LINE-Break: Cryptanalysis and Reverse Engineering of Letter Sealing

Diego F. Aranha
dfaranha@cs.au.dk
Aarhus University
Aarhus, Denmark

Adam Blatchley Hansen
blatchley@cs.au.dk
Aarhus University
Aarhus, Denmark

Thomas Kingo T. Mogensen
kingowassard@hotmail.com
Aarhus University
Aarhus, Denmark

Abstract

We present a security analysis of the messaging service known as LINE, a popular platform used daily by millions of users in Southeast Asia – most notably Japan, Taiwan, Thailand, and Indonesia. More specifically, we focus on its underlying custom end-to-end encryption (E2EE) protocol, known as Letter Sealing v2. Our findings show that Letter Sealing allows a TLS Machine-in-the-Middle attacker or malicious server to violate integrity, authenticity, and confidentiality of communications. The stateless design of the protocol allows message replay, reordering, and blocking attacks without the user being notified. The lack of origin authentication facilitates impersonation attacks, in which the authorship of messages in one-to-one or group chats can be forged by malicious users colluding with the adversary. Lastly, stickers and URL previews present a notable leakage of plaintext, which leads to a violation of confidentiality. To verify the correctness of our findings, we mounted a Machine-in-the-Middle attack on an iOS device, yielding the device's outgoing traffic and the corresponding server responses. Utilizing this setup, we experimentally verified our attacks against the authentic LINE application and an independent implementation. We discuss our findings in comparison to the state-of-the-art E2EE protocols, and conclude that Letter Sealing does not satisfy the requirements expected from a modern E2EE messaging protocol.

CCS Concepts

• **Security and privacy** → **Cryptography**; **Public key (asymmetric) techniques**.

Keywords

E2EE, Letter Sealing, Cryptanalysis, Reverse Engineering

ACM Reference Format:
Diego F. Aranha, Adam Blatchley Hansen, and Thomas Kingo T. Mogensen. 2026. LINE-Break: Cryptanalysis and Reverse Engineering of Letter Sealing. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 1–5, 2026, Bangalore, India.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3779208.3805983

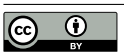
1 Introduction

End-to-end encrypted (E2EE) instant messaging is one of the most widely deployed cryptographic protocols in history, with millions of global users exchanging billions of encrypted messages every day. Much of this success is attributed to the Double-Ratchet algorithm [35], a core part of applications such as Signal, WhatsApp, Wire and Meta Messenger. It combines a public-key ratchet based on the Diffie–Hellman key exchange with a symmetric-key ratchet based on a key derivation function. The resulting combination enables advanced security properties, such as forward secrecy and some form of post-compromise security (self-healing) [9]. Despite the security guarantees given by state-of-the-art E2EE protocols, some popular messengers have chosen to deploy customized protocols with unclear security guarantees and mixed results, such as Telegram [2, 3, 27], Threema [34] and LINE [18, 25, 26].

LINE is a free instant messaging service launched in Japan in 2011. It was later deemed a Super App due to the large variety of services it provides, such as a digital wallet, news, video-on-demand, and even games. LINE is widely used, with an estimated one billion annual users by 2025 and approximately 200 million active monthly users [36]. The application is used most extensively in Southeast Asia, covering roughly 80% to 85% of the population of Japan from 2017 to 2024 [18, 37], and covering 99.5% of people in their 20s in Japan [16, 19]. Due to the widespread adoption, it is used for e-government services within Japan [11], and its revenue peaked at $2.36 billion in 2021 [36]. Previously owned by LINE Corporation, it is now under the ownership of LY Corporation following a merger with Yahoo! Japan and two other companies. The messaging application incorporates a custom end-to-end encryption (E2EE) protocol, coined Letter Sealing (LS), originally developed throughout 2014-2016 and launched at scale in 2016 [12, 29].

Previous work has documented several weaknesses in the original design of the protocol, named Letter Sealing v1 (LSv1), featuring attacks such as replay, forgery, and impersonation attacks [18, 26]. The protocol was updated to v2 in 2019 and has since been updated again in 2021 [12]. With the design of LSv2, LY Corporation claims to have *"all the issues resolved"* [12, 18], which stands in stark contrast to the findings in this work.

Contributions. We continue the effort of analyzing "cryptography in the wild", introduced by Albrecht and Paterson [4], and present a security analysis of LSv2. The analysis is conducted from an examination of the white paper [12], interactions with the live LINE protocol, and partial reverse engineering of its implementation. The experiments were conducted using a Machine-in-the-Middle (MitM) setup against an iOS client using a rogue root certificate, allowing the adversary to tamper with protocol metadata as a malicious server would. We interact with LINE servers through both the official application and an independent JavaScript implementation of the client-side portion of the protocol, called LINEJS[1].



ASIA CCS '26, Bangalore, India

ACM ISBN 979-8-4007-2356-8/26/06
https://doi.org/10.1145/3779208.3805983

[1] LINEJS library in Javascript to create bots: https://github.com/evex-dev/linejs

We present the following attacks:

- *Equivocation attacks*: messages can be replayed, reordered, or dropped without the endpoints being aware, violating transcript consistency. This is inherent to the stateless nature of the protocol, which is insufficiently mitigated by easy-to-bypass server-side countermeasures. We show further attacks against integrity by tampering with read receipts.
- *Impersonation attacks*: the authorship of messages in one-to-one and group chats can be forged by a malicious user colluding with the adversary. This is a direct consequence of the way (group) keys are generated and managed in LSv2, and the lack of origin authentication measures. Combined with the previous attacks, we show how the adversary would be able to forge communications among a subset of parties in a group chat, or infiltrate a group chat and manipulate its subsequent communication.
- *Plaintext leakage attacks*: there is substantial leakage of plaintext through usability features, such as stickers and URL previews. While LSv2 documentation notifies the user of some of the privacy-usability trade-offs around these features, we find that in case of a snooping server, the leakage goes beyond what one would expect from the public documentation.

Ethical concerns. The analysis was performed exclusively using user accounts and devices belonging to the authors, so no additional user was impacted. Our network attacks relied exclusively on the MitM setup in our local networks, hence there was no tampering with the LINE server or APIs. Our findings were disclosed in June 2025 to the LY Corporation Computer Security Incident Response Team and later confirmed by the LSv2 Team, who provided a statement available at our website.[2] We elaborate further on the disclosure process in Section 4.7.

2 E2EE Messaging

There is no clear consensus on a formal definition of end-to-end encryption (E2EE), and the definitions of Secure Messaging vary vastly in the literature. NIST presents the following vague definition of E2EE: "Communications encryption in which data is encrypted when being passed through a network, but routing information remains visible" [32]. Thus, the singular requirement is for the contents of the message to be readable only by the sender and recipient, which amounts to the notion of Confidentiality in the literature. This work will rely on the work of Knodel et. al [28] for the cryptographic requirements on E2EE as a minimum consensus across multiple works [22, 23]. We now unfold their criteria in more detail.

2.1 Security Properties of E2EE

Definition 2.1 (Confidentiality). A system provides message confidentiality if only the sender and intended recipient(s) can read the message plaintext, i.e., messages sent between participants can only be read by the agreed-upon participants in the group, and all participants share the identical group member list.

[2]https://www.linebreak.info

The notion of confidentiality is that contents are hidden from the uninvited. Therefore, confidentiality is broken if the plaintext can be decrypted or recovered at any intermediate device or server.

Definition 2.2 (Integrity). A system provides message integrity when it guarantees that messages have not been modified in transit. If a message has been modified, it must be detected reliably by the recipient(s).

As all digital communication is routed through a series of intermediaries, it is necessary to incorporate mechanisms that detect modifications happening in transit. The notion is extended in [5] to encompass protection against message injection attacks and is termed *Authenticity*.

Definition 2.3 (Authentication). A system provides authentication if the recipient and sender can verify each other's identities in relation to the contents of their communications.

The communicating parties should not rely on intermediaries to verify the identity of the communicating parties to each other. Instead, they should be able to perform such a verification themselves.

Definition 2.4 (Forward secrecy). Forward secrecy prevents attackers from decrypting current ciphertexts through compromising an endpoint's key material in the future.

In other words, forward secrecy ensures that past traffic stays secure, even if an attacker compromises one of the endpoints at a certain point in time. However, as we will see, this imposes certain requirements on the cryptographic protocol: for example, regularly deriving new encryption/decryption keys, and destroying old keys that are no longer required to encrypt or decrypt messages.

All of the above properties can be outlined in various ways. The core intuition is that Confidentiality *hides* the contents, Integrity *preserves* the contents, and Authentication *links* the contents to the correct sender and recipient. Forward secrecy is a more advanced property that introduces a notion of time: messages sent before the compromise should be infeasible to decrypt after the compromise.

