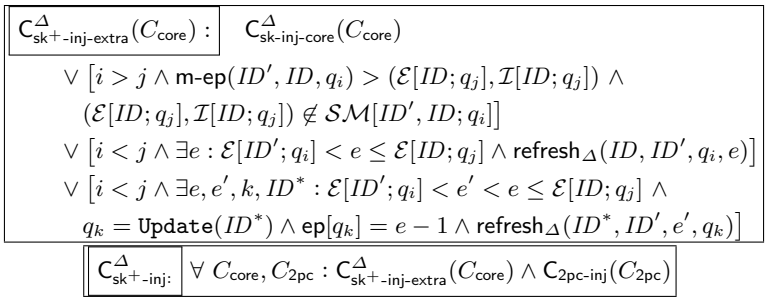

Figure 13. Modified auxiliary injection predicate and injection predicate for Sender Keys+. We remark that the additional logic simply mimics the structure of $\mathsf{C}^{\Delta}_{\mathsf{sk+\text{-}chall}}$.

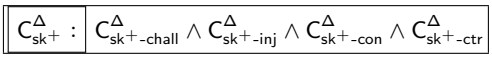

Figure 14. Modified cleanness predicate for Sender Keys+.

6.3 Security of Sender Keys+

In Appendix D, we prove the security of our Sender Keys+ protocol. We do so with respect to the modified cleanness predicate in Figure 14. The proof follows similar steps as the proof for Theorem 1.

Theorem 2. *Let* $\mathsf{SymEnc} := (\mathsf{Enc}, \mathsf{Dec})$ *be a* $(q, \epsilon_{\mathsf{sym}})$-$\mathbf{IND\text{-}CPA}_{\mathsf{SymEnc},b}$ *symmetric encryption scheme,* $\mathsf{Sig} := (\mathsf{Gen}, \mathsf{Sgn}, \mathsf{Ver})$ *a* $(q, \epsilon_{\mathsf{sig}})$-$\mathbf{SUF\text{-}CMA}_{\mathsf{Sig}}$ *signature scheme,* $\mathsf{H} : \{0,1\}^{\lambda} \to \{0,1\}^{\lambda} \times \{0,1\}^{\lambda}$ *a* $(q, \epsilon_{\mathsf{prg}})$-$\mathbf{PRG}_{\mathsf{H}}$ *function,* F *a* $(q, \epsilon_{\mathsf{dprf}})$-$\mathbf{dual\text{-}PRF}_{\mathsf{F}}$ *function,* MAC *a* $(q, \epsilon_{\mathsf{mac}})$-$\mathbf{SUF\text{-}CMA}_{\mathsf{MAC}}$ *message authentication code and* $\mathsf{2PC}$ *a* $(q, \epsilon_{\mathsf{2pc}})$-$\mathbf{2PC\text{-}IND}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}$ *two-party channels scheme for PCS bound* $\Delta > 0$*. Then Sender Keys+ (Figures 23 to 25) is*

$$(q, 2 \cdot \epsilon_{\mathsf{2pc}} + q^3 \cdot (\epsilon_{\mathsf{2pc}} + \epsilon_{\mathsf{sym}} + q \cdot \epsilon_{\mathsf{prg}} + N \cdot q \cdot \epsilon_{\mathsf{dprf}} + q \cdot \epsilon_{\mathsf{mac}}) + q \cdot \epsilon_{\mathsf{sig}})\text{-}\mathbf{M\text{-}IND}_{\mathsf{GM},\mathsf{C}}$$

with respect to predicate $\mathsf{C} = \mathsf{C}^{\Delta}_{\mathsf{sk+}}$ *(Figure 14) and concurrency bound* N*, where two-party channels predicate* $\mathsf{C}_{\mathsf{2pc}}$ *is defined in Figures 21 and 22.*

7 Conclusion and Future Work

In conclusion, our modular approach to modelling Sender Keys has allowed us to identify its main security limitations, some of which we can mitigate while preserving efficiency. We have demonstrated that the protocol does not possess any fundamental security flaws, although it does have notable shortcomings that can be remedied without sacrificing performance. We propose Sender Keys+ as a viable alternative for group messaging when strong PCS is not a critical requirement or regular updates are performed. Interestingly, our modelling of two-party channels has revealed the difficulty of achieving PCS in Sender Keys, even after updates or removals, contradicting folklore assumptions.

In practice, it is common for two-party channels between group members to remain stagnant for extended periods if private communication is not frequent. This degrades the overall group security, underscoring the importance of implementing a regular refresh mechanism by default, especially if PCS updates are implemented. Additionally, Sender Keys is commonly supplemented by additional

mechanisms not considered in our study, such as support for multiple devices and encrypted cloud backups that increase the attack surface.

Looking forward, several research directions emerge. Firstly, our security model can be extended to encompass randomness manipulation, successful message injections, insider threats, and other relevant scenarios. Investigating the practical behaviour of Sender Keys would provide valuable insights for improved modelling and the identification of potential vulnerabilities. Benchmarking both the baseline and extended Sender Keys protocols would also contribute to assessing their practicality. Additionally, it is important to address the challenges that arise when total order is violated, and to design a protocol that avoids the drawbacks associated with decentralised continuous group key agreement (DCGKA) such as the need for multi-round communication [WKHB21]. Towards a more concurrency-friendly Sender Keys protocol, an important direction is the design of a mechanism for resolving ties in control messages that are sent concurrently.

Acknowledgements

We are grateful to David Dervishi for useful insights on Signal's implementation of Sender Keys and for identifying several bugs in an earlier version of the protocol. We are also grateful to anonymous reviewers for their detailed comments and suggestions for improvement.

References

AAN+22a. Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559, 2022. https://eprint.iacr.org/2022/559. (Cited on Page 8.)

AAN+22b. Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 815–844. Springer, Heidelberg, May / June 2022. doi:10.1007/978-3-031-07085-3_28. (Cited on Page 8.)

ACD19. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019. doi:10.1007/978-3-030-17653-2_5. (Cited on Page 6, 8, 10, 36, 38, and 39.)

ACDJ23. Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable Cryptographic Vulnerabilities in Matrix. In *2023 IEEE Symposium on Security and Privacy*, 2023. (Cited on Page 8.)

ACDT20. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020. doi:10.1007/978-3-030-56784-2_9. (Cited on Page 3, 5, 7, 10, 11, 14, 15, 16, 22, and 42.)

ACDT21. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1463–1483. ACM Press, November 2021. doi:10.1145/3460120.3484820. (Cited on Page 3, 6, 8, and 16.)

ACJM20. Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, November 2020. doi:10.1007/978-3-030-64378-2_10. (Cited on Page 3, 4, 5, 6, 7, and 16.)

ADJ24. Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. Device-Oriented Group Messaging: A Formal Cryptographic Analysis of Matrix' Core. In *2024 IEEE Symposium on Security and Privacy (to appear)*, 2024. (Cited on Page 3 and 8.)

AHKM22. Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 69–82. ACM Press, November 2022. doi:10.1145/3548606.3560632. (Cited on Page 7.)

AJM22. Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 34–68. Springer, Heidelberg, August 2022. doi:10.1007/978-3-031-15979-4_2. (Cited on Page 3, 6, 7, and 16.)

AMPS22. Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for telegram. In *2022 IEEE Symposium on Security and Privacy*, pages 87–106. IEEE Computer Society Press, May 2022. doi:10.1109/SP46214.2022.9833666. (Cited on Page 8.)

BBL+22. Olivier Blazy, Ioana Boureanu, Pascal Lafourcade, Cristina Onete, and Léo Robert. How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment. Cryptology ePrint Archive, Report 2022/1090, 2022. https://eprint.iacr.org/2022/1090. (Cited on Page 6 and 39.)

BBR18. Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018. URL: https://hal.inria.fr/hal-02425247. (Cited on Page 3 and 7.)

BBR+23. Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023. URL: https://www.rfc-editor.org/info/rfc9420, doi:10.17487/RFC9420. (Cited on Page 3 and 7.)

BCG22. David Balbás, Daniel Collins, and Phillip Gajland. Analysis and Improvements of the Sender Keys Protocol for Group Messaging. *XVII Reunión española sobre criptología y seguridad de la información. RECSI 2022*, 265:25, 2022. (Cited on Page 8 and 23.)

BCK21. Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the MLS RFC, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021. https://eprint.iacr.org/2021/137. (Cited on Page 8.)

BCV23. David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic Administration for Secure Group Messaging. In *2023 USENIX Security Symposium*, 2023. (Cited on Page 5, 6, 8, and 23.)

BDG+22. Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 213–243. Springer, Heidelberg, November 2022. doi:10.1007/978-3-031-22365-5_8. (Cited on Page 3.)

BDR20. Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, November 2020. doi:10.1007/978-3-030-64378-2_8. (Cited on Page 8.)

BFG+22. Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 784–813. Springer, Heidelberg, August 2022. doi:10.1007/978-3-031-15802-5_27. (Cited on Page 10.)

BRV20. Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 621–650. Springer, Heidelberg, December 2020. doi:10.1007/978-3-030-64840-4_21. (Cited on Page 4, 8, 15, and 25.)

CCD+20. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020. doi:10.1007/s00145-020-09360-1. (Cited on Page 10 and 39.)

CCG16. Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In Michael Hicks and Boris Köpf, editors, *CSF 2016 Computer Security Foundations Symposium*, pages 164–178. IEEE Computer Society Press, 2016. doi:10.1109/CSF.2016.19. (Cited on Page 5.)

CCG+18. Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018. doi:10.1145/3243734.3243747. (Cited on Page 7.)

CEST22. Kelong Cong, Karim Eldefrawy, Nigel P. Smart, and Ben Terner. The key lattice framework for concurrent group messaging. Cryptology ePrint Archive, Report 2022/1531, 2022. https://eprint.iacr.org/2022/1531. (Cited on Page 8.)

CHK21. Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1847–1864. USENIX Association, August 2021. (Cited on Page 8, 15, and 24.)

CJSV22. Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2022. doi:10.1007/978-3-031-15979-4_1. (Cited on Page 10.)

CPZ20. Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1445–1459. ACM Press, November 2020. doi:10.1145/3372297.3417887. (Cited on Page 4.)

CZ22. Cas Cremers and Mang Zhao. Provably post-quantum secure messaging with strong compromise resilience and immediate decryption. Cryptology ePrint Archive, Report 2022/1481, 2022. https://eprint.iacr.org/2022/1481. (Cited on Page 39.)

DFG+23. Gareth T. Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol. In *Advances in Cryptology – CRYPTO 2023 (to appear)*, Cham, 2023. Springer Nature Switzerland. (Cited on Page 8.)

DV19. F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19*, volume 11689 of *LNCS*, pages 343–362. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26834-3_20. (Cited on Page 8 and 39.)

Gal21. Tarek Galal. yowsup, Code Repository, 2021. https://github.com/tgalal/yowsup. (Cited on Page 5.)

HKP+21. Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, November 2021. doi:10.1145/3460120.3484817. (Cited on Page 7.)

IETF23. (IETF) Internet Engineering Task Force. Messaging layer security, mailing list, 2023. https://mailarchive.ietf.org/arch/browse/mls/. (Cited on Page 3.)

Jef20. Kee Jefferys. Session Protocol: Technical implementation details. https://getsession.org/blog/session-protocol-technical-information (accessed July 4th 2023), 2020. (Cited on Page 3 and 8.)

JS18. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96884-1_2. (Cited on Page 8, 25, and 39.)

KGP23. Kien Tuong Truong Kenneth G. Paterson, Matteo Scarlata. Three Lessons From Threema: Analysis of a Secure Messenger. In *2023 USENIX Security Symposium*, 2023. (Cited on Page 8.)

KPPW+21. Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy*, pages 268–284. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00035. (Cited on Page 3, 5, 7, 14, 15, and 16.)

LLM07. Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007*, volume 4784 of *LNCS*, pages 1–16. Springer, Heidelberg, November 2007. (Cited on Page 25.)

M+16. Moxie Marlinspike et al. Signal protocol, 2016. URL: https://github.com/signalapp/libsignal-protocol-java/tree/master/java/src/main/java/org/whispersystems/libsignal. (Cited on Page 3, 4, 5, and 16.)

