Practical Attacks on a Decentralized Secure Messenger Session

Kota Urushigaki
The University of Osaka

Hayato Kimura
NICT & The University of Osaka

Atsushi Tanaka
The University of Osaka

Takanori Isobe
The University of Osaka

Abstract

Session is a widely deployed decentralized messenger application that emphasizes user anonymity and privacy through end-to-end encryption. Session currently employs its own uniquely designed messaging protocol, Session Protocol V1, having migrated from the extensively studied Signal Protocol. In this paper, we conduct a comprehensive, implementation-driven security analysis of the Session Protocol V1, focusing on its 1-to-1 and closed-group communication mechanisms. Our analysis reveals two fundamental design vulnerabilities: the absence of mutual public key authentication and the lack of cryptographic bindings to monotonic sequence counters. Exploiting these weaknesses within the context of actual application environments, we demonstrate three practical attacks: an impersonation attack, a message timestamp forgery attack, and message dropping and replay attacks. These attacks allow malicious server nodes or unprivileged malicious insiders to substitute public keys, silently suppress or duplicate messages, and manipulate the perceived chronological order of conversations. The findings highlight that these exploits severely undermine the fundamental security guarantees of the messenger. Finally, we propose immediate, actionable mitigation strategies to address the identified flaws and secure the protocol against these threats.

1 Introduction

Session [24] is a Swiss decentralized messaging application that provides E2E communication. It is developed and operated by Session Technology Foundation and is designed with a strong emphasis on anonymity and privacy; accordingly, it is often positioned as a tool for privacy-sensitive users such as activists and journalists [15]. As of November 2025, Session is available on major platforms, including iOS, Android, Windows, and macOS, and has been downloaded more than one million times on Google Play alone.[1] This deployment scale makes the security of its messaging protocol important not only from a theoretical perspective but also from a practical one.

Session originated as a fork of the Signal Protocol [31], one of the most widely deployed and extensively studied E2E protocols, and a subject of substantial cryptographic analysis [1,3–5]. Since late 2020, however, Session has transitioned to its own custom protocol design. In this paper, we refer to the currently deployed version of this design as Session Protocol V1 [9,10]. According to Session Technology Foundation, this transition was motivated by the mismatch between Signal's centralized design assumptions and Session's decentralized server architecture [11].

However, custom secure messaging protocols require careful scrutiny, as subtle design and implementation choices can introduce practical vulnerabilities that are not apparent from high-level security goals alone. This concern is particularly important in decentralized settings, where mechanisms originally designed for centralized environments may no longer provide their intended guarantees. Motivated by this concern, we conduct a security analysis of Session Protocol V1, focusing on its E2E messaging mechanisms for 1-to-1 and closed-group chats.

1.1 Our Contributions

In this paper, we present a security analysis of Session Protocol V1, the currently deployed E2E messaging protocol designed by Session Technology Foundation. Our analysis reveals three practical attacks stemming from two core design weaknesses, as summarized in Table 1. These attacks are mounted by a malicious server node, a malicious insider, or a malicious group administrator, depending on the setting. We further demonstrate their feasibility through concrete real-world scenarios. Our study is implementation-driven and is based on the publicly available iOS client-side implementation [27], Session cryptographic wrapper library [20], and the server-side implementation [28]. The attacks identified in this work are as follows.

- **Impersonation Attack (Section 4):** In the 1-to-1 chat

[1] https://play.google.com/store/search?q=Session&c=apps

Table 1: Summary of our attacks against Session. MSN: Malicious Server Node. MI: Malicious Insider. MGA: Malicious Group Administrator. CG: Closed Group.

Attack	Section	Model	Target Setting	Root Cause
Impersonation Attack	4	MSN	1-to-1	Vuln.1 (Lack of Mutual PK Authentication)
Message Timestamp Forgery Attack	5	MI	1-to-1, CG	Vuln.2 (Lack of CTR-bound Symmetric Key)
Message Dropping and Replay Attack	6	MSN/MGA	1-to-1, CG	Vuln.2 (Lack of CTR-bound Symmetric Key)

public-key exchange protocol, Session lacks a mechanism for authenticating the sender's public key. Consequently, a malicious server node can replace the legitimate sender's public key with attacker-controlled key material and impersonate the sender.

- **Message Timestamp Forgery Attack (Section 5):** The Session E2E protocol does not associate messages with monotonic sequence counters; instead, message ordering depends entirely on sender-generated timestamps. As a result, in both 1-to-1 and closed-group chats, a malicious insider can manipulate timestamps to make messages appear earlier or later than they were actually sent.

- **Message Dropping and Replay Attack (Section 6):** The lack of per-message sequence counters allows a malicious server node to selectively drop messages without detection by conversation participants. Furthermore, our analysis of the server-side and client-side implementations reveals that the adversary can execute replay attacks that are effectively undetectable by participants.

These attacks undermine core security properties that users would reasonably expect from a secure messenger. In particular, the impersonation attack can create a false trust relationship between users and thereby enable information theft or follow-on social engineering. The message timestamp forgery attack is especially concerning because it requires no elevated privilege: any legitimate participant can manipulate the apparent chronology of a conversation and thereby alter the semantic meaning of an exchange, such as the interpretation of consent or agreement, without triggering any cryptographic alarm. The message dropping and replay attacks, although they do not directly violate confidentiality, can silently desynchronize participants' views, distort social interpretation, and destabilize coordination in group settings.

To the best of our knowledge, these issues are not documented in the official whitepaper [12] or publicly available sources [24], and they were uncovered through our analysis of the open-source implementation [20, 27, 28]. While some of these issues require protocol-level redesign, we also discuss several immediate mitigation strategies in Section 7. To support reproducibility, the source code is available online[2].

[2] https://github.com/kota5140/session-woot

1.2 Related Work

Löbbecke et al. [17] analyzed the network architecture of Session, focusing on its integration of Onion Requests [14] and Swarms. While Session originated as a fork of the Signal codebase, the authors highlight the difference from the standard Signal Protocol to achieve metadata-free messaging through a decentralized service node network.

In 2021, Quarkslab conducted a third-party security audit of Session's implementation [18]. However, because Session migrated to its current protocol near the end of the audit period, coverage of the deployed design was limited.

Jaeger et al. [8] formally analyzed the "legacy" closed-group E2E protocol in Session and identified several practically exploitable attacks: an insider replay attack where a malicious group member can replay messages to impersonate honest members, a replay attack by external adversaries exploiting unauthenticated timestamps, and a message forgery attack by outsiders with access to an exposed signing key. Specifically, the outsider replay attack exploits the lack of the sender's signature on the timestamp, allowing an external attacker to modify the timestamp and forward past ciphertexts as new messages. At the time of our investigation, the system had transitioned to the new group chat protocol, GroupsV2, where the developers stated this vulnerability was resolved.

In December 2025, while this paper was being prepared, Session Technology Foundation announced a new design for Session Protocol V2 [25], with planned features including perfect forward secrecy (PFS) and post-quantum cryptography (PQC). However, V2 remained at the conceptual stage, and V1 was still the only deployed protocol.

Overall, prior work has examined Session's network layer and parts of its E2E design, but no prior work provides a comprehensive, implementation-driven security analysis of the deployed Session Protocol V1. We address this gap through source-code inspection and practical validation, uncovering previously undocumented vulnerabilities in the current protocol.

1.3 Ethical Considerations

We evaluated the official Session protocol specification [12], its underlying cryptographic library [20], and the source code for both client and server-side implementations [27, 28]. Our study was restricted to passive observation within the

Table 2: Detailed Timeline of Responsible Disclosure

December 2, 2025	Our first disclosure regarding the impersonation attack
January 23, 2026	Second attempt prior to a domestic conference
March 6, 2026	Third attempt addressing all three attacks
March 9, 2026	First response via an alternative email address for Session Technology Foundation
April 2, 2026	Another message indicating that several weeks would be required to respond to our analysis

scope of standard client operations, and we explicitly abstained from active attacks on the mainnet environment. Although a stagenet was available, its functionality was limited to blockchain Proof-of-Stake (PoS) mechanisms rather than messaging services [26]. To the best of our knowledge, our methodology complied with all applicable terms of service and testing policies.