The four defined properties are considered necessary, while the following are deemed *optional* or *desirable* in [28, 34]: Availability, Loss Resilience, Deniability, Post-compromise Security (PCS), Metadata Obfuscation, Disappearing Messages, Traceable Delivery, No Duplication, No Creation, and Closeness. Because these properties are not pivotal to the analysis, nor are they properties of Letter Sealing[3], we will omit further details. Note that certain definitions of Secure Messaging, as found in [5], require a number of these properties, underscoring the lack of consensus. Having established the necessary security properties, let us define the threat models under which the analysis is conducted.

2.2 Threat Models of E2EE

All widely deployed messaging services are centralized, meaning that all messages are transmitted via a central server, which receives messages from the respective senders, temporarily stores them, and forwards them as soon as the recipients are online.

[3]One might argue for Availability, Loss Resilience, and Deniability. While Availability may have some merit, Loss Resilience and Deniability appear to be, at best, incidental side effects rather than fundamental properties of the protocol.

In the present work, we will draw upon the insights and security models presented within the research literature. The definitions of threat models are reproduced from security analyses of similar messaging applications or prior security analyses of LINE [18, 26]. Note that the technical whitepaper from LINE contains no reference to any threat model [12].

Definition 2.5 (E2E adversary). An E2E adversary can intercept, read, and modify any message sent over the network and has full access to the messaging server, thus bypassing the client-to-server (C2S) encryption.

The E2E adversary adequately captures the essence of a compromised server or malicious insider. The necessity of such a threat model arises due to powerful adversaries in the form of intelligence organizations and nation-state actors, for which these messaging servers are prime targets. Any proper security analysis of E2EE is therefore required to assess this threat model [4, 18, 24, 26, 34].

Definition 2.6 (Malicious User). A malicious user is a legitimate party in one-to-one E2EE who tries to break one of the defined security properties of another E2EE session by maliciously deviating from the protocol.

Definition 2.7 (Malicious Group Member). A malicious group member, who is a legitimate group member and possesses a shared group key, tries to break the defined security properties by deviating from the protocol.

Malicious users and group members are less powerful adversaries who operate under the capabilities afforded to any regular user.

Definition 2.8 (Compelled Access). Compelled access defines a scenario in which the adversary has access to a user's device.

Although this threat model is highly invasive, it accurately illustrates a scenario referenced in travel advisories by government bodies, where border control authorities may scrutinize the contents of electronic devices.

Definition 2.9 (Machine in the Middle (MitM)). An adversary has positioned itself in-between a victim and the server, thereby acquiring the capabilities of reading and manipulating all traffic between the victim and the server.

Having defined both the security properties and the threat models, it is evident that E2EE is a compound notion relying on several properties to ensure the protection of users. One might question the necessity of some or all outlined properties, pointing to server-side countermeasures. While these countermeasures can solve specific categories of attacks performed under certain threat models, in the event of an E2E adversary, the users lose the protection of these countermeasures and instead rely on protocol-level protections and security guarantees. Additionally, server-side countermeasures are typically proprietary and not ued to the public, preventing independent verification by third parties [26]. Consequently, each of the properties should be adequately guaranteed by the design of the cryptographic protocol to protect the users. In conclusion, to ensure that E2EE is guaranteed at the protocol level, the E2E adversary is the essence of an E2EE security analysis. See Figure 1 for a visual representation of the E2E and MitM threat models.

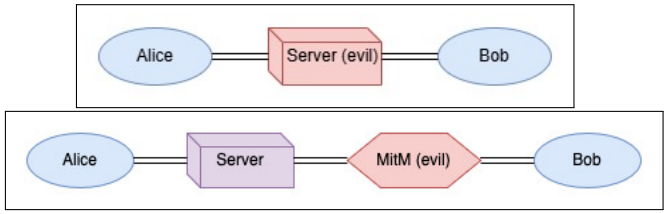


Figure 1: Top: E2E threat model. Bottom: MitM threat model with Bob as the victim.

2.3 Related Work

Most popular encrypted messaging applications have been subjected to security analyses: Signal (and its protocol used in WhatsApp, Meta Messenger), Threema, Matrix, Telegram, LINE, and iMessenger [1–5, 9, 10, 18, 21, 24, 26, 30, 38]. Several of these applications have been found vulnerable to devastating replay and impersonation attacks.

A recent security analysis of Threema, formerly a government-mandated encrypted messaging service, mounted a total of seven realistic attacks under various threat models [34]. The researchers achieved replay and reordering attacks under the E2E threat model, which are similar to our findings, but also uncovered cross-protocol attacks between subprotocols that allow the E2E adversary to obtain encryptions of chosen messages from targeted users (*kompromat* attack). A weaker network attacker in control of the C2S protocol would also be capable of impersonating the client to the server forever by leaking a single ephemeral key or by exploiting cross-protocol interactions to trigger the user into compromising their account. Furthermore, cloning and compression side-channel attacks are presented in the compelled access threat model.

The authors also point to three lessons learned in the analysis: (i) using secure libraries for cryptographic primitives does not lead to secure protocol design; (ii) beware of cross-protocol interactions; and (iii) implement proactive, not reactive, security.

Research in secure messaging revolves mostly around the Signal protocol and the security properties afforded by the underlying Double Ratchet algorithm. Previous security analysis of Letter Sealing conclude by recommending that LINE should instead implement the Signal protocol as the state-of-the-art solution to encrypted messaging [25, 26].

3 Letter Sealing

Letter Sealing (LS) denotes the collection of encryption protocols used in LINE for one-to-one messaging, group messaging, and Voice over Internet Protocol (VoIP) services. Letter Sealing exists in two major versions, as shown in Figure 1, with the current version 2.1 representing a minor revision of v2. Letter Sealing is claimed to provide E2EE for text messages and media streams, ensuring that "*no third parties or LINE Corporation can decrypt private calls and messages*" [12]. The company asserts confidentiality, partial forward security (between clients and servers through TLS), integrity and authenticity as intended security goals.

3.1 Letter Sealing Version 1

LSv1 implemented a custom C2S protocol. E2EE was performed using a standardized encryption mechanism but with a non-standard authentication mechanism. Message metadata was appended and sent with no integrity protection, leading to most of the attacks discovered in the literature [26].

Earlier security analyses of LS and the LINE messaging application have uncovered replay, forgery, impersonation, and key reuse attacks as the primary deficiencies of the original design [18, 26]. Furthermore, forward secrecy is only provided in the C2S encryption [24, 26]. All of these apply to the E2E adversary (e.g., LY Corporation itself). It bears repeating that the primary purpose of E2EE is to protect users from the E2E adversary [26]. With the design of LSv2, LY Corporation claims to have *"all the issues resolved"* [12].

3.2 Letter Sealing Version 2

The protocol was updated to LSv2 as an attempt to address flaws found in previous security analyses. It now utilizes standardized authenticated encryption for payload protection in the form of AES256-GCM. Key agreement stays unchanged and is performed using ECDH over the standardized elliptic curve Curve25519 [8].

	Version 1	Version 2
Key exchange algorithm	ECDH over Curve25519	
Message encryption algorithm	AES256-CBC	AES256-GCM
Message hash function	SHA-256	N/A
Data authentication	AES-ECB with SHA-256 MAC	AES256-GCM
Message data	Encryption and integrity	
Message metadata	Not protected	Integrity

Table 1: Table of cryptographic algorithms in versions 1 and 2 of Letter Sealing (figure from [12]).

Key Generation and Registration. When launching a fresh LINE client and authenticating with the LINE server, the client generates an ECDH key pair consisting of a secret key and a public key (sk, pk). The secret key is stored in the application's private storage, whereas the public key is registered on the LINE server. The server associates the public key with the authenticated client and attributes a unique KeyID to the public key. The KeyID is returned to the user. Users can verify public keys out of band using the public key fingerprints displayed within the app [12].

One-to-One Chats. The following outlines the steps to be performed by the sender in a one-to-one scenario [12]. We denote the users U_i and U_j, and their corresponding keys (pk_{U_i}, sk_{U_i}), and (pk_{U_j}, sk_{U_j}).

(1) Perform ECDH to generate a shared *Secret*. Both users can derive the same secret from their secret key and the other user's public key.
(2) Generate a 16-byte random salt.
(3) Compute the encryption key.
(4) Generate a 12-byte nonce consisting of an 8-byte per-chat counter and a 4-byte randomly generated value.
(5) Compose the metadata (AAD).
(6) Compute the ciphertext and authentication tag, given the key, nonce, plaintext M, and AAD.
(7) Compose the final message.

