Bots can Snoop: Uncovering and Mitigating Privacy Risks of Bots in Group Chats

Kai-Hsiang Chou*
National Taiwan University

Yi-Min Lin*
National Taiwan University

Yi-An Wang
National Taiwan University

Jonathan Weiping Li
National Taiwan University

Tiffany Hyun-Jin Kim
HRL Laboratories

Hsu-Chun Hsiao†
National Taiwan University
Academia Sinica

Abstract

New privacy concerns arise with chatbots on group messaging platforms. Chatbots may access information beyond their intended functionalities, such as sender identities or messages unintended for chatbots. Chatbot developers may exploit such information to infer personal information and link users across groups, potentially leading to data breaches, pervasive tracking, or targeted advertising. Our analysis of conversation datasets shows that (1) chatbots often access far more messages than needed, and (2) when a user joins a new group with chatbots, there is a 3.6% chance that at least one of the chatbots can recognize and associate the user with their previous interactions in other groups. Although state-of-the-art (SoA) group messaging protocols provide robust end-to-end encryption and some platforms have implemented policies to limit chatbot access, no platforms successfully combine these features. This paper introduces SnoopGuard, a secure group messaging protocol that ensures user privacy against chatbots while maintaining strong end-to-end security. Our protocol offers (1) selective message access, preventing chatbots from accessing unrelated messages, and (2) sender anonymity, hiding user identities from chatbots. SnoopGuard achieves $O(\log n + m)$ message-sending complexity for a group of n users and m chatbots, compared to $O(\log(n+m))$ in SoA protocols, with acceptable overhead for enhanced privacy. Our prototype implementation shows that sending a message to a group of 50 users and 10 chatbots takes about 10 milliseconds when integrated with Message Layer Security (MLS).

1 Introduction

Chatbots have recently soared in popularity, partially thanks to the development of powerful Large Language Models (LLMs) [11, 18]. Even before the advent of LLMs, sophisticated rule-based conversational agents have been part of group messaging platforms. They provide a range of functions, from facilitating multilingual communication [66] to regulating chat content and group members [17]. However, this integration into group chats presents complex security challenges that go beyond the scope of typical one-on-one interactions. Unlike one-on-one interactions, where chatbots are the direct recipients, group conversations can inadvertently expose more information than necessary for the chatbot's functionality. For instance, granting full access to all group messages to a chatbot, which is only triggered by predefined commands, clearly violates the principle of least privilege, raising significant privacy concerns. In this work, we focus on two types of excessive information exposure:

(1) Irrelevant Messages: Those messages irrelevant to chatbots might contain sensitive data, including personally identifiable information (PII) like phone numbers and credit card numbers. Even in the absence of explicitly sensitive data, significant personal details can be inferred from dialogues. For instance, arranging a dinner can inadvertently reveal participants' locations. Staab et al. have shown that large language models (LLMs) can accurately infer personal attributes such as gender and location from conversations, even without explicitly mentioning these attributes [62]. These details can be used to build comprehensive profiles of group members, raising significant privacy concerns and exposing users to potential eavesdropping by seemingly harmless chatbots. Edu et al. reported a case where a chatbot developer was caught opening a URL shared in a honeypot group chat [33], reinforcing the claim that chatbots can be used to gather information beyond their intended functionality.

(2) User Identities. Besides processing irrelevant messages, a chatbot developer can link private information to specific users using metadata associated with each message. For example, the Telegram chatbot "Dr.Web" provides a service to check for malicious links in group messages [31]. Since Telegram's API includes a user ID with each message, Dr.Web could potentially record users' browsing history when analyzing shared links, even across multiple groups. This risk is

*Both authors contributed equally to this research.
†Hsu-Chun Hsiao (hchsiao@csie.ntu.edu.tw) is the corresponding author.

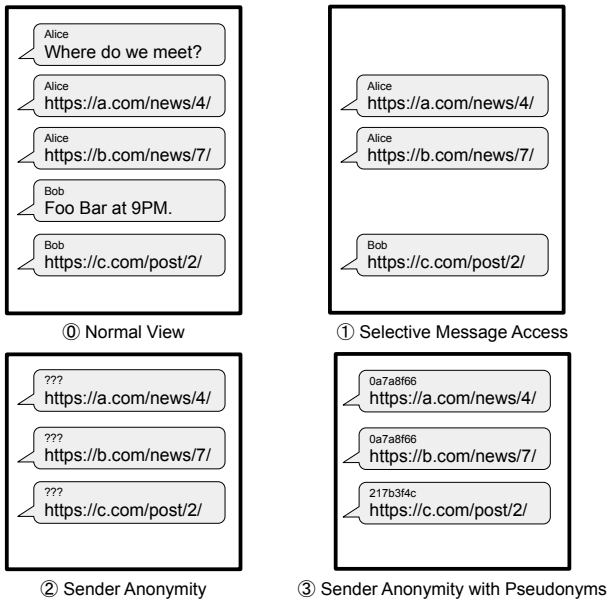


Figure 1: Example views of typical group messages from a *URL-checking bot* (⓪) and messages when our desired privacy policy is enforced (①–③). Selective message access ① ensures that the chatbot can only view messages *relevant* to its functionality (i.e., containing URLs). With sender anonymity ②, the chatbot does not know the sender's identity. With sender anonymity featuring pseudonyms (③), the chatbot can distinguish between senders using pseudonymous identifiers that may change over time.

akin to the privacy concerns posed by ubiquitous third-party cookies, which enable trackers to covertly collect parts of users' browsing histories. In this context, the access to user metadata, which is not essential for Dr.Web's functionality, exemplifies excessive exposure of user identities to chatbots. This risk is exacerbated when user messages not intended for chatbots are linked to users' identities, facilitating online profiling.

To address such privacy concerns, Platforms like Telegram, Slack, and Keybase have introduced various measures to reduce the exposure of user information to chatbots. However, there is a tension between these measures and the goal of implementing end-to-end encryption (E2EE), a common standard in messaging applications that protects user privacy from service providers. Our case studies (§4) reveal that none of the existing platforms sufficiently resolve this tension, highlighting the need for new solutions.

1.1 Our Contributions

This paper identifies two significant privacy issues in group messaging platforms with chatbots: (1) chatbots accessing all messages, regardless of relevance to their functions, and (2) unnecessary disclosure of message senders' identities to chatbots.

To understand the significance of privacy concerns, we conducted case studies on two conversation datasets and show that these problems are prevalence. First, our case study of a Discord chatbot shows that while only 0.24% of messages are intended to the chatbot, it has access to all messages, resulting in excessive information access. Second, our analysis on a Telegram dataset shows that 3.6% of users encounter the same chatbots in multiple groups. This highlights the prevalence of users being identified by the same chatbots across multiple groups. We also conducted a survey on messaging platforms to explore how they address privacy issues. By building chatbots on various platforms, we examined the consistency between their policies and implementations. None of the platforms studied mitigates privacy concerns while maintaining modern E2EE. We highlighted the tension, which our proposed protocol aims to address.

We highlight two key properties for secure group messaging: (1) Selective Message Access, which prevents chatbots from accessing irrelevant messages, and (2) Sender Anonymity, which hides user identities from chatbots, each addressing a distinct privacy issue. Figure 1 illustrates the impact of the two properties from the perspective of a chatbot, showcasing how they enhance privacy within group chats.

To realize these critical privacy properties, we propose SnoopGuard, a secure group messaging protocol that supports both while ensuring strong E2EE. Our design leverages continuous group key agreement (CGKA) schemes to maintain robust security, incorporates individual key management for each chatbot to enable selective message access, and extends existing CGKA protocols to support sender anonymity.

Theoretical analysis shows that SnoopGuard requires $O(\log n + m)$ cryptographic operations to send a message in a group of n users and m chatbots, compared to the $O(\log(n+m))$ required by Message Layer Security (MLS). This demonstrates that our additional privacy protections introduce a manageable overhead for groups with a small number of chatbots, suitable for most practical scenarios. Our prototype implementation, which is based on Signal's Protocol and MLS, demonstrates that sending a message within a group of 50 users and 10 chatbots takes roughly 10 milliseconds on a Mac mini with an M2 processor.

2 Background on Group Chat and Chatbot

2.1 Messaging Platforms

Messaging platforms like WhatsApp, Telegram, LINE, and Slack facilitate real-time communication between users, either one-on-one or in groups. Specifically, their group (multi-user) chat functionality allows a group of people to converse asynchronously. The group initiator has the authority to se-

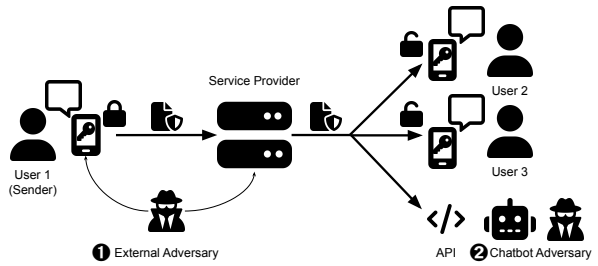


Figure 2: A typical messaging platform involves a service provider forwarding messages among group members. Two primary adversaries against users' privacy in this setting are ❶ malicious service providers with key-compromise capability and ❷ overprivileged chatbots. While state-of-the-art secure group messaging can address ❶ only, our work aims to address both. The icons in the figure are from Font Awesome.

lect members for the group, while other users can join later. Members can send and receive messages at their convenience, without the need to be online simultaneously, as the messaging platform relays messages sent by users to other users when they are online.

Many messaging platforms support adding chatbots to groups, such as Telegram, Discord, and LINE. Group members can interact with chatbots using text-based commands or natural-language prompts, enhancing group communication and productivity by providing convenient access to online services. For example, chatbots can assist multilingual groups with real-time translation [66] and help detect phishing URLs [31] or false information [1] shared by group members. Figure 2 illustrates how messages are delivered within a group of four members, including users 1-3 and a chatbot. The messaging platform runs a centralized server called *service provider* that facilitates the delivery of user 1's messages to both users 2 and 3, as well as to the chatbot.

2.2 Secure Messaging

Modern messaging applications rely on service providers, the servers of messaging platforms, to buffer and deliver messages (as shown in Figure 2). This centralized design presents a risk, as a curious service provider could potentially eavesdrop on user messages. Therefore, the primary goal of secure messaging protocols is to ensure that only senders and receivers can decrypt messages using end-to-end encryption (E2EE). E2EE guarantees that only the parties involved in the communication group can access the plaintext messages, and is now widely accepted as the security standard for messaging platforms. In addition, it is desirable that this security guarantee be resilient to key compromise.

Secure messaging protocols were initially developed for two-party communications and later extended to group settings. In two-party secure messaging, Borisov et al. [16] highlighted the vulnerability of key compromise, showing that traditional public key cryptography, like Pretty Good Privacy (PGP), does not protect messages encrypted *before* a compromise. They introduced the off-the-record (OTR) messaging, where parties continuously negotiate new Diffie-Hellman (DH) session keys and delete old keys to achieve forward secrecy (FS) [40]. Cohn et al. [25] expanded the notion of FS to protect message confidentiality and integrity *after* key compromise, introducing post-compromise secrecy (PCS). They demonstrated that only stateful protocols can provide PCS against full key compromise. Building on OTR, the double ratchet algorithm [54] uses OTR's ratcheting design to generate fresh session keys for each message. This algorithm is the foundation for key agreement in popular messaging apps like WhatsApp, Signal, and Messenger's secret conversations, and has been formally proven to ensure both FS and PCS [3, 24].