In accordance with responsible disclosure practices, we initiated contact with Session Technology Foundation in December 2025. As detailed in Table 2, we conducted multiple disclosure attempts through March 2026, ultimately establishing contact via an alternative address following two unsuccessful attempts. While Session Technology Foundation has acknowledged receipt of our report, a substantive response to our analysis remains pending as of this writing. We remain committed to coordinating with the developers to facilitate the necessary remediation.

2 Session Architecture and Protocols

In this section, we describe the design of Session Protocol V1 architecture as the primary object for our security analysis. The functional core of Session is the E2E protocol, which employs the long-term cryptographic secrets of the users to sign and encrypt messages. The Session E2E protocol supports the core of Session's messaging security and consists of two major chat category, 1-to-1 and closed-group chats. Session provides another type of chat referred to as open group, where users can join public groups anonymously, but we left it out of consideration as it does not adopt E2E communication model. As follows, we elaborate on the 1-to-1 and closed-group chats with respect to their detailed cryptgraphic algorithms as well as the entire Session architecture.

2.1 Cryptographic Primitives and Notation

Session provides `libsession-util`, an open-source cryptographic library only designed for Session Protocol [20]. This library has an crypto_box abstraction provided by Networking and Cryptography Library(NaCl) [2], employing a Curve25519 DiffieHellman key agreement and XSalsa20-Poly1305 as Authenticated Encryption with Additional Data (AEAD) scheme. Let $(idk^{\mathcal{A}}_{priv}, idk^{\mathcal{A}}_{pub})$ denote the long-term signing key pair of user $\mathcal{A}$, while we denote by $(DHIDK^{\mathcal{A}}_{priv}, DHIDK^{\mathcal{A}}_{pub})$ the long-term key pair of user $\mathcal{A}$ for DiffieHellman key agreement. During the account creation process on Session, a user generates a random seed and derives these key pairs deterministically from the seed.

2.2 1-to-1 E2E Protocol

We consider the case where Alice initiates a 1-to-1 chat with a new contact, Bob. The E2E messaging process for 1-to-1 chat is composed of several phases.

1. **Initial Setting (Sect.2.2.1):** Alice obtains Bob's long-term key for key exchange via out-of-band (OOB) channel.
2. **Key Exchange (Sect.2.2.2):** Alice derives a symmetric key from her ephemeral private key and Bob's long-term public key.
3. **Encryption/Wrapping (Sect.2.2.3):** Alice encrypts the messages, appends metadata such as outer timestamp, and signs the resulting concatenated data.
4. **Message Routing (Sect.2.2.4):** Alice routes messages to a randomly chosen service node via the network, which validates and stores them. Bob subsequently polls a server node for undelivered messages and retrieves them.
5. **Unwrapping/Decryption (Sect.2.2.5):** Bob unwraps and decrypts ciphertexts polled from server nodes and displays the messages in order on the client.

2.2.1 Initial Setting

To derive a symmetric key with Bob, Alice needs Bob's long-term public key. Alice retrieves Bob's public key through the invitation. Session offers two invitation methods: manual entry of the account ID and scanning of a QR code.

Account ID Entry: Alice obtains the account ID of Bob through OOB channel(such as other messaging applications), and she directly enters the ID into the form on the client.

QR Code Scanning: Alice obtains Bob's long-term public key by scanning the QR code displayed on his device, which encodes the key.

In summary, in both methods, Bob provides his $DHIDK^{Bob}_{pub}$ to Alice via the application, allowing Alice to recognize the key and open a 1-to-1 chat to send new messages to Bob.

This process is performed directly between clients, not via server nodes. At this stage, however, only Alice has obtained Bob's public key, and unless it is communicated through some direct means, Bob has no information about Alice's public key $DHIDK_{pub}^{Alice}$.

2.2.2 Key Exchange

After obtaining Bob's long-term public key, Alice generates a new ephemeral key pair for each message as shown in the following equations, and derives a symmetric key K and *nonce* from her private key and Bob's public key.

$$eph_{priv}^{Alice},\ eph_{pub}^{Alice} \leftarrow \mathsf{X25519.KeyGen}()$$

$$K \leftarrow \mathsf{X25519.ECDH}(eph_{priv}^{Alice},\ DHIDK_{pub}^{Bob})$$

$$nonce \leftarrow \mathsf{BLAKE2}(eph_{pub}^{Alice},\ DHIDK_{pub}^{Bob})$$

2.2.3 Encryption/Wrapping

Alice first creates a signed message M by signing the message m with her signining private key idk_{priv}^{Alice}. Since 1-to-1 (as well as closed-group) protocol employs the long-term key to sign the message, Session does not provide the cryptographic deniability. The following equations describe the procedure for generating the signature.

$$m' \leftarrow m \mid\mid idk_{pub}^{Alice} \mid\mid DHIDK_{pub}^{Bob}$$

$$Sig_{Alice} \leftarrow \mathsf{Ed25519.Sign}(idk_{priv}^{Alice},\ m')$$

$$M \leftarrow m \mid\mid idk_{pub}^{Alice} \mid\mid Sig_{Alice}$$

Next, Alice encrypts the signed message M using the symmetric key K and a nonce, concatenates the ciphertext with her ephemeral public key eph_{pub}^{Alice}, and treats the result as the payload. She then appends a timestamp to this payload and serializes it. This timestamp is generated based on the client device's local Unix time. Subsequently, Alice signs the serialized data along with *metadata* such as $(\texttt{"store"} \mid\mid namespace \mid\mid timestamp_{outer})$ using her long-term identity private key idk_{priv}^{Alice}, and transmits the signed package with the signing key to the network. Note that inner timestamp attached to the payload is used by the receiving client to determine message ordering, whereas the outer timestamp allows server nodes to identify expired messages and mitigate replay attacks reported by Jaeger et al. [8].

$$C \leftarrow \mathsf{Salsa20\text{-}Poly1305.Enc}(K,\ nonce,\ M)$$

$$payload \leftarrow eph_{pub}^{Alice} \mid\mid C$$

$$body \leftarrow (payload,\ timestamp_{inner},\ \ldots)$$

$$Sig'_{Alice} \leftarrow \mathsf{Ed25519.Sign}(idk_{priv}^{Alice},\ body,\ metadata)$$

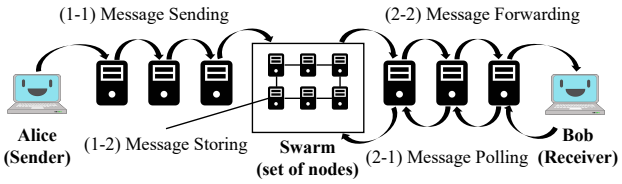


Figure 1: Message Routing on the Session Network

2.2.4 Message Routing

We first describe the architectural components of the Session's network layer (hereafter referred to as the Session Network), followed by a detailed message routing procedure. This procedure is partitioned into two primary phases: (1) message sending and storing and (2) message polling and forwarding. The overall architecture of the Session Network is illustrated in Figure 1. We verified the message handling logic by analyzing both the iOS client-side [27] and the server-side implementations [21, 28].