MP16a. Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm, 2016. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf. (Cited on Page 3, 8, 16, and 36.)

MP16b. Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 283:10, 2016. (Cited on Page 10.)

PP22. Jeroen Pijnenburg and Bertram Poettering. On secure ratcheting with immediate decryption. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part III*, volume 13793 of *LNCS*, pages 89–118. Springer, Heidelberg, December 2022. doi:10.1007/978-3-031-22969-5_4. (Cited on Page 8 and 39.)

PR18. Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96884-1_1. (Cited on Page 8, 25, and 39.)

RMS18. Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 415–429, London, UK, 2018. IEEE. doi:10.1109/EuroSP.2018.00036. (Cited on Page 8 and 23.)

Wha20. WhatsApp. WhatsApp Encryption Overview Technical white paper, v.3, oct 2020. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. (Cited on Page 3, 4, 5, 16, 18, and 23.)

WKHB21. Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2024–2045. ACM Press, November 2021. doi:10.1145/3460120.3484542. (Cited on Page 3, 8, 9, 10, 14, 15, 22, 29, and 38.)

A Deferred Preliminaries

Definition 5 (Symmetric Encryption). *A symmetric encryption scheme* $\mathsf{SymEnc} \coloneqq (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is defined as the following tuple of* PPT *algorithms.*

$\mathsf{k} \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$**:** *Given the security parameter* 1^λ *(encoded in unary) the generation algorithm returns a key* $\mathsf{k} \in \mathcal{K}$.
$c \xleftarrow{\$} \mathsf{Enc}(\mathsf{k}, m)$**:** *Given a key* k *and a message* m*, the encryption algorithm returns a ciphertext* c.
$m \leftarrow \mathsf{Dec}(\mathsf{k}, c)$**:** *Given a key* k *and a ciphertext* c*, the decryption algorithm returns a message* m.

We say that SymEnc *is correct if for any message* $m \in \mathcal{M}$ *and any key* $\mathsf{k} \in \mathcal{K}$ *it holds that,*

$$\Pr\left[\mathsf{Dec}(\mathsf{k}, c) = m \;\middle|\; \begin{matrix} \mathsf{k} \xleftarrow{\$} \mathsf{Gen}(1^\lambda) \\ c \xleftarrow{\$} \mathsf{Enc}(\mathsf{k}, m) \end{matrix}\right] = 1,$$

where the probability is taken over the random coins of Enc*, and* $\mathcal{M}$ *and* $\mathcal{K}$ *denote the message space and key space respectively.*

Definition 6 (IND-CPA security of SymEnc). *Let* SymEnc *be a symmetric encryption scheme.* Chosen plaintext attack *security, or IND-CPA security, for* SymEnc *is defined via the game* $\mathbf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{SymEnc},b}$ *depicted in Figure 15. We define the advantage of adversary* $\mathcal{A}$ *in* $\mathbf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{SymEnc},b}$ *as*

$$\mathsf{Adv}^{\mathbf{ind\text{-}cpa}}_{\mathsf{SymEnc}}(\mathcal{A}) \coloneqq \left|\Pr[\mathbf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{SymEnc},1} \Rightarrow 1] - \Pr[\mathbf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{SymEnc},0} \Rightarrow 1]\right|.$$

We say that SymEnc *is* $(\epsilon, q_{\mathtt{Enc}})$-$\mathbf{IND\text{-}CPA}_{\mathsf{SymEnc}}$ *if for all* PPT *adversaries* $\mathcal{A}$ *who make at most* $q_{\mathtt{Enc}}$ *queries to* Enc*, we have* $\mathsf{Adv}^{\mathbf{ind\text{-}cpa}}_{\mathsf{SymEnc}}(\mathcal{A}) \leq \epsilon$.

Game $\mathbf{IND\text{-}CPA}^{\mathcal{A}}_{\mathsf{SymEnc},b}$

$\mathsf{k} \xleftarrow{\$} \mathsf{SymEnc.Gen}(1^\lambda)$
$m_0, m_1 \leftarrow \bot$
$(m_0, m_1, \mathsf{st}) \leftarrow \mathcal{A}^{\mathtt{Enc}}$
require $|m_0| = |m_1|$
$b \xleftarrow{\$} \{0,1\}$
$c^* \xleftarrow{\$} \mathsf{Enc}(k, m_b)$
$b' \leftarrow \mathcal{A}(c^*, \mathsf{st})$
return $[\![b = b']\!]$

Oracle $\mathtt{Enc}(m)$

require $m \notin \{m_0, m_1\}$
$c \xleftarrow{\$} \mathsf{Enc}(k, m)$
return c

Figure 15. IND-CPA security for SymEnc.

Definition 7 (PRG security of H). *Let* H *be a function* $\mathsf{H} : \mathcal{S} \to \mathcal{W} \times \mathcal{K}$. PRG *security for* H *is defined via a game* $\mathbf{PRG}^{\mathcal{A}}_{\mathsf{H}}$ *depicted in Figure 16. Let* $\mathsf{Adv}^{\mathbf{prg}}_{\mathsf{H}}(\mathcal{A}) \coloneqq \left|\Pr[\mathbf{PRG}^{\mathcal{A}}_{\mathsf{H}} \Rightarrow 1] - \frac{1}{2}\right|$ *be the advantage of adversary* $\mathcal{A}$ *in* $\mathbf{PRG}^{\mathcal{A}}_{\mathsf{H}}$*. We say that* H *is* (ϵ, q)-$\mathbf{PRG}_{\mathsf{H}}$ *if for all* PPT *adversaries* $\mathcal{A}$ *who make at most* q *oracle queries, we have* $\mathsf{Adv}^{\mathbf{prg}}_{\mathsf{H}}(\mathcal{A}) \leq \epsilon$.

Definition 8 (Message Authentication Code). *A message authentication code* $(\mathsf{MAC.Gen}, \mathsf{MAC.Tag}, \mathsf{MAC.Ver})$ *is defined as the following tuple of* PPT *algorithms.*

Game $\mathbf{PRG}_{\mathsf{H}}^{\mathcal{A}}$	**Oracle** $\mathtt{RoR}$
01 $b \xleftarrow{\$} \{0,1\}$	04 **if** $b = 0$:
02 $b' \leftarrow \mathcal{A}^{\mathtt{RoR}}$	05 $(w,k) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda$
03 **return** $[\![b = b']\!]$	06 **if** $b = 1$:
	07 $s \xleftarrow{\$} \{0,1\}^\lambda$
	08 $(w,k) \leftarrow \mathsf{H}(s) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$
	09 **return** (w,k)

Figure 16. Pseudorandom generator security for H.

$\mathsf{k} \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$**:** *Given the security parameter* 1^λ *(encoded in unary) the generation algorithm returns a key* $\mathsf{k} \in \mathcal{K}$.
$\tau \xleftarrow{\$} \mathsf{Tag}(\mathsf{k}, m)$**:** *Given a key* k *and a message* m*, the tag generation algorithm returns a tag* τ.
$m \leftarrow \mathsf{Ver}(\mathsf{k}, m, \tau)$**:** *Given a key* k*, a message* m *and a tag* τ*, the verification algorithm returns a bit* $b \in \{0,1\}$.

We say that MAC *is correct if for any message* $m \in \mathcal{M}$ *and any key* $\mathsf{k} \in \mathcal{K}$ *it holds that,*

$$\Pr\left[\mathsf{Ver}(\mathsf{k}, m, \tau) = 1 \;\middle|\; \begin{matrix} \mathsf{k} \xleftarrow{\$} \mathsf{Gen}(1^\lambda) \\ \tau \xleftarrow{\$} \mathsf{Tag}(\mathsf{k}, m) \end{matrix}\right] = 1,$$

where the probability is taken over the random coins of Tag*, and* $\mathcal{M}$ *and* $\mathcal{K}$ *denote the message space and key space respectively.*

Definition 9 (SUF-CMA security of MAC). Let MAC be a message authentication scheme. *Strong existential unforgeability*, or SUF-CMA security, for MAC is defined via the game $\mathbf{SUF\text{-}CMA}_{\mathsf{MAC}}^{\mathcal{A}}$ depicted in Figure 17. We define the advantage of adversary $\mathcal{A}$ in $\mathbf{SUF\text{-}CMA}_{\mathsf{MAC}}^{\mathcal{A}}$ as

$$\mathsf{Adv}_{\mathsf{MAC}}^{\mathbf{suf\text{-}cma}}(\mathcal{A}) \coloneqq \Pr[\mathbf{SUF\text{-}CMA}_{\mathsf{MAC}}^{\mathcal{A}} \Rightarrow 1].$$

We say that MAC is (ϵ, q)-$\mathbf{SUF\text{-}CMA}_{\mathsf{MAC}}$ if for all PPT adversaries $\mathcal{A}$ who make at most q oracle queries we have $\mathsf{Adv}_{\mathsf{MAC}}^{\mathbf{suf\text{-}cma}}(\mathcal{A}) \leq \epsilon$.

Game $\mathbf{SUF\text{-}CMA}_{\mathsf{MAC}}^{\mathcal{A}}$	**Oracle** $\mathtt{Mac}(m)$
01 $k \xleftarrow{\$} \mathsf{MAC.Gen}(1^\lambda)$	06 $\tau \xleftarrow{\$} \mathsf{MAC.Tag}(k, m)$
02 $Q \leftarrow \emptyset$	07 $Q \leftarrow Q \cup \{m, \tau\}$
03 $(m, t) \leftarrow \mathcal{A}^{\mathtt{Mac}}$	08 **return** τ
04 **require** $(m, t) \notin Q$	
05 **return** $[\![\mathsf{MAC.Ver}(k, m, t) = 1]\!]$	

Figure 17. SUF-CMA security for MAC.

Definition 10 (Dual PRF F**).** *Let* F *be a function* $\mathsf{F} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$*. PRF security for* F *is defined via the game* $\mathbf{PRF}_{\mathsf{F}}^{\mathcal{A}}$ *depicted in Figure 18. Let* $\mathsf{Adv}_{\mathsf{F}}^{\mathbf{prf}}(\mathcal{A}) \coloneqq \left|\Pr[\mathbf{PRF}_{\mathsf{F}}^{\mathcal{A}} \Rightarrow 1] - \frac{1}{2}\right|$ *be the advantage*

of adversary $\mathcal{A}$ *in* $\mathbf{PRF}_{\mathsf{F}}^{\mathcal{A}}$. *We say that* F *is* (ϵ, q)-$\mathbf{PRF}_{\mathsf{F}}$ *if for all* PPT *adversaries* $\mathcal{A}$ *who make at most* q *oracle queries, we have* $\mathsf{Adv}_{\mathsf{F}}^{\mathbf{prf}}(\mathcal{A}) \leq \epsilon$. *We define a function* $\mathsf{F}^{\mathit{Swap}}(k, x) := \mathsf{F}(x, k)$. *The dual PRF security of* F *is the PRF security of* $\mathsf{F}^{\mathit{Swap}}$.

$$\mathsf{Adv}_{\mathsf{F}}^{\mathbf{dual\text{-}prf}}(\mathcal{A}) := \max\{\mathsf{Adv}_{\mathsf{F}}^{\mathbf{prf}}(\mathcal{A}), \mathsf{Adv}_{\mathsf{F}^{\mathit{Swap}}}^{\mathbf{prf}}(\mathcal{A})\}.$$

Game $\mathbf{PRF}_{\mathsf{F}}^{\mathcal{A}}$	**Oracle** $\mathtt{RoR}(x)$
01 $b \xleftarrow{\$} \{0,1\}$	06 **if** $b = 0$:
02 $k \xleftarrow{\$} \mathcal{K}$	07 **return** $\mathsf{F}(k, x)$
03 $f \xleftarrow{\$} \mathsf{Func}[\mathcal{X}, \mathcal{Y}]$	08 **if** $b = 1$:
04 $b' \leftarrow \mathcal{A}^{\mathtt{RoR}}$	09 **return** $f(x)$
05 **return** $[\![b = b']\!]$	

Figure 18. Pseudorandom function security for $\boldsymbol{G}$.