To extend secure messaging from two to multiple parties, a straightforward design is to use pairwise secure channels between each two members, but the updating complexity is linear to the group size, lacking scalability for large groups. Another approach is the Sender Keys Protocol [69], which Signal and WhatsApp use for large groups; each member generates their own encryption key called a *sender key* and distributes the key to each group member through pairwise secure channels. The Sender Keys Protocol provides constant-time update and FS, but does not provide PCS [7, 13].

Recently, continuous group key agreement (CGKA) [4] was introduced as a unifying framework for group key agreement schemes, supporting asynchronous operations and strong security guarantees. CGKA schemes frequently update group keys to maintain confidentiality after potential key compromises, achieving FS and PCS. Tree-based CGKA protocols use key trees to reduce update complexity to logarithmic time. For example, Asynchronous Ratcheting Tree (ART) [26] is based on a Diffie-Hellman tree [48], while TreeKEM [12] uses a hash tree for greater efficiency. Message Layer Security (MLS), an IETF standard [9], adopts TreeKEM for group key management, and the security of MLS, TreeKEM, and related variants has been thoroughly analyzed [4–6]. Although state-of-the-art secure group messaging, such as MLS, can defend against malicious service providers with key compromise capability (❶ in Figure 2), to our knowledge, no existing protocols can protect user privacy against overprivileged chatbots as well (❷ in Figure 2).

2.3 Threat Model and Assumptions

We consider two types of adversaries: *overprivileged chatbots* and *external adversaries*, as shown in Figure 2. External adversaries include common adversaries considered in previous literature on secure messaging [55, 67].

Overprivileged Chatbots. Overprivileged chatbots are insider adversaries participating in group conversations as regular members. They have access to the group messages, group

metadata, and group events. We assume that chatbots are passive adversaries. In other words, active attacks, such as sending malicious group modification messages [55], are out of scope. While collusion between chatbots is possible, we assume no collusion between a chatbot and a group member, since such collusion would allow the chatbot to trivially access all information known to the member.

External Adversaries. External adversaries, including semi-honest service providers, have key compromise capabilities, allowing them to learn the keys on a user's device [25]. Following the typical assumption in PCS literature, we assume that a device will be compromised for only a finite period, after which the user will regain control and perform at least one secure operation. We assume that the service provider and chatbots do not collude with each other and acknowledge that such collusion could introduce new privacy risks outside the scope of this work.

We assume that the service provider removes users' network-level metadata (such as source IP addresses) before forwarding messages to chatbots. This assumption is practical in real-world scenarios and allows us to focus solely on ensuring application-level anonymity.

3 Privacy Issues of Chatbots in Group Chats

We present two case studies examining privacy risks from overprivileged chatbots in group settings. These studies analyze to what extent chatbots can access excessive message content and metadata beyond their functional requirements. Our analysis addresses the following research questions (RQs):

- **RQ1-a:** Are chatbots overprivileged in a single group?
- **RQ1-b:** If so, how many unnecessary messages and sender identities can a chatbot access in a group?
- **RQ2:** How likely are users to be identified by the same chatbots across multiple groups?

Our case studies used two datasets: DISCO [63] and Pushshift [10]. DISCO contains 1.5M Discord messages from 323.6K users across four groups (November 2019-October 2020). With readable English content but little group overlap, DISCO is better suited for analyzing excessive access in a single group (RQ1). Pushshift comprises 2.2M Telegram users across 27.8K groups (September 2015-November 2019), making it ideal for studying privacy leaks related to cross-group user linkage (RQ2).[1]

For ethical considerations and practical constraints, we focus on analyzing public datasets. Although these public datasets have limitations in representing private group chat dynamics, they enable a glance at potential privacy concerns, particularly regarding excessive access permissions and bot presence across multiple groups. We leave it to future work to investigate the exposure of sensitive content in private group conversations.

3.1 Case Study 1: Unnecessary Access

Methodology. To answer RQ1, we conducted a case study of one chatbot in the DISCO dataset, analyzed its access privileges, and measured the amount of data exposure. We chose depth over breadth due to the time-intensive nature of manual code and documentation analysis.

Since the dataset predated the widespread adoption of large language models, we focused on identifying rule-based chatbots by searching for recurring message patterns. This approach led us to discover the "Discord Gophers Bot," which responds to messages prefixed with `?go` by providing tutorial links.[2] For instance, sending `?go tour` to the group prompts the chatbot to reply with "A Tour of Go <https://tour.golang.org/welcome/1>," directing the user to the official Go tutorial. Similarly, the chatbot responds to `?go channels` by providing a YouTube tutorial link: "Understanding channels <https://www.youtube.com/watch?v=KBZlN0izeiY>."

Overprivileged Bots. To investigate whether a chatbot is overprivileged, we reviewed Discord Gophers Bot's source code and discovered a command handler snippet that processes messages starting with `?go`: `if !strings.HasPrefix(m.Content, "?go") { return }`. Additionally, we found that the chatbot accesses the message sender's user ID and username through `Author.ID` and `Author.Username` attributes. Although the exact permissions granted to the chatbot are undocumented, this code analysis indicates that it requires broad permissions to read all messages and identify their senders, which is beyond its functional requirements.

Disclosure of Irrelevant Messages. Our analysis revealed that of 246,685 messages in the bot's group, only 580 (0.24%) started with `?go` command. However, the bot had access to all messages—424 times more than necessary ($(246,685 - 580)/580$). Among those messages the chatbot should not have access, we discovered several containing PII, including one case where a user accidentally shared his/her email address while pasting error logs. Such an incident exemplifies the privacy risks overprivileged chatbots pose in group settings. As of this writing, the bot remains active in the Gophers server, though a January 2022 update implemented Discord slash commands, restricting the bot's access to only pre-registered command messages. As Section 4.2 will dis-

[1] The Pushshift dataset includes only Telegram "channels," which are public chatrooms open to any user. In Telegram, "groups" are chatrooms restricted to approved users. For consistency, we will use the term "group" to refer to "Telegram channels" throughout this discussion.

[2] Although the chatbot's name was anonymized, consultation with the Gophers community confirmed it as the open-source project Discord Gophers Bot" (https://github.com/discord-gophers/dgobot).

cuss, slash commands are now officially recommended due to their enhanced privacy and usability.

Unnecessary Disclosure of User Metadata. Recall that our code review revealed the chatbot's access to the sender's user ID and username. However, the primary purpose of Discord Gophers Bot is to recommend tutorials, a function that does not require the sender's information. While the bot includes an admin feature to modify tutorial links, this feature only requires distinguishing admin from non-admin users, which is achievable through methods using pseudonyms. Moreover, our analysis of 246,685 messages revealed no instances of link modifications by admins, suggesting that the sender identification was unnecessary for all recorded conversations.

This case study shows how chatbots can be overprivileged by accessing unnecessary message content and metadata (RQ1-a). Moreover, when overprivileged, bots can accumulate large volumes of potentially sensitive information beyond their operational needs (RQ1-b).

3.2 Case Study 2: Cross-Group Identification

Methodology. When a user encounters the same chatbot in multiple groups where sender identities are exposed, the chatbot can link the user across these groups and aggregate inferred information. To quantify the probability of cross-group user identification (RQ2), we analyzed the Pushshift dataset [10], focusing on Telegram chatbots with disabled privacy restrictions. Despite Telegram's default message access limitations, the chatbot metadata revealed that 718 (45.5%) of 1,577 analyzed chatbots operated with the privacy mode disabled, granting them complete message access. We reconstructed a member list for each group by analyzing account metadata and messages.[3] We quantified user-chatbot relationships by counting how many groups each unique user-chatbot pair shared (i.e., encounter count).

Prevalence of Cross-Group Chatbots. Among the 718 chatbots studied, 253 (35.2%) appeared in multiple groups, 44 (6%) appeared in more than 10 groups, and three appeared in over 100 groups, significantly increasing the possibility of cross-group tracking.

User-Chatbot Encounters. Among 1,168,344 users who joined at least one group, 42,508 (3.6%) users encountered the same chatbot in multiple groups, with one user having repeated encounters with 134 different chatbots. Moreover, among the 4,155,927 user-chatbot pairs encountered at least once, 97,813 pairs (2.4%) were encountered multiple times, and 96 pairs were encountered more than 10 times, with extreme cases of two pairs being encountered 55 times. These patterns demonstrate the extensive user data collection potential of an overprivileged chatbot.

[3] We focused only on active members who sent messages, as the dataset does not include complete member lists.

4 Tension Between Mitigating Privacy Issues and Achieving E2EE

This section examines the privacy practices of popular group messaging platforms. Our survey shows that these messaging platforms face a fundamental tension between protecting user privacy from chatbots and enabling E2EE. While some platforms protect user privacy by filtering messages and hiding metadata, these protections are not trivially enforceable in the presence of E2EE, which prevents the inspection of message content. Our findings highlight the need for a practical solution to reconcile the two competing privacy goals.

4.1 Methodology

To understand the tension, we examined how popular messaging services address below privacy goals and analyzed their design decisions.

- **Mitigating Privacy Issues of Chatbots:** (1) Chatbots should only have access to necessary messages for their intended function. (2) Chatbots should be unable to identify the sender of any message (e.g., through the permanent user identifier or profile information).
- **Achieving E2EE:** Messages should always be sent over E2EE channels that provide both FS and PCS.

We focused on globally popular platforms [28] that support either E2EE or chatbot policies for analysis, including WhatsApp, Viber, Telegram, LINE, Discord, and Signal. We also included Slack, which is mentioned in previous work on chatbot-related security issue [20], and Keybase, which features cryptography-based restrictions on bot access control in groups [46].

For platforms that officially support chatbots (§4.2), we investigated chatbot permissions by creating chatbots using the official APIs and following the respective platform guidelines. On each platform, we set up a group, added a chatbot, and tested its behavior by sending messages in various formats. These included standard messages, messages that mentioned or tagged the chatbot, and messages that use predefined formats (e.g., commands) on platforms that support such functionalities. We then used the chatbot APIs to analyze the messages the chatbot received and the metadata included in these messages. We also investigated whether message contents were encrypted using E2EE by using Wireshark to capture and analyze network packets.

For platforms without official chatbot support (§4.3), we searched for well-known unofficial open source libraries that claim to enable chatbot functionality, and analyzed their code. We chose not to create chatbots using these libraries because the programmatically-simulated user's E2EE status and message access policies are predictably identical to those of regular users. Also, using these libraries would violate the platforms' terms of service.

4.2 Platforms with Chatbot Support

Most platforms that support chatbots in group chats enforce policies to regulate their access control. In the following discussion, we examine the design measures these platforms employ to prevent chatbots from learning excessive information, and evaluate whether they maintain E2EE in the presence of chatbots.

Telegram. Telegram restricts message access for chatbots, but still exposes the sender's identity in messages and does not supports E2EE. By default, chatbots in a group operate in *Privacy Mode*, allowing them to access only system messages or messages that mention them using pre-registered commands [64]. However, they can still access metadata about the sender [43]. Additionally, Telegram's MTProto 2.0 protocol [42] only implements group chats using server-client encryption rather than E2EE [65].

Slack. Slack provides two different permission models for chatbots: the granular bot permission scopes and the legacy bot permission scopes [60]. The granular bot permission scopes are designed to improve privacy by implementing a new scope that limits chatbots to messages in which they are specifically mentioned. When mentioned, the chatbot can identify the sender within the group using a unique identifier, but this identifier does not directly link to the sender's profile, thereby achieving pseudonymity against chatbots. In contrast, chatbots under the legacy bot permission scopes have unrestricted access to messages. Slack encrypts all messages and data at rest and in transit. However, it does not provide E2EE, which means that messages can be decrypted by Slack on their servers [61].