Onion Routing and Swarm: The Session Network is fundamentally constructed from the Onion Routing Protocol reffered to as Onion Requests (which is undergoing a transition to Lokinet[3]) and *Swarm* [12, 17]. The Onion Routing Protocol functions as an anonymous routing infrastructure comprised of a chain of three relay server nodes between a client and a Swarm. A Swarm constitutes a set of parallel server nodes deterministically mapped from either a recipient's long-term public key (in the 1-to-1 protocol) or a group public key (in the closed-group protocol) [21]. Specifically, the public key is projected into a 64-bit integer Swarm space, and the nearest Swarm is selected on a circular ring over $[0, 2^{64})$ [20]. Since cryptographic public keys are uniformly distributed over their domain, the probability that any particular Swarm is responsible for a given user is approximately $1/n$, where n is the total number of Swarms. Given a specific public key, the operator of a malicious server node might attempt to strategically position their node within the corresponding Swarm. To prevent such targeted attacks, the assignment of a new server node to its initial Swarm is governed by a separate deterministic algorithm [17, 21].

In the following, we describe how a message from Alice is stored on a server node and subsequently retrieved by Bob through the Session network layer as introduced above.

(1) **Message Sending/Storing:** Firstly, Alice selects an entry server node within the Swarm mapped from Bob's public key $DHIDK_{pub}^{Bob}$ and routes the message using the Onion Routing Protocol. The entry server node authenticates the message by verifying the signature and the validity of the metadata (incl. timestamp window)

[3] Low-Latency Anonymous Routing Protocol (LLARP) service operating at the network layer

against the Alice's public key idk_{pub}^{Alice}. Upon successful verification, the server node facilitates intra-Swarm propagation by broadcasting the message to the peer server nodes. Then, the server node computes the hash value of the message to check for duplicates. Given that each message includes a unique timestamp and other data, hash collisions for identical message are highly unlikely. If no duplicate is found, the message is persisted in the database as a new entry alongside its hash value. Similarly, each server node in the Swarm receiving the propagated message independently performs signature verification, computes the message hash to perform deduplication and stores the message upon successful verification, provided it is not a duplicate. Once these operations are completed, the entry server node returns a response back to Alice.

$$\mathsf{Ed25519.Verify}(idk_{pub}^{Alice}, Sig'_{Alice})$$

$$message_Hash \leftarrow \mathsf{BLAKE2}(body)$$

(2) **Message Polling/Forwarding:** When the receiver Bob attempts to retrieve messages while online, Bob first sends a message retrieval request to a randomly chosen server node in his Swarm using the Onion Routing Protocol. This request includes Bob's public key idk_{pub}^{Bob} and signed metadata (`"retrieve" ||` $timestamp$) which defends against replay attack to scrape stored messages for further analysis. Additionally, the request also includes the hash of Bob's last received message from that server node, enabling the server node to synchronize and transmit only new messages to reduce overhead. Then the server node verifies the signature; if successful, it proceeds to forward the undelivered message to Bob.

2.2.5 Unwrapping/Decryption

During message retrieval, Bob obtains a server-computed message hash and employs it as a primary key to scan his local database to avoid duplicate message entries. The message is accepted and persisted only if its hash does not collide with any previously stored entries. Bob then unwraps the retrieved message and separates the payload from its metadata such as inner timestamp. Next, Bob derives the symmetric key K and $nonce$ from his private key $DHIDK_{priv}^{Bob}$ and Alice's ephemeral public key eph_{pub}^{Alice}, and decrypts the ciphertext. Finally, Bob verifies the decrypted signed message M using Alice's signing public key idk_{pub}^{Alice}. The following equations describe the procedure of decryption and signature verification.

$$payload,\ timestamp_{inner},\ \cdots \leftarrow body$$

$$eph_{pub}^{Alice}\ ||\ C \leftarrow payload$$

$$nonce \leftarrow \mathsf{BLAKE2}(eph_{pub}^{Alice},\ DHIDK_{pub}^{Bob})$$

$$K \leftarrow \mathsf{X25519.ECDH}(eph_{pub}^{Alice},\ DHIDK_{priv}^{Bob})$$

$$M \leftarrow \mathsf{Salsa20\text{-}Poly1305.Dec}(K,\ nonce,\ C)$$

$$m\ ||\ idk_{pub}^{Alice}\ ||\ Sig_{Alice} \leftarrow M$$

$$\mathsf{Ed25519.Verify}(idk_{pub}^{Alice},\ m',\ Sig_{Alice})$$

The chronological ordering of messages on Bob's device is strictly governed by the signed timestamp provided by Alice. This ordering design is likely intended to prevent out-of-order delivery, where a later message is displayed before earlier ones. Unlike centralized architectures that rely on a single trusted server to provide a global reference clock, Session's decentralized nature makes establishing a total ordering of messages inherently challenging, as it lacks a unified source of absolute time.

2.3 Closed Group E2E Protocol

The closed-group E2E protocol in Session inherits its foundational cryptographic primitives from the 1-to-1 communication protocol. Since the detailed specification of the encryption and decryption processes do not fundamentally affect our analysis of message ordering integrity, we omit the granular steps previously discussed in the 1-to-1 setting (Section 2.2). However, to maintain the security properties unique to closed group messaging, it introduces an additional phase for group configuration and state management. In this section, we summarize the key features of this design and describe the components essential to our analysis.

2.3.1 Group Configuration

In Session, a closed group is created and managed by a privileged entity refered to as an administrator. To initialize a group, an administrator first generates a dedicated long-term identity key pair on the client side. The long-term public key is then deterministically mapped to single Swarm on the Session Network. Server nodes within the Swarm mapped to the group public key are responsible for storing and relaying encrypted group information and messages. Next, the administrator derives a shared key and nonce via a Key Derivation Function (KDF) and persists them, along with a signed group configuration, on the corresponding Swarm. This key is then distributed to each member through 1-to-1 chats. Finally, members use signed OOB invitations to retrieve the configuration and decrypt the group state.

$$ipk_{\text{priv}}^{\text{group}},\ ipk_{\text{pub}}^{\text{group}} \leftarrow \mathsf{Ed25519.KeyGen}()$$

$$DHIDK_{\text{priv}}^{\text{group}},\ DHIDK_{\text{pub}}^{\text{group}} \leftarrow \mathsf{X25519.KeyConvert}()$$

2.3.2 Wrapping/Unwapping

In this section, we describe the specifications for message wrapping, unwrapping, and the subsequent display logic on the recipient's device. The wrapping process in the closed-group E2E protocol is essentially identical to that of 1-to-1 communication. Consider a scenario where Alice, a group member, sends a message to Bob within the same group.

Alice encrypts the message using the group shared key obtained during the group configuration phase (Section 2.3.1) and a fresh nonce generated on her client. Specifically, Alice signs the message with her long-term identity private key idk_{priv}^{Alice} and appends the Unix timestamp generated by her client. The timestamped message is then encrypted using a newly generated nonce and the group shared key, followed by serialization. Note that timestamp is appended before the encryption in closed-group protocol. Finally, Alice attaches metadata, such as $(\texttt{"store"} \mid\mid \textit{namespace} \mid\mid \textit{timestamp})$, signs the entire package with idk_{priv}^{Alice}, and transmits it to a server node in the Swarm corresponding to the group public key.

The message polling procedure for closed groups follows the same flow as 1-to-1 chats. Bob randomly selects a server node within the Swarm and transmits the hash of the last received message. The server node forwards all new messages arrived after that hash, along with their respective hash values. Bob's client then performs deduplication, relying on these hashes to filter out any previously processed messages. Subsequently, Bob deserializes the payload and decrypts it using the group's shared key. Upon extracting the inner timestamp and verifying the message signature, the recipient's client displays the content according to the chronological order of the extracted timestamp. As with the 1-to-1 protocol, the message ordering on the recipient's side depends entirely on the timestamps generated by the sender's client.

3 Attack Settings

In this section, we describe the threat models and security goals considered in our analysis of Session.

3.1 Threat Models

In the Session E2E protocol, neither client nodes nor server nodes are assumed to be fully trusted. We consider three primary threat models: (1) a malicious insider who can send arbitrary messages to recipients in 1-to-1 or closed-group chats; (2) a malicious group administrator who can create a closed group and subsequently modify its configuration; and (3) a malicious server node operating within the Session network. We formally define these adversaries and their capabilities as follows.

