Analysis and Improvements of the Sender Keys Protocol for Group Messaging

David Balbás
IMDEA Software Institute,
Universidad Politécnica de Madrid
Spain
david.balbas@imdea.org

Daniel Collins
EPFL
Switzerland
daniel.collins@epfl.ch

Phillip Gajland
Max Planck Institute for Security & Privacy,
Ruhr-University Bochum
Germany
phillip.gajland@mpi-sp.org

***Abstract*—Messaging between two parties and in the group setting has enjoyed widespread attention both in practice, and, more recently, from the cryptographic community. One of the main challenges in the area is constructing secure (end-to-end encrypted) and efficient messaging protocols for group conversations. The popular messaging applications WhatsApp and Signal utilise a protocol in which, instead of sharing a single group key, members have individual *sender keys*, which are shared with all other group members. The Sender Keys protocol is claimed to offer forward security guarantees. However, despite its broad adoption in practice, it has never been studied formally in the cryptographic literature.**

In this paper we present the first analysis of the Sender Keys protocol along with some prospective improvements. To this end, we introduce a new cryptographic primitive, develop a game-based security model, present a security analysis in the passive and active settings, and propose several improvements to the protocol.

***Index Terms*—Secure Messaging, Group Messaging, Signal, WhatsApp, Sender Keys.**

I. Introduction

Messaging applications such as WhatsApp, Facebook Messenger, Signal and Telegram have enjoyed widespread adoption and form an integral part of communications for billions of people. All of the aforementioned applications rely, to a varying degree, on cryptography to provide diverse forms of authenticity and secrecy.

Among end-to-end encrypted messaging solutions (this excludes Telegram and Facebook Messenger by default, among others), there exist diverse cryptographic solutions. For two-party messaging, Signal's Double Ratchet Protocol [1] is the most popular choice in practice, and many solutions also exist in the cryptographic literature [2], [3], [4], [5], [6]. For group messaging, the naive solution, as used by Signal Messenger for small groups, of adopting Double Ratchet sessions among every pair of group members does not scale well. Thus, recent work such as the Messaging Layer Security (MLS) standardization effort [7] aims to construct secure group messaging protocols where the complexity of group operations (adding and removing members, updating key material) is sublinear in the group size [8], [9], [10], [11], [12].

Nevertheless, the popular messaging applications WhatsApp and Signal (for large groups) use a protocol for group messaging [13], [14] that does not involve sharing a unique group key that evolves over time. This differs from MLS, and from the group key agreement abstraction followed there [9], [10], [11]. This protocol, called Sender Keys, has not been formally studied in the literature despite its widespread adoption.

A. Secure Group Messaging

Two standard security notions prevail in the literature both for two-party and group messaging. The first is *forward security* (FS), which protects the confidentiality of past messages in the event of a key exposure and can be achieved using just symmetric cryptography (for example by iteratively hashing symmetric keys). The second is *post-compromise security* (PCS), which ensures that security can be restored after a key exposure in certain adversarial settings [15], typically when the adversary is passive for some period of time. FS- and PCS-oriented key evolution mechanisms are commonly known as *ratcheting*.

Both properties apply to the confidentiality and authenticity of sent messages and can be captured formally in a security game. There exist different formalisations of security in the literature, but most of them 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 yet limited capabilities) as in [9], as a semi-active adversary which can schedule messages arbitrarily [10], or as an active adversary that can inject messages [11], [16]. In many protocols, including Sender Keys and MLS, the DS relies mainly on some centralized infrastructure (the *central server* hereafter).

Some messaging protocols also require additional infrastructure to deal with user authentication or security. This may include Public Key Infrastructure (PKI), or, in the case of Sender Keys, secure two-party messaging channels established between each pair of users. Achieving security in multiple groups simultaneously is outside the scope of this work, and requires additional precautions detailed in [17].

B. Sender Keys

In a Sender Keys group $\boldsymbol{G}$, every user $\mathsf{ID} \in \boldsymbol{G}$ owns a so-called *sender key* which is shared with all group members. A sender key is a tuple $\mathsf{send\text{-}k} = (\mathsf{spk}, \mathsf{ck})$, where spk is a public signature key (with a private counterpart ssk), and ck is a symmetric *chain key*. Every time a user ID sends a message m to the group, ID encrypts m using a *message key* mk that is deterministically derived from its chain key ck. Upon message reception, group members also derive mk to decrypt

the message. Messages are authenticated by appending the sender's signature.