Definition 11 (Digital signature). *A digital signature scheme* $\mathsf{Sig} := (\mathsf{Gen}, \mathsf{Sgn}, \mathsf{Ver})$ *is defined as the following tuple of* PPT *algorithms.*

$(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{Gen}$**:** *creates a public-private key pair.*
$\sigma \xleftarrow{\$} \mathsf{Sgn}(\mathsf{sk}, m)$**:** *generates a signature* σ *from a message* m *and the secret key* sk.
$b \leftarrow \mathsf{Ver}(\mathsf{pk}, \sigma, m)$**:** *outputs* $b \in \{0,1\}$*, indicating acceptance or rejection, given a signature* σ*, a message* m *and a public key* pk.

We say that the signature scheme is correct if for any $m \in \mathcal{P}$ *and all choices of randomness, if* $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{Gen}$ *and* $\sigma \xleftarrow{\$} \mathsf{Sgn}(\mathsf{sk}, m)$*, then* $\mathsf{Ver}(\mathsf{pk}, \sigma, m) = 1$.

Definition 12 (SUF-CMA security of Sig). Let Sig be a signature scheme. *Strong existential unforgeability*, or SUF-CMA security, for Sig is defined via the game $\mathbf{SUF\text{-}CMA}_{\mathsf{Sig}}^{\mathcal{A}}$ depicted in Figure 19. We define the advantage of adversary $\mathcal{A}$ in $\mathbf{SUF\text{-}CMA}_{\mathsf{Sig}}^{\mathcal{A}}$ as

$$\mathsf{Adv}_{\mathsf{Sig}}^{\mathbf{suf\text{-}cma}}(\mathcal{A}) := \Pr[\mathbf{SUF\text{-}CMA}_{\mathsf{Sig}}^{\mathcal{A}} \Rightarrow 1].$$

We say that Sig is (ϵ, q)-$\mathbf{SUF\text{-}CMA}_{\mathsf{Sig}}$ if for all PPT adversaries $\mathcal{A}$ who make at most q oracle queries we have $\mathsf{Adv}_{\mathsf{Sig}}^{\mathbf{suf\text{-}cma}}(\mathcal{A}) \leq \epsilon$.

B Security Model for Two-Party Channels

In this section, we outline the security model for the 2PC primitive introduced in Section 2.1. We provide an in-depth explanation of channel epochs and indices, describe the security game and the cleanness predicates, and discuss the potential extensions and limitations of our model.

Game SUF-CMA$^{\mathcal{A}}_{\mathsf{Sig}}$	**Oracle** $\mathtt{Sign}(m)$
01 $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{Sig.Gen}(1^\lambda)$	06 $\sigma \xleftarrow{\$} \mathsf{Sig.Sgn}(\mathsf{sk}, m)$
02 $Q \leftarrow \emptyset$	07 $Q \leftarrow Q \cup \{(m, \sigma)\}$
03 $(m, \sigma) \leftarrow \mathcal{A}^{\mathtt{Sign}}(\mathsf{pk})$	08 **return** σ
04 **require** $(m, \sigma) \notin Q$	
05 **return** $[\![\mathsf{Sig.Ver}(\mathsf{pk}, \sigma, m) = 1]\!]$	

Figure 19. SUF-CMA security for Sig.

Channel Epochs and Indices. Our notion of channel epochs is exactly the notion of epochs as defined in [ACD19] used to model the Double Ratchet protocol [MP16a]. For a given two-party channel, ID and ID' are each associated with a channel epoch e_{2pc}, which corresponds to how many times the direction of communication has changed, alongside an index i_{2pc}, indicating how many Send calls have been made by a sender or the latest message received by a receiver. Initially, the sender (say ID) sets $e_{\mathsf{2pc}} = 0$ and the receiver ID' sets $e_{\mathsf{2pc}} = -1$. Thereafter, party ID (resp. ID') is the sender in even (resp. odd) channel epochs, and the receiver in odd (resp. even) channel epochs. When a party is a sender in channel epoch e_{2pc} and receives a message from $e_{\mathsf{2pc}} + 1$, they advance to $e_{\mathsf{2pc}} + 1$; likewise they advance channel epochs when they are a receiver and then send a message. When a party sends as a sender, they increment their index i_{2pc}, and as a receiver they set i_{2pc} to the message received with the highest index. Note that i_{2pc} represents the number of messages sent or received for a given channel epoch e_{2pc}; i_{2pc} is set to 0 whenever a party has just become a sender again. Hence, we define a total ordering on channel epochs and indices $(e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$ such that $(e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) \leq (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}})$ when $e_{\mathsf{2pc}} < e'_{\mathsf{2pc}}$, or $e_{\mathsf{2pc}} = e'_{\mathsf{2pc}}$ and $i_{\mathsf{2pc}} < i'_{\mathsf{2pc}}$.

Oracles for Correctness and Security. To capture both correctness and security we employ several oracles that the adversary can query. We describe them briefly before diving into the detail of the correctness and security notions.

$\mathtt{InitCh}(ID, ID')$**:** This oracle initializes the two-party channel between parties ID and ID'.

$\mathtt{Send}(ID, ID', m)$**:** This oracle allows the sending of a message m (i.e., generating a ciphertext) from party ID to party ID'.

$\mathtt{Challenge}(ID, ID', m_0, m_1)$**:** This oracle generates a message challenge where the adversary $\mathcal{A}$ provides two messages m_0 and m_1 of the same length, and party ID sends m_b to party ID'.

$\mathtt{Receive}(ID, C)$**:** This oracle delivers the ciphertext C to party ID.

$\mathtt{Expose}(ID)$**:** This oracle leaks the state of party ID to the adversary.

Correctness. Now we provide an overview of a correctness game between a challenger and a computationally unbounded adversary. Correctness in the $\mathsf{2PC}$ scheme will follow a structure similar to the security game but with specific adaptations. The adversary begins by invoking the $\mathtt{InitCh}$ oracle to establish a secure channel between ID and ID'. Subsequently, the adversary can dynamically query the $\mathtt{Send}$ and $\mathtt{Receive}$ oracles to execute the protocol. Note that the $\mathtt{Expose}$ oracle is permitted, providing the adversary with leaked state information. However, access to the $\mathtt{Challenge}$ oracle is disallowed. It is worth noting that the adversary may only call $\mathtt{Receive}$ with inputs that were output by $\mathtt{Send}$.

Game 2PC-IND$^{\mathcal{A}}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$

01 **for all** ID :
02 $\quad\gamma[ID] \xleftarrow{\$} \mathsf{Init}(1^\lambda, ID)$
03 $\mathcal{CH}[\cdot], \mathcal{S}[\cdot], \mathtt{EI}[\cdot] \leftarrow \bot$
04 $b' \leftarrow \mathcal{A}^{\mathtt{InitCh},\ldots,\mathtt{Expose}}$
05 **require** C
06 **return** b'

Oracle $\mathtt{InitCh}(ID, ID')$

07 $\mathsf{acc} \xleftarrow{\$} \mathsf{InitCh}(ID', \gamma[ID])$
08 **return** acc

Oracle $\mathtt{Send}(ID, ID', m)$

09 $(C, e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) \xleftarrow{\$} \mathsf{Send}(m, ID', \gamma[ID])$
10 **require** $C \neq \bot$
11 $\mathcal{S}[ID, ID'] \xleftarrow{\cup} \{(C, e_{\mathsf{2pc}}, i_{\mathsf{2pc}})\}$
12 $\mathtt{EI}[ID, ID'] \leftarrow (e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$
13 **return** $(C, e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$

Oracle $\mathtt{Challenge}(ID, ID', \mathsf{m}_0, \mathsf{m}_1)$

14 **require** $|\mathsf{m}_0| = |\mathsf{m}_1|$
15 $(C^*, e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) \xleftarrow{\$} \mathsf{Send}(m_b, ID', \gamma[ID])$
16 **require** $C^* \neq \bot$
17 $\mathcal{CH}[ID, ID'] \xleftarrow{\cup} \{(C^*, e_{\mathsf{2pc}}, i_{\mathsf{2pc}})\}$
18 $\mathcal{S}[ID, ID'] \xleftarrow{\cup} \{(C^*, e_{\mathsf{2pc}}, i_{\mathsf{2pc}})\}$
19 $\mathtt{EI}[ID, ID'] \leftarrow (e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$
20 **return** (C^*, e, i)

Oracle $\mathtt{Receive}(ID, C)$

21 $(m, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) \leftarrow \mathsf{Recv}(C, \gamma[ID])$
22 **require** $m \neq \bot$
23 **if** $(C, e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) \notin \mathcal{S}[ID', ID]$:
24 $\quad$**return** $(b, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$ // Forgery
25 $\mathcal{CH}[ID', ID] \xleftarrow{-} \{(C, e_{\mathsf{2pc}}, i_{\mathsf{2pc}})\}$
26 **if** $(e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) > \mathtt{EI}[ID', ID]$:
27 $\quad\mathtt{EI}[ID', ID] \leftarrow (e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$
28 **return** $(\bot, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$

Oracle $\mathtt{Expose}(ID)$

29 **require** $\mathcal{CH}[ID', ID] = \emptyset \;\forall\; ID'$
30 **return** $\gamma[ID]$

Figure 20. 2PC-IND security for 2PC. Lines in teal correspond only to bookkeeping and state update operations. Dictionaries $\mathcal{CH}, \mathcal{S}$ and $\mathtt{EI}$ store challenged messages, sent messages, and channel epoch-index respectively.

The predicates used in the correctness analysis are modified accordingly. The challenge predicate is no longer needed due to the absence of the `Challenge` oracle. Instead, a modified cleanness predicate is employed that consists only of a injection predicate that only allows for honestly generated ciphertexts. The game always returns 0, unless the reception of an honestly generated message fails. The 2PC scheme accommodates out-of-order messages through immediate decryption, allowing for efficient message processing. The attacker is limited to delivering compromised messages pertaining to non-current epochs. If a user is exposed in channel epoch e_{2pc} and acts as the sender, they can decrypt a message from channel epoch $e_{\mathsf{2pc}}+1$. In summary, correctness of the 2PC scheme ensures the faithful delivery of messages between sender and receiver, assuming no interference from an active attacker.

Security Game. The security definition used for two-party channels relies on the **2PC-IND**$^{\mathcal{A}}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ game in Figure 20. The game starts by initializing the states of all parties and initializing the dictionaries $\mathcal{CH}$ and $\mathcal{S}$, which store challenge ciphertexts and all sent (including challenge) ciphertexts, respectively. It also initializes `EI` as a variable that tracks the channel epoch-index pair of a given channel $[ID, ID']$. Then, the adversary $\mathcal{A}$ can adaptively query all the oracles listed above. Finally, $\mathcal{A}$ outputs a guess b' of b given that the game execution has been *clean*, i.e., that the cleanness predicate $\mathsf{C}_{\mathsf{2pc}}$ holds. $\mathcal{A}$ can win the game in two different ways. Firstly, it can make a correct guess of the bit b; note that the only operation that depends on b is the `Challenge` oracle. Ciphertexts C^* generated by $\mathtt{Challenge}(ID, ID', m_0, m_1)$ are stored

in $\mathcal{CH}$, and removed from $\mathcal{CH}$ once they are delivered. We use $\mathcal{CH}$ to prevent trivial forgeries, as if there is any challenge C^* that has not yet been delivered to ID' (and such that, by correctness, ID' must be able to decrypt it), then $\mathcal{A}$ cannot leak the state of ID' via $\texttt{Expose}(ID')$.

Secondly, $\mathcal{A}$ can directly obtain b by making a successful forgery that is accepted via $\texttt{Receive}(ID, C)$. To leak b to $\mathcal{A}$, `Receive` checks that C does not correspond to a message sent by the sender ID' in epoch $(e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$, where $(ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$ are outputs of the internal Recv call.