Definition 1. (Malicious Insider, MI) A Malicious Insider (MI) is a legitimate participant in a 1-to-1 or closed-group conversation who can arbitrarily modify messages sent from their client and retain received messages locally beyond the sender's control. The requirements for assuming this role are minimal: the adversary only needs a valid Session account and participation in the target conversation.

Definition 2. (Malicious Group Administrator, MGA) A Malicious Group Administrator (MGA) is a malicious insider with administrative privileges in a closed group. In addition to normal participant capabilities, an MGA can create a group, modify its configuration, and abuse administrator privileges while remaining indistinguishable from a legitimate administrator to other group members. An MGA may also collude with a malicious server node, or operate one directly, to mount compound attacks such as selective message dropping or replay attacks.

Definition 3. (Malicious Server Node, MSN) A Malicious Server Node (MSN) is a server in the Session network that appears honest from the users' perspective but deviates from the prescribed protocol behavior. An MSN can manipulate, suppress, replay, or selectively forward protocol messages, and may collude with a Malicious Group Administrator (MGA) to target specific messages or group members.

3.2 Security Goals

As a security goal of the protocol, the official whitepaper [12] defines data integrity as follows.

- **Integrity:** Messages are received intact and unmodified, and any modification can be detected.

Furthermore, as fundamental security goals that Session should satisfy, we adopt the notions of confidentiality and authentication defined by Unger et al. [35] for general messaging applications.

- **Confidentiality:** Only the intended recipients are able to read a message. Specifically, the message must not be readable by a server operator that is not a conversation participant.
- **Authentication:** All participants in a conversation receive proof from all other participants whom they believe to be in the conversation that those participants possess the known long-term secret keys. Furthermore, each participant can verify that a message was sent by the claimed sender.

In addition, we consider the following security properties, which the Session E2E protocol is expected to satisfy.

- **Public Key Authenticity:** During the public-key exchange for 1-to-1 chats, each user should be able to verify the authenticity of the other party's public key. If

a malicious server node (MSN) attempts to replace a user's legitimate public key with one controlled by the attacker, the protocol should detect the attack and abort the procedure. Otherwise, an unauthenticated adversary may bypass the intended authentication guarantees and gain unauthorized access to the conversation.

- **Message Ordering:** The protocol should provide a cryptographically protected notion of message order, so that recipients can detect message reordering, message dropping, replay, or other anomalies in the flow of communication. In particular, the displayed order of messages should not depend solely on sender-controlled metadata such as timestamps.

If public-key authenticity is not satisfied, an unauthorized and unauthenticated adversary may be able to participate in a 1-to-1 chat, thereby violating the broader authentication guarantee that Session is expected to provide. Similarly, if message ordering is not protected, a malicious insider can manipulate timestamps to cause messages to be displayed in an incorrect order on the recipient's device. Such manipulation undermines the authenticity and integrity of the temporal context of the communication.

4 Impersonation Attack

In this section, we present an impersonation attack against 1-to-1 chats. We describe an attack in which a malicious server node (MSN) (Definition 3) on the communication path impersonates an arbitrary sender. We then identify the underlying cryptographic vulnerability that enables this attack and discuss its feasibility and impact.

4.1 Attack Overview

We begin by defining the underlying vulnerability that enables the attack.

Vulnerability 1. (Lack of Mutual Public Key Authentication) In 1-to-1 chats, Session does not provide an effective mechanism for the message recipient to verify the authenticity of the sender's public key. As a result, the recipient cannot distinguish the legitimate sender's public key from attacker-substituted key material.

This attack exploits the above vulnerability, namely the lack of mutual public-key authentication. We consider a scenario in which a malicious server node (MSN) impersonates a legitimate sender, as illustrated in Figure 2.

The attack proceeds in three stages.

1. The sender, Alice, initiates the initialization and key-exchange phases to establish a 1-to-1 chat with the receiver, Bob (Section 2.2).

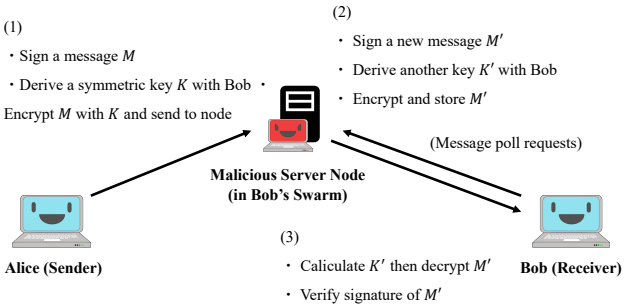


Figure 2: Overview of the impersonation attack.

2. Acting as an on-path active adversary, the MSN intercepts the exchange and replaces Alice's legitimate public key and encrypted message with attacker-controlled key material and a forged ciphertext.

3. Bob, unaware of the interception, derives a shared secret and decrypts the forged message using the MSN's substituted public key.

Because of Vulnerability 1, Bob cannot distinguish Alice's legitimate public key from the public key substituted by the MSN. Consequently, the MSN can successfully impersonate Alice and inject forged encrypted messages.

Session's 1-to-1 chat employs a Trust-on-First-Use (TOFU) model for initial key establishment. During the initialization phase, Alice can verify Bob's public-key authenticity through an out-of-band (OOB) channel, such as an account ID or QR code (Section 2.2.1). However, this verification is strictly unidirectional: while it allows Alice to authenticate Bob's identity, it does not provide a reciprocal guarantee for Bob. Therefore, this mechanism is insufficient to prevent the impersonation attack described in this section. Furthermore, as discussed in Section 4.3, this impersonation attack is significantly more feasible to execute against Session than against Signal [32] or WhatsApp [16], even though they rely on the same TOFU model.

4.2 Attack Procedure

This subsection describes the detailed procedure of the impersonation attack by a malicious server node (MSN), illustrated in Figure 2. We show that an MSN can cause a legitimate receiver to establish communication with attacker-controlled key material, thereby enabling sender impersonation.

We consider a scenario where a legitimate sender, Alice, attempts to establish a secure channel with a legitimate receiver, Bob. Let Eve denote the Malicious Server Node (MSN) acting as the adversary. The attack follows the state transitions summarized in Figure 3 and proceeds through the following three stages:

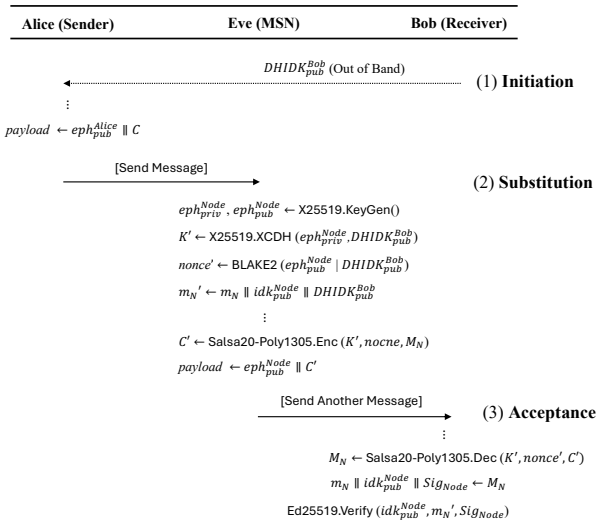


Figure 3: Impersonation attack by a malicious server node

(1) **Initiation (Sender Client):** Alice initiates the session by encrypting a payload containing the message, its digital signature, and her long-term public key using symmetric encryption. She then transmits the resulting ciphertext and her ephemeral public key (eph_{pub}^{Alice}) to a server node within the Swarm (Section 2.2.3). The metadata sent together with the message is signed using her long-term secret key (idk_{priv}^{Alice}). The shared key K used for encryption is generated via ECDH key exchange between her ephemeral secret key (eph_{priv}^{Alice}) and the receiver's long-term public key $DHIDK_{pub}^{Bob}$.