Forward security is provided by using a fresh message key for every message; every time a message is sent, the chain key is hashed forward using a key derivation function. In other words, chain keys are symmetrically ratcheted.

The protocol also requires that there exist confidential and authenticated two-party communication channels between every pair of users. These are used for sharing sender keys in the event of parties being added or removed from the group.

C. Contributions

The main scientific contributions of our paper are the following:

- We introduce a new cryptographic primitive, Group Messenger (GM), which is suitable for messaging protocols like Sender Keys that are not necessarily based on group key agreement.
- We formally describe Sender Keys, based on a code analysis of Signal's source code [14], and WhatsApp's security white paper [13].
- We present a security model for (single-group) Group Messenger. We do so via a security game that considers an active adversary who can interact with several oracles. The game is parametrised by a cleanness predicate that captures forward-secure group messaging.
- We carry out a security analysis for passive and active adversaries and detail prospective fixes and improvements to the Sender Keys protocol.

We note that this is a preliminary and shortened version of our work.

II. Primitive Syntax

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 $\perp$. 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 introduce a cryptographic primitive that we call *Group Messenger* $\mathsf{GM} := (\mathsf{Init}, \mathsf{Send}, \mathsf{Recv}, \mathsf{Exec}, \mathsf{Proc})$, similar to other group messaging abstractions such as Continuous Group Key Agreement (CGKA) [9]. In contrast to CGKA, our primitive does not model key agreement (as this does not neatly capture the Sender Keys protocol), but rather sending and receiving messages. The syntax is as follows.

- $\gamma \xleftarrow{\$} \mathsf{Init}(1^\lambda, \mathsf{ID})$: Given the security parameter 1^λ and a user identity ID, the probabilistic initialisation algorithm returns an initial state γ.
- $(C, \gamma') \xleftarrow{\$} \mathsf{Send}(m, \gamma)$: Given a message m, and a state γ, the probabilistic sending algorithm returns a ciphertext C and a new state γ'.
- $(m, \gamma') \leftarrow \mathsf{Recv}(C, \gamma)$: Given a ciphertext C, and a state γ, the deterministic receiving algorithm returns a message m and a new state γ'.
- $(C, \gamma') \xleftarrow{\$} \mathsf{Exec}(\mathtt{cmd}, \boldsymbol{IDs}, \gamma)$: Given a command $\mathtt{cmd} \in \{\mathtt{crt}, \mathtt{add}, \mathtt{rem}\}$, a list of user identities $\boldsymbol{IDs}$, and a state γ, the probabilistic execution algorithm returns a ciphertext C and a new state γ'.
- $\gamma' \leftarrow \mathsf{Proc}(C, \gamma)$: Given a ciphertext C, and a state γ, the deterministic processing algorithm returns a new state γ'.

Note that there are separate algorithms for sending / receiving application messages and for executing / processing changes to the group. Furthermore, should two-party protocols be required under the hood to implement the group primitive (this is not the case for CGKAs such as TreeKEM [9]), then these are outside the scope of this definition.

III. Protocol Description

In this section, we introduce a formal description of the Sender Keys protocol in accordance with our Group Messenger syntax.

A. Protocol Setup

1) Central server: The Delivery Service (DS) relies on a central server which provides total ordering of messages and authenticates users initially (i.e. it acts as a PKI). In practice, the DS is also responsible for managing two-party channels.

2) Two-party channels: The protocol assumes that there exist authenticated and secure two-party communication channels (for example using Signal's Double Ratchet protocol [1] as done in WhatsApp [13]) between every pair of protocol users. This assumption can be realized via the Delivery Service and asynchronous, PKI-aided key-exchange mechanisms such as X3DH [18].

3) Primitives: The protocol uses standardised [19], [20] underlying cryptographic primitives:

- Two different Key Derivation Functions (KDF) $H_1, H_2 : \{0,1\}^\lambda \to \{0,1\}^\lambda$. These are used to derive message keys mk and chain keys ck, respectively.
- A symmetric encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$.
- A digital signature scheme $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$.

In Signal's implementation of Sender Keys [14], the KDFs are instantiated as $H_1(m) := \mathsf{HMAC}(\mathtt{0x01}, m)$ and $H_2(m) := \mathsf{HMAC}(\mathtt{0x02}, m)$. We note that in Signal's two-party sessions signatures are not used. Instead, messages are authenticated by computing a MAC based on a function of the message key.

In WhatsApp [13], the HMAC used for the KDF is `HMAC-SHA256`, the symmetric encryption scheme is `AES-256` in CBC mode, and the signature scheme is `ECDSA` with `Curve25519`.