Discord. Discord employs a fine-grained permission model for chatbots, covering message access in groups. By default, chatbots cannot read messages unless explicitly mentioned. Discord recommends using the new *Interactions* API [29], where chatbots can interact with messages with slash commands, buttons, or menus. Additionally, chatbot developers can request permission to access all messages, but this requires manual verification and approval, including checking the presence of a privacy policy, as outlined in Discord's criteria for approval [30]. On top of that, Discord does not hide the sender identities from the chatbots and does not support E2EE.

LINE. Group chats involving chatbots in LINE are not end-to-end encrypted, and privacy issues from chatbots are not mitigated. LINE offers group E2EE through a mechanism called "Letter Sealing," which is similar to the Sender Keys Protocol and provides FS [50, 51]. However, Letter Sealing is automatically disabled in groups when a chatbot is added. Furthermore, LINE does not implement any specific measures to reduce the privacy risks associated with chatbots, which have the same access privileges as regular users, including full access to messages and sender identities [27].

Keybase. All messages shared in Keybase Chat are transmitted through channels with E2EE, and it allows certain messages to be revealed to the chatbot while maintaining E2EE. Keybase [45] uses a strategy similar to Telegram's privacy mode to restrict the chatbot's access to messages. In addition to messages that explicitly mention the chatbot using `@Bot`, Keybase also shares messages that match commands pre-registered by the chatbot, offering flexibility. Message metadata reveals the sender's username, allowing the chatbot to learn the sender's identity.

For messages intended for the chatbot, Keybase uses bot-specific keys shared between the group members and the chatbot to encrypt the messages. The bot-specific key is derived from the team key, a key shared by all group members except the chatbots, and is then sent to the chatbot, encrypted with the chatbot's user key. By encrypting with the bot-specific key, both the chatbot and the group members can decrypt the message. Messages not intended for the chatbot are encrypted using the team key. As a result, the chatbot is aware of these messages but cannot decrypt the content since it lacks access to the team key [46, 47]. Due to its similar design to the Sender Keys protocol, it likely does not support PCS. This claim is supported by the absence of PCS in their documentation [47] or security assessment reports [56], and the lack of key renegotiation features in the current design. Furthermore, because group members are responsible for initiating the creation of bot-specific keys, chatbots cannot initiate interactions or update keys, even if the keys are compromised.

4.3 Platforms Lacking Chatbot Support

Platforms like WhatsApp, Viber, and Signal do not natively support chatbot functionality in group chats. As a workaround, some developers create automated users to simulate chatbots on these platforms. This has been confirmed by reviewing the source code of third-party chatbots [23, 52]. Since these unofficial chatbots operate as regular users, they have the same permissions, including full access to all group messages and sender identities. While WhatsApp, Viber, and Signal offer E2EE in group chats, as outlined in their respective documentation [58, 68, 69], most of these platforms use a Sender Keys-like protocol, which does not support PCS. Only Signal's private groups [59], which use pairwise encryption for small groups, support PCS.

4.4 Findings

Our survey revealed two common approaches among popular group messaging platforms, highlighting the tension between mitigating overprivileged chatbots and achieving E2EE.

Most platforms, such as Telegram, Discord, and Slack, forgo E2EE when interacting with chatbots, allowing service providers to control what information should be shared with chatbots. In contrast, some platforms, such as WhatsApp,

Viber, and Signal, focus on providing E2EE for group chats and do not support chatbots in group chats. However, the development of unregulated third-party bots, while retaining full E2EE capabilities, may have access to all messages. A notable exception to these two approaches is Keybase. It allows E2EE and message filtering to coexist, but lacks some essential security features found in modern E2EE protocols. In addition, none of these group messaging platforms effectively address the privacy concerns associated with unnecessarily revealing sender identities. Table 1 summarizes each platform's support for the desired privacy goals.

Our findings suggest that a novel technique is needed to resolve the tension between mitigating overprivileged chatbots and achieving E2EE. In the rest of the paper, we will present our proposed protocol to resolve this tension.

	WhatsApp*	Viber*	Signal*	LINE	Telegram	Discord	Slack	Keybase
Group E2EE	●	●	●	●	-	-	-	●
Group E2EE (w\ bots)	●	●	●	-	-	-	-	●
PCS	-	-	◐	-	-	-	-	-
Message access control	-	-	-	-	●	●	●	●
Hide sender	-	-	-	-	-	-	●	-

●: fully support, ◐: partially support, -: no support

Table 1: Comparison of security properties on popular messaging platforms. Signal partially supports PCS only in private groups. Platforms marked with * do not officially support chatbots.

5 SnoopGuard: A Privacy-Preserving Secure Group Messaging Protocol

We aim to design a secure group messaging protocol that addresses two privacy concerns posed by overprivileged chatbots: (1) lack of message access control, and (2) unnecessary disclosure of sender identities, all while ensuring robust E2EE with FS and PCS. Our solution, SnoopGuard, builds on a tree-based continuous group key agreement (CGKA) scheme, enhancing the traditional key tree structure to overcome its limitations and support our desired properties. This design enables selective message access and sender anonymity. Additionally, we introduce extensions to support pseudonymity and enhance user privacy by hiding message triggers from service providers.

5.1 Desired Properties and Challenges

We first outline the desired properties of our messaging protocol: *selective message access* and *sender anonymity* to address the two privacy issues, and challenges achieving them using existing protocols.

Selective Message Access. The concept of selective message access refers to the ability to restrict chatbots from accessing messages that are irrelevant to their functionality. Implementing this poses several challenges. First, E2EE makes it impractical for service providers to filter messages for chatbots, as platforms like Telegram and Discord currently do, since the providers cannot read the encrypted messages. Second, to maintain confidentiality, chatbots should not have access to the encryption keys used for messages not intended for them, as possessing the ciphertext alone would allow them to decrypt the messages if they had the necessary keys.

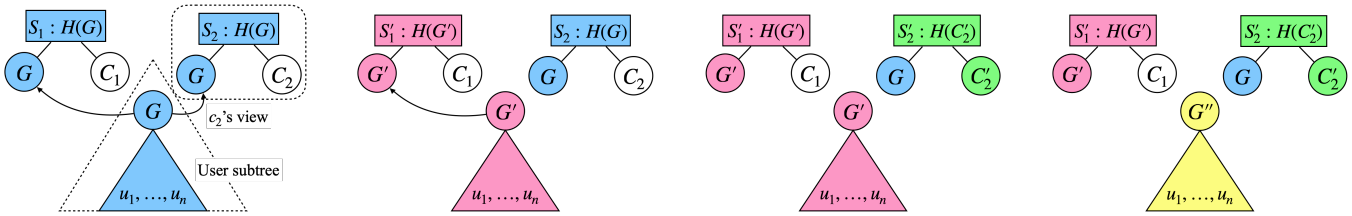
However, existing group messaging protocols such as Sender Keys do not facilitate selective key access, as they use static keys that do not change per message. Similarly, CGKA-based protocols that treat chatbots as users require all participants, including chatbots, to catch up with all key updates, also preventing selective access. Keybase's protocol allows selective key access by having a chatbot-specific sender key that is shared by all group members, but does not allow chatbots to update their keys and does not achieve PCS. Therefore, there is a need for a group messaging protocol that performs message filtering on the client side and implements a key management strategy that allows selective key access for chatbots, ensuring that they can only decrypt messages meant for them.

Sender Anonymity. Sender anonymity aims to hide the sender's identity from chatbots. Chen et al. define a form of sender anonymity called internal group anonymity (IGA), which guarantees sender indistinguishability among all group members [19]. However, our scenario requires a directional version: while users should be indistinguishable to chatbots, users should still be able to distinguish chatbots. To address this, we introduce a new notion called selective IGA, which provides directional sender anonymity. We also aim to achieve pseudonymity, another notion of sender anonymity that allows chatbots to differentiate between users without knowing their real identities, as in the case of anti-spam bots.

Despite this, no existing group messaging protocol provides the required level of sender anonymity. In the Sender Keys protocol, each sender can be uniquely identified through their individual sender keys. Similarly, in tree-based CGKA schemes like ART and TreeKEM, the "update path" [9] reveals node information from the sender's node to the root, which can be used to identify the sender. To address this, Chen et al. proposed Anonymous ART (AART) [19], which supports IGA, but does not support selective message access for the same reasons as other CGKA schemes.

5.2 Design Overview

Our solution is built on a tree-based CGKA scheme, as it is the most common key agreement scheme that achieves both FS and PCS. Before detailing the specific components of



(a) Users perform an update to obtain the group secret G and update the shared secrets for C_1, C_2. Secrets updated in this phase are colored blue.

(b) Users perform an update to obtain the group secret G' and update the shared secrets for C_1. Secrets updated in this phase are colored red.

(c) C_2 performs an update. Secrets updated during this phase are colored green. Updates initiated by chatbots will not trigger an update for the user subtree.

(d) Users trigger an update only for user subtree, resulting the group secret G''. Secrets updated during this phase are colored yellow.

Figure 3: Illustration of CMRT with users $u_1, \ldots, u_n$ and chatbots c_1, c_2. Users share the group secret G from the *user subtrees* (triangles), while C_1, C_2 are secrets for chatbots c_1, c_2, respectively. The rectangles represent secrets shared between the group and each chatbot. The arrows indicate secret assignments, and the lines indicate parent-child relationships, where a child knows the secret of its parent. For example, in (b), G' represents the group secret, while S_1' is the secret shared between $u_1, \ldots, u_n$ and C_1. In (c), the chatbot can send the fresh key S_2' to the users using the public key derived from G.

our protocol, we first explain how our approach addresses the challenges outlined in Section 5.1. The core of our approach lies in modifying the structure of key tree used in CGKA. We concatenate the tree root of the CGKA with several new "root nodes," each representing a separate group key.

Selective Message Access. One of the main challenge achieving selective message access is that chatbots may share inconsistent keys with users, since if chatbots do not receive all messages, they may miss key updates. However, chatbots using traditional CGKA require all key update messages to maintain key consistency. To solve this, our construction employs multiple root nodes, each corresponding to a separate group key—one for each chatbot. Each chatbot maintains its own key state shared with the user group, and its key is updated only when a message is specifically intended for that chatbot (as illustrated in Figure 3). If no key update occurs for a given chatbot, its key remains unchanged, allowing it to continue encrypting messages with the current key.

Selective IGA. Tree-based CGKA protocols typically fail to provide sender anonymity because the sender's identity can be inferred from the *update path*—the path from the sender's node to the root. To mitigate this, we modify the structure of key tree by separating group members into two subtrees: a *user subtree* containing all user nodes, and a *chatbot subtree* containing only the chatbot nodes (as shown in Figure 3). During a user-initiated key update, the chatbot is only made aware of the root of the user subtree, rather than the full details of the individual nodes within the subtree. This design allows the chatbot to compute a shared secret with the entire group without knowing which specific user initiated the update, thereby preserving the sender's anonymity.

The resulting structure of key tree, called the Compressed Multi-Roots Tree (CMRT), achieves these two additional security properties by assigning each chatbot a dedicated subtree while sharing a root node with the user subtree. From the users' perspective, their tree connects to multiple root nodes, each corresponding to a different chatbot. In contrast, each chatbot perceives itself as part of a smaller 3-node tree that links only to the root of the user subtree. Key updates are managed efficiently, with users only storing the root nodes, optimizing storage while maintaining asynchronous states across different chatbots, as shown in Figure 3. This "multi-root" structure is "compressed" to balance data storage efficiency with strong security guarantees.

5.3 Building Blocks

This subsection presents the building blocks of our construction, including cryptographic primitives, a formal definition of CGKA, and TreeKEM, the CGKA construction that serves as the foundation for our solution. Throughout this section, $\leftarrow\$$ denotes assignment from a randomized algorithm, while $\leftarrow$ represents assignment from a deterministic one.