The pseudocode equivalent to these steps is outlined below.

$$
\begin{aligned}
&Secret = ECDH(pk_{U_j}, sk_{U_i}) = ECDH(pk_{U_i}, sk_{U_j})\\
&salt = randombytes(16)\\
&key = SHA256(Secret||salt||"Key")\\
&nonce(12) = counter[8]||randombytes(4)\\
&AAD = recipID||senderID||senderKeyID||recipKeyID||version||type\\
&(C, tag) = AES_{GCM}(key, nonce, M, AAD)\\
&Msg = version||type||salt||C||tag||nonce||senderKeyID||recipKeyID
\end{aligned}
$$

Only the final message *Msg* is sent to the recipient through the server. Note that the sending party U_i is expected to know the following values in advance, or communicate with the server to learn them: (a) pk_{U_j}, (b) *counter*, a 64-bit per-chat counter stored locally by U_i, representing the number of messages U_i has sent in this chat, (c) *recipID*, the identifier of the recipient's account, acquired through adding them as a friend, and (d) *recipKeyID*, a server-generated identifier denoting the version of the recipient's public key. The recipient U_j decrypts by computing the same symmetric key when performing ECDH with sk_{U_j} and pk_{U_i}, then following the same key derivation steps.

While the client-stored 64-bit counter protects against nonce reuse, in the current implementation of LS the receiver does not utilize it to keep a consistent protocol state between parties. Moreover, no digital signature or origin authentication mechanism (other than GCM tags) is present in the protocol. Therefore, it is respectively vulnerable to replay and impersonation attacks.

Group Chats. To facilitate group messaging in LINE, the group must first establish a common secret through the key exchange protocol. A member of the group - "*typically the first member wishing to send a message to the group*" [12] - performs the following steps:

(1) Generate a new ECDH key pair (pk_g, sk_g).
(2) Designate the private key, sk_g, to be the *group key*.
(3) Retrieve public keys of all other group members: $[pk_{U_j}, pk_{U_k}, ...]$.
(4) Derive pairwise symmetric encryption keys: $key_j = ECDH(sk_{U_i}, pk_{U_j})$, $key_k = ECDH(sk_{U_i}, pk_{U_k})$, etc.
(5) Encrypt the group key sk_g under each of the symmetric keys, creating one encryption for each member and compiling them into a list.

The equivalent pseudocode is outlined below in the same order:

$$
\begin{aligned}
&newKeyPair =_{\$} (pk_g, sk_g)\\
&groupKey = sk_g\\
&memberPublicKeys = [pk_{U_j}, pk_{U_k}, ...]\\
&key_j = ECDH(sk_{U_i}, pk_{U_j}), \quad key_k = ECDH(sk_{U_i}, pk_{U_k}), \quad ...\\
&encryptedGroupKeys = [E_{key_j}(groupKey), E_{key_k}(groupKey)...]
\end{aligned}
$$

The list of encrypted group keys is sent to the LINE server, where it is stored. It is unclear how the server correlates each user to their associated entry of this array, and it is also unclear from the specification which form of symmetric encryption is used to encrypt the keys, i.e., which mode of operation is used for AES. LINEJS points to CBC mode in their implementation[4]. Key rotation is repeated upon group changes, such as adding or removing members [12]. LSv2 in enabled in groups with up to 50 members.

[4]Specifically, refer to the file located at `/packages/linejs/base/e2ee/mod.ts`

Sending a Message to the Group. Group messaging functions analogously to one-to-one chats. For ease of reference, the pseudocode representation is kept in full, while the description highlights only the differences. We assume all group members hold the group key sk_g. Sending a message from U_i to the group consists of the following alterations from the one-to-one protocol:

(1) Compute a shared group secret based on the sender's public key: $Secret = ECDH(sk_g, pk_{U_i})$.
(2) Unchanged
(3) Unchanged
(4) Unchanged
(5) Replace *recipID* with *groupID* and *recipKeyID* with *groupKeyID*.
(6) Unchanged
(7) Once again, replace *recipKeyID* with *groupKeyID*.

The equivalent pseudocode is outlined below in the same order:

$$Secret = ECDH(sk_g, pk_{U_i})$$
$$salt = randombytes(16)$$
$$key = SHA256(Secret||salt||''Key'')$$
$$nonce(12) = counter[8]||randombytes(4)$$
$$AAD = groupID||senderID||senderKeyID||groupKeyID||version||type$$
$$(C, tag) = AES_{GCM}(key, nonce, M, AAD)$$
$$Msg = version||type||salt||C||tag||nonce||senderKeyID||groupKeyID$$

Only the final message *Msg* is sent to the group. Notice that the secret is derived identically by all group members, namely, using the shared group key and the public key of the sender. Once more, the server and client implementations do not check the counter within the nonce to ensure a consistent protocol state across parties. Consequently, the group protocol is also vulnerable to replay and impersonation attacks.

4 Security Analysis

The security analysis has been conducted under the following limitations: (i) Our geographical region has limited the amount of available authentication flows and the variety of accessible services and account types. Therefore, the analysis cannot touch upon the functionality related to phone numbers or the use of Official Accounts and Business Accounts. (ii) Every attempt at removing the certificate pinning on an Android device yielded an unusable application, partly due to mitigations encountered when performing authentication through Google accounts. For this reason, we mount MitM attacks by installing a rogue self-signed certificate in the iOS device, which makes the LINE client trust as legitimate because the app does not implement any form of certificate pinning on iOS. This yields the outgoing traffic and the corresponding responses, allowing us to prototype the attacks both as a network TLS MitM attacker (as a malicious router between the client and LINE) and to simulate tampering of metadata by a malicious LINE server. We employed the `mitmproxy` software[5] and its scripting capability. However, we did not manage to route all traffic through our MitM setup; more specifically, we were unable to capture and manipulate incoming messages. Thus, our evidence for impersonation attacks relies on the indistinguishability between legitimate and manipulated traffic.

[5]mitmproxy - an interactive HTTPS proxy: https://www.mitmproxy.org/

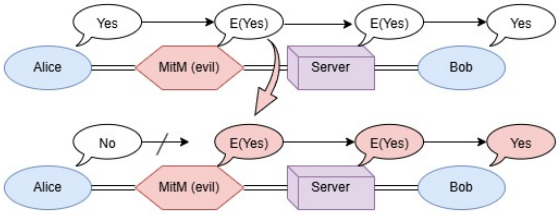


Figure 2: A replay attack in the MitM threat model of one-to-one communications. The MitM adversary replays Alice's first message. *E(Yes)* denotes an encryption of "Yes".

Finally, throughout this section, the practicality of certain attacks with respect to server countermeasures is left open. This is due to the necessary threat model of these attacks and our lack of access to the LINE servers, making it infeasible to properly test the attack strategies. We note the attack strategies for completeness, hoping to incite others to discover the exact requirements for mounting the attacks, as we also build on the works of others.

4.1 Replay Attacks

The first and simplest attack uncovered during our analysis is the capability to *replay* encrypted messages. LSv2 lacks a cryptographically-bound protocol state: each client maintains an independent message counter that is neither synchronized nor validated by the endpoints. Nonces and counters are thus accepted without verification, and server-side modifications further obscure state tracking. As TLS is used for C2S communications and protects against replays, the threat model is limited to MitM and E2E adversaries.

Figure 2 gives an overview of the attack strategy. The MitM can be mounted on either party. Should it be mounted on the sender, all chat members will receive the replayed message. Should it be mounted on a recipient, only the targeted party will receive it. When Alice sends a message to Bob within an E2EE packet, the server returns a modified message object back to her device. Specifically, an additional counter value *reqSeq*, included in metadata, is overwritten. As per our observations, the recipient accepts arbitrary counter values, regardless of whether they contain the expected value, proving that clients do not store counter values of their counterparts to maintain protocol state. Furthermore, the message received by the recipient does not contain the unauthenticated metadata counter value at all, providing evidence that the counter is either utilized as a server-side replay mitigation or to ensure that messages arrive only once if the sender is experiencing an unstable Internet connection.

Mitigations to Subvert. To test replay attacks, we hardcoded the randomness and the counter. Upon resending an identical message, we encountered a server-side mitigation that only occurs if the message is an exact copy of a previous message, which is unlikely in a real-world scenario due to the sampling of random nonce and salt. To subvert the mitigation, one can alter *reqSeq*, as it appears in the unauthenticated metadata. In practice, communications are Thrift-encoded [20], so integer counters are converted to byte arrays of type `Uint8Array`. Modifications to one of the bytes

representing the counter will subvert the server-side mitigation, making it insufficient against replay attacks.