B. State

The state γ of a user ID contains a sender key $\mathsf{send\text{-}k} := (\mathsf{spk}, \mathsf{ck})$, where spk denotes the the signature public key and ck the chain key, as well as a secret signature key ssk, each belonging to ID. The state also maintains a list of current group members $\boldsymbol{G}$. Additionally, for each user $\mathsf{ID} \in \boldsymbol{G}$ the sender key $\mathsf{send\text{-}k}_{\mathsf{ID}}$, a list of skipped message keys $\{\mathsf{mk}^i_{\mathsf{ID}}, \dots\}$, used for out-of-order delivery, and the counter i are stored. If ID leaks its state, we say that it suffered a *state compromise*.

C. Algorithms

We describe the Sender Keys protocol according to our Group Messenger primitive defined in Section II. The description follows Signal's reference implementation [14] regarding sender key ratcheting as well as message encryption and decryption. The details of the Exec and Recv algorithms are inferred from [13], but we cannot assert that our interpretation is entirely faithful to WhatsApp's implementation. A simplified example of a 3-message conversation is shown in Figure 1.

1) State initialization: The Init algorithm initializes the state variables of users; in practice this is done at install time.

2) Group creation: The creation of a group is carried out via the $\mathsf{Exec}(\texttt{crt}, \boldsymbol{G}, \gamma)$ algorithm which takes a list of prospective members $\boldsymbol{G} := \{\mathsf{ID}_1, \ldots, \mathsf{ID}_{|\boldsymbol{G}|}\}$ as input. All parties are assumed to have pre-established two-party communication sessions. The group creator $\gamma.\mathsf{ME}$ generates a chain key $\mathsf{ck} \xleftarrow{\$} \{0,1\}^\lambda$ and a signature key pair $(\gamma.\mathsf{spk}, \gamma.\mathsf{ssk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$. Then, it sends its sender key $\gamma.\mathsf{send\text{-}k} = (\gamma.\mathsf{spk}, \mathsf{ck})$ to each $\mathsf{ID} \in \boldsymbol{G}$ individually via their secure two-party channel.

3) Message sending: To send the i^{th} message m_i, a user $\gamma.\mathsf{ME}$ calls the $\mathsf{Send}(m_i, \gamma)$ algorithm which does the following:

- Derive a new message key mk from the symmetric part of its sender key (i.e. the chain key) as $\mathsf{mk}^i_{\mathsf{ME}} \leftarrow H_1(\mathsf{ck}^i_{\mathsf{ME}})$.
- Encrypt the message m_i as $c_i \xleftarrow{\$} \mathsf{Enc}(\mathsf{mk}^i_{\mathsf{ME}}, m_i)$.
- Ratchet the chain key ck[1] as $\mathsf{ck}^{i+1}_{\mathsf{ME}} \leftarrow H_2(\mathsf{ck}^i_{\mathsf{ME}})$.
- Jointly sign the ciphertext c_i, the message index i and the sender's identity ME as $\sigma_i \xleftarrow{\$} \mathsf{Sig}(\mathsf{ssk}, (c_i, i, \mathsf{ME}))$
- Send $C := (c_i, i, \mathsf{ME}, \sigma_i)$ to all group members via the DS.

If $\gamma.\mathsf{send\text{-}k} = \perp$, then $\gamma.\mathsf{ME}$ must first generate a fresh sender key and distribute it as in the group creation. This occurs every time a member sends their first message after entering the group or after a member removal. We emphasise that ciphertexts are not sent over the preexisting two-party channels that are used to communicate sender keys, but rather over the network (i.e., via the Delivery Service) itself.

4) Message reception: Upon reception of a message $C = (c_i, i, \mathsf{ID}, \sigma_i)$, the receiver calls $\mathsf{Recv}(C, \gamma)$. First, Recv verifies ID and the signature as $\mathsf{Ver}(\mathsf{send\text{-}k}[\mathsf{ID}].\mathsf{spk}, \sigma_i, (c_i, i, \mathsf{ID}))$ (note that if $\mathsf{send\text{-}k}[\mathsf{ID}] = \perp$, the receiver must wait until a new sender key is sent by ID). If the check passes, then the algorithm proceeds as follows:

- If $\mathsf{send\text{-}k}[\mathsf{ID}].\mathsf{ck}$ is at iteration i:
  derive $\mathsf{mk}_i \leftarrow H_1(\mathsf{send\text{-}k}[\mathsf{ID}.\mathsf{ck}])$,
  decrypt c_i as $m_i \leftarrow \mathsf{Dec}(\mathsf{mk}_i, c_i)$ and erase mk_i,
  and refresh $\mathsf{send\text{-}k}[\mathsf{ID}.\mathsf{ck}] \leftarrow H_2(\mathsf{send\text{-}k}[\mathsf{ID}.\mathsf{ck}])$.
- If $\mathsf{send\text{-}k}[\mathsf{ID}].\mathsf{ck}$ is at iteration $j < i$, refresh the chain key $i - j$ times as $\mathsf{send\text{-}k}[\mathsf{ID}.\mathsf{ck}] \leftarrow H_2(\mathsf{send\text{-}k}[\mathsf{ID}.\mathsf{ck}])$ while storing message keys $\mathsf{mk}_j, \ldots \mathsf{mk}_{i-1}$ (up to N_{max} keys). Then, obtain mk_i, decrypt c_i, and erase mk_i.
- If $\mathsf{send\text{-}k}[\mathsf{ID}].\mathsf{ck}$ is at iteration $j > i$, search for a stored mk_j, attempt to decrypt c_i, and erase mk_j. If unsuccessful, output $\perp$.

[1]Note that this practice is safer (better FS) than first evolving the chain key and then deriving a message key, since it allows for the immediate deletion of the chain key used to derive the message key.

5) Membership changes: To add a new user ID to the group, a member calls $\mathsf{Exec}(\texttt{add}, \{\mathsf{ID}\}, \gamma)$. This produces a notification message T sent to all group members including ID. Separately, $\gamma.\mathsf{ME}$ sends every member's sender key to ID using their two-party channel. As mentioned earlier, ID only generates and sends its own sender key when they send their first message.

To remove a user ID from the group, a member calls $\mathsf{Exec}(\texttt{rem}, \{\mathsf{ID}\}, \gamma)$ and sends the notification T to all members.

6) Message processing: Group changes are processed via $\mathsf{Proc}(T, \gamma)$. If a user ID is added, $\gamma.\mathsf{ME}$ simply updates the list of group members $\gamma.G$. If ID is removed, $\gamma.\mathsf{ME}$ deletes all sender keys, including his own. In this scenario, the group "starts over", namely all members must generate and send a new sender key (i.e. a new signature key pair and symmetric key) to the members.

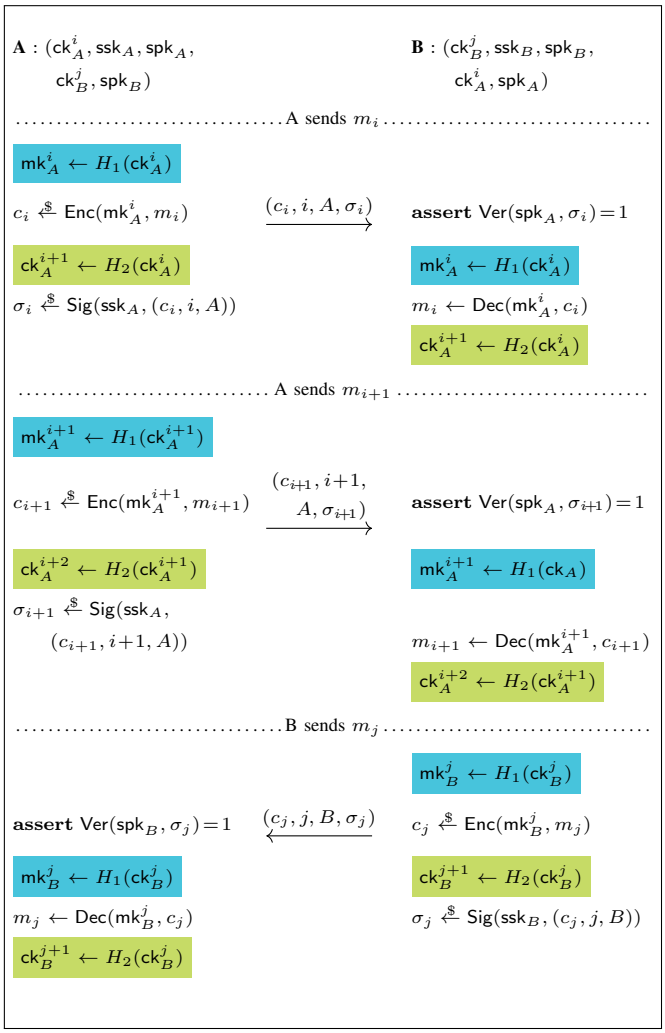

Figure 1: Sending/receiving messages between two group members for a three-message (in-order) conversation. Ephemeral message keys mk are deleted immediately after use. A's initial sender key is $(\mathsf{ck}^i_A, \mathsf{ssk}_A)$ and B's initial sender key is $(\mathsf{ck}^j_B, \mathsf{ssk}_B)$.