We note that the challenge-and-send style of our game is analogous to the security game for GM. This game is also multi-user as it captures all channels at once, extending other single-user security models such as those in [ACD19, WKHB21]. Our modelling presents some similarities with the modelling of two-party channels in [WKHB21], but differs from it in several aspects. Our Init algorithm does not require the public key of the counterpart as opposed to theirs, and correctness is captured as part of the security model. Most importantly, the adversary is not allowed to attempt forgeries (the model only captures confidentiality of sent messages) or out of order delivery.

Predicates. The game is parametrised by the two-party channels cleanness predicate $\mathsf{C}_{\mathsf{2pc}}$, which we divide into two sub-predicates as $\mathsf{C}_{\mathsf{2pc}} \coloneqq \mathsf{C}_{\mathsf{2pc\text{-}chal}} \wedge \mathsf{C}_{\mathsf{2pc\text{-}inj}}$. Both sub-predicates are additionally parametrised by the PCS bound Δ. We follow the blueprint of [ACD19] for the predicate definition.

$$
\begin{array}{|l|l}
\hline
\mathsf{C}_{\mathsf{2pc\text{-}chal}} : & \forall (i, ID, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) : q_i = \texttt{Expose}(ID) \wedge \\
& (e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) = \max\{\texttt{EI}[ID, ID'; q_i], \texttt{EI}[ID', ID; q_i]\},\ \nexists\ (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}, j) : (i < j) \wedge \\
& \big[\ (q_j = \texttt{Challenge}(ID', ID, \cdot, \cdot) \wedge (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}) = \texttt{EI}[ID', ID; q_j] \wedge e'_{\mathsf{2pc}} < e_{\mathsf{2pc}} + \Delta) \vee \\
& (q_j = \texttt{Challenge}(ID, ID', \cdot, \cdot) \wedge (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}) = \texttt{EI}[ID, ID'; q_j] \wedge e'_{\mathsf{2pc}} < e_{\mathsf{2pc}} + \Delta)\ \big] \\
\hline
\end{array}
$$

Figure 21. Predicate $\mathsf{C}_{\mathsf{2pc\text{-}chal}}$ where $\mathcal{A}$ makes oracle queries $q_1, \ldots, q_q$.

Challenge (Figure 21). Suppose that $\mathcal{A}$ exposes a party ID and later makes a `Challenge` query involving ID (either as a sender or as a receiver) and some other party ID'. Essentially, the predicate requires that the challenge message then must belong to a channel epoch e'_{2pc} that is Δ or more epochs past the channel epoch e_{2pc}, where e_{2pc} corresponds to the largest epoch value between ID and ID' at exposure time.

$$
\begin{array}{|l|l}
\hline
\mathsf{C}_{\mathsf{2pc\text{-}inj}} : & \forall (i, ID, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) : q_i = \texttt{Expose}(ID) \wedge \\
& (e_{\mathsf{2pc}}, i_{\mathsf{2pc}}) = \max\{\texttt{EI}[ID, ID'; q_i], \texttt{EI}[ID', ID; q_i]\}, \\
& \nexists\ (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}, j, C) : (i < j) \wedge (\min\{e^*_{\mathsf{2pc}}, e'_{\mathsf{2pc}}\} < e_{\mathsf{2pc}} + \Delta) \wedge \\
& (e'_{\mathsf{2pc}}, i'_{\mathsf{2pc}}) = \min\{\texttt{EI}[ID, ID'; q_j], \texttt{EI}[ID', ID; q_j]\} \wedge \\
& \big[\ (q_j = \texttt{Receive}(ID', C) = (\cdot, ID, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \wedge (C, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \notin \mathcal{S}[ID, ID'; q_j]) \vee \\
& (q_j = \texttt{Receive}(ID, C) = (\cdot, ID', e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \wedge (C, e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}}) \notin \mathcal{S}[ID', ID; q_j])\ \big] \\
\hline
\end{array}
$$

Figure 22. Predicate $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$ where $\mathcal{A}$ makes oracle queries $q_1, \ldots, q_q$.

Injection (Figure 22). For an intuitive description, let C be a forged ciphertext that, when processed by Recv by some ID', claims to be from sender ID and on epoch-index $(e^*_{\mathsf{2pc}}, i^*_{\mathsf{2pc}})$. Then, the

predicate requires that if $\mathcal{A}$ exposes ID and later attempts to inject C to some ID', both the epoch e'_{2pc} of the (ID, ID') channel at injection time and e^*_{2pc} are Δ or more epochs further from the channel epoch e_{2pc} at exposure time. Considering both epochs $e_{\mathsf{2pc}}, e^*_{\mathsf{2pc}}$ prevents injections on out-of-order messages. Self-injections (i.e., where $ID = ID'$) are also restricted since the adversary can trivially mount such an attack on the Double Ratchet [ACD19].

In order to model the injection predicate for our GM primitive, we abuse notation and refer directly to $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$. In the context of GM, we parametrise the predicate by a ciphertext C_{2pc}, and simply replace the belonging to $\mathcal{S}$ by the equivalent condition in the GM security game $(\cdot, C_{\mathsf{2pc}}) \notin \mathcal{M}[ID, ID'; q_j]$.

Extensions. Our security model adopts the core requirements of the modelling in [ACD19]. Our cleanness predicates are designed to suit a large class of two-party messaging protocols parametrized by the PCS bound Δ. As a consequence, our analysis is not as fine-grained as possible, but we gain in readability and modularity.

The recent work of Blazy et al. [BBL+22] classifies different two-party messaging protocols based on their resilience to different adversarial and the resulting PCS guarantees. Future work could incorporate these factors into our modelling of two-party channels. Another direction would be to parametrise the PCS bound based on whether a party is exposed while acting as a sender or receiver on the channel (as opposed to considering a worst-case Δ), as done in One could take an even more fine-grained approach in the style of [CCD+20] also. A bound of $\Delta = 1$ would be possible when only considering receiver exposure.

We also note that if one replaces the Double Ratchet with another protocol, either keeping [ACD19, CZ22, PP22] or dropping [JS18, PR18, DV19] support for out-of-order message delivery along the way, it is possible to consider less restrictive injection predicates. For instance, by using signatures, one no longer needs to restrict self-injections. We also restrict injections corresponding to out-of-order delivery (all epochs $< e_{\mathsf{2pc}} + \Delta$ in $\mathsf{C}_{\mathsf{2pc\text{-}inj}}$); notice that restricting winning injections only on ciphertexts in-transit is sufficient as done in [ACD19].

C Theorem 1 Proof: Sender Keys Security

We here prove Theorem 1, i.e., the security of our Sender Keys protocol. Let $\mathcal{A}$ be an adversary against the Sender Keys protocol that plays the GM message indistinguishability game $\mathbf{M\text{-}IND}^{\mathcal{A}}_{\mathsf{GM},b,\mathsf{C}}$ (Figure 2) with respect to cleanness predicate $\mathsf{C} = \mathsf{C}^{\Delta}_{\mathsf{sk}}$ (Figure 9), and let $q_1, \ldots, q_q$ be the oracle queries of $\mathcal{A}$ in a given execution. The proof follows a series of hybrid games, where Game 0 is the original game in Figure 2.

Exposed keys and key sequences. Before diving into the details, we characterise the set of exposed chain keys $\mathsf{ExpKeys}_{\mathsf{ck}}$ as those keys that can be (trivially) derived by the adversary following its state exposure queries. To this end, we observe that all chain keys (and also message keys) generated during protocol execution are uniquely identified by three parameters: the epoch number e, the key index i, and the owner ID; we label them as $\mathsf{ck}^{e,i}_{ID}$ and $\mathsf{mk}^{e,i}_{ID}$. Given a user ID, its chain keys form *sequences* such that the key in $(e, i+1)$ is deterministically derived from the key in (e, i) as $\mathsf{ck}^{e,i+1}_{ID} = H_1(\mathsf{ck}^{e,i}_{ID})$ in the chain. Formally, let $q_i = \texttt{Deliver}(ID, (T_{\mathsf{core}}, C_{\mathsf{2pc}}))$ where T_{core} is either: a remove message for $ID' \neq ID$, a create message, a message that adds ID to the group, or an update message for ID, and $q_j = \texttt{Deliver}(ID, (T'_{\mathsf{core}}, C'_{\mathsf{2pc}}))$ where T'_{core} is either the next remove

message for any $ID' \in \boldsymbol{G}$ for $\boldsymbol{G}$ from the perspective of ID before q_i where $j > i$, or an update message for ID; otherwise, $q_j = q_{q+1}$ (where $\mathcal{E}[ID; q_{q+1}]$ denotes the state of $\mathcal{E}[ID]$ *after* query q_q) if no such query was made. Let also $e = \mathcal{E}[ID; q_i]$ and $\tilde{e} = \mathcal{E}[ID; q_j]$. Then,

$$\mathsf{ck}_{ID}^{e,0}, \ldots, \mathsf{ck}_{ID}^{e,i_e}, \mathsf{ck}_{ID}^{e+1,0}, \ldots, \mathsf{ck}_{ID}^{\tilde{e}-1,i_{\tilde{e}}}$$

is the chain key sequence for ID in epochs e to $\tilde{e}$. Note since $\mathsf{ck}_{ID}^{e,0}$ is the first key generated after a removal or update that $\mathsf{ck}_{ID}^{e,0}$ is generated using fresh randomness and distributed to all parties via two-party channels.

For every $q_j = \mathtt{Expose}(ID_j)$ query such that $(e_j, i_j) \leftarrow \mathsf{m\text{-}ep}(ID_j, ID', q_j)$, all the chain keys of ID' that are exposed are exactly those in the chain key sequence of ID' starting from epoch (e_j, i_j). If we denote this set by $\mathsf{EpCK}_{ID'}^{(j)}$, then we have that

$$\mathsf{ExpKeys}_{\mathsf{ck}} = \bigcup_{j=1}^{k} \left\{ \mathsf{ck}_{ID'}^{e,i} : ID' \in \boldsymbol{G}[ID_j; q_{i_j}] \wedge (e,i) \in \mathsf{EpCK}_{ID'}^{(j)} \right\}.$$

where $\boldsymbol{G}[ID_j; q_{i_j}] = ID_j.\gamma.\boldsymbol{G}$ represents the view of the group of ID_j at the time of query q_{i_j}. Note that, for any group member, if one of its keys in a chain key sequence is in $\mathsf{ExpKeys}_{\mathsf{ck}}$, all the subsequent keys until either a removal or update for ID_j is processed are also in $\mathsf{ExpKeys}_{\mathsf{ck}}$.

Hybrid games. We define the main sequence of games below. We then bound the corresponding advantages by a series of lemmas.

Game 0 This is the original $\mathbf{M\text{-}IND}_{\mathsf{GM},b,\mathsf{C}}^{\mathcal{A}}$ game, parameterised by a cleanness predicate C.

Game 1 In this game, we remove the **return** b conditions whenever $\mathtt{Deliver}(ID, (T_{\mathsf{core}}, C_{\mathsf{2pc}}))$ or $\mathtt{Receive}(ID, (C_{\mathsf{core}}, C_{\mathsf{2pc}}))$ are called such that C_{2pc} was not previously output in some previous oracle query that outputs a ciphertext or control message.

Game 2 In this game, we remove the **return** b condition in the `Receive` and `Deliver` oracles, so that they always returns nothing. Hence, the adversary cannot win the game by injecting.

Game 3 In this game, we allow a single call to the `Challenge` oracle, as opposed to arbitrarily many calls.

Game 4 In this game, all chain keys and corresponding message keys in ID's key sequence that includes ck such that $H_2(\mathsf{ck}) = \mathsf{mk}$, where mk is the message key used in the underlying Send call in the $\mathtt{Challenge}(ID, \cdot, \cdot)$ query (if it exists), are all replaced by uniformly random values. In addition, all $\mathsf{2PC}$ ciphertexts that transmit the chain key ck or keys earlier in the key sequence leading to ck are replaced with encryptions of 0^ℓ where ℓ is the length of the message encrypted.

To complete the proof, Game 4 is simulated by an IND-CPA SymEnc adversary. In the lemmas hereafter, we assume that an adversary $\mathcal{B}$ simulating a hybrid for $\mathcal{A}$ can efficiently determine whether $\mathcal{A}$ has violated the cleanness predicate, and aborts execution since $\mathcal{B}$ can no longer win.