Counter Encodings. We reverse engineering the server behavior to understand how counters are managed for replay detection. Certain choices of counter values allow for unlimited replays of an identical message, while other values encounter the mentioned server-side mitigation when replayed. Table 2 contains an overview of how different counter values are Thrift encoded. All values that either encode to 0 or whose encoding begins with an odd number yield the possibility of endless replays, subverting the server-side mitigations. We experimentally verified the attack strategy against the authentic LINE application under the MitM adversary, utilizing `0xFF` (255) as the counter value. The replay attack is undetected by the sender, leaving no indication of compromise.

Through a binary search, we discovered a cut-off at the counter value 560493232128. This is equal to 522 GiB and can be written as 522×2^{30}. Any positive smaller value will not allow for replays, as the server flags the replay and drops the packet[6]. From Table 2, one can see that Thrift performs a zigzag encoding in which the sign bit becomes the least significant bit of the encoding. This has the consequence that positive numbers are encoded to even numbers, while negative numbers are encoded to odd numbers. It is further clear from Table 2 that the Thrift encoding is cyclic from 0 to 560493232128 and back[7]. Lastly, the encodings vary in length, affecting the final length of the E2EE packet.

As mentioned, the server detects replays for positive counter values smaller than 560493232128. Due to the structure of the Thrift encoding, we hypothesize that the server makes a check for expected counter values but forgets to handle a large set of values, namely those that encode to 0 and negative integers that encode to an odd number.

Practical Implications. A current mitigation against replay attacks is the use of TLS. This ensures that an adversary cannot replay a captured packet over HTTPS. Therefore, the attack is relevant under the E2E and MitM threat models. Note that the adversary is unable to read the contents of the message unless the adversary is a member of the chat in question. Secondly, when a user deletes the LINE application (or the application storage) from their device, it forces their device to generate a new key pair upon reinstallation or reauthentication, as the old keys are deleted. Hence, old messages are encrypted under an obsolete key, thwarting attempts at replaying these specific messages.

For a one-to-one messaging scenario, the issue has the immediate consequence that the malicious server can replay any and all messages endlessly. In a group messaging scenario, key rotation happens whenever members are added to or removed from the group. Thus, the malicious server can replay any and all messages originating from any user within a group, as long as the group configuration has not changed since the original message was sent, i.e., the list of messages that can be replayed *resets* when the group changes, assuming key rotation is correctly performed.

In a one-to-one messaging scenario, the MitM can be mounted on either party. The adversary can swap the E2EE contents of a legitimate message with those of an old message, altering the communication in real time. Alternatively, the adversary can craft new packets using a captured E2EE packet, creating communication that the parties did not initiate.

In a group messaging scenario, the adversary must mount the MitM attack on all parties to make the communications look legitimate. Apart from that, the attack strategy is identical to the one-to-one scenario.

[6]It appears the server deletes its cache of counter values for each party once per day.
[7]Note that while the encoding scheme is cyclic, the pattern eventually deviates from the expected pattern outlined in Table 2.

Value	Encoded Bytes (Uint8Array)
0	**0**
-1	**1**
1	2
-2	**3**
2	4
560493232126 (522 GiB - 2)	252, 255, 255, 255, 15
560493232127 (522 GiB - 1)	254, 255, 255, 255, 15
560493232128 (522 GiB)	**255**, 255, 255, 255, 15
560493232129 (522 GiB + 1)	**253**, 255, 255, 255, 15
560493232130 (522 GiB + 2)	**251**, 255, 255, 255, 15
-560493232126 (522 GiB - 2)	**251**, 255, 255, 255, 15
-560493232127 (522 GiB - 1)	**253**, 255, 255, 255, 15
-560493232128 (522 GiB)	**255**, 255, 255, 255, 15
-560493232129 (522 GiB + 1)	254, 255, 255, 255, 15
-560493232130 (522 GiB + 2)	252, 255, 255, 255, 15
2 · 560493232128 (522 GiB)	**0**
-2 · 560493232128 (522 GiB)	**0**
NaN	**0**
MAX_VALUE	**0**
MIN_VALUE	**0**
NEGATIVE_INFINITY	**0**
POSITIVE_INFINITY	**0**

Table 2: Thrift encoding of counter values. Values in bold obtain successful replays.

4.2 Message Reordering and Blocking

A second consequence of the stateless design is that the ordering of messages becomes the server's responsibility. The server can manipulate timestamp generation and message arrivals, thereby influencing the semantics of any conversation. As these values are unauthenticated metadata, the MitM adversary holds equivalent capabilities.

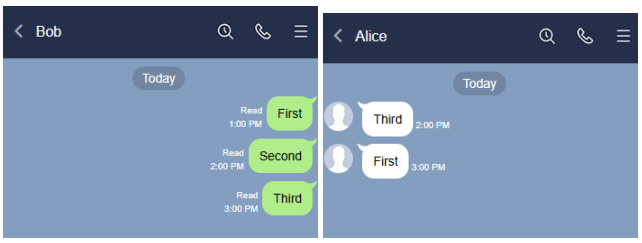


Figure 3: A MitM adversary blocks and reorders messages.

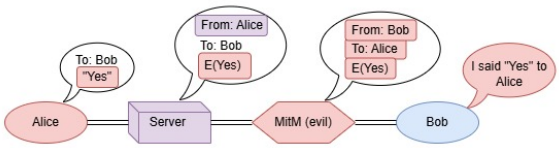


Figure 4: An impersonation attack under the threat model of a Malicious User colluding with the MitM adversary. Alice prepares a malicious message with honest metadata. The server infers the sender. The MitM maliciously alters the metadata.

The recipient device orders the messages based on server-side timestamps. Even though the sender includes a local counter to all outgoing messages, this counter is not used for determining the ordering of messages, as the server alters the metadata value in transit. An E2E adversary can thus arbitrarily order messages for each recipient client. Furthermore, the recipient does not detect message loss. Fig. 3 demonstrates how the user would see read receipts on all messages, even though one was never delivered.

On a practical level, the server or the MitM adversary can inject past messages and reorder or drop current messages, manipulating the semantics of the conversation, although the contents are unavailable to them.

4.3 Impersonation Attacks

All encryption and authentication operations in LSv2 are performed with shared secrets and public keys available to all chat members. Therefore, any chat member can create valid messages on behalf of others within the same chat, and there can be no guarantee as to the origin of a message. This is true for both one-to-one and group messaging. The server designates the sender of a message by overwriting the source field within the metadata of each message.

In other words, Alice can craft a valid message from Bob and send it to the server. Under the E2E threat model, the colluding server can modify the message. Under the MitM threat model, the message must be modified in transit from the server to Bob, as shown in Fig. 4. In either case, the correct alterations to the message will ensure that the arriving message will be displayed as if Bob were the legitimate author. The vulnerability is also present within group chats, as all members can compute the key material necessary to impersonate any other group member. The symmetric keys used for encryption and authentication do not protect against manipulating the sender and recipient fields within a message, making it possible to craft a message that is a cryptographically valid message from another party. The practicality of the attack boils down to routing, server-side mitigations, and client-side mitigations. Under the same threat model, the attack can similarly be performed through intercepting an honest message from Bob to a group and manipulating it to contain plaintext of the adversary's choosing, without Bob discovering the attack. Consequently, the adversary is thus capable of delivering individually tailored message contents to each group member, without any of them being able to detect the attack.

The Flaw. Let us revisit the cryptographic operations within LS that allow for an impersonation attack. Firstly, the parties compute a common secret through ECDH. In a one-to-one scenario, it is derived as follows:

$$Secret = ECDH(pk_{Alice}, sk_{Bob}) = ECDH(pk_{Bob}, sk_{Alice})$$

In a group scenario, it is instead derived as follows:

$$Secret = ECDH(pk_{Sender}, sk_{Group})$$

In either scenario, the secret can be computed by all chat members, because it only depends on public information and private keys available locally. Any party can also generate a salt and nonce. The encryption key is deterministically generated from the shared secret, the salt, and a hardcoded string. Hence, we can assume without loss of generality that the following values are pre-defined before a message is constructed: shared secret, salt, key, and nonce.

One-to-One Impersonation. To aid the reader, Alice's identifiers are highlighted in red, while Bob's identifiers are highlighted in blue. Alice would send an honest message to Bob as follows:

$$AAD = BobID||AliceID||AliceKeyID||BobKeyID||2||type$$
$$(C, tag) = AES_{GCM}(key, nonce, M, AAD)$$
$$Msg = 2||type||salt||C||tag||nonce||AliceKeyID||BobKeyID$$

An honest message from Bob to Alice would look different:

$$AAD = AliceID||BobID||BobKeyID||AliceKeyID||2||type$$
$$(C, tag) = AES_{GCM}(key, nonce, M, AAD)$$
$$Msg = 2||type||salt||C||tag||nonce||BobKeyID||AliceKeyID$$

The only difference between the two messages lies in the positions of account identifiers and KeyIDs, which are known to both parties.