IV. Security Model

We propose a model of security for our Group Messenger primitive that captures the security suitable for an authenticated and forward-secure group messaging scheme. We introduce a game played between a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ and a challenger.

A. Game Description

At the beginning of the game, a bit b is uniformly sampled which parametrises the game. To win, the adversary either has to guess b or carry out a successful forgery in a *clean* protocol run. The game is parameterised by a *cleanness* predicate (sometimes safety [9] predicate) that captures the exact security of the protocol, namely the authenticity and confidentiality of group messages.

In this work, we assume that the two-party communication channels used for sending sender keys are perfectly secure; i.e., always confidential and authenticated. This assumption is not easily met in practice, but allows us to capture the essence of Sender Keys alone.

For simplicity, the game starts with a pre-established group $\boldsymbol{G}$ where every member $\mathsf{ID} \in \boldsymbol{G}$ already has everyone's honestly generated sender key. Then, $\mathcal{A}$ can interact with several oracles:

- $\mathcal{O}^{\mathsf{Challenge}}(\mathsf{ID}, m_0, m_1)$. The adversary receives a ciphertext C_b corresponding to ID sending m_b, i.e., the output of $C_b \leftarrow \mathsf{Send}(m_b, \gamma_{\mathsf{ID}})$.
- $\mathcal{O}^{\mathsf{Send}}(\mathsf{ID}, m)$. ID sends an application-level message m using the Send algorithm, producing a ciphertext C.
- $\mathcal{O}^{\mathsf{Receive}}(\mathsf{ID}, \mathsf{ID}', C)$. ID calls Recv on an application-level ciphertext C claimed to be from user ID'. If C has not been generated honestly via the $\mathcal{O}^{\mathsf{Send}}(\mathsf{ID}, m)$ oracle and receiving is successful, b is leaked to $\mathcal{A}$.
- $\mathcal{O}^{\mathsf{Add}}(\mathsf{ID}, \mathsf{ID}')$. ID adds ID' to the group via Exec, generating a control message T.
- $\mathcal{O}^{\mathsf{Remove}}(\mathsf{ID}, \mathsf{ID}')$. ID removes ID' from the group via Exec, generating a control message T.
- $\mathcal{O}^{\mathsf{Deliver}}(\mathsf{ID}, T)$. A control message T is delivered to ID who calls Proc.
- $\mathcal{O}^{\mathsf{Expose}}(\mathsf{ID})$. The current state γ of ID leaks to $\mathcal{A}$.
- $\mathcal{O}^{\mathsf{ExpMK}}(\mathsf{ID}, i)$. The i-th message key mk of ID leaks to $\mathcal{A}$. No message encrypted under this key can be challenged (neither before nor after exposure).

After q oracle queries, the adversary outputs a guess b' of b. Note that $\mathcal{A}$ can win the game either by guessing a challenge correctly or by injecting a forged message via $\mathcal{O}^{\mathsf{Receive}}$ successfully.

B. Cleanness

For the particular case of Sender Keys, we describe the cleanness predicate which defines the following conditions for a valid game:

- We define the event $\mathsf{refresh}(\mathsf{ID})$ that occurs when: 1) some member has been removed from the group, and 2) member ID processes this change (note the lack of a PCS update option).
- After exposing *any* user ID, all adversarial calls to $\mathcal{O}^{\mathsf{Challenge}}$ on future messages are disallowed until $\mathsf{refresh}(\mathsf{ID})$ occurs for *every* $\mathsf{ID} \in \boldsymbol{G}$. This extends to all challenges on skipped messages (out-of-order) that ID has not received at exposure time.
- After exposing *a specific* user ID', $\mathcal{A}$ cannot win the game by impersonating ID' via $\mathcal{O}^{\mathsf{Receive}}(\mathsf{ID}, \mathsf{ID}', C)$ for a forgery C until a new $\mathsf{refresh}(\mathsf{ID})$ event occurs.

We remark that our security notion is adaptive insofar as users can adaptively expose users. Under our cleanness predicate, we consider limited injection queries, i.e., partially active security. Our modelling further assumes that the underlying two-party channels are perfectly secure, and thus we leave it as important future work to examine security where, e.g., state exposures on the underlying channels are allowed and the consequent security guarantees are captured.