Cryptographic Primitives. The public-key encryption (PKE) scheme $\mathsf{PKE} = (\mathsf{PKEG}, \mathsf{PKEnc}, \mathsf{PKDec})$ consists of key generation $(\mathsf{sk}, \mathsf{pk}) \leftarrow\$ \mathsf{PKEG}(1^\lambda)$, encryption $c \leftarrow\$ \mathsf{PKEnc}(\mathsf{pk}, m)$ using the public key pk, and decryption $m \leftarrow \mathsf{PKDec}(\mathsf{sk}, c)$ that retrieves the original message m from the ciphertext c. We also use a digital signature scheme $\mathsf{Sig} = (\mathsf{Sign}, \mathsf{Vf})$, where $s \leftarrow\$ \mathsf{Sign}(\mathsf{sk}, m)$ signs a message m, and $\mathsf{Vf}(\mathsf{pk}, s, m)$ verifies it using the corresponding public key. Additionally, a collision-resistant hash function $\mathsf{H} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ is used.

Continuous Group Key Agreement (CGKA). As detailed in Alwen et al. [4], a CGKA scheme comprises operations for key agreement in secure group communication, enabling all members to share a common secret key. This scheme

allows for dynamic adjustments in group membership, including adding or removing members. The shared key is updated whenever there are changes to the membership or upon request by any group member. For instance, the key can be refreshed with every sent message, which helps guarantee that future key compromises do not impact the security of previously sent messages. The syntax of CGKA is defined as follows, as outlined by Alwen et al. [4].

Definition 1. *A CGKA scheme is a tuple of the following algorithms* $\mathsf{CGKA} = (\mathsf{init}, \mathsf{create}, \mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{proc})$*:*

- $\gamma \leftarrow\$ \mathsf{init}(\mathsf{ID})$ *takes a user ID* ID *and outputs an initial state* γ*.*
- $(\gamma', T) \leftarrow\$ \mathsf{create}(\gamma, \mathsf{ID}_1, \ldots, \mathsf{ID}_n)$ *takes a state* γ *and a list of user IDs* $\mathsf{ID}_1, \ldots, \mathsf{ID}_n$*. This outputs a new state* γ' *and control message* T*.*
- $(\gamma', T) \leftarrow\$ \mathsf{add}(\gamma, \mathsf{ID})$ *takes a state* γ *and a user* ID *to add, and outputs a new state* γ' *and control message* T*.*
- $(\gamma', T) \leftarrow\$ \mathsf{rem}(\gamma, \mathsf{ID})$ *takes a state* γ *and a user* ID *to remove, and outputs a new state* γ' *and control message* T*.*
- $(\gamma', T) \leftarrow\$ \mathsf{upd}(\gamma)$ *takes a state* γ *and outputs a new state* γ' *and control message* T*.*
- $(\gamma', k) \leftarrow \mathsf{proc}(\gamma, T)$ *takes a state* γ *and a control message* T *or* W*. This outputs an updated state* γ' *and a fresh group key* k*.*

To initiate the CGKA scheme, each user, identified by ID, starts by initializing their state with $\mathsf{init}(\mathsf{ID})$. Upon initialized, a user can create a group using create with the initial members' identifiers. Subsequent group modifications, such as adding members with add, removing them via rem, or updating personal key material with upd, each triggers an update to the group key and generates a control message T. Group members keep their states synchronized by processing these control messages via proc, ensuring they all share the same group key k.

TreeKEM. TreeKEM [12] is a tree-based CGKA scheme constructed with a hashing key tree. Each node in this tree holds a secret accessible only to the members within its subtree, and each member is assigned to a leaf node. The secret of the root node serves as the shared secret for the entire group. In TreeKEM, let $S_i \in \{0,1\}^\lambda$ denote a member's i-th secret from the leaf. The secret S_i is computed as the hash of the secret S_{i-1} from one of its child nodes, specifically the last child that updates the secret. Additionally, each node contains a pair of public-private keys (sk_i, pk_i) generated from its secret S_i using PKEG. The node information can be computed by $(sk_i, pk_i) \leftarrow \mathsf{PKEG}(S_i)$ where $S_i \leftarrow H(S_{i-1})$.

To perform key update, a member (associated with one of the leaf node) randomly generates a new secret and iteratively computes the secrets along the path to the root node via hashing. The member then notifies other members of the new secrets by encrypting them with the public keys of the sibling nodes. Specifically, for a node v with a new secret S', the member encrypts S' using the public key of node $\mathsf{sibling}(v)$ and sends the ciphertext to the members under $\mathsf{sibling}(v)$, where $\mathsf{sibling}(v)$ denotes the sibling node of v. The member also publishes all new public keys along the updated path.

5.4 SnoopGuard Operations

We now introduce the core operations of our secure group messaging protocol, SnoopGuard. SnoopGuard extends existing secure group messaging protocols by adding chatbot-specific operations: chatbot addition/removal, message sending by chatbots, and trigger functions. It uses CMRT, which relies on an existing tree-based CGKA scheme for user subtree management, to enable selective message access and selective IGA. While SnoopGuard handles communication between users and chatbots, interactions among users continue to rely on the underlying secure group messaging protocols. We assume that the service provider acts as a public key infrastructure (PKI) where chatbots register their public keys and signed trigger functions. The pseudocode for the baseline protocol is presented in Appendix B.

Trigger Function. To enable client-side message filtering, we introduce the concept of *trigger function* represented as $f_{\mathsf{CID}} : \mathcal{M} \to \{0,1\}$, where $\mathcal{M}$ denotes the message space and CID denotes the unique identifier of a chatbot. The function returns 1 if a message is considered relevant to a chatbot identified by CID, and 0 otherwise. The function is evaluated on user's device before a message is sent. The function can be implemented in various form, such as a snippet code executed in a sandbox environment, or a checkbox that asks for permission.

State Initialization. The CMRT state for each entity, including a user or a chatbot, is denoted as γ. For users, the CMRT state includes the state of the underlying CGKA scheme $\gamma.s0$ and a dictionary $\gamma.\mathsf{cbts}$ indexed by chatbot identifier to record leaf nodes information for each chatbot in a group. For chatbots, the CMRT state consists of its identifier $\gamma.\mathsf{ME}$, the root public key of the user subtree $\gamma.\mathsf{gpk}$, a long-term PKE key pair $(\gamma.\mathsf{sk}_{\mathsf{CID}}, \gamma.\mathsf{pk}_{\mathsf{CID}})$ for identity authentication, and a PKE key pair $(\gamma.\mathsf{csk}, \gamma.\mathsf{cpk})$ associated with the chatbot's tree node.

When a chatbot identified by CID is initialized, it registers the long-term public key $\mathsf{pk}_{\mathsf{CID}}$, the trigger function f_{CID} and its signature $s \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathsf{CID}}, f_{\mathsf{CID}})$ to the service provider.

Group Creation. We assume only a user can create a group, and a group initially contains only users. The process of creating a group with identities $\mathsf{ID}_1, \ldots, \mathsf{ID}_n$ involves the following steps: (1) A user with state γ initiates the group using the CGKA scheme's create algorithm: $(\gamma.s0, T) \leftarrow \mathsf{CGKA.create}(\gamma, \mathsf{ID}_1, \ldots, \mathsf{ID}_n)$. (2) This user then broadcasts

the control message T to the users identified by $\mathsf{ID}_1, \ldots, \mathsf{ID}_n$. (3) Upon receiving the message, each user processes the control message T : $(\gamma.s0, k) \leftarrow \mathsf{proc}(T)$.

Adding or Removing Users. Similar to group creation, a user with state γ calls the CGKA scheme using $(\gamma.s0, T) \leftarrow \mathsf{CGKA.add}(\mathsf{ID}')$ or $(\gamma.s0, T) \leftarrow \mathsf{CGKA.rem}(\mathsf{ID}')$, where ID' is the identifier of the user to be added or removed. After that, the user broadcasts the control message T to all user members, which is then processed by $(\gamma.s0, k) \leftarrow \mathsf{proc}(T)$, resulting in a fresh group key k.

Adding a Chatbot. Adding a chatbot identified by a chatbot ID CID to a group involves four steps: (1) The initiating user retrieves chatbot's public key $\mathsf{pk}_{\mathsf{CID}}$, samples a leaf secret k, computes chatbot subtree's public key cpk by $(\mathsf{csk}, \mathsf{cpk}) \leftarrow \mathsf{PKEG}(k)$, and encrypts the secret: $e \leftarrow \mathsf{PKEnc}(\mathsf{pk}_{\mathsf{CID}}, k)$. (2) The user broadcasts root public key of user's subtree $\gamma.s0.\mathsf{gpk}$, the encrypted secret e, and public key cpk of chatbot's subtree, to all group members and the chatbot. (3) All users retrieve the chatbot's public key $\mathsf{pk}_{\mathsf{CID}}$, trigger function f_{CID}, and the associated signature s from the service provider and store the chatbot's information into their respective states: $\gamma.\mathsf{cbts}[\mathsf{CID}] \leftarrow (f_{\mathsf{CID}}, \gamma.s0.\mathsf{gsk}, \mathsf{cpk})$ if the signature is successfully verified by $\mathsf{Vf}(\mathsf{pk}_{\mathsf{CID}}, s, f_{\mathsf{CID}})$. (4) The chatbot decrypts the initial secret $k \leftarrow \mathsf{PKDec}(\gamma.\mathsf{sk}_{\mathsf{CID}}, e)$, and stores the received root public key gpk, and the derived PKE key pair $(\mathsf{csk}, \mathsf{cpk}) \leftarrow \mathsf{PKEG}(k)$. This process initializes a TreeKEM key tree between the user group and the chatbot.

Removing a Chatbot. Removing a chatbot identified by a chatbot ID CID involves the following steps. (1) The initiating user broadcasts the decision to remove the chatbot to all group members. (2) All users then clear the corresponding entry from their records: $\gamma.\mathsf{cbts}[\mathsf{CID}] \leftarrow \perp$. (3) The removed chatbot also clears its information about the user group: $\gamma.\mathsf{gpk} \leftarrow \perp$.

Users Sending a Message to Chatbot. When a user with state γ sends a message to chatbots, the process involves initiating an underlying CGKA (user subtree) key update (step 2) and a TreeKEM update (step 2 and 4), and encrypting the message using the newly derived shared secret (step 3). The steps are as follows: (1) Obtains a fresh group key k from the CGKA scheme: $(\gamma.s0, T_0) \leftarrow \mathsf{CGKA.upd}(\gamma.s0); (\gamma.s0, k) \leftarrow \mathsf{proc}(\gamma.s0, T_0)$. (2) Computes the group PKE key pair $(\mathsf{gsk}, \mathsf{gpk}) \leftarrow \mathsf{PKEG}(k)$ and a new message encryption key $k' \leftarrow \mathsf{H}(k)$. (3) Encrypts the message m: $c \leftarrow \mathsf{Enc}(k', m)$ and prepares the control message $T = (T_0, c, \mathsf{gpk})$. (4) For each chatbot identified by CID that requires the message (i.e., $f_{\mathsf{CID}}(m) = 1$), retrieves the chatbot's public key $\mathsf{cpk} \leftarrow \gamma.\mathsf{cbts}[\mathsf{CID}].\mathsf{cpk}$, appends $(\mathsf{CID}, \mathsf{PKEnc}(\mathsf{cpk}, k'))$ to the control message T, and updates the group secret key for that chatbot $\gamma.\mathsf{cbts}[\mathsf{CID}].\mathsf{gsk} \leftarrow \mathsf{gsk}$. (5) Finally, broadcasts the control message T to the group, including the chatbots.