Lemma 1. *There exists an adversary $\mathcal{B}_1$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}_{\mathsf{GM},\mathsf{C}}^{\mathbf{g0}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{GM},\mathsf{C}}^{\mathbf{g1}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}^{\mathbf{2pc\text{-}ind}}(\mathcal{B}_1).$$

Proof. We proceed by constructing $\mathcal{B}_{1,b'}$, a $\mathbf{2PC\text{-}IND}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ adversary that simulates Game 0/Game 1 for adversary $\mathcal{A}$ depending on $\mathbf{2PC\text{-}IND}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ bit b and $\mathbf{M\text{-}IND}_{\mathsf{GM},b',\mathsf{C}}^{\mathcal{A}}$ bit b'. At

a high level, $\mathcal{B}_{1,b'}$ simulates all $\mathbf{M\text{-}IND}^{\mathcal{A}}_{\mathsf{GM},b,\mathsf{C}}$ oracle queries using its own oracles and otherwise simulating locally, except every $\texttt{Challenge}(ID, \mathsf{m}_0, \mathsf{m}_1)$ is simulated as if the challenger's bit is b'. In more detail, $\mathcal{B}_{1,b'}$ replaces all:

- $\mathsf{2PC.InitCh}(ID')$ calls from ID by the output of query $\texttt{InitCh}(ID, ID')$.
- $\mathsf{2PC.Send}(\mathsf{m}, ID')$ calls from ID by the output of $\texttt{Send}(ID, ID', \mathsf{m})$.
- $\mathsf{2PC.Recv}(C)$ calls from ID as follows: $\mathcal{B}_{1,b'}$ first calls $\texttt{Receive}(ID, C)$, which outputs $(b, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$. If $b \neq \perp$, $\mathcal{B}_{1,b'}$ returns b to its challenger and stops simulating. Otherwise, by cleanness and construction of Recv (argued below), C must have been previously output by a $\texttt{Send}(ID, ID', \mathsf{m})$ call. In this case, $\mathcal{B}_{1,b'}$ thus replaces the Recv call with $(m, ID', e_{\mathsf{2pc}}, i_{\mathsf{2pc}})$.

In addition:

- If $\mathcal{A}$ calls $\texttt{Expose}(ID)$, $\mathcal{B}_{1,b'}$ uses the output of its own $\texttt{Expose}(ID)$ call and its state from locally simulating to respond to $\mathcal{A}$'s query.
- Finally, $\mathcal{B}_{1,b'}$ outputs the same bit as $\mathcal{A}$.

Note that by construction of Sender Keys, after $\mathsf{2PC.Recv}$ is called in Recv and Proc calls, the output is checked so it is "appropriate" for the context it is called in (i.e. it is consistent with the received C_{core} or T_{core} (e.g., by checking ID and epoch matches with the input C). In addition, in a given Sender Keys $\mathsf{Recv}/\mathsf{Proc}$ call that does *not* invoke $\mathsf{2PC.Recv}$, $C_{\mathsf{2pc}} = \perp$ is enforced, preventing GM forgeries that include an arbitrary C_{2pc} value that is simply ignored. Thus, if $\mathcal{B}_{1,b'}$ outputs $(b, \dots)$ from $\texttt{Receive}$ such that $b \neq \perp$, it must be that a valid $\mathsf{2PC}$ forgery was made. In addition, by the concurrency cleanness predicate which disallows forgeries on $\texttt{Deliver}$, the **return** b condition in $\texttt{Deliver}$ call is never reached.

Using the triangle inequality, we have

$$\begin{aligned}
\mathsf{Adv}^{\mathbf{g0}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) &= \left|\Pr[G^1_0 \Rightarrow 1] - \Pr[G^0_0 \Rightarrow 1]\right| \\
&\leq \left|\Pr[G^1_0 \Rightarrow 1] - \Pr[G^1_1 \Rightarrow 1]\right| + \left|\Pr[G^0_0 \Rightarrow 1] - \Pr[G^0_1 \Rightarrow 1]\right| \\
&\quad + \left|\Pr[G^1_1 \Rightarrow 1] - \Pr[G^0_1 \Rightarrow 1]\right| \\
&\leq \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_{1,1}) + \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_{1,0}) + \mathsf{Adv}^{\mathbf{g1}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A})
\end{aligned}$$

where the inequality $\left|\Pr[G^{b'}_0 \Rightarrow 1] - \Pr[G^{b'}_1 \Rightarrow 1]\right| \leq \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_{1,b'})$ holds because the simulation is perfect given $\texttt{Receive}$ never outputs $(b, \dots)$ with $b \neq \perp$, and when $\texttt{Receive}$ does output such a $(b, \dots)$, $\mathcal{B}_{1,b'}$'s advantage is at least as large as $\mathcal{A}$'s since $\mathcal{B}_{1,b'}$ always outputs the correct bit. The result follows. □

Lemma 2. *There exists an adversary $\mathcal{B}_2$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g1}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + q \cdot \mathsf{Adv}^{\mathbf{suf\text{-}cma}}_{\mathsf{Sig}}(\mathcal{B}_2).$$

Proof. Let E be the event that $\mathcal{A}$ in a clean execution of Game 1 (i.e., when cleanness predicate C evaluates to true) calls $\texttt{Receive}(ID, (C_{\mathsf{core}}, C_{\mathsf{2pc}}))$ that outputs $m \neq \perp$ for some ID such that $(C_{\mathsf{core}}, \vec{C})$ was not previously output by $\texttt{Send}$, where C_{2pc} is some (correct) $\mathsf{2PC}$ ciphertext by definition of Game 1. Observe first that Game 1 and Game 2 are identical given $\neg E$. Note that at most q signature keys are sampled and sent over the two party channels during the game's execution:

the first Send call in an epoch and the (single) $\mathsf{Exec}(\mathtt{crt}, \boldsymbol{IDs}, \cdot)$ call result in one signature key being sampled in each call.

Then, let E_i be the event that that $\mathcal{A}$'s first call to $\mathtt{Receive}$ with argument $(ID, (C_{\mathsf{core}}, C_{\mathsf{2pc}}))$ satisfying the conditions above is such that internal Recv call outputs ID' corresponding to the i-th signature key sampled by the challenger.

We define SUF-CMA adversary $\mathcal{B}_{ID',e}$ who simulates for Game 1 adversary $\mathcal{A}$ given E_i holds. $\mathcal{B}_i$ simulates as follows. $\mathcal{B}_i$ locally simulates for $\mathcal{A}$ and responds to all of $\mathcal{A}$'s queries except queries involving the i-th sender key sampled: let ID be the key holder. For these queries, ID sets spk in variable SK to the SUF-CMA public key pk. $\mathcal{B}_i$ generates all signatures associated with spk via SUF-CMA oracle $\mathtt{Sign}$.

Finally, consider when $\mathcal{A}$ makes their first query of the form $\mathtt{Receive}(ID, C)$ that returns $m \neq \bot$ that was not previously output by $\mathtt{Send}$. Observe that $C = (C_{\mathsf{core}} = (M, \sigma), C_{\mathsf{2pc}})$ and $M = (c, (e, i), \mathsf{kc}', \mathsf{i_{ck}}', ID)$ where σ is a signature on $(c, (e, i), \mathsf{i_{ck}}, ID)$. As forgeries on C_{2pc} are disallowed, only C_{core} can possibly be the source of the forgery. By construction of Recv, $(c, (e, i), \mathsf{kc}', \mathsf{i_{ck}}', ID)$ must be different from values previously input to Recv for a non-bottom value to be output, and, by definition of event E_i, C_{core} must differ from values previously output by $\mathtt{Send}$. Moreover, by cleanness, $\mathcal{A}$ must not have been able to make a state exposure that enables it to access signature secret key ssk. Thus, the simulation is well-defined and signature σ is a valid forgery, and so $\mathcal{B}_i$ extracts σ from C and returns (m, σ) with $m = M$ as above to its SUF-CMA challenger.

Finally, we have

$$\begin{aligned}
\mathsf{Adv}^{\mathbf{g1}}_{\mathsf{GM,C}}(\mathcal{A}) &= \big|\Pr[G_1^1 \Rightarrow 1 \wedge \neg E] - \Pr[G_1^0 \Rightarrow 1 \wedge \neg E] \\
&\quad + \Pr[G_1^1 \Rightarrow 1 \wedge E] - \Pr[G_1^0 \Rightarrow 1 \wedge E]\big| \\
&\leq \mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM,C}}(\mathcal{A}) + \big|\Pr[G_1^1 \Rightarrow 1 \wedge E] - \Pr[G_1^0 \Rightarrow 1 \wedge E]\big| \\
&\leq \mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM,C}}(\mathcal{A}) + \Pr[E] \\
&\leq \mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM,C}}(\mathcal{A}) + \sum_i \Pr[E_i] \\
&\leq \mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM,C}}(\mathcal{A}) + \sum_i \mathsf{Adv}^{\mathbf{suf\text{-}cma}}_{\mathsf{Sig}}(\mathcal{B}_i) \\
&\leq \mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM,C}}(\mathcal{A}) + q \cdot \mathsf{Adv}^{\mathbf{suf\text{-}cma}}_{\mathsf{Sig}}(\mathcal{B}_2),
\end{aligned}$$

where the last step holds by combining each $\mathcal{B}_i$ into $\mathcal{B}_2$. □

Lemma 3. *There exists an adversary $\mathcal{B}_3$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM,C}}(\mathcal{A}) \leq q_{chal} \cdot \mathsf{Adv}^{\mathbf{g3}}_{\mathsf{GM,C}}(\mathcal{B}_3)$$

where $q_{chal} \leq q$ denotes the number of $\mathtt{Challenge}$ *oracle queries made by $\mathcal{A}$.*

Proof. We adopt the same high-level strategy as the proof of Lemma 6 in [ACDT20]. Let H_0 be exactly Game 2 with $b = 0$. For $i \in [1, q_{chal}]$, let H_i be exactly H_{i-1} except the i-th query $\mathtt{Challenge}(ID, m_0, m_1)$ uses m_1 (i.e., acts as if the challenge bit is $b = 1$). Observe first that we have $\mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM,C}}(\mathcal{A}) = |\Pr[H_0 \Rightarrow 1] - \Pr[H_{q_{chal}} \Rightarrow 1]|$. We will show, for $i \in [1, q_{chal}]$, that there exists adversary $\mathcal{B}_{3,i}$ playing Game 2 such that $|\Pr[H_i \Rightarrow 1] - \Pr[H_{i-1} \Rightarrow 1]| = \mathsf{Adv}^{\mathbf{g3}}_{\mathsf{GM,C}}(\mathcal{B}_{3,i})$. The claimed result then follows by applying the sequence of hybrids and the triangle inequality.

$\mathcal{B}_{3,i}$ simulates as follows. For $\mathcal{A}$'s first $i-1$ $\texttt{Challenge}(ID, m_0, m_1)$ calls, $\mathcal{B}_{3,i}$ calls $\texttt{Send}(ID, m_1)$ and returns the result. For $\mathcal{A}$'s ith $\texttt{Challenge}(ID, m_0, m_1)$ call, $\mathcal{B}_{3,i}$ calls $\texttt{Challenge}(ID, m_0, m_1)$ and returns the result. For $\mathcal{A}$'s subsequent $\texttt{Challenge}(ID, m_0, m_1)$ calls, $\mathcal{B}_{3,i}$ calls $\texttt{Send}(ID, m_0)$ and returns the result. If $\mathcal{A}$ ever makes an `Expose` query that would trivially allow for them to decrypt any challenge ciphertext, or has previously called `Expose` such that the resulting `Challenge` query would be trivially decryptable, $\mathcal{B}_{3,i}$ aborts. Note that this condition can be efficiently determined based on $\mathcal{A}$'s oracle queries. $\mathcal{B}_{3,i}$ processes all other queries using its own oracles.