(2) **Substitution (Malicious Server Node):** The MSN, Eve, discards Alice's public key and encrypted message, and replaces them with her own key material and a forged ciphertext. Specifically, she derives a new shared key K' using ECDH between her own ephemeral secret key and Bob's long-term public key. Eve then constructs a valid protocol message carrying the forged ciphertext, signs the associated metadata with her own long-term secret key (idk_{priv}^{Node}), and forwards it to Bob through the Swarm. These operations remain cryptographically consistent with the protocol because Eve uses valid attacker-controlled keys.

(3) **Acceptance (Recipient Client):** The receiver Bob retrieves the ciphertext and the attacker's ephemeral public key ($eph^{Node}\,pub$) from his Swarm, and then performs decryption and signature verification. However, because the sender's long-term public key is not authenticated before this exchange, Bob accepts Eve's substituted public key ($DHIDK^{Node}\,pub$) without detecting the mismatch. As a result, all cryptographic checks succeed under Eve's key material, leading Bob to believe that he has established a secure session with Alice.

4.3 Feasibility and Impact

We evaluate the feasibility and potential impact of the impersonation attack. Similar to Signal [32] and WhatsApp [16], Session adopts a TOFU model; consequently, the authenticity of a sender's public key is not guaranteed during the initial key exchange. In contrast to the centralized infrastructure of Signal and WhatsApp, Session's decentralized design enables unprivileged entities to operate a server node by staking the requisite amount of cryptocurrency. This paradigm implies that an adversary can practically integrate into the network infrastructure without compromising a central authority and mount impersonation attacks against random users at a predictable economic cost. In the following sections, we first analyze the probability of successfully targeting a randomly chosen user (Section 4.3.1). Next, we estimate the financial cost required to execute the attack (Section 4.3.2). Finally, we examine the practical efficacy of existing mitigation strategies as surfacing in the real-world user interface (Section 4.3.3), highlighting inherent limitations in the current countermeasures.

4.3.1 Feasibility on Random Users

The security of Session's network layer, like that of other blockchain-based infrastructures, is closely tied to its degree of decentralization. According to the official block explorer of the Session Network, as of November 2025, the network consists of 1,596 active server nodes and 274 Swarms [19]. This corresponds to an average of approximately $1{,}596/274 \approx 5.8$ nodes per Swarm.

As mentioned in Section 2.2.4, newly joined nodes cannot arbitrarily choose which Swarm to join, in order to reduce the risk that a Swarm becomes dominated by malicious or low-performance nodes. Assuming that node assignment to Swarms is random, the probability that a single attacker-operated server node is assigned to the Swarm corresponding to a specific account ID is approximately $1/1596$. Once assigned to the target Swarm, the probability that an attacker-controlled node serves as the entry node for the sender's initial message is approximately $0.17 (= 1/5.8)$.

Therefore, even a single malicious node yields a non-negligible probability of success. When the attacker targets multiple users, the cumulative probability of success increases proportionally, making the attack practically relevant rather than purely theoretical.

Attack Example: The attack is particularly concerning in targeted settings. For example, an adversary may attempt to impersonate an employee of a large organization, a journalist, or a public figure. Even if the probability of successfully targeting any particular account is modest, the large number of potential targets within such groups substantially increases the overall chance of success. Once a false trust relationship is established, the adversary may exploit it to obtain sensitive

information or carry out follow-on social engineering attacks. This makes the attack especially dangerous in high-value communication settings where identity authenticity is critical.

4.3.2 Cost Required for the Attack

When targeting users at random, the primary economic barrier to mounting this impersonation attack is the staking cost required to operate a server node in the Session network. According to the official documentation [23], operating an individual node requires a stake of 25,000 SESH tokens. Based on a market price of approximately 0.108 USD per SESH in late 2025, the entry cost for a malicious server node (MSN) is roughly $2,700$ USD.

This relatively low financial threshold suggests that the attack is not limited to nation-state adversaries with substantial resources; it is also within reach of well-funded criminal groups or persistent individual attackers. By increasing the number of nodes under their control, an adversary can linearly increase the probability of being assigned to a target's Swarm[4]. As a result, the probability of a successful impersonation attack grows cumulatively with the attacker's capital investment.

4.3.3 Current Implementations and Safeguards

To mitigate this impersonation attack, Session implements several application-level safeguards when communication is established with a new contact.

First, the recipient client employs a *Message Request* mechanism. Messages from unregistered senders are diverted to a separate inbox, visually isolating them from established conversations and encouraging user caution. Second, once a connection has been established, the recipient can view the sender's public-key fingerprint through the profile interface. By manually comparing this fingerprint with the sender's actual key over an OOB channel, users can in principle verify its authenticity. While these measures may provide some deterrence against impersonation attacks, they have two primary limitations.

- **Suboptimal Verification Format:** The OOB verification process in Session necessitates the manual comparison of 32-byte hexadecimal strings. In contrast, Signal employs *Safety Numbers*, a QR-code-based OOB verification mechanism that provide a more usable authentication workflow. Prior work by Dechand et al. [5] showed that hexadecimal-string comparison yields significantly lower attack-detection rates and worse user perception than natural-language or numeric representations. Turner et al. [34] further observed that, even for numeric fingerprints, longer formats lead to higher miss rates than shorter ones. These findings suggest that, although 32-byte fingerprints provide strong cryptographic entropy, they are suboptimal for security mechanisms that rely on manual human verification.
- **Vulnerability to Public-Key Mining:** Session allows users to generate arbitrary identity key pairs during account creation. A malicious server node can exploit this flexibility to search for a fraudulent key pair whose fingerprint partially matches that of a legitimate sender. Concretely, given $DHIDK_{pub}^{Alice}$, the attacker can mine a forged key pair $DHIDK_{pub}^{Node}$ that matches selected bytes of Alice's fingerprint. As demonstrated by Kimura et al. in the context of Nostr [13], such public-key mining can significantly reduce the probability that an average user detects an ongoing impersonation attack during manual comparison. More concretely, as discussed by Dechand et al. [5], if an adversary can control both a 24-bit prefix and a 24-bit suffix, the brute-force search space is 2^{48}. Assuming a user base of one million and 274 Swarms as of November 2025, each Swarm manages approximately $10^6/274 \approx 2^{12}$ users. Consequently, for an MSN targeting users within its assigned Swarm, the effective search space for finding at least one useful collision is reduced to approximately 2^{36} operations. This level of complexity is well within practical reach using commodity hardware, especially because the search can be carried out offline and in parallel.

Beyond these technical limitations, Session does not adequately communicate the importance of public-key verification to its users. Unlike Signal, which explicitly explains the security role of Safety Numbers [29, 30, 33], Session's user interface merely displays the identity key without explaining its significance for authentication. As a result, this UI/UX limitation further weakens the practical effectiveness of the protocol's manual verification mechanism.

5 Message Timestamp Forgery Attack

In this section, we present a class of message timestamp forgery attacks that exploit temporal weaknesses in the Session protocol. These attacks allow an adversary to manipulate the apparent timeline of a conversation, thereby undermining the integrity of its temporal context.

5.1 Attack Overview

In both 1-to-1 and closed-group chats, a Malicious Insider (MI) (Definition 1) can perform a timestamp forgery attack on messages that they send. By manipulating client-generated timestamps, the MI can alter the chronological order in which messages are displayed on the recipient's client, thereby distorting the semantic context of the conversation. In practice,

[4] Cases in which the attacker controls a large fraction of the network are outside the scope of the whitepaper [12].