In order to impersonate Bob, Alice must create a message that appears to originate from Bob[8], and get it routed to Bob's device. This can be done through colluding with the E2E adversary or mounting a MitM attack on Bob. First, Alice maliciously crafts a message in which the AAD field is identical to a message from Bob to Alice. However, the final message object keeps the positions of the KeyIDs intact, making the message fields indistinguishable from a valid message from Alice, making the server forward the message to Bob.

$$AAD = AliceID||BobID||BobKeyID||AliceKeyID||2||type$$
$$(C, tag) = AES_{GCM}(key, nonce, M, AAD)$$
$$Msg = 2||type||salt||C||tag||nonce||AliceKeyID||BobKeyID$$

In transit, the colluding E2E or MitM adversary can swap the KeyIDs to make it indistinguishable to how the message would have appeared if Bob were the legitimate sender. Note that Alice only maintains the original positioning of the KeyIDs to subvert a server mitigation. This allows the attack strategy to generalize to the MitM adversary, without the colluding MitM adversary needing to know Alice's key, as it simply swaps fields in the message sent. In the full E2E adversary setting, colluding with Alice and knowing her key, the server can simply send messages with arbitrary contents directly to Bob.

[8]Concretely, Bob's main device would believe Bob utilized a secondary device to send the message. Yet, Bob does not need to have a secondary device activated for impersonation to work.

Group Impersonation. Assume again that the secret, salt, key, nonce, and message are fixed. A legitimate message from Bob to a group is computed and formatted as follows:

$$AAD = GroupID||BobID||BobKeyID||GroupKeyID||2||type$$
$$(C, tag) = AES_{GCM}(key, nonce, M, AAD)$$
$$Msg = 2||type||salt||C||tag||nonce||BobKeyID||GroupKeyID$$

Once more, the message is not cryptographically bound to Bob, as the message is computed from values computable by all group members, particularly by the malicious Alice.

For Alice's impersonation of Bob to work in a group setting, the metadata must be modified to adjust the direction and authorship of the message before it arrives to Bob's client. Since group messages have the group identifier as the recipient and the *groupKeyId* as the *recipKeyId*, no change is needed for these fields. Analogously to one-to-one impersonation, the remaining clients require only slight modifications. In summary, the necessary modifications for impersonation attacks are relatively simple, and the resulting message is indistinguishable from a legitimate message sent by the victim.

To carry out the attack, we used two local LINEJS clients with accounts named Alice and Bob. Alice prepared the message as outlined above. To mimic a compromised server, we captured the packet before it was received by Bob's client and performed the defined manipulations. From Bob's perspective, the message was indistinguishable from what he would have received from the server if he had been the legitimate sender of the message.

Under the E2E assumption, the attack is simple and scales well to larger groups. The MitM threat model suffers from scalability issues within group chats, as the adversary must mount MitM attacks on every group member whom they wish to receive the forged messages, as server-side mitigations (which an E2E adversary could trivially disable) blocked our attempts to send the modified messages through on behalf of another user. Note that the attack is trivial against a group with 50+ members, as LS is disabled [13].

4.4 Forging Read Receipts

A read receipt is a special type of message sent from one party to another to indicate which messages have been read. LINE implements read receipts as simple server-appended UNIX timestamps on each message. This means that an E2E adversary can tamper with receipts, achieving the capability of blocking messages towards a victim, and subsequently forging a read receipt on their behalf. Thus, the server can deliver messages to a subset of group members while the message will appear to have been read by all. Such behavior may be interpreted as proof of the victim's liveness, (even in situations where the adversary has no access to any of the victims devices or credentials) and could be miscontrued as implicit compliance with the contents of the ongoing conversation. Finally, this allows the server to forge read receipts on messages it has blocked, tricking the sender into thinking the message was not only sent but also received and read.

4.5 Putting All Together: Stealthy Chat Attacks

Under the threat model of a malicious group member colluding with the E2E adversary, it is feasible to craft full conversations stealthily. A malicious server can manipulate metadata at will, drop messages for certain users, and forge read receipts. The malicious group member (or the server) can add a metadata status field which disables recipient notifications for that message. From these, one can build a large-scale, high-impact impersonation attack that crafts full conversations for any subset of group members.

This would allow the server to cause each group member to see a full unique conversation with arbitrary messages from any other group members, complete with read receipts for their own "sent" messages. Although such an attack could be easily detected if group members make contact through a separate communication channel and compare chat logs, it is easy to imagine a scenario where controlling a well-chosen group chat so completely could cause harm. Especially if the users did not consider this to be possible as they were under the belief that the chat was still securely end-to-end encrypted.

A more subtle attack could involve the adversary simply sending messages on behalf of a user to the rest of the group chat. From the perspective of other users, these messages would be seemingly indistinguishable from legitimate messages, even when looked at from the victims own device. This would allow the adversary to plant incriminating evidence, create an excuse for law enforcement to investigate or detain a person of interest, or otherwise put the victim in a vulnerable or dangerous position.

Because group key rotations are handled server-side without cryptographic binding to group membership, we observe that malicious users can trigger illegitimate key rotations to cause denial-of-service, maintain access after removal, or even gain illegitimate group membership. Finally, the reuse of group keys across communication types (e.g., VoIP) exposes a potential avenue for passive eavesdropping on group calls.

4.6 Plaintext Leakage Attacks

In our research we also identified two quality-of-life features which, in LINE's default configuration, can potentially leak information about the contents of a private chat to the LINE server.

Sticker Leakage. When users of LINE write text messages, the application continuously recommends icons and stickers related to the contents of what the user is typing. This is a popular feature among users, with the option for users to create and sell their own custom sticker packs. As of 2019, there were already over 4.9 million sticker sets available for purchase.[9] Despite the fact that many users routinely send stickers as standalone messages, using them in place of text to convey important information [33], stickers sent in LINE chats are *not* covered under LS.[10]

Beyond this, we also found that the sticker-preview function can leak information about the contents of text messages. The app does not download all possible sticker suggestions in advance. Instead, as a user is typing, if the app wants to suggest a sticker it does not already have saved in its cache, the sticker will be fetched from a remote server through a unique ID placed within the URL. The returned icon or sticker relates to the contents of the message currently being written. For example, the first time a user starts typing the word "money", the app would fetch previews for the stickers in Fig. 5.

[9] https://www.linecorp.com/en/pr/news/en/2019/2699
[10] https://help.line.me/line/ios/?contentId=50000087

Figure 5: Sticker previews rendered when user types *money*.

The leakage of un-cached stickers takes place in three stages, all of which allow an adversary to determine parts of the contents of a conversation in real-time. To increase the effective leakage, an adversary can utilize their LINE application to accurately determine which stickers and words are correlated:

(i) The first leakage happens while the victim is composing a message. The application constantly looks for appropriate stickers, each of which perform a lookup to the server if they are not cached within the app. Thus, all new users will perform a massive amount of lookups until their app has cached the corresponding stickers.
(ii) The second leakage is more precise and happens when the user clicks a recommended sticker. In that case, the sticker is enlarged and its corresponding animation is downloaded from the server and played as a preview. Due to the layout of the preview UI, two related sticker animation previews are simultaneously downloaded.
(iii) The third leakage happens when the user finally sends an animation, at which point the metadata for the sticker package is downloaded from the server.

We would like to highlight that (ii) can leak when a user *considers* sending a sticker for the first time (when they tap it to select, but before they decide to send,) even if they change their mind and never actually send the sticker. Likewise, (i) leaks stickers related to any keywords typed into the message field *as they are typed*, even if the user then deletes them without sending.

LY Corporation claims in [13] that the sticker lookup feature is *anonymous*. While it is true that the user's authentication token is not present in the lookup, the user-agent and language fields form an identifier that, although not necessarily unique, should aid in tracking specific devices. Specifically, LINE formats user-agent headers with three partial entries, e.g., `LINE/2025.415.1834 CFNetwork/3826.400.120 Darwin/24.3.0`. These fields represent the application version, the networking framework version, and the operating system version, respectively. The language field contains entries of the form: `language-region da, en-gb;q=0.8`, denoting the preferred languages and a preference score. We argue that these values, in addition to classic TLS fingerprinting techniques which we will not discuss in depth here, constitute enough of an identifier to substantially challenge the claim of anonymity.