Note that if $\mathcal{A}$'s (multi-challenge) execution satisfies the cleanness predicate, then so too does $\mathcal{B}_{3,i}$'s. To see this, note that $\mathcal{B}_{3,i}$ and $\mathcal{A}$ make the same queries to all oracles except for `Challenge` and `Send`. In particular, since $\mathcal{B}_{3,i}$ makes the same `Expose` queries as $\mathcal{A}$ (given $\mathcal{B}_{3,i}$ does not abort) and less `Challenge` queries, there are the same or possibly less opportunities for the challenge predicate to fail in $\mathcal{B}_{3,i}$'s execution as compared to $\mathcal{A}$'s. Moreover, the additional `Send` queries that $\mathcal{B}_{3,i}$ makes do not affect any predicates. Thus, if $\mathcal{B}_{3,i}$'s challenge bit b is 0, then $\mathcal{B}_{3,i}$ perfectly simulates H_{i-1}, and similarly $\mathcal{B}_{3,i}$ perfectly simulates H_i given $b = 1$. The result follows. □

Lemma 4. *There exists an adversary $\mathcal{B}_4$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g3}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq q \cdot (\mathsf{Adv}^{\mathbf{g4}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_4) + q \cdot \mathsf{Adv}^{\mathbf{prg}}_{\mathsf{H}}(\mathcal{B}_4))$$

Proof. The proof of this lemma proceeds via hybrid sub-games. Consider the (restricted) chain key sequence in an execution of Game 3 starting from epoch e until key epoch (ID, e', i') corresponding to the output of Send in the $\texttt{Challenge}(ID, \cdot, \cdot)$ call, if it exists. If it does not exist, then Game 3 adversary $\mathcal{A}$ has no advantage as their execution is independent of the challenge bit. Otherwise, we replace all chain keys and their corresponding message keys in this sequence by uniformly random values. Note that, in the protocol in Figure 23, we model H_1 and H_2 as a PRG $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. Namely, $\mathsf{H}(\mathsf{ck}_i) = (\mathsf{ck}_{i+1}, \mathsf{mk}_i)$ outputs an updated chain key and a new message key.

Let E_{ck} be the event where call $\texttt{Challenge}(ID, \cdot, \cdot)$ is made such that the underlying Send call uses message key mk iteratively derived from some ck, where $\mathsf{ck} = \mathsf{ck}^{e,0}_{ID}$, for some e, is the start of ID's corresponding chain key sequence. Observe that at most q such chain key sequences are possible at the start of an execution of Game 3 where $\mathcal{A}$ makes q oracle queries, and each such ck is associated with a corresponding ID (the user who sampled ck). Denote for simplicity this chain key sequence by $\mathsf{ck}_1, \ldots, \mathsf{ck}_m$ (where $\mathsf{ck}_1 := \mathsf{ck}^{e,0}_{ID}$, $\mathsf{ck}_m := \mathsf{ck}^{e',i'}_{ID}$ and $m \leq q$). For each E_{ck}, we can construct a sequence of hybrids H_i for $i = -1, 0, \ldots, m$ as follows. Game H_{-1} is as in the original Game 3. Game H_0 differs from game H_{-1} in that all two-party ciphertexts encrypting ck or ancestors in its key sequence are replaced with encryptions of dummy message 0^ℓ for messages of length ℓ. For $i \geq 1$, game H_{i-1} differs from game H_i in that, in the latter, we replace both mk_{i-1} and ck_i by uniformly random $r_{i-1} \xleftarrow{\$} \mathcal{K}$ and $s_i \xleftarrow{\$} \mathcal{W}$, respectively. Finally, game H_m is Game 4; all non-exposed keys are independent.

We first construct $\mathbf{2PC\text{-}IND}^{\mathcal{A}}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ adversary $\mathcal{B}$ that simulates for adversary $\mathcal{A}$ playing (as we will argue) game H_{-1} or H_0 depending on its challenger's bit. $\mathcal{B}$ simulates similarly to $\mathcal{B}_1$ in the proof of Lemma 1 except when simulating $\mathsf{2PC.Send}(\mathsf{m}, ID')$ calls. Here, instead of replacing all such calls with the output of $\texttt{Send}(ID, ID', \mathsf{m})$, $\mathcal{B}$ replaces calls that encrypt ck or its key sequence ancestors in m with the output of $\texttt{Challenge}(ID, ID', \mathsf{m}, 0^{|\mathsf{m}|})$. $\mathcal{B}$ otherwise simulates identically. Observe that in a clean execution of H_{-1}, chain key ck_1 must not be exposed, and that $\mathcal{B}$, who is parametrised by ck, can deduce exactly which $\mathsf{2PC.Send}$ calls to replace with an `Challenge` call. It follows that $\mathcal{B}$ simulates H_{-1} given the challenge bit is 0 and H_0 given it is 1.

Note in a clean execution of H_0 that the starting key in the chain $\mathsf{ck}_{ID}^{e,0} \notin \mathsf{ExpKeys}_{\mathsf{ck}}$ is generated by ID from fresh randomness. Besides, $\mathsf{ck}_{ID}^{e,0}$ is only sent over two-party channels that contains no information about the key due to the previous hop (noting in a clean execution that the channels must have healed if previously compromised), so it is hidden from $\mathcal{A}$. Now, let $\mathcal{A}$ be an adversary that interpolates between any two games H_{i-1} and H_i. Then, we can create an adversary $\mathcal{B}$ against PRG indistinguishability from $\mathcal{A}$ as follows. Since by induction that in H_{i-1} the seed ck_{i-1} of the PRG is a uniformly random value, $\mathcal{B}$ simply embeds a PRG challenge in mk_{i-1} and ck_i (recall that we consider a PRG with an expansion factor of 2 such that $\mathsf{H}(\mathsf{ck}_{i-1}) = (\mathsf{ck}_i, \mathsf{mk}_{i-1})$. Then, $\mathcal{B}$ simulates the rest of the game locally and returns the guess of $\mathcal{A}$.

It follows that the simulation is perfect. By considering all events E_{ck} and the union bound, it follows that there exists $\mathcal{B}_4$ so that:

$$\mathsf{Adv}^{\mathbf{g3}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq q \cdot (\mathsf{Adv}^{\mathbf{g4}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_4) + q \cdot \mathsf{Adv}^{\mathbf{prg}}_{\mathsf{H}}(\mathcal{B}_4))$$

□

Lemma 5. *There exists an adversary $\mathcal{B}$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g4}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{ind\text{-}cpa}}_{\mathsf{SymEnc}}(\mathcal{B})$$

Proof. We reduce to the security of the encryption scheme. Let $\mathcal{A}$ be an adversary against Game 4. Then, we can build an adversary $\mathcal{B}$ against the IND-CPA security of the encryption scheme. Let b^* be the (hidden) bit that parameterises the IND-CPA game of $\mathcal{B}$. Then, $\mathcal{B}$ simulates Game 4 for $\mathcal{A}$ except for the challenge query $q^* = \texttt{Challenge}(ID^*, m_0, m_1)$, where it proceeds as follows. $\mathcal{B}$ receives m_0, m_1 from $\mathcal{A}$ and forwards them to the IND-CPA challenger, who outputs a ciphertext c^*. Then, $\mathcal{B}$ crafts a ciphertext C^* as if it originated from ID^* and sends it to $\mathcal{A}$. The simulation continues until the game finishes, and $\mathcal{B}$ returns the same guess b' as $\mathcal{A}$.

As the message key mk used to encrypt the challenge message in the original Game 4 is a uniformly random key, as we argued above, the simulation is perfect. Hence, the lemma follows. □

Finally, Theorem 1 follows by combining the sequence of hybrids above.

Proof strategy for Corollary 1. The hybrids are defined similarly except that they differ in the definition of Game 1 and Game 4. Let the resulting sequence of games be denoted Game 1', ..., Game 4'. In Game 1', all two-party channel ciphertexts that cannot be trivially decrypted by the adversary are replaced with encryptions of dummy strings of the form 0^ℓ. Game 4' differs as in Game 4 except that 2PC ciphertexts are not changed. We can then essentially directly use the above lemmas except for Lemmas 1 and 4:

- For Game 1', note that since $\mathbf{2PC\text{-}IND}_{\mathsf{2PC},b,\mathsf{C}_{\mathsf{2pc}},\Delta}$ adversary $\mathcal{B}_{1,b'}$ is given all of $\mathcal{A}$'s queries in advance, it can efficiently deduce which 2PC.Send queries to replace with `Challenge` and `Send` depending on which ciphertexts can be trivially exposed by $\mathcal{A}$ or not. It then follows that
$$\mathsf{Adv}^{\mathbf{g0}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{g1'}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_1)$$
- For Game 4', the reduction no longer needs to guess ck, since this information can be efficiently derived from the sequence of queries $q_1, \ldots, q_q$ initially given to $\mathcal{A}$. A sequence of hybrids H_i for $i \geq 0$ can then be directly constructed; note we can ignore the hop between H_{-1} and H_0 since Game 1 already handles this. It then follows that
$$\mathsf{Adv}^{\mathbf{g3'}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{g4'}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + q \cdot \mathsf{Adv}^{\mathbf{prg}}_{\mathsf{H}}(\mathcal{B}_4)$$

The result follows.

D Theorem 2 Proof: Sender Keys+ Security

In this section, we prove Theorem 2.

From our Sender Keys+ protocol, we model $\mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3$ as a PRG $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$, and the KDF F used for the update operation as a dual PRF. Note the proof would still work with respect to the Sender Keys predicate $\mathsf{C}^{\Delta}_{\mathsf{sk}}$ (Figure 9) as the new predicate $\mathsf{C}^{\Delta}_{\mathsf{sk+}}$ (Figure 14) is strictly less restrictive. Towards the proof, we re-define the notion of key sequences introduced in Appendix C. We now consider chain key sequences for ID starting with $\mathsf{ck}^{e,0}_{ID}$ where zero or more update operations from $ID' \neq ID$ are applied to a chain key of the form $\mathsf{ck}^{e',i'}_{ID}$. In addition, updates from parties $ID \neq ID'$ now result in a new key sequence.

Hybrid games. We define the main sequence of games below.

Game 0 This is the original $\textbf{M-IND}^{\mathcal{A}}_{\mathsf{GM},b,\mathsf{C}}$ game, parameterised by a cleanness predicate C.

Game 1 In this game, we remove the **return** b conditions whenever $\texttt{Deliver}(ID, (T_{\mathsf{core}}, C_{\mathsf{2pc}}))$ or $\texttt{Receive}(ID, (C_{\mathsf{core}}, C_{\mathsf{2pc}}))$ are called such that C_{2pc} was not previously output in some previous oracle query that outputs a ciphertext or control message.

Game 2 In this game, we completely remove the **return** b condition in the `Deliver` oracle and in the `Receive` oracle for forgeries that occur when predicate $\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}})$ is true for ciphertexts of the form C_{core}.

Game 3 In this game, we allow a single call to the `Challenge` oracle, as opposed to arbitrarily many calls.

Game 4 In this game, all chain keys and corresponding message and MAC keys in ID's key sequence that includes ck such that $H_2(\mathsf{ck}) = \mathsf{mk}$ and $H_3(\mathsf{ck}) = \tau\mathsf{k}$, where mk (resp. $\tau\mathsf{k}$) is the message key (resp. MAC key) used in the underlying Send call in the $\texttt{Challenge}(ID, \cdot, \cdot)$ query (if it exists), are all replaced by uniformly random values. In addition, all $\mathsf{2PC}$ ciphertexts that transmit the chain key ck or keys earlier in the key sequence leading to ck are replaced with encryptions of 0^ℓ where ℓ is the length of the message encrypted.

Game 5 In this game, we completely remove the **return** b condition in the `Receive` oracles. Hence, the adversary cannot win the game by injecting.

To complete the proof, Game 5 is simulated by an IND-CPA SymEnc adversary.