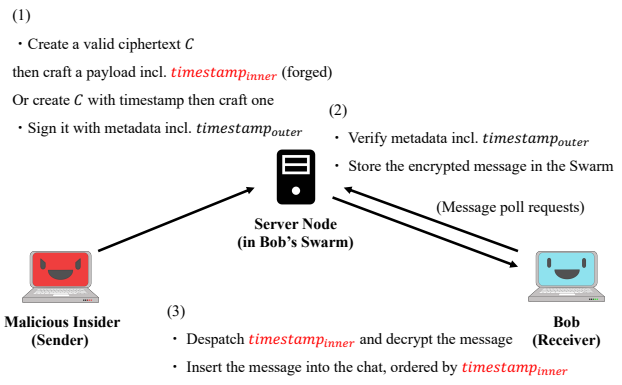


Figure 4: Overview of the timestamp forgery attack. $timestamp_{inner}$ dictates the fabricated insertion point in Bob's chat, while $timestamp_{outer}$ enforces replay protection at the server nodes.

this enables an attacker to misrepresent the order of events or retrospectively change the apparent meaning of an exchange.

The attack is fundamentally enabled by the following vulnerability, namely the lack of counter-bound symmetric keys in the E2E protocol.

Vulnerability 2. (Lack of Counter-bound Symmetric Key) In the absence of per-message counters or sequence numbers, message order is entirely contingent upon the temporal information provided by client-issued timestamps.

In the Session E2E protocol for both 1-to-1 and closed-group chats, the sender generates two distinct timestamps that serve different purposes. The outer timestamp is used by server nodes for replay protection. In contrast, the inner timestamp is included inside the protected message content. In 1-to-1 chats, it is embedded in the payload, whereas in closed-group chats, it is included in the encrypted plaintext. The recipient uses this inner timestamp to determine the position of the message in the conversation timeline. To ensure authenticity and integrity, the sender signs the inner timestamp together with the protected content and the relevant metadata, including the outer timestamp, before transmission. However, our analysis shows that these messages are not cryptographically bound to unique counters or monotonically increasing sequence numbers. As a result, a malicious sender can arbitrarily manipulate the inner timestamp without causing any cryptographic verification failure. Figure 4 illustrates the separation between the inner and outer timestamps and shows how this design enables timestamp forgery.

Moreover, we find that neither the server nodes nor the receiving clients enforce a sufficiently strict temporal validity window. In the absence of a robust expiration policy or logical-clock-style consistency check, the system cannot reliably detect whether a message has been artificially shifted forward or backward in time.

5.2 Attack Procedure

The message timestamp forgery attack is carried out through direct manipulation of the sender's client, exploiting the absence of strict temporal validation at both the server and client sides. The attack proceeds through the following three stages.

(1) **Fabrication (Malicious Insider):** In either a 1-to-1 or closed-group chat, the Malicious Insider (MI) initiates the attack by attaching a fabricated, non sequential timestamp to a message. Because the protocol does not bind the message to a cryptographic counter, the MI can arbitrarily shift the timestamp forward or backward in time. The MI then signs the protected content together with the relevant metadata, including a valid outer timestamp, and transmits the resulting message to a server node.

(2) **Relay (Server Node):** The server node, or Swarm, receives the packet and verifies that the outer timestamp is valid, after which it stores the message in the storage layer. Crucially, the server relays the message to recipients without validating the temporal range of the attached inner timestamp and without appending any mandatory arrival time metadata. This lack of server-side enforcement allows the forged temporal information to remain intact.

(3) **Rendering (Recipient Client):** The recipient's client fetches the message from the server. During rendering, the client interprets the fabricated inner timestamp as the authoritative basis for message ordering and displays the message accordingly within the conversation timeline.

By following this procedure, the MI can successfully distort the semantic context of the conversation, for example by making a message appear to have been sent before or after another message in a way that changes the apparent meaning of the exchange.

5.3 Feasibility and Impact

The proposed attack is highly feasible because it requires no elevated privileges, such as administrative access to server nodes. The only prerequisite is that the adversary acts as an MI in a legitimate 1-to-1 or group conversation. By trivially manipulating message timestamps, an MI can undermine the chronological integrity of the conversation.

In most messaging platforms, users implicitly trust timestamps and rely on the displayed order of messages to interpret the context of a conversation. This trust creates an opportunity for serious abuse. Consider, for example, a contract negotiation. An MI could send a message containing an additional, previously undisclosed clause and assign it a backdated timestamp so that it appears before the victim's subsequent message of agreement, such as "I accept these terms." In this way, the MI creates a forged record in which the victim appears to

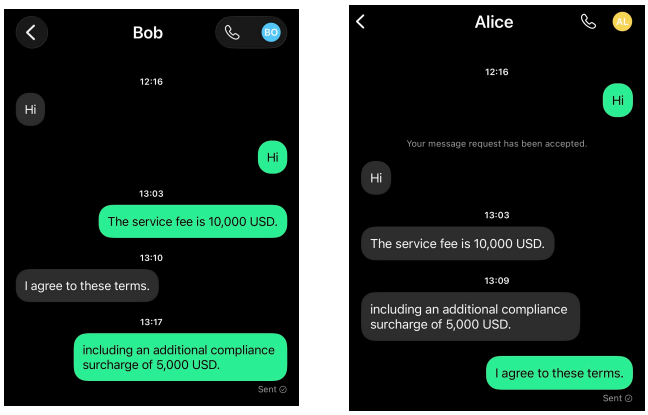


Figure 5: Attacker's client before re-login (left) and the recipient's client (right). By clearing the local state and logging in to Session again, the attacker restores message ordering, achieving consistency with the recipient's view.

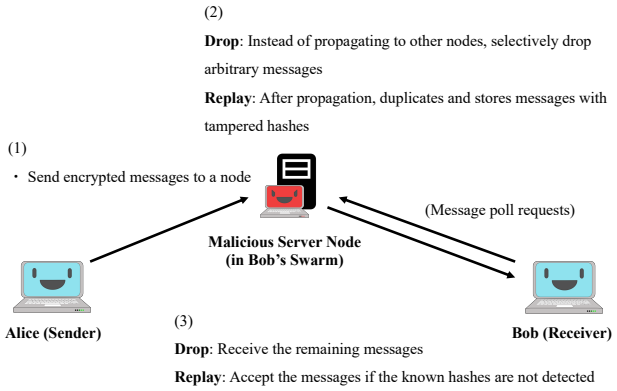


Figure 6: Overview of the message dropping and replay attack.

have consented to the inserted clause, thereby changing the apparent meaning of the entire exchange.

This attack also highlights the fragility of deniability[5] in practical implementations of the Session protocol. Although the 1-to-1 and group E2E protocols use long term signing keys, which already weaken cryptographic deniability, the platform attempts to mitigate this limitation through client-side measures such as message deletion and ephemeral handling of signatures after verification [10].

However, these protections provide only limited practical protection. A malicious insider can bypass them by externally storing received message and signature pairs before the official client discards them. In the contract example above, the attacker can later present these stored signatures to a third party, such as a court, turning what might otherwise be dismissed as a screenshot into verifiable cryptographic evidence. Thus, even though the protocol attempts to preserve an ephemeral user experience, the persistence of cryptographic proofs substantially amplifies the legal and social impact of the attack.

Attack Example: To illustrate the attack concretely, consider a 1-to-1 conversation involving a contractual agreement, as shown in Figure 5. Suppose that the malicious insider and the victim are negotiating the terms of a service. The MI first sends a message stating: "The service fee is 10,000 USD." The victim subsequently replies: "I agree to these terms."

After receiving the victim's confirmation, the MI appends a new message that introduces an additional clause, such as: "including an additional compliance surcharge of 5,000 USD." Instead of presenting this modification transparently, the MI assigns the modified message a fabricated timestamp that places it chronologically before the victim's acceptance message.

Because the protocol does not bind messages to monotonically increasing counters, and because the recipient's client renders messages according to the attached timestamps, the conversation is displayed in the following order: first the original fee statement, then the newly inserted clause, and finally the victim's agreement. As a result, it appears as though the victim consented to the modified terms.