The granularity of the leakage (separate identifiable lookups on the first appearance, first highlight and first send,) combined with the fact that sticker messages are not encrypted with LS, means that the server may see 3 *anonymous* sticker lookup requests for some sticker, then see an identifiable plaintext sending of the sticker, and potentially another *anonymous* lookup from the receiver, all in a very short period of time. This allows the server to easily tie heuristically correlated groups of lookups to specific users, compounded again if the users are using personal or rarely used stickers.

We acknowledge that the correlation between the recommended stickers and the written plaintext is not always crystal clear, making the exact leakage of the attack hard to quantify. Thus, we leave open the practicality of exploiting the sticker lookup leakage for determining the exact contents of a full conversation.

While we present the attack under the MitM or E2E threat model, we conjecture that the current leakage will also apply to a weaker adversary without the capability to subvert TLS. For example, imagine surveilling a user who is sending a text message of the same length every 10 minutes from a new burner phone. Under the claims of LS, if we monitor their encrypted network traffic, we should not be able to distinguish if they are sending the same message as before or a different one. However, if one of the times they start typing, we see a large spike in TLS traffic from their device, this could signify that they typed something different than before, causing the app to download many new sticker previews.

Similarly, based on the fact that the URL of each sticker contains a unique identifier, and that the resulting stickers, animations, and sticker packages have varying lengths, it may be possible to leak metadata around which stickers were downloaded just from TLS metadata. We acknowledge that a number of things can thwart an attempt to perform such leakage through TLS, and we leave the practicality of this more general attack open.

URL Preview Leakage. When a user sends a URL in an encrypted chat, LINE will by default generate a preview of the webpage linked, and add this to the message.

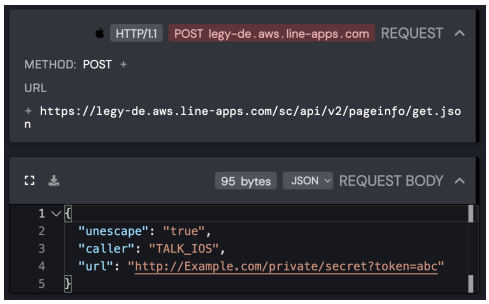


Figure 6: Data received by the server when a user sends URL through LINE.

Any URL sent within a chat will simultaneously create a request to the LINE server, in which the URL is the body of the request. The server renders a preview and returns it to the client.

As seen in Fig. 6, this includes not only the URL but all URL-encoded parameters, which could leak not only the website, but credentials, authentication tokens or other highly sensitive information to the server.

Once more, the contents of a chat are leaked, breaking confidentiality. The potential of leaking authentication tokens and private file-sharing links might lead to further breaches of confidentiality or pose a threat to the integrity of the leaked resources.

Documentation and Mitigations. Earlier work has also documented this behavior, correctly stating that the leakage defeats the purpose of E2EE, but no fix has been implemented [6]. As a response to our disclosure, LINE highlighted that they document these leakages in their encryption report [13], where they also note that, regarding URL previews, "*Users can opt-out of this function in the settings*".

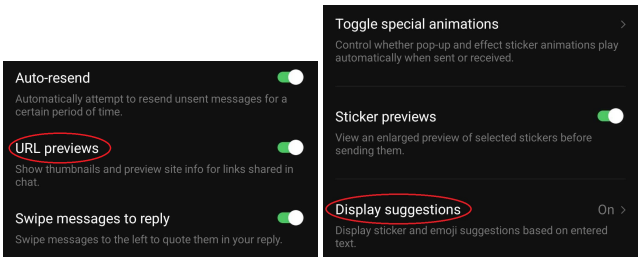


Figure 7: URL and Sticker Preview settings in LINE app.

We would like to highlight that this setting, seen in Fig. 7, is enabled by default for all users. The description gives no indication that it could have any security or privacy implications, and the toggle is not located in the `Privacy` section but under `Chat`, alongside UI options such as "*swipe to reply*".

As such, we believe even a highly technical and privacy-conscious user may not immediately understand the potential privacy implications of these toggles from how they are presented in LINE.

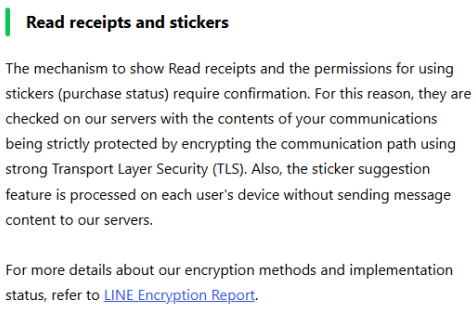


Figure 8: LINE Help Center claim regarding stickers.

Users in the LINE app who click through *"This chat is protected by Letter Sealing" -> "learn more about Letter Sealing"* are directed to the LINE Help Center webpage on LS[11].

While the Help Center page correctly warns the user that "*URL information is processed on our servers*" and guides on how concerned users could disable URL previews, the page's description of sticker suggestions (see Fig. 8) only claims that the sticker suggestion feature will not send "*message content*" to the LINE servers. If the user then clicks through to the 2024 LINE Encryption Report [13] that they will find the Statement:

"Sticker keyword: For LINE to be able to recommend a suitable sticker for the chat context, the LINE client will check the certain keywords in the messages. And if there's a match, it will be sent to the server anonymously."

Beyond our previous concerns about the practical traceability of the claimed *anonymous lookups*, we worry the Help Center, (the documentation the app directs users to) using the phrasing *"is processed on each user's device without sending message content to our servers."* could give some users a false sense of security that sticker lookups do not send any sensitive data based on message content to the server. Uninformed users may not realize that sticker lookups will in reality sometimes send traffic to the server which is highly correlated with the users' typed messages.

4.7 Summary of the Findings

As a result of the discovered vulnerabilities and practical attacks, LINE suffers from a lack of several crucial security properties of modern E2EE [5, 28]. Notably, only confidentiality is partially reached in the MitM and E2E threat models. However, one cannot effectively trust confidentiality without authentication and integrity, as both the source and the encrypted contents themselves are insufficiently protected.

The sole key material employed within LS is long-term keys. Therefore, there cannot exist a guarantee of forward secrecy, as the notion requires a distinction between key material at different points in time. Thus, LINE offers forward secrecy solely from client to server through its use of TLS. The E2E and MitM adversaries can thus bypass forward secrecy: a single private key leaked to the E2E or MitM adversary would forfeit all security guarantees for all of the user's communications. It takes but a single key to decrypt all of a user's past, present, and future messages. Earlier analysis of LS also has pointed to the use of digital signatures as a countermeasure to their uncovered attacks [26]. Origin authentication mechanisms would also solve the current impersonation attacks, therefore it is unclear what criteria LY Corporation used to design LSv2.

While the shift from LSv1 to LSv2 has ensured that the protocol is based on standardized cryptographic primitives, this gives no guarantee that LINE's composition of the operations results in a secure protocol. Even if a redesign of LS may solve several of our mentioned attacks against the protocol, it is not applied in several scenarios, such as groups of 50+ members, chats containing official accounts, and stickers, to name a few.

Aftermath. After the initial disclosure process, LINE has published an updated encryption report [15] and LSv2 specification [14]. While we did not notice any change in the specification related to our results, the encryption report elaborates on the risk of leakage. It justifies the design decision to perform server-side URL previews as a way to prevent phishing, and discusses mitigations to the sticker preview privacy issue using differential privacy and federated learning, although technical details are insufficient to reach a conclusion. The report further claims that "*This approach does not send plaintext keywords to the server*", which misses the point that sticker lookups leak client-side keywords without the need to send them directly to the server. In combination with their statement, we understand that LINE finds server-side countermeasures convincing and sufficient to mitigate our attacks in practice, which contradicts the threat model of a malicious LINE server.

[11] https://help.line.me/line/android/?contentId=50000087, 12/12/2025

5 Recommendations

We have two sets of recommendations, one for maintainers, designers and developers of the LINE protocol, and another for users.

5.1 For the LINE Developers

The current iteration of LS has been found to violate all of the fundamental security properties of E2EE. Therefore, we strongly recommend the adoption of the Signal protocol, as all of the uncovered attacks would become infeasible. Should LY Corporation choose to improve LS instead, the following key components must be carefully tended to: (i) The introduction of a protocol state or a uniqueness mechanism is essential to thwart replay attacks. The same mechanism could potentially be used to secure the ordering of messages. (ii) To combat impersonation attacks, it is imperative to introduce per-user secret key material in message authentication, such as signatures, or a logical separation between key material for sending and receiving, allowing users to cryptographically distinguish parties. (iii) The use of encryption protocols must be extended to stickers and read receipts, and the extensive list of exceptions for LS should be pruned considerably. (iv) Key rotation for groups must contain a stronger cryptographic binding to the members and must ensure the integrity and correctness of the generated key material. These properties should prevent illegitimate and malicious key rotation, respectively. Recent work by Downling et al. [17] retrofits a Double Ratchet into LSv2 and may serve as a starting point.