Lemma 6. *There exists an adversary $\mathcal{B}_1$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g0}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{g1}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_1).$$

Proof. The proof is essentially identical to that of Lemma 1 so we omit it. □

Lemma 7. *There exists an adversary $\mathcal{B}_2$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g1}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + q \cdot \mathsf{Adv}^{\mathbf{suf\text{-}cma}}_{\mathsf{Sig}}(\mathcal{B}_2).$$

Proof. The proof follows the same high-level idea as for Lemma 2. That is, we consider events E_i for $i \in [1, q']$ for some $q \leq q$ such that the first successful forgery is made using the i-th signature

key pair sampled. For this proof, we consider forgeries now over both `Receive` and `Deliver` rather than just `Receive`. By construction of Sender Keys+, `Deliver` forgeries given $\mathsf{C}^{\Delta}_{\mathsf{sk\text{-}inj\text{-}core}}(C_{\mathsf{core}})$ is true only occur as a result of a signature forgery. Thus, by a very similar reduction to Lemma 2, the result follows. Note that unlike in the proof for Section 5, we have not yet completely disallowed injections on `Receive`: we still allow forgeries that are permitted by $\mathsf{C}^{\Delta}_{\mathsf{sk}+}$ but disallowed by $\mathsf{C}^{\Delta}_{\mathsf{sk}}$. □

Lemma 8. *There exists an adversary $\mathcal{B}_3$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g2}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq q_{chal} \cdot \mathsf{Adv}^{\mathbf{g3}}_{\mathsf{GM},\mathsf{C}}(\mathcal{B}_3)$$

where $q_{chal} \leq q$ denotes the number of `Challenge` *oracle queries made by $\mathcal{A}$.*

Proof. The proof is identical to that of Lemma 3 so we omit it. □

Lemma 9. *There exists an adversary $\mathcal{B}_4$ with similar running time to $\mathcal{A}$ such that*

$$\begin{aligned}\mathsf{Adv}^{\mathbf{g3}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq q^2 \cdot \big(&\mathsf{Adv}^{\mathbf{g4}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + \mathsf{Adv}^{\mathbf{2pc\text{-}ind}}_{\mathsf{2PC},\mathsf{C}_{\mathsf{2pc}},\Delta}(\mathcal{B}_4) + \\ &q \cdot \mathsf{Adv}^{\mathbf{prg}}_{\mathsf{H}}(\mathcal{B}_4) + N \cdot q \cdot \mathsf{Adv}^{\mathbf{dual\text{-}prf}}_{\mathsf{F}}(\mathcal{B}_4)\big)\end{aligned}$$

where N is the concurrency bound (c.f. Section 6.2).

Proof. The proof diverges from the proof of Lemma 4 in order to handle the new update mechanism. As in Lemma 4, we consider the event E_{ck} where the $\texttt{Challenge}(ID, \cdot, \cdot)$ call invokes Send with key mk in the key sequence starting from ck. Lemma 4 then constructs hybrids $H_{-1}, H_0, \ldots, H_m$ given E_{ck}.

Observe that, fixing ck, mk could have been derived from zero or more update operations initiated by $ID' \neq ID$. Now, by cleanness, after invoking the security of the two-party channels, mk is hidden from the adversary. If no update operations were made, then it must be that ck is not exposed. If there was one update operation, then by cleanness, either the update secret r or ck are hidden (or possibly both).

Let i be the i-th last update operation applied to form mk, where $i \in [0, q']$ for some $q' < q$. Let $E_{\mathsf{ck},i}$ be the event that the i-th last update operation is hidden from the adversary for $i \geq 1$, and $i = 0$ where ck itself is secure. Observe that $E_{\mathsf{ck}} = \cup_{i \in [0,q']} E_{i,q}$ for some $q' < q$.

Our proof strategy then is as follows. For each $E_{\mathsf{ck},i}$, we first hop by invoking the security of the two-party channels to replace all two-party ciphertexts that communicate 'safe' value r or ck (or a descendent in the key sequence) with encryptions of dummy messages (hopping between H_{-1} and H_0). The simulation proceeds analogously to that in Lemma 4. Then (hopping to H_m):

- For $i = 0$, we hop by iteratively replacing all relevant H_j calls using the PRG assumption; there are at most $N \cdot q$ such queries. We replace each F call (of which there are at most q) along the way with a uniform value by the dual PRF assumption, keying F with ck'.
- For $i \geq 1$, we first hop by replacing the first call to F made by the challenger with r with a uniform value using the dual PRF assumption and keying F with r. We then hop using the PRG assumption on H_j and keying PRF F thereafter with ck'; note there are at most $N \cdot i \leq N \cdot q$ such calls.

By a similar argument to Lemma 4, the hop with the two-party channels and the hops with the PRG and dual PRF are sound. The result follows by combining the sequence of hybrids.

Lemma 10. *There exists an adversary $\mathcal{B}_5$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g4}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{g5}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) + q \cdot \mathsf{Adv}^{\mathbf{suf\text{-}cma}}_{\mathsf{MAC}}(\mathcal{B}_5).$$

Proof. Let E be the event that a successful forgery to `Receive` is made in a clean execution of Game 4. Note Game 4 and 5 are identical given $\neg E$. Let E_i be the event that $\mathcal{A}$'s first forgery with `Receive` is with respect to the i-th key sequence starting from ck; note $E = \cup_{i\in[1,q']} E_i$ for some $q' \leq q$. Observe by the previous game hop that the corresponding MAC key $\tau\mathsf{k}$ is uniform. We construct a $\mathbf{SUF\text{-}CMA}_{\mathsf{MAC}}$ adversary $\mathcal{B}$ that simulates for Game 4 adversary $\mathcal{A}$ given E_i holds. $\mathcal{B}$ locally simulates all calls except MAC query using $\tau\mathsf{k}$ on message m during the simulation is replaced with the output of query `Mac`. Finally, when $\mathcal{A}$ makes a successful forgery using $\tau\mathsf{k}$, $\mathcal{B}$ extracts the tag t from the message and returns it to its challenger. The simulation is perfect and so the result follows by a similar derivation as in Lemma 2. □

Lemma 11. *There exists an adversary $\mathcal{B}$ with similar running time to $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathbf{g5}}_{\mathsf{GM},\mathsf{C}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathbf{ind\text{-}cpa}}_{\mathsf{SymEnc}}(\mathcal{B})$$

Proof. The proof is essentially identical to that of Lemma 5 so we omit it. □

E Sender Keys and Sender Keys+ Protocol Description

We introduce the full Sender Keys and Sender Keys+ protocols in Figures Figures 23 to 25, extending the descriptions in Section 4 and Section 6.

Below, we also make some additional remarks intended to help the reader to parse the pseudocode, which is inherently complex, not least due to the additional variables and logic that is required by our modifications.

Sent and unsent sender keys. When a new user *ID* joins and a member *ME* processes the message via $\mathsf{Proc}(T = (\mathsf{add}, \cdot), \cdot)$, *ID* does not receive the sender key of *ME* until *ID* speaks again. Hence, *ME* needs to keep track of this newly added user; it does so via the $\mathsf{no\text{-}SK}[\cdot]$ dictionary. Namely, $\mathsf{no\text{-}SK}[ID] = \mathbf{true}$ in the view of *ME* if *ME* has not sent his sender key to *ID* yet. This functionality is captured in the SendToMissing algorithm.

Sending control messages without a sender key. A different scenario is that *ME* calls Exec and generates a control message T which, in Sender Keys+, needs to be signed. In the event that *ME* does not have a working sender key yet (e.g., due to a recent removal), then *ME* generates an ephemeral sender key containing only a signature key spk. The key is immediately distributed over the two-party channels via OneTimeSpk. We remark that whenever both SendToMissing and OneTimeSpk are executed in the same algorithm (such as in Exec), only one of them will output a non-blank ciphertext, depending on whether the caller's signature key $\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{spk}$ exists or not.

Index updates. Most of the protocol logic behind our update mechanism is explained in Section 6. Due to the synchronization issues mentioned there, the update initiator sends his view (message epoch) of everyone else's sender key. This information is stored in the $\mathsf{Upd\text{-}Ind}[\cdot]$ dictionary, which is sent as part of the control message.

Additional state variables. The state variables $\mathsf{max\text{-}i_{ck}}[\cdot]$ and $\mathsf{last\text{-}kc}$ were omitted in Figure 4. Essentially, these variables keep track of the maximum index $\mathsf{max\text{-}i_{ck}}[ID, \mathsf{kc}]$, corresponding to the sender key $\mathsf{SK}[ID, \mathsf{kc}]$, for which a message was sent. This is critical to determine what skipped message keys (if any) should be stored in $\mathcal{MK}$ so that chain keys can eventually be deleted for forward security. This synchronisation mechanism occurs in the Recv algorithm via the two-party channels, where $\mathsf{last\text{-}kc}$ specifies the last key counter the bound $\mathsf{max\text{-}i_{ck}}$ refers to.

$\underline{\mathsf{Init}(ID, 1^\lambda)}$

$\gamma.ME \leftarrow ID$
$\gamma.(\mathsf{ssk}, \boldsymbol{G}, \mathsf{ep}, \mathsf{i}_{ME}, \mathsf{last\text{-}kc}) \leftarrow \bot$
$\gamma.\mathsf{SK}[\cdot,\cdot] \leftarrow \bot$ // $(ID, \mathsf{ck}) \rightarrow (\mathsf{spk}_{ID}, \mathsf{ck}_{ID}, \mathsf{i_{ck}})$
$\gamma.\mathcal{MK}[\cdot,\cdot] \leftarrow \bot$ // $(ID, (\mathsf{kc}, \mathsf{i_{ck}})) \rightarrow (\mathsf{mk}, \tau\mathsf{k})$
$\gamma.\mathsf{no\text{-}SK}[\cdot] \leftarrow \mathbf{false}$ // $ID \rightarrow$ bool
$\gamma.\mathsf{max\text{-}i_{ck}}[\cdot,\cdot] \leftarrow \bot$ // $\mathsf{kc} \rightarrow \mathsf{i_{ck}}$
$\gamma.\mathsf{kc}[\cdot] \leftarrow \bot$ // $ID \rightarrow \mathsf{kc}$
$\gamma.\mathsf{rs}[\cdot] \leftarrow \bot$ // $\mathsf{kc} \rightarrow$ random coin
$\gamma.\gamma_{\mathsf{2pc}} \leftarrow \mathsf{2PC.Init}(ID)$
return γ

$\underline{\mathsf{Send}(\mathsf{m}, \gamma)}$

require $ME \in \boldsymbol{G}$
$\vec{C}[\cdot] \leftarrow \bot$
if $\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{i_{ck}} = \bot$:
  // Sample sender key if needed
  $\vec{C} \leftarrow$ PreSendFirst()
if $\mathsf{i}_{ME} = 0$:
  $\vec{C} \leftarrow (\vec{C},$ SendToMissing())
$(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{ck})$
$c \xleftarrow{\$} \mathsf{Enc}(\mathsf{mk}, \mathsf{m})$
UpdateCK$(ME, \mathsf{kc}[ME])$
$M \leftarrow (c, (\mathsf{ep}, \mathsf{i}_{ME}), \mathsf{kc}[ME], \mathsf{i_{ck}}, ME)$
$\tau \leftarrow \mathsf{MAC.Tag}(\tau\mathsf{k}, M)$
$\sigma \xleftarrow{\$} \mathsf{Sig.Sgn}(\mathsf{ssk}, M, \tau)$
return $C := ((M, \tau, \sigma), \vec{C})$

$\underline{\mathsf{Recv}(C = ((M, \tau, \sigma), C_{\mathsf{2pc}}), \gamma)}$

parse M **as** $(c, (e, i), \mathsf{kc}', \mathsf{i_{ck}}', ID)$
require $ID \in \boldsymbol{G}$
if $\mathsf{SK}[ID, \mathsf{kc}'] = \bot$:
  // Receive ID's sender key via 2PC if needed
  $((\mathsf{SK}[ID, \mathsf{kc}'], \mathsf{kc}^*, \mathsf{ep}', \mathsf{max\text{-}i_{ck}}', \mathsf{last\text{-}kc}'), ID^*, \cdot, \cdot) \leftarrow$ $\mathsf{2PC.Recv}(C_{\mathsf{2pc}})$
  require $ID^* = ID \wedge \mathsf{ep}' = e \wedge \mathsf{kc}^* = \mathsf{kc}$
  DeleteOldCK$(ID, \mathsf{max\text{-}i_{ck}}', \mathsf{last\text{-}kc}')$
else require $C_{\mathsf{2pc}} = \bot$
require $e \leq \mathsf{ep}$
require $\mathsf{Sig.Ver}(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{spk}, \sigma, M, \tau)$
$(\mathsf{mk}, \tau\mathsf{k}) \leftarrow$ UpdateKeysRecv()
require $\mathsf{MAC.Ver}(\tau\mathsf{k}, M, \tau)$
$\mathsf{m} \leftarrow \mathsf{Dec}(\mathsf{mk}, c)$
return (m, ID, e, i)