Importantly, no cryptographic verification fails during this process. All signatures remain valid, and the protocol detects no anomaly because the forged timestamp is treated as legitimate metadata. The manipulation therefore remains within the bounds of valid protocol behavior while fundamentally altering the semantic meaning of the exchange.

6 Message Dropping and Replay Attack

In this section, we present message dropping and replay attacks mounted by a Malicious Server Node (MSN) (Definition 3). An MSN can selectively suppress or replay arbitrary messages in both 1-to-1 and closed-group chats.

As captured by Vulnerability 2, the protocol does not cryptographically bind messages to unique sequence counters. As a result, recipients cannot reliably determine whether a message was intentionally suppressed in transit or maliciously replayed by a server node, as illustrated in Figure 6.

Furthermore, we distinguish our findings from the replay attack reported by Jaeger et al. [8]. While their attack exploits the absence of cryptographic signatures on timestamps in the legacy closed-group protocol, we demonstrate a payload duplication attack in GroupsV2 that arises from the protocol's inherent design vulnerability discussed above.

[5] Deniability is a security property stating that a participant in a chat cannot provide cryptographic proof to an external third party that a specific message was sent by another participant.

6.1 Attack Procedure

The attacks described in this section are carried out by a Malicious Server Node (MSN) located in the recipient's Swarm. Below, we describe the concrete procedures for message dropping and replay attacks.

Packet Suppression (Message Dropping): If the sender selects the MSN as the entry node of the Swarm for message sending, the MSN intentionally discards the packet before it is propagated to the other server nodes in the Swarm. Because the protocol does not provide sequence based gap detection, the recipient's client has no reliable way to determine that a message has been suppressed in transit. As a result, the dropped message simply disappears from the recipient's view without triggering any explicit error or warning.

Payload Duplication (Replay Attack): The MSN duplicates the received ciphertext locally and stores each copy with a distinct forged hash value, while propagating the original ciphertext to the other server nodes in the Swarm. Because the protocol does not bind each message to a unique nonce or sequence counter, the recipient's client cannot distinguish between a legitimate message and a duplicated one resulting from a replay attack. Importantly, the client-side deduplication logic relies entirely on the integrity of server generated hashes. Consequently, if the recipient selects the MSN as the entry node for message polling, the MSN can replay a previously delivered ciphertext under a forged hash value. The recipient's client then accepts it as a fresh and legitimate message, causing multiple duplicate entries of the replayed content to be rendered in the chat interface.

The whitepaper [12] does not discuss dropping or replay attacks stemming from Vulnerability 2. Furthermore, hash computations at the implementation level are utilized to mitigate benign message redundancies inherent to Session's asynchronous decentralized design, rather than serving as a countermeasure against replay attacks by an MSN.

6.2 Feasibility and Impact

Executing this attack only requires the adversary to operate a server node that serves as the entry point during message sending (message dropping) or polling (replay attack). Because the staking requirement for running a node in the Session Network is not prohibitively high, the practical feasibility of mounting such attacks is consistent with our earlier cost analysis in Section 4.3.2. In addition, given an average of 5.8 server nodes per Swarm (as explained in Section 4.3.1), the probability of a user selecting an adversary-controlled MSN as the entry node reaches near-100% after a limited number of sending or polling iterations. These results indicate that the adversary can execute the attacks against random users with an almost guaranteed success rate.

Although these attacks do not directly compromise message confidentiality, they still introduce significant security risks.

- When a message is dropped, the sender remains unaware of the suppression and may interpret the lack of a response as intentional silence, neglect, or exclusion. As a result, the conversational state becomes inconsistent across participants without triggering any cryptographic warning.
- When a message is replayed, the recipient may perceive the sender as repeatedly sending the same message, acting urgently, or experiencing a technical malfunction. In this way, the adversary can manipulate social interpretation without altering the message content itself.
- By selectively dropping or replaying messages over time, an adversary can gradually distort the conversational context, damage interpersonal trust, and destabilize coordination within a group.
- If a Malicious Group Administrator (MGA) colludes with an MSN, the threat becomes substantially more severe. During group initialization, the MGA can repeatedly generate long term key pairs until obtaining one that maps to a Swarm containing the colluding MSN. This precomputation enables selective dropping or replay attacks against specific members or messages, allowing covert manipulation of group dynamics while remaining indistinguishable from normal network behavior.

These attacks show that delivery integrity and message ordering are security critical properties even when confidentiality remains intact.

7 Mitigations

We identified multiple previously undocumented vulnerabilities in the Session E2E protocol through an audit of its specification and implementation. Although fully addressing these issues requires substantial changes to the core protocol design, the ongoing development of Session Protocol V2, with its focus on Perfect Forward Secrecy (PFS) and post-quantum security, provides an opportunity to introduce stronger long-term defenses. At the same time, practical countermeasures are needed to reduce risk in currently deployed applications. In this section, we discuss both immediate mitigations and longer term design directions.

7.1 Out-of-Band Public Key Authentication

As described in Section 4, the Session E2E protocol adopts a TOFU model and provides only one-sided authentication: the message sender can authenticate the receiver's public key in advance through an OOB channel, but the receiver does not equivalently authenticate the sender's public key. As a result,

a malicious server node can impersonate a legitimate sender during the initial exchange. As discussed in Section 4.3.3, Session utilizes a 32 byte hexadecimal-string comparison, which yields significantly lower attack-detection rates and worse user perception. Furthermore, public key mining remains a computationally feasible threat in the current server node implementation, which is made worse by the suboptimal fingerprint verification.

To address this issue, we propose a mutual public key authentication mechanism performed between users, inspired by Safety Numbers used in the Signal Protocol [30, 33]. Under this approach, both participants verify the authenticity of their respective public keys by either scanning a partner's QR code or manually comparing the displayed digits in person. This method achieves a robust trade-off between user experience (UX) and public key authentication.

Furthermore, while the current Session interface displays public keys as contiguous 32-byte hexadecimal strings for manual OOB comparison, our proposal adopts a more resilient format similar to Safety Numbers in Signal. By converting these keys into numeric representations and partitioning them into discrete blocks, the system can reduce user detection errors during the verification process.

7.2 Ratchet Mechanism and Its Alternatives

To fundamentally address Vulnerability 2, the protocol needs a cryptographic mechanism that binds each message to an ordered state. One principled approach is to reintroduce a ratchet mechanism, which was used in earlier generations of secure messaging protocols and remains standard in systems such as Signal [32], WhatsApp [16], Threema [7, 22], and Matrix [6]. Ratcheting mechanisms derive fresh session keys over time and can provide properties such as Perfect Forward Secrecy (PFS). Specifically, the Double Ratchet algorithm integrates these with asymmetric exchanges to achieve Post-Compromise Security (PCS). More importantly in our setting, it cryptographically links each message to an evolving sequence of states, enabling recipients to detect temporal manipulation, replay, and omission. Although the final rendering order in the user interface remains an implementation choice, the integrity of the underlying message stream can still be protected.

However, reintroducing a full ratchet mechanism would require substantial architectural changes. Session previously moved away from the Signal based design in part because of implementation complexity, and the removal of ratcheting was one consequence of that transition. Therefore, although a ratchet based redesign is desirable as a long-term solution, a lighter weight mitigation is also needed for currently deployed systems.

As a near term countermeasure, we propose temporal window validation. Under this approach, both server nodes and recipient clients enforce a predefined acceptance window for message timestamps at the time of delivery. Messages whose timestamps fall outside the permitted range are rejected or flagged as suspicious. In a decentralized system such as Session, the choice of window size must balance usability and security: a wider window improves robustness against clock skew and network delay, whereas a narrower window provides stronger resistance against timestamp forgery.