We require that either the CGKA's control message T_0 is removed from the control message T before forwarded to the chatbots, or T_0 is encrypted in a way similar to MLS's Private Message [9], so that chatbots cannot decrypt T_0 nor learn the sender's identity. When sending a message to both users and chatbots, this process operates in parallel with the original secure group messaging protocol, which handles ciphertext generation and key updates for users.

A user with state γ receiving the control message T updates the keys using the following steps: (1) Computes the fresh group key through: $(\gamma.s0, k) \leftarrow \mathsf{CGKA.proc}(T_0)$. (2) Computes the group PKE key pair $(\mathsf{gsk}, \mathsf{gpk}) \leftarrow \mathsf{PKEG}(k)$. (3) For each CID in the control message, updates the group secret key $\gamma.\mathsf{cbts}[\mathsf{CID}].\mathsf{gsk} \leftarrow \mathsf{gsk}$.

A chatbot with state γ receiving the control message T first checks if its identifier CID is included. If not, the message is deemed unrelated to this bot. If the identifier is found, the chatbot proceeds to decrypt the message using the following steps: (1) Decrypts the chatbot encryption key k' from the ciphertext e associated with CID: $k' \leftarrow \mathsf{PKDec}(\gamma.\mathsf{csk}, e)$. (2) Decrypts the message content m using $m \leftarrow \mathsf{Dec}(k', c)$. (3) Updates the group public key in its state to $\gamma.\mathsf{gpk} \leftarrow \mathsf{gpk}$.

Chatbot Sending a Message. When a chatbot with state γ and identified by CID sends a message to users, it issues a TreeKEM update and encrypts the message using the newly derived shared secret. The steps are as follows: (1) Randomly generates a key: $k \leftarrow\!\!\$\ \{0,1\}^{\lambda}$. (2) Computes the message encryption key $k' \leftarrow \mathsf{H}(k)$ and the PKE key pair $(\mathsf{csk}, \mathsf{cpk}) \leftarrow \mathsf{PKEG}(k)$ for chatbot's tree node. (3) Encrypts the message m: $c \leftarrow \mathsf{Enc}(k', m)$ and encrypts the encryption key using group public key: $e \leftarrow \mathsf{PKEnc}(\gamma.\mathsf{gpk}, k')$ (4) Updates the chatbot private key $\gamma.\mathsf{csk} \leftarrow \mathsf{csk}$. (5) Finally, broadcasts the entire control message $T = (\mathsf{CID}, c, e, \mathsf{cpk})$ to the group.

A user with state γ receiving the control message T decrypts the message using the following steps: (1) Decrypts the encryption key using the dedicated group private key for CID: $k' \leftarrow \mathsf{PKDec}(\gamma.\mathsf{cbts}[\mathsf{CID}].\mathsf{gsk}, e)$. (2) Decrypts the message content: $m \leftarrow \mathsf{Dec}(k', c)$. (3) Updates the chatbot public key $\gamma.\mathsf{cbts}[\mathsf{CID}].\mathsf{cpk} \leftarrow \mathsf{cpk}$.

5.5 SnoopGuard Extensions

We present two SnoopGuard extensions: pseudonym visibility for stateful chatbot applications and trigger concealment from service providers for enhanced user privacy.

Pseudonymity Achieving pseudonymity involves additional two steps: (1) The group member registers a pseudonym anonymously ; (2) When a group member sends a message using the pseudonym, the chatbot verifies that the message truly comes from the authenticated sender behind the pseudonym. Our protocol should ensure that in both steps, the chatbot remains unaware of the exact identity of the member.

In our protocol, a pseudonym contains a PKE key pair used as message signing key. In the first step, a member generates an ephemeral identity key pair (sk_e, pk_e) and registers the

identity by broadcasting the public key pk_e to all chatbots using the baseline protocol, then each chatbot records pk_e as a new ephemeral identity in the group. In the second step, the member signs the message using sk_e, and the chatbot verifies the signature with pk_e to check the legitimacy of the message sender. Because baseline protocol provides sender anonymity, the chatbot cannot associate two pseudonyms with the same user. The users can easily change their pseudonyms by registering new ones.

Trigger Concealment from Service Providers Chatbots are typically triggered by messages matching specific patterns, which leaks information about message content to service providers. For example, if a phishing detection bot is triggered, the provider might deduce the message contains a URL. To enhance user privacy, senders can transmit control messages to chatbots without including key updates for those that should not be triggered. In step (4) of the procedure for sending a message to a chatbot, described in Section 5.4, when $f_{\mathsf{CID}}(m)$ is false, meaning that the chatbot identified by CID should not receive the message, the sender appends (CID, r), where r represents the random bytes that should be indistinguishable from the actual ciphertext of the key update, thereby "hiding" the triggering event. These chatbots would not be able to decrypt the new secret and would therefore drop the message.

5.6 Integration with Existing Secure Group Messaging Protocols

SnoopGuard needs to run alongside a secure group messaging protocol. There are two strategies for integration, depending on whether the underlying secure group messaging protocol uses a CGKA key tree.

Protocols without CGKA. Our protocol can coexist with secure group messaging protocols that do not rely on CGKA, such as the Sender Keys Protocol, without interfering with their operations. When creating a group and updating members, in addition to the original key exchange procedures, such as sharing sender keys, users also maintain a complete CMRT, including the user's subtree. When sending a message, the sender follows the original procedure to communicate with other users, which, in the case of the Sender Keys Protocol, involves encrypting messages with their sender key, and follows SnoopGuard to send messages to chatbots by encrypting with each chatbot's root key and issuing key updates.

Protocols with CGKA. The advantages of our protocol can be highlighted when integrated with secure group messaging protocols that use tree-based CGKA, such as MLS. In these cases, the protocol's key management already relies on a tree-based CGKA key tree to generate a shared secret for group members, and CMRT can utilize the existing tree as the user's subtree. This further reduces the overhead of maintaining the SnoopGuard, since the effort of storing and maintaining the user's subtree is already included in these protocols.

6 Evaluation

6.1 Security Analysis

	E2EE		Sender Anonymity	SMA
	FS	PCS		
Sender Keys	●	-	-	-
Keybase	●	-	-	●
MLS (TreeKEM)	●	●	-	-
AART [19]	●	●	IGA	-
SnoopGuard	●	●	Selective IGA	●

Table 2: Security comparisons between secure group messaging protocols. SMA stands for Selective Message Access.

This section provides a security analysis to show that our protocol satisfies the desired security properties, while achieving FS and PCS, as shown in Table 2. The analysis is based on the assumption that the underlying CGKA scheme satisfies FS and PCS. We use this fact to conclude that our protocol also satisfies the two properties, while also achieving selective message access. For sender anonymity, the analysis relies on the assumption that leaf secret chosen during key updates of TreeKEM is uniformly random.

We assume that the hash function H, used in both TreeKEM and our protocol, functions as a pseudorandom generator (PRG). This implies that if H receives random input, its output will be uniformly random. Furthermore, we assume the public key encryption (PKE) scheme employed is IND-CPA secure, which guarantees that an adversary cannot distinguish between the ciphertexts of any two chosen plaintexts. The formal security definition for these cryptographic primitives is included in Appendix A.

Also, a CGKA scheme should already satisfy the following two security properties, according to the security definition of key indistinguishability by Alwen et al. [4]. To formalize, we first define the epoch t, which is a protocol execution counter that advances whenever a control message is processed. Let γ_t denotes the state at epoch t and k_t denotes the group key at epoch t, we have the following relation for each group member: $(\gamma_t, k_t) \leftarrow \mathsf{proc}(\gamma_{t-1}, T)$ for any legitimate control message T.

- **Forward Secrecy**: For an external adversary who has access to all control messages T and compromises a member's state γ_t, the adversary should not be able to distinguish any key k_i for $i < t$ from a uniform random distribution.

- **Post-Compromise Security**: For an external adversary who has access to all control messages T and compromises a member's state, but a group member successfully

creates a commit at epoch t without the adversary's control, the adversary should not be able to distinguish any key k_i for $i > t$ from a uniform random distribution.

Forward Secrecy. Assuming that no chatbot is compromised, we show that for an external adversary with access to messages T and a compromised member state γ_t, any key k_i for $i < t$ remains indistinguishable from uniform random. First, the adversary gains no meaningful information about previous keys from the compromised keys because the underlying CGKA protocol ensures forward secrecy, making any key k generated before the compromise indistinguishable from random. By the definition of PRG, the group keys k' generated before the compromise also retain this indistinguishability, as $k' = \mathsf{PRG}(k)$. Second, the adversary gains no meaningful information from control messages containing encrypted previous keys, as the IND-CPA security of the PKE scheme ensures that these ciphertexts reveal no meaningful information about the plaintext keys and are indistinguishable from encryptions of random values.

Post-Compromise Security. Assuming that no chatbot is compromised, we show that for an external adversary with access to messages T and the state of a compromised member, if a group member performs an uncompromised operation at epoch t, then the subsequent key k_i for $i > t$ is indistinguishable from a uniform random distribution. This claim is based on principles similar to those supporting FS, where post-compromise group keys generated by a PRG are indistinguishable from random. In addition, the IND-CPA security of the PKE scheme ensures that control messages reveal no meaningful information about the future keys to an attacker.

Sender Anonymity. Assuming that the new leaf secret chosen during key updates of TreeKEM is uniformly random, we show that a chatbot adversary cannot significantly distinguish any message bundle between any two group members. Information in the message bundle comprises a constant chatbot identifier, a group public key derived from a uniformly random leaf secret via PRG operations, and a ciphertext secured under IND-CPA is indistinguishable from random.

Selective Message Access. We show that a chatbot adversary within a group cannot distinguish any unauthorized key k from a uniform random distribution by adhering to both forward secrecy and post-compromise security. For a chatbot adversary $\mathcal{A}$ authorized with key k_t at epoch t, forward secrecy ensures that all previous keys k_i for $i < t$ are secure and indistinguishable from random because they are generated via a PRG from a uniformly random source. In addition, these keys maintain indistinguishability under IND-CPA security because the control messages do not contain ciphertexts encrypted with the public key of $\mathcal{A}$. On the other hand, post-compromise security protects all subsequent keys k_i for $i > t$, preventing $\mathcal{A}$ from obtaining knowledge of these unauthorized keys and thus protecting future communications within the group.

6.2 Performance Analysis

This section analyzes the overhead of our protocol, both theoretically and empirically, and compares it to other secure group messaging protocols.

6.2.1 Theoretical Performance Analysis

In our protocol, the setup phase for a group with n users and m chatbots involves $O(n+m)$ public-key encryption (PKE) and hash operations, where the TreeKEM of user subtree construction requires $O(n)$ PKE and hashes, and computing shared keys for each chatbot takes $O(m)$ PKE and hashes. Sending a message to chatbots requires $O(\log n + m)$ complexity, where updating user subtree requires $O(\log n)$ PKE and hashes, and computing shared keys for each chatbot takes $O(m)$ PKE and hashes. Pseudonymity adds minimal overhead. The storage requirement is $O(n+m)$ for users and $O(1)$ for chatbots. The comparative result is presented in Table 3, and a more detailed analysis is presented in Appendix C.

6.2.2 Experimental Performance Evaluation

We implemented SnoopGuard using Go. Our prototype implementation includes two versions: one extends the existing `libsignal` library [58], where individual chats follow the Signal Protocol and group chats follow the Sender Keys Protocol; the other version is based on the Messaging Layer Security (MLS) protocol, using the `go-mls` library [22]. We conducted our experiments on a Mac mini equipped with Apple M2 processor and 16GB of RAM.