V. Security

We claim that Sender Keys, as described in Section III, is secure with respect to our security model. However, the security captured by our cleanness predicate is sub-optimal, in the sense that forward security can be strengthened for authentication, as we introduce in Section V-C. Here we introduce a security analysis, but leave a security proof and more accurate modelling for future work.

A. Passive adversaries

For a (semi-)passive adversary which does not attempt to inject messages via $\mathcal{O}^{\mathsf{Receive}}$ (but can still schedule messages arbitrarily), we claim that Sender Keys is secure with respect to the cleanness predicate, given that the symmetric encryption scheme is IND-CPA secure. Towards proving this:

- If the KDF is a one-way function, message keys mk_i can be exposed independently; the compromise of a message key never affects the confidentiality of other keys or messages. Hence, giving adversarial access to $\mathcal{O}^{\mathsf{ExpMK}}(\mathsf{ID}, i)$ does not impact the cleanness predicate.
- Also assuming one-wayness of the KDF, forward secrecy holds trivially except for out-of-order messages.
- Assuming that the two-party channels are secure, all users recover from state exposure via $\mathcal{O}^{\mathsf{Expose}}(\mathsf{ID})$ after a removal is made effective. Note that, outside our model, security of the two-party channels may also degrade after a state exposure, leaving room for further attacks.

B. Single- vs multi-key

In a Sender Keys group, each user is associated with a different symmetric key and thus the state comprises $O(n)$ secret material at all times. Since users encrypt and then hash forward using their own key when sending each message, users can safely send messages concurrently and with some inter-member message reordering.

For large groups, however, this scaling behaviour may represent a bottleneck. Consequently, one can envision trade-offs between the amount of concurrency supported and the amount of secret material required to be stored at a given point a time. The other extreme of the spectrum would be when all users maintain the *same* single symmetric chain. In situations where users are not expected to concurrently send messages this allows the secret state size to reduce to $O(1)$ without degrading security. To deal with concurrency in this setting, a central server which rejects all but the first (for example) message and requests re-transmission for other users could then be employed.

In MLS, a new group secret (chosen by a single user) is established each epoch, from which point all $O(n)$ application keys are derived for a given point in time; MLS additionally supports out-of-order message delivery within a given epoch. In Sender Keys, each user chooses their own key; thus, the security of ciphertexts in the presence of a passive adversary is

contingent upon users initially sampling their key with enough entropy.

C. Active attacks

In an active adversary scenario, the security of Sender Keys is sub-optimal. In particular, we note two issues. First, consider a simple group $\boldsymbol{G} = \{\mathsf{ID}_1, \mathsf{ID}_2\}$ and the following sequence of oracle queries:

- $q_1 = \mathcal{O}^{\mathsf{Send}}(\mathsf{ID}_1, m)$ which generates the i-th message C encrypted under mk and signed under ssk_1.
- $q_2 = \mathcal{O}^{\mathsf{Expose}}(\mathsf{ID}_1)$, where $\mathcal{A}$ obtains ssk_1, but not mk.
- $q_3 = \mathcal{O}^{\mathsf{ExpMK}}(\mathsf{ID}_1, i)$, leaking mk.
- $\mathcal{A}$ crafts $c' = Enc(m', \mathsf{mk})$, signs it under ssk_1 and forges a C'.
- $q_4 = \mathcal{O}^{\mathsf{Deliver}}(\mathsf{ID}_2, C')$, as the i-th message.

Note that q_4 is a forbidden query by our cleanness predicate in Section IV-B. q_4 attempts to inject a message that corresponds to key material utilized *before* the state exposure, hence one can envision stronger forward security where queries like q_4 are allowed. In this case, the Sender Keys adversary would win the game. We describe how to achieve such stronger security in Section V-D by strengthening signatures.

Technically, the query q_3 can be replaced by $\mathcal{O}^{\mathsf{Expose}}(\mathsf{ID}_k)$, or even be omitted as the adversary can still create a valid forgery by altering the metadata in C (such as the sender's identity) without crafting a new c'. This attack can occur naturally if ID_2 is offline when m is first sent.

An attack of a similar nature can also occur if the same signature key is re-used across groups, and they are refreshed at different times, as pointed out in [17].

The second issue we note is that the implementation of the Exec algorithm can be problematic if messages are not authenticated correctly. This led to attacks in the past such as the *burgle into the group* or the *acknowledgement forgery* attacks in [21]. Securing control messages and group membership changes is possible as introduced in [16].