A substantial mitigation against plaintext leakage is the current use of caching within the app. Once a sticker has been downloaded – either the recommendation, the corresponding animation, or the full sticker package – it is cached on the phone. However, we have seen repeated downloads of the same sticker package metadata in our weeks of testing the feature, which may allow continuous leakage. Thus, while the attack enjoys increased effectiveness against new users as well as users who continuously utilize new stickers, it appears to remain ongoing at some level for all users. Our main recommendation is two-fold: (i) Make the app download all sticker packages available to the user when the app is booted. This would remove the leakage. Should a client receive a sticker that is not cached, it should ask for the entire package to be downloaded, leaking only the package identifier. (ii) Letter Sealing should be enabled for stickers, to protect both confidentiality, but also to protect users against the server sending stickers on behalf of any user to any other user. Currently, if LS were to be updated, removing our general impersonation attacks outlined above, the lack of LS for stickers would still present a considerable threat to users.

URL previews in Signal are generated in a sandbox environment locally on the client device [31]. While this feature is not strictly part of the Signal protocol, it adequately protects users from the leakage seen in the current LINE implementation. We acknowledge that Signal's solution presents a range of different security challenges, hence, we are hesitant to recommend that LINE should engineer a similar solution. Otherwise, the URL preview feature could be disabled by default and present a warning to the user once toggled.

Finally, LINE should include strong mitigations against MitM attacks consistently across operating systems. The current setup required to mount MitM attacks on an unlocked iOS device can be performed within minutes.

5.2 For LINE users

For LINE users, there are some steps which can be taken to immediately improve their personal security, depending on their personal risk profile.

Passive E2E Adversary. We start with users who wish to maintain privacy against a passive E2E adversary, that is an adversary who has gained (or been granted) access to eavesdrop on either the TLS session to the server, or eavesdrop on the server itself.

Disabling Sticker previews prevents potentially leaking contents the user is typing via sticker lookups to any such adversary. This also prevents potential leakage of information to third party network adversaries via patterns in the encrypted TLS traffic to the server, by preventing extra TLS requests from being sent when you type words for the first time which trigger new sticker previews.

Disabling URL previews prevents leaking any plaintext URLs sent in the chat in plaintext to any such adversary, as seen in Fig. 6. Beyond this, users should also be aware that stickers themselves are not encrypted, meaning any stickers sent will be received in plaintext by such an adversary.

As such, if a user wishes to ensure the contents of a given chat remain fully confidential against any potential eavesdropping E2E adversary, they should disable URL and Sticker previews, as seen in Fig. 7, and refrain from using plaintext messaging options in that chat, such as stickers.

Active E2E Adversary. It is much more difficult for an end user to mitigate the danger of an E2E adversary, who will actively interfere with the user's network traffic, including blocking, modifying or creating new network traffic.

If users follow the guidance above, confidentiality of individual messages may still be maintained. However, the user should be aware that if targeted by such an adversary, they cannot rely on the integrity or authentication of any messages they send or receive.

Such an adversary could block any sent message without alerting sender or receiver, (still showing the sender a *"read receipt"*) send arbitrary stickers to any party at will, or replay previous messages arbitrarily many times. Furthermore, if the adversary compromises (or collaborates with) any member of a chat or group chat, they may be able to send arbitrary messages to any subset of the group, on behalf of any other group member.

As such, in settings where a user wishes to communicate reliably and securely even in the presence of a potential active E2E adversary, the LINE protocol should not be used.

As previously discussed, the E2E adversary model is important because it captures not only when the owner of the server is dishonest, but also the threat of compromise by threat actors or nation states, or classified demands or coercion from law enforcement or intelligence agencies in any of the jurisdictions the hosting company operates in.

Some users may only worry about their data being harvested by a passive adversary who is recording every single message going through the server for later analysis. While others such as journalists, political activists or other "high risk" targets may worry about being directly targeted by an active E2E adversary attempting to silence, mislead, coerce or incriminate them.

We respect that some of the features we advise against using, such as the wide range of stickers suggested in conversation, are the very reasons some users choose to use LINE. It is up to the individual user to make their own judgment as to how they value functionality and convenience against the privacy of full E2EE, and we hope our findings lead to more informed user decisions.

6 Conclusion

Overall, our findings in Table 3 demonstrate that LSv2 fails to provide several core E2EE properties under realistic adversarial models. While AES-GCM and Curve25519 are sound primitives, their composition in a stateless, unauthenticated protocol undermines the intended security. Mitigations should include explicit message authentication codes over protocol metadata, cryptographically enforced counters, per-message key evolution (ratcheting), and authenticated key-confirmation. Adoption of a modern construction such as the Signal Double-Ratchet or Sender Keys [7] (per-sender authentication keys) would resolve most identified issues.

Target	Attack	Threat model
All chats	Replay	E2E or MitM
	Reordering	E2E or MitM
	Blocking	E2E or MitM
	Forge receipts	E2E or MitM
	Sticker leakage	E2E or MitM
	URL leakage	E2E or MitM
One-to-one chats	Impersonation	Malicious User w/ E2E
	Impersonation	Malicious User w/ MitM
Group chats	Impersonation	Malicious Member w/ E2E
	Impersonation	Malicious Member w/ MitM*

Table 3: A summary of practically exploitable attacks, their threat models, and references to the relevant sections. *MitM must be mounted on all targeted group chat members.

Our security analysis arrives again at a recurrent point in the cryptographic literature, which bears repeating: deploying custom-made cryptographic protocols has the inherent risk of repeating design flaws that have been already successfully resolved in robust and standard ways in other protocols. Following previous work [34], our security analysis also teaches some valuable lessons to the real-world cryptographic protocol designer/implementer, beyond the usual maxims of the field. We discuss these lessons below.

Include as much as possible in the scope of E2EE. All types of communication contents should be considered sensitive, because it is hard to anticipate how users will engage with the application and the design should assume the worst case. Stickers are completely unencrypted despite being a core part of LINE's value proposition, and users regularly sending stickers as standalone messages to convey information [33]. Bringing stickers, and other alternative forms of chat content into the E2EE layer provided by LS should be a high priority. Metadata should be protected as well, as the manipulation of metadata can amplify the impact of other attacks. For example, the absence of checking message counters, sender identity or read receipts can be used to dramatically influence the semantics of communication. This lesson can be learned as an application of the proactive security advice from [34].

Security-driven design. Secure messaging is difficult! Messengers hoping to live up to strong security guarantees should be designed from the ground up with security in mind. The gradual improvement in encryption across message and media types detailed in the Line Encryption Report [13] is a strong indication of trying to build security into an already existing protocol. When trying to retroactively add security to an existing protocol, it is important to start by designing a provably secure final protocol, capturing all the desired security properties, and have every update be another step towards that goal. Is it not sufficient to just choose well-implemented cryptographic primitives or reuse secure protocols in different contexts [34]. If each new version is driven by another self-contained quick update to fix the current most pressing vulnerability, it can lead to design decisions like adding an entirely separate unauthenticated counter to the metadata, instead of using the counter authenticated by GCM; or seemingly unimportant elements such as read receipts being completely ignored.

Finally, the approach of gradual and asymmetrical improvements across products can lead to a complicated patchwork of security claims, with varying security models and different claims for different features. What seems like a reasonable trade-off to designers behind the scenes with a deep understanding of the trade-offs involved in modifying the existing infrastructure may not be at all obvious to an end user, who simply sees that messages are E2EE, and has no reason to expect this does not generalize to stickers. While many of these tradeoffs are documented in the LINE encryption report, settings and explanations in the app itself seem to focus solely on usability, or note the lack of encryption only by omission.

Beware of evolving threat models. While business choices such as priotitizing sticker functionality over encrypted stickers could be expected for an average chat application, LINE is not like any other chat application.

For example, *"Out of Taiwan's population of 23.42 million, LINE's monthly active users have reached 22 million as of 2024"*[12], the Taiwanese government gives government LINE accounts as a primary point of contact for emergency information during natural disasters[13] and government officials LINE accounts are so integrated into the government communication network that LINE is launching bespoke blue badges to formally verify accounts belonging to goverment staff[14].

In countries like Taiwan, while LINE may have started as a chat app, it has over time become closer to critical infrastructure. This kind of societal and governmental dependence not only raises consumers expectations of security, making clear communication about insecure elements more important, but also makes it a clear target for malicious actors. This evolution in LINE's role carries with it a natural evolution of the required threat model it needs to respect. As such, we believe ensuring full E2EE, to ensure privacy and security even in the face of a compromised server, is critical.