Figure 4 shows the time required to add a chatbot to groups of different sizes, measured from the initiation of the addition process to its completion for all users. Regardless of the underlying protocol, adding a chatbot to a group of 50 members takes about 2 milliseconds in our protocol, while it takes about 30 milliseconds in both original protocols. Adding a pseudonymous chatbot to a group incurs higher overhead due to pseudonym registration, but this can be minimized by combining registration with the user's first message to the chatbot, avoiding extra roundtrips.

Figure 5 shows the time required to send a message to group members and chatbots, from the start of encryption to the point where all users and chatbots have decrypted the message. Sending a message to 50 members and 10 chatbots takes about 10 milliseconds when integrated with MLS and about 5 milliseconds when integrated with the Sender Keys Protocol. The overhead of sending a message increases linearly with the number of chatbots, which is consistent with the theoretical analysis. Pseudonymity introduces a small additional overhead due to the signature processes involved.

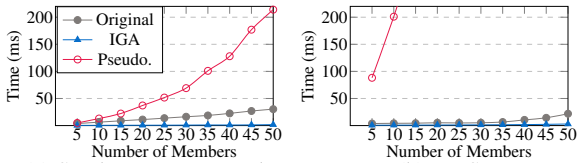
To evaluate the performance of our protocol on resource-constrained devices, we also benchmarked chatbot addition and message encryption on low-end containers, as shown in Figure 7 and Figure 8 in the appendix. We conclude that our

		number of exponentiations			number of encryptions or hashes		
		sender	per user	per chatbot	sender	per user	per chatbot
Sender Keys	setup	$O(n+m)$	$O(n+m)$	$O(n+m)$	$O(n+m)$	$O(n+m)$	$O(n+m)$
	ongoing	0	0	0	$O(1)$	$O(1)$	$O(1)$
(A)ART	setup	$O(n+m)$	$O(\log(n+m))$	$O(\log(n+m))$	0	0	0
	ongoing	$O(n+m)$	$O(n+m)$	$O(n+m)$	$O(1)$	$O(1)$	$O(1)$
MLS	setup	$O(n+m)$	$O(1)$	$O(1)$	$O(n+m)$	$O(\log(n+m))$	$O(\log(n+m))$
	ongoing	$O(\log(n+m))$	$O(1)$	$O(1)$	$O(\log(n+m))$	$O(\log(n+m))$	$O(\log(n+m))$
Ours	setup*	$O(n+m)$	$O(1)$	$O(1)$	$O(n+m)$	$O(\log n)$	$O(1)$
	ongoing	$O(\log n+m)$	$O(1)$	$O(1)$	$O(\log n+m)$	$O(\log n)$	$O(1)$

*: The overhead for registering pseudonyms, which is equivalent to sending a message from each user to the chatbot, is omitted.

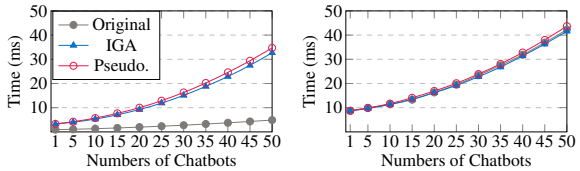
Table 3: Computation complexity comparison. n = number of group members, m = number of chatbots.

messaging protocol does not introduce prohibitive overhead for users or chatbots, even on lower-end devices. A more detailed analysis and the full results of our experiment are presented in Appendix D.



(a) Sender Keys Protocol (b) MLS

Figure 4: Adding Chatbot with Sender Anonymity



(a) Sender Keys Protocol (b) MLS

Figure 5: Sending Messages with Sender Anonymity

7 Discussions

7.1 Integrating Other Privacy Mechanisms

Our modular design supports seamless integration with existing privacy solutions for group messaging. The user subtree component of CMRT is built on an abstraction of the CGKA scheme, allowing it to be replaced by other CGKA-based protocols, provided they satisfy the requirements for FS and PCS. For instance, integrating administrated CGKA [8] could additionally strengthen group administrative controls. Metadata-hiding techniques can also be employed to enhance user privacy against group outsiders. For instance, applying CGKA-based membership-hiding methods [34, 41], which encrypts and authenticates group metadata using the shared group key, can ensure membership anonymity. Similarly, applying MLS's Private Message [9], which encrypts control messages, can conceal sender identities.

7.2 Limitations of SnoopGuard

The effectiveness of SnoopGuard heavily relies on the accuracy of the trigger function in determining the relevance of messages. A poorly designed or maliciously crafted trigger function could mark all messages as relevant, granting a chatbot unrestricted access despite SnoopGuard's protections. Addressing this issue may be beyond the scope of the messaging protocol itself. Instead, a vetting process, either manual or automated, could be implemented to mitigate such abuses of trigger functions. Additionally, while we identify two key privacy properties against overprivileged chatbots, namely selective message access and sender anonymity, the list may not be exhaustive. As new risks emerge and additional privacy properties are proposed, SnoopGuard can serve as a foundation for future refinements.

7.3 Roadmap Toward a Privacy-Preserving Solution for Chatbots

Developing privacy-preserving chatbots for group messaging platforms is a complex challenge that involves both technical and user-centric considerations. While SnoopGuard provides a foundational framework by addressing two key privacy properties at the protocol level, there remain broader areas for exploration.

Enhancing Usability. One important direction is to improve the usability of privacy-preserving chatbots. This includes designing intuitive interfaces that inform users of the chatbot's presence, clearly communicate chatbot permissions, and enable seamless workflows for granting or restricting access. For example, Keybase displays chatbot permissions on installation pages and notifies users that a chatbot can read their messages by displaying an icon next to those messages [45]. However, to our knowledge, no user studies have been conducted to evaluate the effectiveness of design approaches for communicating chatbot permissions.

Designing Permission Model. Refining permission models for chatbots is another important area of research to enhance privacy protections. Previous research on Android permis-

sions has explored automated, context-aware permission decision support [37, 70]. Building on this approach, future work could focus on developing dynamic and adaptive permission frameworks designed for evolving group dynamics and user needs. These frameworks could integrate natural language processing (NLP) techniques to help users decide whether a chatbot can read certain messages or a message should be sent anonymously, thereby reducing users' cognitive load.

Understanding User Perceptions. Understanding users perceptions, acceptance, and trust of privacy-preserving chatbots is critical. Several studies have explored users' privacy concerns and perceptions of chatbots in one-on-one interactions [38, 39, 44], but group chat settings remain underexplored. Understanding how users perceive privacy features, what concerns they prioritize, and how they interact with these systems is essential to improving adoption and engagement.

8 Related Work

This work presents a secure group messaging protocol protecting users' privacy from chatbots. In the previous sections, we reviewed secure messaging protocols and messaging platforms supporting chatbots. This section further considers related work on chatbots' security and privacy issues and permission frameworks in smart devices.

8.1 Chatbot Security

Several studies have conducted large-scale security evaluations of chatbots on modern messaging platforms. Edu et al. [33] analyzed over 15,000 Discord chatbots and found that over 40% of the chatbots examined ask for permission to access message history, but less than 5% of them offer a privacy policy. Similarly, Chen et al. [20] analyzed design flaws in chatbot-like third-party apps on Business Collaboration Platforms (BCP), such as Slack. Their analysis showed that these apps can steal messages or impersonate users. The use of runtime policy checks and explicit user confirmation are suggested as countermeasures. These studies underscore the privacy risks associated with chatbots, providing strong motivation for our work.

Biswas [14] proposed methods for service providers to filter encrypted messages sent to chatbots using searchable encryption [15]. While their objective aligns with our goal of selective message access, their approach lacks robust E2EE properties like FS. Nonetheless, their work highlights the potential for achieving selective message access through service providers while protecting message confidentiality.

8.2 Permission Frameworks in Smart Devices

Mobile and web app permissions are well-studied, with findings showing that user-consent models often fail to inform users effectively about privacy risks. Chia et al. [21] conducted a large-scale study on Facebook apps, Chrome extensions, and Android apps. They revealed that many apps use misleading tactics to request excessive permissions. Similarly, Felt et al. [35, 36] found Android's permission system ineffective because developers often fail to follow the least privilege principle, leading to over-privileged apps, and users often ignore or misunderstand permission warnings, leading to uninformed decisions. These challenges highlight the importance of ensuring that the trigger function in our protocol is both developer-friendly and capable of clearly communicating permissions to users.

Smart speakers, like chatbots that constantly listen to messages, face similar challenges due to their always-on microphones. Manikonda et al. [53] found that users reported heightened privacy concerns after learning about the always-listening nature of smart speakers. Lau et al. [49] identified widespread user misunderstandings and limited use of privacy controls. Moreover, both Dubois et al. [32] and Schönherr et al. [57] demonstrated that smart speakers, when actively listening for activation words, can be unintentionally triggered, leading to unintended recordings. Chatbots in group chats, which may also operate in an always-on mode, could raise similar concerns when chatbots are mistakenly triggered.

9 Conclusion

This paper identifies two key privacy issues in group messaging protocols involving chatbots, and highlights the lack of existing platform that adequately addresses these issues while preserving robust E2EE. To address these challenges, we introduce selective message access and two forms of sender anonymity, and propose a secure group messaging protocol called SnoopGuard, which efficiently manages multiple keys among group members and chatbots while ensuring sender anonymity. The theoretical analysis confirms the possibility of achieving the claimed properties without significant overhead, and our implementation demonstrates both its efficiency and its seamless integration into existing secure messaging services. Finally, we present a roadmap toward a more comprehensive solution to this problem, aiming to raise awareness and inspire future research on this critical issue.

Acknowledgments

We thank the anonymous reviewers and shepherd for their valuable comments. We also thank Mahmood Sharif and Hsien-En Tzeng for their insightful feedback. This research was supported in part by the National Science and Technology Council of Taiwan under grants 112-2223-E-002-010-MY4 and 113-2634-F-002-001-MBK, and National Taiwan University under grant 114L7848.

Ethical Consideration

Our study is based on two publicly available conversation datasets, both of which were collected from public chat rooms and compliant with the policies of the respective messaging services. The datasets are fully anonymized, and we do not attempt to re-identify any member, except a chatbot as clearly discussed in Section 3. When consulting about the chatbot, one of the researchers used their authentic Discord account and asked in a public channel.

Telegram, Slack, and Discord are aware of the chatbot security issues to the extent that they already provide mitigations that do not work with E2EE. For platforms without official chatbot support, such as WhatsApp, disguising chatbots as users is against their terms of service. Therefore, we see no imminent need to inform them about the issues. We are in the process of informing LINE about the issues.

Open Science

The analysis scripts for the Pushshift Telegram dataset used in Section 3.2, the basic chatbot implementations for various messaging services discussed in Section 4.2, and the prototype implementation of SnoopGuard mentioned in Section 6.2.2 are available at https://github.com/csienslab/snoopguard-artifact and https://zenodo.org/records/14729613.

References

[1] Auntie meiyu, your trusted fact-checking confidant. https://checkcheck.me/en/. Accessed on 2024-02-07.

[2] Geekbench 6 - cross-platform benchmark. https://www.geekbench.com/. Accessed on 2025-01-02.

[3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–158. Springer, 2019.

[4] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the ietf mls standard for group messaging. In *Annual International Cryptology Conference*, pages 248–277. Springer, 2020.

[5] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1463–1483, 2021.

[6] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In *Theory of Cryptography*, pages 261–290. Springer, 2020.

[7] David Balbás, Daniel Collins, and Phillip Gajland. Analysis and improvements of the sender keys protocol for group messaging. In *XVII Reunión española sobre criptología y seguridad de la información. RECSI 2022*, volume 265, page 25. Ed. Universidad de Cantabria, 2022.

[8] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1253–1270, 2023.

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

[10] Jason Baumgartner, Savvas Zannettou, Megan Squire, and Jeremy Blackburn. The pushshift telegram dataset. In *Proceedings of the international AAAI conference on web and social media*, volume 14, pages 840–847, 2020.