D. Proposed Modifications and Tweaks

1) Ratcheting signature keys: The attack shown in the previous section can be mitigated if signature keys are also ratcheted. A simple fix is to introduce a *chain* of signature keys, where an ephemeral signature key is created every time a message or block of messages is sent.

Let $(\mathsf{ssk}, \mathsf{spk})$ be ID's signature key pair, where spk is part of its Sender Key. Then, 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 do as follows:

- Encrypt the i^{th} message m with the corresponding mk, $c \xleftarrow{\$} \mathsf{Enc}(\mathsf{mk}, m)$.
- Sign $C \leftarrow (c, i, \mathsf{ID}, \mathsf{spk}')$ as $\sigma \xleftarrow{\$} \mathsf{Sig}(\mathsf{ssk}, C)$
- Send the tuple (σ, C) to the group.
- Replace ssk by ssk'.

Upon reception of a message, ID' will:

- Verify the signature as $\mathsf{Ver}(\mathsf{spk}, \sigma, C)$ and decrypt the ciphertext c.
- Replace spk by spk' in ID's sender key.

Note that this countermeasure involves a notable overhead and entropy consumption (although only for the sender), so it may not be desirable in all scenarios, or for all sent messages. Users could also replace their signature keys on-demand or on a time schedule to trade more post-compromise security for performance.

2) Randomness manipulation: We note that Sender Keys, as described in Section III, 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 or with part of the state, as in [2] and the classic NAXOS trick for authenticated key exchange [22].

3) Refresh option for PCS: Post compromise security (PCS) is generally achieved when introducing fresh randomness by establishing new group secrets. In the case of Sender Keys, it could be beneficial to establish new sender keys across the group at some intervals, for instance via an *update* group operation, as noted in [17]. However, if only Alice updates her sender key, a passive adversary would still be able to eavesdrop on messages sent by any other group member. Furthermore, as the sender key is group-specific, any messages sent by Alice in another group are also susceptible to eavesdropping. We refer the reader to Appendix A of [17] for a more detailed discussion and conclude that the sender keys is not a suitable approach towards achieving a PCS-secure group messenger.

VI. Conclusion and Future Work

The Sender Keys protocol is a convenient protocol for group chats which is simple to implement, achieves a fair degree of end-to-end forward security, and deals well with concurrency. Nevertheless, we remark that no security is gained by employing multiple sender keys with respect to a single group key given randomness used to generate the keys is honest (i.e., in our model). Furthermore, forward security is sub-optimal for message authentication, which can be fixed by ratcheting signature keys.

Our results are under the assumption that two-party channels are secure, which is clearly not the case in practice. Therefore, more powerful attacks may arise under a more realistic model where exposure of two-party channels is possible. We also note that, in the real world, the security of messaging apps can also be broken in different ways than attacking the protocol. Additional features such as conversation transcript backups and multi-device support highly increase the attack surface and should be avoided in applications where security is critical.

Our analysis leaves multiple directions for future work, such as a rigorous formalization of the security model with a fine-grained analysis of the two-party channels and user compromise, more precise cleanness predicates, and especially concrete security reductions.

Acknowledgements

This work has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation program under project PICOCRYPT (grant agreement No. 101001283), and by a grant from Nomadic Labs and the Tezos foundation. The last author

was supported by DFG under Germanys Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] M. Marlinspike and T. Perrin, "The double ratchet algorithm," 2016. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf 1, 2

[2] J. Jaeger and I. Stepanovs, "Optimal channel security against fine-grained state compromise: The safety of messaging," in *Advances in Cryptology – CRYPTO 2018, Part I*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10991. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 33–62. 1, 5

[3] B. Poettering and P. Rösler, "Asynchronous ratcheted key exchange," Cryptology ePrint Archive, Report 2018/296, 2018, https://eprint.iacr.org/2018/296. 1

[4] F. B. Durak and S. Vaudenay, "Bidirectional asynchronous ratcheted key agreement with linear complexity," in *IWSEC 19: 14th International Workshop on Security, Advances in Information and Computer Security*, ser. Lecture Notes in Computer Science, N. Attrapadung and T. Yagi, Eds., vol. 11689. Tokyo, Japan: Springer, Heidelberg, Germany, Aug. 28–30, 2019, pp. 343–362. 1

[5] J. Alwen, S. Coretti, and Y. Dodis, "The double ratchet: Security notions, proofs, and modularization for the Signal protocol," in *Advances in Cryptology – EUROCRYPT 2019, Part I*, ser. Lecture Notes in Computer Science, Y. Ishai and V. Rijmen, Eds., vol. 11476. Darmstadt, Germany: Springer, Heidelberg, Germany, May 19–23, 2019, pp. 129–158. 1