Acknowledgements. We thank Nadim Kobeissi for useful discussions and feedback on earlier versions of this paper; the LINEJS developers for help with the framework; the LY Corporation CSIRT members and LS designers for a constructive disclosure process.

[12] https://www.lycorp.co.jp/en/story/20250226/linetaiwan.html
[13] https://www.gov.tw/News_Content_37_764076
[14] https://moda.gov.tw/press/press-releases/17092

References

[1] Martin R. Albrecht, Sofia Celi, Benjamin Dowling, and Daniel Jones. 2023. Practically-exploitable Cryptographic Vulnerabilities in Matrix. In *SP*. IEEE, 164–181.

[2] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, Eyal Ronen, and Igors Stepanovs. 2025. Analysis of the Telegram Key Exchange. In *EUROCRYPT (8) (Lecture Notes in Computer Science)*. Springer, 212–241.

[3] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. 2026. Four Attacks and a Proof for Telegram. *J. Cryptol.* 39, 1 (2026), 10.

[4] Martin R. Albrecht and Kenneth G. Paterson. 2024. Analyzing Cryptography in the Wild: A Retrospective. *IEEE Secur. Priv.* 22, 6 (2024), 12–18.

[5] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT (1) (Lecture Notes in Computer Science)*. Springer, 129–158.

[6] Ionut Arghire. 2020. Link Previews in Chat Apps Pose Privacy, Security Issues: Researchers. https://www.securityweek.com/link-previews-chat-apps-pose-privacy-security-issues-researchers/.

[7] David Balbás, Daniel Collins, and Phillip Gajland. 2023. WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs. In *ASIACRYPT (5) (Lecture Notes in Computer Science)*. Springer, 307–341.

[8] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography (Lecture Notes in Computer Science)*. Springer, 207–228.

[9] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2020. A Formal Security Analysis of the Signal Messaging Protocol. *J. Cryptol.* 33, 4 (2020), 1914–1983.

[10] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 2018. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *CCS*. ACM, 1802–1819.

[11] LY Corporation. 2020. Smart city transformation of local governments (in Japanese). https://linegov.com/service/partner.php.

[12] LINE Corporation. 2021. LINE Encryption Overview. https://scdn.line-apps.com/stf/linecorp/en/csr/line-encryption-whitepaper-ver2.1.pdf.

[13] LY Corporation. 2024. LINE Encryption Report (2024). https://www.lycorp.co.jp/en/privacy-security/security/transparency/encryption-report/2024/.

[14] LINE Corporation. 2025. LINE Encryption Overview. https://www.lycorp.co.jp/en/privacy-security/line-encryption-whitepaper-ver2.2.pdf.

[15] LY Corporation. 2025. LINE Encryption Report (2025). https://www.lycorp.co.jp/en/privacy-security/security/transparency/encryption-report/2025/.

[16] Statista Research Department. 2025. Share of people who use LINE in Japan in fiscal year 2024, by age group. https://www.statista.com/statistics/1077541/japan-line-penetration-rate-by-age-group/.

[17] Benjamin Dowling, Prosanta Gope, Mehr U Nisa, and Bhagya Wimalasiri. 2026. Drawing the LINE: Cryptographic Analysis and Security Improvements for the LINE E2EE Protocol. arXiv:2602.18370 [cs.CR] https://arxiv.org/abs/2602.18370

[18] Antonio M. Espinoza, William J. Tolley, Jedidiah R. Crandall, Masashi Crete-Nishihata, and Andrew Hilts. 2017. Alice and Bob, who the FOCI are they?: Analysis of end-to-end encryption in the LINE messaging application. In *FOCI @ USENIX Security Symposium*. USENIX Association.

[19] SB Telecom Europe. 2024. 2024 Social Media in Japan. https://www.digitalmarketingforasia.com/wp-content/uploads/2024/07/2024-Social-Media-in-Japan.pdf.

[20] The Apache Software Foundation. 2024. Thrift Compact protocol encoding. https://github.com/apache/thrift/blob/master/doc/specs/thrift-compact-protocol.md.

[21] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. 2016. Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage. In *USENIX Security Symposium*. USENIX Association, 655–672.

[22] Daniel Kahn Gillmor, Niels ten Oever, and avri doria. 2015. *Human Rights Protocol Considerations Glossary*. Internet-Draft. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-dkg-hrpc-glossary/01/ Work in Progress.

[23] Britta Hale and Chelsea Komlo. 2022. On End-to-End Encryption. *IACR Cryptol. ePrint Arch.* 2022 (2022), 449.

[24] Takanori Isobe, Ryoma Ito, and Kazuhiko Minematsu. 2023. Cryptanalysis on End-to-End Encryption Schemes of Communication Tools and Its Research Trend. *J. Inf. Process.* 31 (2023), 523–536.

[25] Takanori Isobe and Kazuhiko Minematsu. 2018. Breaking Message Integrity of an End-to-End Encryption Scheme of LINE. In *ESORICS (2) (Lecture Notes in Computer Science)*. Springer, 249–268.

[26] Takanori Isobe and Kazuhiko Minematsu. 2020. Security Analysis and Countermeasures of an End-to-End Encryption Scheme of LINE. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* 103-A, 1 (2020), 313–324.

[27] Jakob Jakobsen and Claudio Orlandi. 2016. On the CCA (in)Security of MTProto. In *SPSM@CCS*. ACM, 113–116.

[28] Mallory Knodel, Sofia Celi, Olaf Kolkman, and Gurshabad Grover. 2024. Definition of End-to-end Encryption. Cryptology ePrint Archive, Paper 2024/2085. https://eprint.iacr.org/2024/2085

[29] LINE. 2016. Letter Sealing gets enhanced! https://line-en-official.weblog.to/archives/1060089042.html.

[30] Felix Linker, Ralf Sasse, and David A. Basin. 2025. A Formal Analysis of Apple's iMessage PQ3 Protocol. In *USENIX Security Symposium*. USENIX Association, 5015–5034.

[31] Joshua Lund. 2019. I link therefore I am. https://signal.org/blog/i-link-therefore-i-am/.

[32] NIST. 2015. end-to-end encryption. https://csrc.nist.gov/glossary/term/end_to_end_encryption.

[33] Michaela Oberwinkler. 2023. Digital stickers in Japanese LINE communication. *The Interdisciplinary Journal of Image Sciences* 38 (2023), 238–262. Issue 2. doi:10.1453/1614-0885-2-2023-15758

[34] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. 2023. Three Lessons From Threema: Analysis of a Secure Messenger. In *USENIX Security Symposium*. USENIX Association, 1289–1306.

[35] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/. Revision 1, 2016-11-20.

[36] Leonardo Pires. 2024. LINE Revenue and Growth Statistics. https://usesignhouse.com/blog/line-stats/.

[37] World Population Review. 2025. LINE Users by Country 2025. https://worldpopulationreview.com/country-rankings/line-users-by-country.

[38] Douglas Stebila. 2024. Security analysis of the iMessage PQ3 protocol. *IACR Cryptol. ePrint Arch.* 2024 (2024), 357.

A Protocol Format

We rely on LINEJS to interface with the LINE API. It implements a *self-bot*, allowing users to create automated clients that act on the user's behalf. We briefly describe how server messages are formatted in the current LINE implementation. The noteworthy fields are the following:

```
{
  type: "SEND_MESSAGE",
  reqSeq: -1,
  message: {
    from: SenderID,
    to: RecipID,
    contentMetadata: { e2eeVersion: "2" },
    chunks: [
      <salt>, <C || tag>, <nonce>,
      <senderKeyID>, <recipKeyID>
    ],
  }
}
```

The following are relevant for security, as they denote data from a cryptographic context or for the correct routing of messages:

- `from` and `to` specify the origin and the recipient accounts (or recipient group).
- The sequence number, `reqSeq`, could potentially protect against replays.
- `e2eeVersion` specifies the cryptographic protocol version, with LSv2 being used here.
- `type` is a server-appended field containing values such as "`SEND_MESSAGE`" or "`RECEIVE_MESSAGE`", presumably facilitating the use of multiple LINE clients.
- `chunks` contains the E2EE message.

In practice, `SenderID` and `RecipID` are strings prefixed by `u` followed by 32 seemingly random hexadecimal characters, e.g., `u1234567890abcdef1234567890abcdef`. The contents of `chunks` are five buffers containing hexadecimal byte sequences, e.g., `<Buffer 00 50 52 26>`. The `to` field is used by the server to determine which user to route the message to. From the sender's perspective, the `from` field is optional, and the server overwrites this field to contain the account identifier (*MID*) of the sender.