[11] Jeff Beckman. 120+ chatbot statistics for 2024 (already mainstream). https://techreport.com/statistics/software-web/chatbot-statistics/. Accessed on 2024-09-01.

[12] 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.

[13] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the signal double ratchet algorithm. In *Annual International Cryptology Conference*, pages 784–813. Springer, 2022.

[14] Debmalya Biswas. Privacy preserving chatbot conversations. In *IEEE International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, 2020.

[15] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology - EUROCRYPT 2004*, pages 506–522. Springer, 2004.

[16] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84, 2004.

[17] BotoStore. Group butler. https://botostore.com/c/groupbutler_bot/. Accessed on 2024-02-01.

[18] BYBY.DEV. Top 8 llm-powered ai chatbots. https://byby.dev/ai-chatbots, Nov 2023. Accessed on 2024-02-07.

[19] Kaiming Chen and Jiageng Chen. Anonymous end to end encryption group messaging protocol based on asynchronous ratchet tree. In *Information and Communications Security*, pages 588–605. Springer, 2020.

[20] Yunang Chen, Yue Gao, Nick Ceccio, Rahul Chatterjee, Kassem Fawaz, and Earlence Fernandes. Experimental security analysis of the app model in business collaboration platforms. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2011–2028, 2022.

[21] Pern Hui Chia, Yusuke Yamamoto, and N Asokan. Is this app safe? a large scale study on application permissions and risk signals. In *Proceedings of the 21st international conference on World Wide Web*, pages 311–320, 2012.

[22] Cisco. go-mls. https://github.com/cisco/go-mls/. Accessed on 2024-02-08.

[23] codigoencasa. Chatbot library. https://github.com/codigoencasa/bot-whatsapp?tab=readme-ov-file. Accessed on 2024-02-07.

[24] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33:1914–1983, 2020.

[25] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016.

[26] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1802–1819, 2018.

[27] LY Corporation. Messaging api overview. https://developers.line.biz/en/docs/messaging-api/overview/. Accessed on 2024-01-29.

[28] David Curry. Messaging app revenue and usage statistics (2024). https://www.businessofapps.com/data/messaging-app-market/, Jan 2024. Accessed on 2024-02-07.

[29] Discord. Discord developer portal — documentation — application commands. https://discord.com/developers/docs/interactions/application-commands. Accessed on 2024-09-04.

[30] Discord. Message content intent review policy – developers. https://support-dev.discord.com/hc/en-us/articles/5324827539479-Message-Content-Intent-Review-Policy#h_01FJ9G3JV1H0HN25C4V7X3RHD4. Accessed on 2024-09-03.

[31] Dr.Web. Dr.web bot for telegram. https://free.drweb.com/drweb+telegram/. Accessed on 2024-02-07.

[32] Daniel J Dubois, Roman Kolcun, Anna Maria Mandalari, Muhammad Talha Paracha, David Choffnes, and Hamed Haddadi. When speakers are all ears: Characterizing misactivations of iot smart speakers. *Proceedings on Privacy Enhancing Technologies*, 2020.

[33] Jide Edu, Cliona Mulligan, Fabio Pierazzi, Jason Polakis, Guillermo Suarez-Tangil, and Jose Such. Exploring the security and privacy risks of chatbots in messaging services. In *Proceedings of the 22nd ACM internet measurement conference*, 2022.

[34] Keita Emura, Kaisei Kajita, Ryo Nojima, Kazuto Ogawa, and Go Ohtake. Membership privacy for asynchronous group messaging. In *International Conference on Information Security Applications*, pages 131–142. Springer, 2022.

[35] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 627–638, New York, NY, USA, 2011. Association for Computing Machinery.

[36] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, pages 1–14, 2012.

[37] Hongcan Gao, Chenkai Guo, Yanfeng Wu, Naipeng Dong, Xiaolei Hou, Sihan Xu, and Jing Xu. Autoper: Automatic recommender for runtime-permission in android applications. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 107–116. IEEE, 2019.

[38] Miriam Gieselmann and Kai Sassenberg. The more competent, the better? the effects of perceived competencies on disclosure towards conversational artificial intelligence. *Social Science Computer Review*, 41(6):2342–2363, 2023.

[39] Ece Gumusel. A literature review of user privacy concerns in conversational chatbots: A social informatics approach: An annual review of information science and technology (arist) paper. *Journal of the Association for Information Science and Technology*, 2024.

[40] Christoph G Günther. An identity-based key-exchange protocol. In *Advances in Cryptology — EUROCRYPT '89*, pages 29–37. Springer Berlin Heidelberg, 1990.

[41] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide metadata in mls-like secure group messaging: simple, modular, and post-quantum. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1399–1412, 2022.

[42] Telegram Messenger Inc. Mtproto mobile protocol. https://core.telegram.org/mtproto. Accessed on 2024-01-29.

[43] Telegram Messenger Inc. Telegram bot api. https://core.telegram.org/bots/api. Accessed on 2024-02-07.

[44] Carolin Ischen, Theo Araujo, Hilde Voorveld, Guda van Noort, and Edith Smit. Privacy concerns in chatbot interactions. In *Chatbot Research and Design: Third International Workshop, CONVERSATIONS 2019, Amsterdam, The Netherlands, November 19–20, 2019, Revised Selected Papers 3*, pages 34–48. Springer, 2020.

[45] Keybase. End-to-end encryption for things that matter. https://keybase.io/. Accessed on 2024-06-10.

[46] Keybase. Introducing bots on keybase. https://keybase.io/blog/bots. Accessed on 2024-06-10.

[47] Keybase. Keybase book: Keybase docs. https://book.keybase.io/docs/chat/. Accessed on 2024-09-03.

[48] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):60–96, 2004.

[49] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. Alexa, are you listening? privacy perceptions, concerns and privacy-seeking behaviors with smart speakers. *Proceedings of the ACM on human-computer interaction*, 2(CSCW):1–31, 2018.

[50] LINE Corporation. Technical whitepaper - line encryption overview. https://d.line-scdn.net/stf/linecorp/en/csr/line-encryption-whitepaper-ver2.1.pdf, Nov 2021. Accessed on 2024-02-07.

[51] LINE Corporation. Line transparency report. https://linecorp.com/en/security/encryption/2021h1, Jan 2022. Accessed on 2024-02-07.

[52] Pedro S. Lopez. whatsapp-web.js. https://github.com/pedroslopez/whatsapp-web.js. Accessed on 2024-02-08.

[53] Lydia Manikonda, Aditya Deotale, and Subbarao Kambhampati. What's up with privacy? user preferences and privacy concerns in intelligent personal assistants. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 229–235, 2018.

[54] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, Nov 2016. Accessed on 2024-02-07.

[55] 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. IEEE, 2018.

[56] Keegan Ryan, Thomas Pornin, and Shawn Fitzgerald. Protocol security review. https://keybase.io/docs-assets/blog/NCC_Group_Keybase_KB2018_Public_Report_2019-02-27_v1.3.pdf, 2019.

[57] Lea Schönherr, Maximilian Golla, Thorsten Eisenhofer, Jan Wiele, Dorothea Kolossa, and Thorsten Holz. Exploring accidental triggers of smart speakers. *Computer Speech & Language*, 73:101328, 2022.

[58] Signal. libsignal. https://github.com/signalapp/libsignal/tree/main/rust/protocol/src. Accessed on 2024-02-08.

[59] Signal. Private group messaging. https://signal.org/blog/private-groups/, 2014. Accessed on 2024-09-05.

[60] Slack. Permission scopes. https://api.slack.com/scopes. Accessed on 2024-01-29.

[61] Slack. Security at slack. https://a.slack-edge.com/964df/marketing/downloads/security/Security_White_Paper_2020.pdf. Accessed on 2024-01-30.

[62] Robin Staab, Mark Vero, Mislav Balunović, and Martin Vechev. Beyond memorization: Violating privacy via inference with large language models. *arXiv preprint arXiv:2310.07298*, 2023.

[63] Keerthana Muthu Subash, Lakshmi Prasanna Kumar, Sri Lakshmi Vadlamani, Preetha Chatterjee, and Olga

Baysal. Disco: A dataset of discord chat conversations for software engineering research. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 227–231, 2022.

[64] Telegram Messenger Inc. Telegram privacy policy. https://telegram.org/privacy, Apr 2023. Accessed on 2024-02-07.

[65] Telegram Messenger Inc. Telegram support: End-to-end encryption faq. https://tsf.telegram.org/manuals/e2ee-simple, 2023. Accessed on 2024-02-07.

[66] Yandex Translate. Telegram bot. https://yandex.com/support/translate-mobile/bot.html. Accessed on 2024-02-07.

[67] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. Sok: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, 2015.

[68] Rakuten Viber. Viber encryption overview. https://www.viber.com/app/uploads/viber-encryption-overview.pdf. Accessed on 2024-02-04.

[69] WhatsApp. Whatsapp encryption overview. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, Jan 2023. Accessed on 2024-02-07.

[70] Primal Wijesekera, Joel Reardon, Irwin Reyes, Lynn Tsai, Jung-Wei Chen, Nathan Good, David Wagner, Konstantin Beznosov, and Serge Egelman. Contextualizing privacy decisions for better prediction (and protection). In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

A Cryptographic Primitives

A.1 Pseudorandom Generators

Let $\mathcal{W}$ be the domain and range of the pseudorandom generator (PRG). A PRG is a function $\mathsf{PRG} : \mathcal{W} \to \mathcal{W}$ that, given an input U uniformly sampled from $\mathcal{W}$, produces an output $\mathsf{PRG}(U)$ that is indistinguishable from a uniformly random element $U' \in \mathcal{W}$. The security of a PRG is measured by the advantage for an attacker $\mathcal{A}$ has in distinguishing between $\mathsf{PRG}(U)$ and U', denoted as $\mathsf{Adv}^{\mathsf{PRG}}_{prg}(\mathcal{A})$.

Definition 2. *A* PRG *scheme is secure if for all efficient adversaries $\mathcal{A}$ and a security parameter λ, there is a negligible function* negl *such that* $\mathsf{Adv}^{\mathsf{PRG}}_{prg}(\mathcal{A}) \leq \mathsf{negl}(\lambda)$.

A.2 Public Key Encryption

A public-key encryption (PKE) scheme $\mathsf{PKE} = (\mathsf{PKEG}, \mathsf{PKEnc}, \mathsf{PKDec})$ consists of the following algorithms:

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\ \mathsf{PKEG}(s)$: Generates a PKE key pair from a secret s.
- $c \leftarrow\!\$\ \mathsf{PKEnc}(\mathsf{pk}, m)$: Encrypts the message m using the public key pk and outputs a ciphertext c.
- $m \leftarrow \mathsf{PKDec}(\mathsf{sk}, c)$: Decrypts the ciphertext c using sk and outputs the message m.