[6] F. Balli, P. Rösler, and S. Vaudenay, "Determining the core primitive for optimally secure ratcheting," in *Advances in Cryptology – ASIACRYPT 2020, Part III*, ser. Lecture Notes in Computer Science, S. Moriai and H. Wang, Eds., vol. 12493. Daejeon, South Korea: Springer, Heidelberg, Germany, Dec. 7–11, 2020, pp. 621–650. 1

[7] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol," Internet Engineering Task Force, Internet-Draft draft-ietf-mls-protocol-14, May 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-14 1

[8] K. Bhargavan, R. Barnes, and E. Rescorla, "TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)," Inria Paris, Research Report, May 2018. [Online]. Available: https://hal.inria.fr/hal-02425247 1

[9] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security analysis and improvements for the IETF MLS standard for group messaging," in *Advances in Cryptology – CRYPTO 2020, Part I*, ser. Lecture Notes in Computer Science, D. Micciancio and T. Ristenpart, Eds., vol. 12170. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 17–21, 2020, pp. 248–277. 1, 2, 4

[10] K. Klein, G. Pascual-Perez, M. Walter, C. Kamath, M. Capretto, M. Cueto, I. Markov, M. Yeo, J. Alwen, and K. Pietrzak, "Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement," in *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 268–284. 1

[11] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk, "Continuous group key agreement with active security," in *TCC 2020: 18th Theory of Cryptography Conference, Part II*, ser. Lecture Notes in Computer Science, R. Pass and K. Pietrzak, Eds., vol. 12551. Durham, NC, USA: Springer, Heidelberg, Germany, Nov. 16–19, 2020, pp. 261–290. 1

[12] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Modular design of secure group messaging protocols and the security of MLS," in *ACM CCS 2021: 28th Conference on Computer and Communications Security*, G. Vigna and E. Shi, Eds. Virtual Event, Republic of Korea: ACM Press, Nov. 15–19, 2021, pp. 1463–1483. 1

[13] WhatsApp, "WhatsApp Encryption Overview Technical white paper, v.3," Oct. 2020, https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. 1, 2, 3

[14] M. Marlinspike *et al.*, "Signal protocol." [Online]. Available: https://github.com/signalapp/libsignal-protocol-java/tree/master/java/src/main/java/org/whispersystems/libsignal 1, 2, 3

[15] K. Cohn-Gordon, C. J. F. Cremers, and L. Garratt, "On post-compromise security," in *CSF 2016: IEEE 29st Computer Security Foundations Symposium*, M. Hicks and B. Köpf, Eds. Lisbon, Portugal: IEEE Computer Society Press, jun 27-1 2016, pp. 164–178. 1

[16] D. Balbás, D. Collins, and S. Vaudenay, "Cryptographic Administrators for Secure Group Messaging," Cryptology ePrint Archive. 1, 5

[17] C. Cremers, B. Hale, and K. Kohbrok, "The complexities of healing in secure group messaging: Why cross-group effects matter," in *USENIX Security 2021: 30th USENIX Security Symposium*, M. Bailey and R. Greenstadt, Eds. USENIX Association, Aug. 11–13, 2021, pp. 1847–1864. 1, 5

[18] M. Marlinspike and T. Perrin, "The x3dh key agreement protocol," 2016. [Online]. Available: https://signal.org/docs/specifications/x3dh/x3dh.pdf 2

[19] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," IETF Internet Request for Comments 2104, Feb. 1997. 2

[20] H. Krawczyk and P. Eronen, "Hmac-based extract-and-expand key derivation function (hkdf)," Internet Requests for Comments, RFC Editor, RFC 5869, May 2010, http://www.rfc-editor.org/rfc/rfc5869.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5869.txt 2

[21] P. Rösler, C. Mainka, and J. Schwenk, "More is less: On the end-to-end security of group chats in signal, whatsapp, and threema," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 415–429. 5

[22] B. A. LaMacchia, K. Lauter, and A. Mityagin, "Stronger security of authenticated key exchange," in *ProvSec 2007: 1st International Conference on Provable Security*, ser. Lecture Notes in Computer Science, W. Susilo, J. K. Liu, and Y. Mu, Eds., vol. 4784. Wollongong, Australia: Springer, Heidelberg, Germany, Nov. 1–2, 2007, pp. 1–16. 5