$\underline{\text{UpdateKeysRecv}()}$

$\mathsf{i_{ck}} \leftarrow \mathsf{SK}[ID, \mathsf{kc}']$
// Store skipped keys in $\mathcal{MK}$ given out-of-order delivery
while $\mathsf{i_{ck}} < \mathsf{i_{ck}}'$:
  $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck})$
  $\mathcal{MK}[ID, (\mathsf{kc}', \mathsf{i_{ck}}')] \leftarrow (\mathsf{mk}, \tau\mathsf{k})$
  UpdateCK(ID, kc')
  $\mathsf{i_{ck}}'$++
if $\mathsf{i_{ck}} > \mathsf{i_{ck}}'$:
  require $\mathcal{MK}[ID, (\mathsf{kc}', \mathsf{i_{ck}}')] \neq \bot$
  $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow \mathcal{MK}[ID, (\mathsf{kc}', \mathsf{i_{ck}}')]$
  // Delete stored key for forward security
  $\mathcal{MK}[ID, (\mathsf{kc}', \mathsf{i_{ck}}')] \leftarrow \bot$
else :
  $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ID', \mathsf{kc}'].\mathsf{ck})$
  UpdateCK(ID, kc')
return $(\mathsf{mk}, \tau\mathsf{k})$

Figure 23. Sender Keys and Sender Keys+ protocol description (part 1 of 3). Text in black colour corresponds to standard Sender Keys. Coloured text corresponds to the modifications in Sender Keys+ from Section 6.2: blue text corresponds to securing control messages via signatures, teal text corresponds to MACing for forward security and violet text corresponds to PCS updates.

Figure 24. Sender Keys and Sender Keys+ protocol description (part 2 of 3).

UpdateCK(ID, kc')

131 $\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck} \leftarrow \mathsf{H}_2(\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{ck})$
132 $\mathsf{SK}[ID, \mathsf{kc}'].\mathsf{i}_{\mathsf{ck}} \leftarrow \mathsf{SK}[ID, \mathsf{kc}'].\mathsf{i}_{\mathsf{ck}} + 1$
133 **if** $ID = ME$: $\mathsf{i}_{ME} \leftarrow \mathsf{i}_{ME} + 1$

SendToMissing()

134 // Send my sender key to new parties
135 **if** $\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{spk} = \bot$: **return**
136 $\mathsf{m} \leftarrow (\mathsf{SK}[ME, \mathsf{kc}[ME]], \mathsf{kc}[ME]$, ep, $\mathsf{max\text{-}i_{ck}}$[last-kc], last-kc)
137 $\vec{C}[\cdot] \leftarrow \bot$
138 **for all** $ID \in \boldsymbol{G} \setminus \{ME\}$:
139 **if** no-SK$[ID]$:
140 $(\vec{C}[ID], \cdot, \cdot) \xleftarrow{\$} \mathsf{2PC.Send}(ID, \mathsf{m})$
141 no-SK$[ID] \leftarrow$ **false**
142 **return** $\vec{C}$

PreSendFirst()

143 $(\mathsf{spk}, \mathsf{ssk}) \xleftarrow{\$} \mathsf{Sig.Gen}$
144 $\mathsf{ck} \xleftarrow{\$} \{0,1\}^\lambda$
145 $\mathsf{max\text{-}i_{ck}}$[last-kc] $\leftarrow \mathsf{SK}[ME,$ last-kc$].\mathsf{i}_{\mathsf{ck}}$
146 $\mathsf{SK}[ME, \mathsf{kc}[ME]] \leftarrow (\mathsf{ck}, \mathsf{spk}, 0)$
147 $\mathsf{m} \leftarrow (\mathsf{SK}[ME, \mathsf{kc}[ME]], \mathsf{kc}[ME]$, ep, $\mathsf{max\text{-}i_{ck}}$[last-kc], last-kc)
148 $\vec{C}[\cdot] \leftarrow \bot$
149 **for all** $ID' \in \boldsymbol{G} \setminus \{ME\}$:
150 $(\vec{C}[ID'], \cdot, \cdot) \xleftarrow{\$} \mathsf{2PC.Send}(ID', \mathsf{m})$
151 no-SK$[ID'] \leftarrow$ **false**
152 last-kc $\leftarrow \mathsf{kc}[ME]$
153 **return** $\vec{C}$

OneTimeSpk()

154 $\vec{C}[\cdot] \leftarrow \bot$
155 **if** $\mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{spk} = \bot$:
156 $(\mathsf{spk}, \mathsf{ssk}) \xleftarrow{\$} \mathsf{Sig.Gen}$
157 $\mathsf{SK}[ME, \mathsf{kc}[ME]] \leftarrow (\mathsf{spk}, \bot, \bot)$
158 $\mathsf{m} \leftarrow (\mathsf{SK}[ME, \mathsf{kc}[ME]], \mathsf{kc}[ME]$, ep, $\mathsf{max\text{-}i_{ck}}$[last-kc], last-kc)
159 **for all** $ID \in \boldsymbol{G} \setminus \{ME\}$:
160 $(\vec{C}[ID], \cdot, \cdot) \xleftarrow{\$} \mathsf{2PC.Send}(ID, \mathsf{m})$
161 last-kc $\leftarrow \mathsf{kc}[ME]$
162 **return** $\vec{C}$

DeleteOldCK(ID, $\mathsf{max\text{-}i_{ck}}'$, last-kc$'$)

163 **if** $\mathsf{SK}[ID,$ last-kc$'].\mathsf{ck} = \bot$: **return**
164 **while** $\mathsf{SK}[ID,$ last-kc$'].\mathsf{i}_{\mathsf{ck}} <$ $\mathsf{max\text{-}i_{ck}}'$:
165 $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ID,$ last-kc$'].\mathsf{ck})$
166 $\mathcal{MK}[ID,$ (last-kc$', \mathsf{SK}[ID,$ last-kc$'].\mathsf{i}_{\mathsf{ck}})] \leftarrow (\mathsf{mk}, \tau\mathsf{k})$
167 UpdateCK(ID, last-kc$'$)
168 $\mathsf{SK}[ID,$ last-kc$'].\mathsf{ck} \leftarrow \bot$

Exec(cmd $=$ upd, ME, γ)

169 **require** $ME \in \boldsymbol{G}$
170 Upd-Ind$[\cdot] \leftarrow \bot$
171 $(\mathsf{spk}, \mathsf{ssk}) \xleftarrow{\$} \mathsf{Sig.Gen}$
172 $r \xleftarrow{\$} \{0,1\}^\lambda$
173 Upd-Ind$[ME] \leftarrow \mathsf{SK}[ME, \mathsf{kc}[ME]].\mathsf{i}_{\mathsf{ck}}$
174 $\mathsf{ck} \xleftarrow{\$} \{0,1\}^\lambda$
175 $\mathsf{SK}[ME, \mathsf{kc}[ME]+1] \leftarrow (\mathsf{spk}, \mathsf{ck}, \mathsf{i}_{\mathsf{ck}})$
176 $\mathsf{m} \leftarrow (\mathsf{SK}[ME, \mathsf{kc}[ME]+1], \mathsf{kc}[ME]+1, \mathsf{ep}$, $\mathsf{max\text{-}i_{ck}}$[last-kc], last-kc, r)
177 $\vec{C}[\cdot] \leftarrow \bot$
178 **for all** $ID \in \boldsymbol{G} \setminus \{ME\}$:
179 $(\vec{C}[ID], \cdot, \cdot) \xleftarrow{\$} \mathsf{2PC.Send}(ID, \mathsf{m})$
180 Upd-Ind$[ID] \leftarrow \mathsf{SK}[ID', \mathsf{kc}[ID]].\mathsf{i}_{\mathsf{ck}}$
181 $\mathsf{rs}[\mathsf{kc}[ME]] \leftarrow r$
182 $T \leftarrow (\mathsf{upd}, ME, \mathsf{kc}[ME]+1, \mathsf{ep}+1,$ Upd-Ind)
183 **return** $(T, \sigma := \mathsf{Sig.Sgn}(\mathsf{ssk}, T), \vec{C})$

Proc($(T = (\mathsf{upd}, ID, \mathsf{kc}', \mathsf{ep}',$ Upd-Ind$), \sigma, C_{2\mathsf{pc}}), \gamma$)

184 **require** $\mathsf{Sig.Ver}(\mathsf{SK}[ME].\mathsf{spk}, \sigma, T)$
185 **if** $ID = ME$:
186 **require** $C_{2\mathsf{pc}} = \bot$
187 $r \leftarrow \mathsf{rs}[\mathsf{kc}']$; $\mathsf{rs}[\mathsf{kc}'] \leftarrow \bot$
188 **else** :
189 $((\mathsf{SK}[ID, \mathsf{kc}[ID]+1], \mathsf{kc}^*, \mathsf{ep}^*,$ $\mathsf{max\text{-}i_{ck}}'$, last-kc$', r), ID^*, \cdot, \cdot) \leftarrow \mathsf{2PC.Recv}(C_{2\mathsf{pc}})$
190 **require** $ID^* = ID$
191 **require** $\mathsf{ep}' = \mathsf{ep}^* + 1$
192 **require** $\mathsf{kc}^* = \mathsf{kc}[ID] + 1$
193 **require** $\mathsf{ep}' = \mathsf{ep} + 1$
194 $\mathsf{ep} \leftarrow \mathsf{ep}'$; $\mathsf{i}_{ME} \leftarrow 0$
195 $\mathsf{kc}[ID] \leftarrow \mathsf{kc}[ID] + 1$
196 DeleteOldCK(ID, $\mathsf{max\text{-}i_{ck}}'$, last-kc$'$)
197 // Hash forward N times before hashing with r
198 **for all** $ID' \in \boldsymbol{G} \setminus \{ID\}$:
199 **if** $\mathsf{SK}[ID', \mathsf{kc}[ID']].\mathsf{ck} = \bot$:
200 **continue** (line 198)
201 **while** $\mathsf{SK}[ID', \mathsf{kc}[ID']].\mathsf{i}_{\mathsf{ck}} <$ Upd-Ind$[ID']$:
202 $(\mathsf{mk}, \tau\mathsf{k}) \leftarrow (\mathsf{H}_1, \mathsf{H}_3)(\mathsf{SK}[ID', \mathsf{kc}[ID']].\mathsf{ck})$
203 $\mathcal{MK}[ID', (\mathsf{kc}[ID'], \mathsf{SK}[ID', \mathsf{kc}[ID']].\mathsf{i}_{\mathsf{ck}})] \leftarrow (\mathsf{mk}, \tau\mathsf{k})$
204 UpdateCK($ID', \mathsf{kc}[ID']$)
205 $\ell \leftarrow \mathsf{SK}[ID', \mathsf{kc}[ID']].\mathsf{i}_{\mathsf{ck}} -$ Upd-Ind$[ID']$
206 **require** $\ell < N$
207 $\mathsf{ck}' \leftarrow \mathsf{SK}[ID', \mathsf{kc}[ID']].\mathsf{ck}$
208 **do** $N - \ell$ times : $\mathsf{ck}' \leftarrow \mathsf{H}_2(\mathsf{ck}')$
209 $\mathsf{SK}[ID', \mathsf{kc}[ID']].\mathsf{ck} \leftarrow \mathsf{F}(\mathsf{ck}', r)$
210 **return true**

Figure 25. Sender Keys and Sender Keys+ protocol description (part 3 of 3).