IND-CPA security. Consider the following game:

$\underline{\text{IND-CPA}^{\mathcal{A}}_{\mathsf{PKE}}(\lambda)}$
$b \leftarrow\!\$\ \{0,1\}$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\ \mathsf{PKEG}(1^\lambda)$
$(m_0, m_1) \leftarrow\!\$\ \mathcal{A}(1^\lambda, \mathsf{pk})$
$c \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
$b' \leftarrow\!\$\ \mathcal{A}(1^\lambda, c)$
return $1_{b=b'}$

Definition 3. *Let* $\mathsf{Adv}^{\mathsf{PKE}}_{\text{IND-CPA}}(\mathcal{A})$ *denote the advantage of adversary $\mathcal{A}$ winning the* IND-CPA *game, A* PKE *scheme is CPA − secure if for all efficient adversaries $\mathcal{A}$ and a security parameter λ, there is a negligible function* negl *such that*

$$\mathsf{Adv}^{\mathsf{PKE}}_{\text{IND-CPA}}(\mathcal{A}) \leq \mathsf{negl}(\lambda).$$

B Pseudoprocedure of SnoopGuard

This section presents the pseudoprocedure of our group messaging protocol in Figure 6. A user with state γ sends a message m by invoking $\mathsf{usr\text{–}send}(\gamma, m)$, which generates a control message T. Other users update their states by calling $\mathsf{proc}(\gamma, T)$, while only the triggered chatbots can successfully decrypts the message m by invoking $\mathsf{cbt\text{–}recv}(\gamma, T)$, where γ represents their respective states. Similarly, the chatbots send a message with cbt–send, which generates control message T to be processed by usr–recv. The operations set–pk and get–pk represent incorruptible operations for registering and retrieving metadata associated with a chatbot identified by CID from the service provider.

C Theoretical Performance Analysis of the Secure Group Messaging

This section extends Section 6.2.1 by presenting a more detailed analysis of the theoretical performance of our secure group messaging protocol.

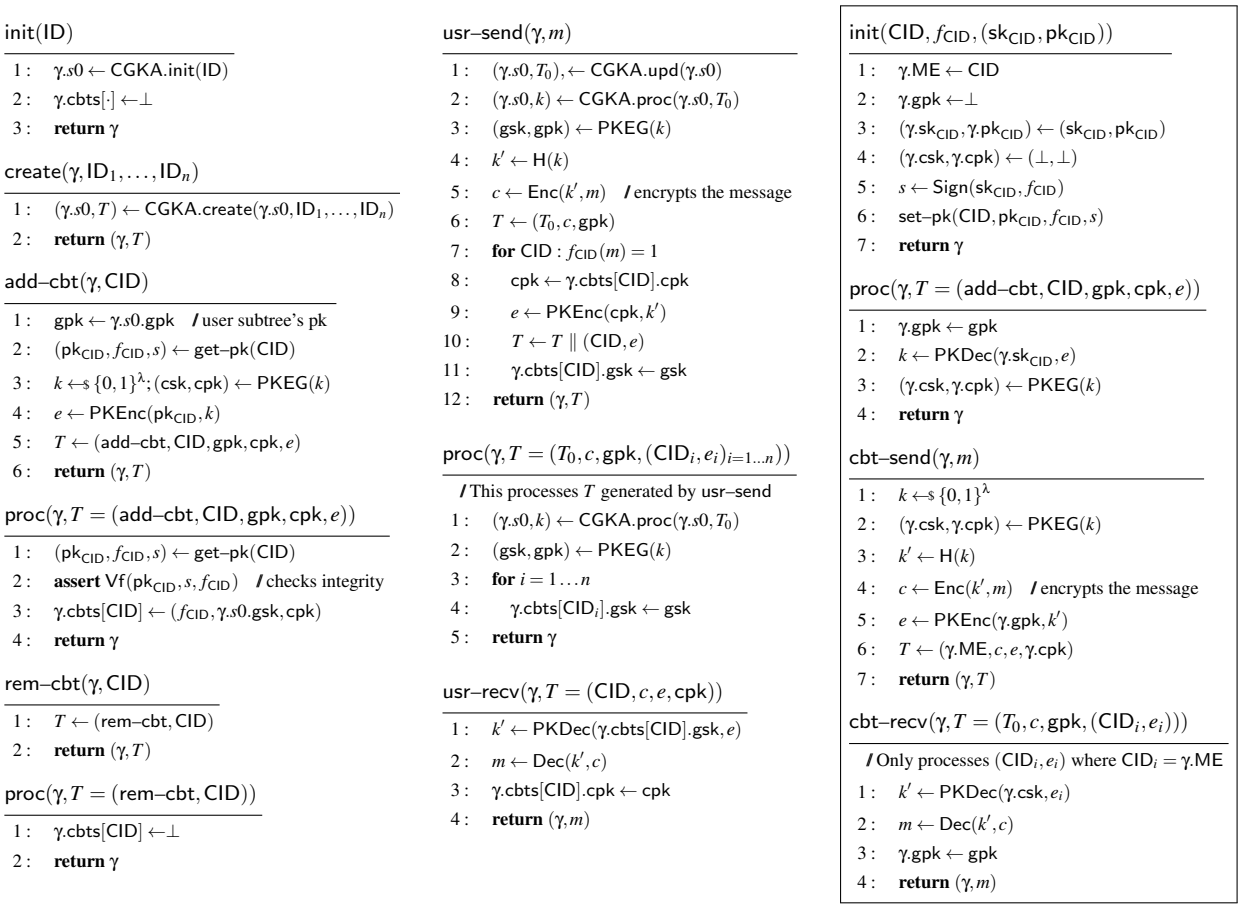


Figure 6: The SnoopGuard protocol. Unboxed algorithms are used by users while boxed algorithms are used by chatbots.

Baseline. For setup phase, the group initiator uses $O(n+m)$ PKE operations and $O(n+m)$ symmetric operations. The construction of the TreeKEM with n members involves $O(n)$ PKE operations and hash operations, respectively. To compute the shared secret for each chatbot, the initiator also performs $O(m)$ PKE operations and hash operations, respectively. For the receivers, each user requires $O(1)$ PKE operations and $O(\log n)$ hash operations to initiate the TreeKEM. For each chatbot, it takes $O(1)$ PKE operations to decrypt the secret, but only $O(1)$ hash operations to compute the shared secret due to the unbalanced tree structure.

Suppose a user sends a message to the chatbots. The message sender performs $O(\log n+m)$ PKE operations and symmetric operations, respectively. Updating the TreeKEM involves $O(\log n)$ public key generations and hash operations. Each chatbot takes $O(1)$ PKE operations and hash operations for the sender to do the key update and message encryption, respectively, and there are m chatbots, imposing $O(m)$ overhead. Message recipients, including users and chatbots, perform identical actions as in the setup phase to update the secret, resulting in the same overhead.

Adding a chatbot to the group is almost the same as sending a message to a chatbot. The initiator performs $O(\log n)$ PKE operations and symmetric operations, respectively, to update the TreeKEM. Both the chatbot and other members perform the same actions as in the setup phase to update the secret.

Pseudonymity. For the setup phase, registering a pseudonym is equivalent to sending a message to the chatbot. For the ongoing phase, using pseudonyms requires both sender and receiver $O(1)$ additional PKE operations to create and verify the signature. This does not affect the overall complexity.

Trigger Concealment. The sender fakes key updates using random bytes and sends them to chatbots that should not receive the message. This is equivalent to triggering all chatbots, and therefore maintaining the same complexity for the ongoing phase.

Storage Overhead. The storage overhead for each user is $O(n+m)$, which includes $O(n)$ keys for the TreeKEM and $O(m)$ keys for all the chatbots. This is equivalent to the MLS with $n+m$ members. However, each chatbot only needs to store $O(1)$ public key for users' subtree. This represents an advantage compared to MLS, which requires $O(m+n)$ storage for each chatbot.

D Experimental Performance Evaluation of the Secure Group Messaging

This section extends Section 6.2.2 and presents the full results of our experiments, as well as a more detailed analysis of the results.

Adding a Chatbot. To demonstrate that adding a chatbot has an acceptable overhead, we measure the time from the initiation of the invitation to the chatbot until all members and chatbots complete the necessary key exchanges. In the case of a pseudonymous chatbot, all users register their pseudonyms.

Figure 4 illustrates the time spent adding a chatbot to a group with varying group sizes, based on different levels of sender anonymity. For comparison, we also include the traditional scenario where the chatbot is treated as a user. For the Signal Protocol (Sender Keys Protocol), adding a chatbot to an IGA-secure group takes significantly less time because there is no need to distribute sender keys. For the MLS protocol, the original protocol takes slightly more time due to the larger group size, as chatbots are counted as members and therefore cause more overhead when adding members.

Sending a Message. The timer starts when the sender's client takes the plaintext and stops when the last receiver decrypts and outputs the message content. For each experiment, we maintain a fixed group size of 50 members and measure the time it takes for all group members and chatbots to receive the message. We assume that all chatbots will receive the message regardless of the content, showing the worst-case scenario. Our experiments focus on the efficiency of our protocols with respect to the number of chatbots. We did not consider the effect of group size because we use the underlying protocol for group members, and the efficiency of the Sender Keys Protocol and MLS are not relevant in our experiments.

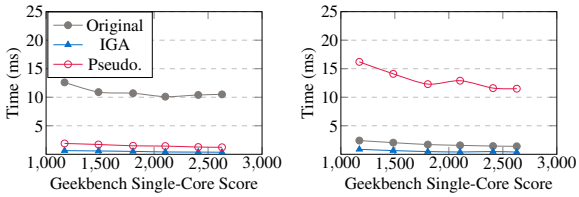
Figure 5 shows the time required to send a message to group members and chatbots with varying levels of sender anonymity, as well as whether to conceal triggers from the service provider. We also include the traditional scenario for comparison. Consistent with the theoretical analysis, our protocol introduces overhead linear to the number of chatbots. Pseudonymity results in slightly higher overhead, possibly due to the additional signature processes.

Performance on Resource-Constrained Devices. Most users access messaging services on mobile devices. To demonstrate that our protocol performs well under mobile-like constraints, we simulate its behavior in environments with limited CPU resources. Specifically, we run our protocol in Docker containers with CPU allocations ranging from 0.5 to 1.0 CPUs. Each container was benchmarked using Geekbench 6 [2], a widely-used tool for assessing mobile device performance, to understand how the allocated resources perform. After establishing the performance baseline, we simulate the protocol in the constrained environment. For the chatbot addition experiment, we measure the time required for the device to generate all the necessary information, complete the key exchange, and register a pseudonym, if applicable. For the message sending scenario, we compute the time required for the device to generate all ciphertexts of the messages and, if necessary, the required signatures. Unlike previous experiments that evaluate overall system performance, this setup isolates the computation performed on a single device to evaluate the protocol's suitability for resource-constrained environments.

The CPU constraints used in our experiments resulted in Geekbench 6 single-core scores ranging from 1,171 to 2,629. These scores represent a spectrum from budget smartphones (e.g., Qualcomm Snapdragon 865, MediaTek Dimensity 8100) to high-end devices (e.g., Apple M2, Apple A16) as of 2024.

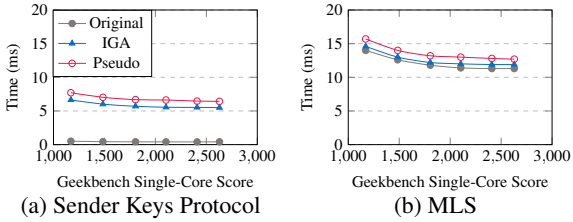
Figure 7 shows the results of the chatbot addition experiment for a group with 50 members and 30 chatbots. Without pseudonymity, the overhead remains below 15 ms for all CPU configurations. With pseudonymity enabled, the overhead increases slightly due to pseudonym generation, but remains below 20 ms even on the most constrained CPU setting. Figure 8 shows the results of the message sending experiment for the same group configuration. Using the Signal protocol, the time to generate ciphertexts remains below 5 ms, even with the lowest CPU allocation. While MLS introduces more overhead, it remains below 10 ms for all configurations.

These results show that SnoopGuard performs efficiently across devices, from low-end to high-end, with minimal latency even under significant CPU constraints, making it suitable for diverse mobile devices.



(a) Sender Keys Protocol (b) MLS

Figure 7: Adding chatbot with sender anonymity under various CPU constraints



(a) Sender Keys Protocol (b) MLS

Figure 8: Sending messages with sender anonymity under various CPU constraints