Moreover, we propose client-side hash-based deduplication to mitigate replay attacks. Under this approach, the recipient client recomputes the hash of each incoming message from entry server nodes and performs a lookup against a local deduplication cache. If a match is found, the message is identified as a replay and discarded.

This design aligns with the findings of Kimura et al. [13], who identified a binding vulnerability in the Nostr E2E protocol. They observed that recipients' trust in sender-provided IDs allows for forgery attacks where malicious content is bound to legitimate IDs. To mitigate this, they proposed mandatory ID recalculation on the recipient's client, ensuring that any data failing the hash consistency check is immediately discarded to prevent cache poisoning.

Together, these immediate and actionable mitigations provide fine-grained protection within the temporal window, effectively blocking the re-injection of forged or identical messages with minimal computational overhead.

8 Conclusions

Session is a decentralized messenger designed to provide strong anonymity and privacy, without requiring a phone number or email address for account registration. In this paper, we analyzed the deployed Session E2E messaging protocol for both 1-to-1 and closed-group chats and identified three practical attacks: an impersonation attack (Section 4), a message timestamp forgery attack (Section 5), and message dropping and replay attacks (Section 6). We further evaluated the feasibility and impact of each attack and showed that all of them can be mounted with practical resources. In particular, the message timestamp forgery attack is especially concerning because it requires no elevated privilege and can be carried out by an ordinary participant in a legitimate conversation.

Our findings highlight the risks of deploying custom secure messaging protocols without sufficiently preserving the guarantees provided by well-studied designs. Although Session Protocol V1 has been publicly available for several years, the vulnerabilities identified in this work had not been previously documented. This suggests that rigorous security evaluation must go beyond high-level protocol descriptions and include careful inspection of implementation behavior and real-world attack feasibility. We hope that our analysis will help improve Session's current protocol and inform the design of more secure future messaging systems.

Acknowledgments

This work was supported by JST CRONOS, Japan, Grant Number JPMJCS25N1, JSPS KAKENHI, Grant Number JP24H00696 and JST ACT-X, Japan, Grant Number JPMJAX25M8.

References

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2019.

[2] Daniel J. Bernstein. Cryptography in NaCl, 2009. https://cr.yp.to/highspeed/naclcrypto-20090310.pdf [Accessed: 2026-3-3].

[3] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 784–813. Springer, 2022.

[4] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[5] Sergej Dechand, Dominik Schürmann, Karoline Busse, Yasemin Acar, Sascha Fahl, and Matthew Smith. An empirical study of textual Key-Fingerprint representations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 193–208, Austin, TX, August 2016. USENIX Association.

[6] Matrix.org Foundation. Matrix.org, 2026. https://matrix.org/ [Accessed: 2026-4-24].

[7] Threema GmbH. Threema – secure communication for individuals and companies – Threema, 2026. https://threema.com/en/ [Accessed: 2026-3-3].

[8] Joseph Jaeger and Akshaya Kumar. Analyzing group chat encryption in MLS, Session, Signal, and Matrix. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques Madrid, Spain, May 4-8, 2025, Proceedings, Part VIII*, volume 15608 of *Lecture Notes in Computer Science*, pages 272–301. Springer, 2025.

[9] Kee Jefferys. The Session protocol: A new standard for decentralized private messaging, 2020. https://getsession.org/blog/introducing-the-session-protocol [Accessed: 2026-3-3].

[10] Kee Jefferys. Session protocol: Technical implementation details, 2020. https://getsession.org/blog/session-protocol-technical-information.

[11] Kee Jefferys. The Session protocol: What's changing — and why?, 2020. https://getsession.org/session-protocol-explained [Accessed: 2026-3-3].

[12] Kee Jefferys, Maxim Shishmarev, and Simon Harman. Session: A model for end-to-end encrypted conversations with minimal metadata leakage. *CoRR*, abs/2002.04609, 2020.

[13] Hayato Kimura, Ryoma Ito, Kazuhiko Minematsu, Shogo Shiraki, and Takanori Isobe. Not in the prophecies: Practical attacks on Nostr. In *10th IEEE European Symposium on Security and Privacy, EuroS&P 2025, Venice, Italy, June 30 - July 4, 2025*, pages 585–606. IEEE, 2025.

[14] U.S. Naval Research Laboratory. Onion routing, 2026. https://www.onion-router.net/ [Accessed: 2026-3-3].

[15] Alex Linton. Private messaging apps: Who needs a secure messenger — and why?, 2019. https://getsession.org/who-needs-private-messaging [Accessed: 2026-3-3].

[16] WhatsApp LLC. Whatsapp | secure and reliable free private messaging and calling, 2026. https://www.whatsapp.com/ [Accessed: 2026-3-3].

[17] Johannes Löbbecke. Session: Secure messenger with additional measures for metadata protection. In *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2020*, volume NET-2020-11-1, pages 21–25, November 2020.

[18] Charlie Boulo Marwan Anastas. Audit of Session secure messaging application, 2021. https://blog.quarkslab.com/audit-of-session-secure-messaging-application.html [Accessed: 2026-3-3].

[19] Oxen. Oxen blockchain explorer, 2025. https://oxen.observer/ [Accessed: 2026-3-3].

[20] Oxen. Github - oxen-io/libsession-util: Session utilities library, 2026. https://github.com/session-foundation/libsession-util [Accessed: 2026-3-3].

[21] Oxen. Github - oxen-io/oxen-core: Oxen, 2026. https://github.com/oxen-io/oxen-core [Accessed: 2026-3-3].

[22] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from Threema: Analysis of a secure messenger. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1289–1306, Anaheim, CA, August 2023. USENIX Association.

[23] Session Technology Foundation. Running a Session node, 2025. https://docs.getsession.org/contribute-to-the-session-network/running-a-session-node [Accessed: 2026-3-3].

[24] Session Technology Foundation. Session | send messages, not metadata. | private messenger, 2025. https://getsession.org/ [Accessed: 2026-3-3].

[25] Session Technology Foundation. Session protocol v2: PFS, post-quantum and the future of private messaging, 2025. https://getsession.org/blog/session-protocol-v2 [Accessed: 2026-3-3].

[26] Session Technology Foundation. Session stagenet node setup, 2025. https://docs.getsession.org/contribute-to-the-session-network/testnet/session-stagenet-node-setup [Accessed: 2026-3-3].

[27] Session Technology Foundation. Github - session-foundation/session-ios: Session iOS - a decentralized, onion routed, private messenger, 2026. https://github.com/session-foundation/session-ios [Accessed: 2026-3-3].

[28] Session Technology Foundation. Github - session-foundation/session-storage-server: Oxen storage server, 2026. https://github.com/session-foundation/session-storage-server [Accessed: 2026-3-3].

[29] Signal Technology Foundation. How to protect yourself on Signal – Signal support, 2026. https://support.signal.org/hc/en-us/articles/9932632052378-How-to-protect-yourself-on-Signal/ [Accessed: 2026-3-3].

[30] Signal Technology Foundation. Signal » blog » safety number updates, 2026. https://signal.org/blog/safety-number-updates/ [Accessed: 2026-3-3].

[31] Signal Technology Foundation. Signal » documentation, 2026. https://signal.org/docs/ [Accessed: 2026-3-3].

[32] Signal Technology Foundation. Signal » home, 2026. https://signal.org/ [Accessed: 2026-3-3].

[33] Signal Technology Foundation. What is a safety number and why do i see that it changed? – Signal support, 2026. https://support.signal.org/hc/en-us/articles/360007060632-What-is-a-safety-number-and-why-do-I-see-that-it-changed/ [Accessed: 2026-3-3].

[34] Dan Turner, Siamak F. Shahandashti, and Helen Petrie. The effect of length on key fingerprint verification security and usability. In *Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023, Benevento, Italy, 29 August 2023- 1 September 2023*, pages 23:1–23:11. ACM, 2023.

[35] 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.