Three Lessons From Threema: Analysis of a Secure Messenger

Kenneth G. Paterson
Applied Cryptography Group, ETH Zurich

Matteo Scarlata
Applied Cryptography Group, ETH Zurich

Kien Tuong Truong
Applied Cryptography Group, ETH Zurich

Abstract

We provide an extensive cryptographic analysis of Threema, a Swiss-based encrypted messaging application with more than 10 million users and 7000 corporate customers. We present seven different attacks against the protocol in three different threat models. As one example, we present a cross-protocol attack which breaks authentication in Threema and which exploits the lack of proper key separation between different sub-protocols. As another, we demonstrate a compression-based side-channel attack that recovers users' long-term private keys through observation of the size of Threema encrypted backups. We discuss remediations for our attacks and draw three wider lessons for developers of secure protocols.

1 Introduction

Threema is a Swiss encrypted messaging application. It has more than 10 million users [33] and more than 7,000 on-premise customers [36]. It is among the top Android apps in Switzerland, Germany, Austria, Canada, and Australia in the "paid for" category [8]. It has undergone independent security audits [39, 38]. Along with Signal and Telegram, Threema was widely advertised as a more secure alternative to WhatsApp [57, 24, 58] in the wake of Facebook's acquisition of the company [24] and of the January 2021 change in WhatsApp Terms of Service [32, 62]. Threema is the messaging app of choice of the Swiss Government and of the Swiss Army [25], where its use is mandated for all official communications [60]. Threema is also used by German politicians, including the current Chancellor, Olaf Scholz [53]. The company behind Threema makes strong claims for its security [35] and advertises its Swiss, non-US jurisdiction as a virtue.[1]

In general, secure messaging protocols are designed to provide end-to-end encrypted communication between two (or more) parties. Modern designs like Signal achieve not only confidentiality, integrity, origin authentication and verification of correct ordering of the exchanged messages, but also:

- **session key security**: compromise of a session key or ephemeral values in one session should not affect the security of keys in other, parallel sessions;
- **fine-grained forward secrecy**: compromise of session keys in an on-going session should not affect security of earlier keys and exchanged messages in that session and compromise of long-term keys should not undermine the security of completed sessions; and
- **post-compromise security**: after a compromise of all key material, it should be possible to regain security after a protocol execution, provided the adversary is passive during that execution.

Such properties have been established for Signal for two-party messaging [14, 43, 15, 21].

At the same time, the current landscape of messaging apps includes a wide variety of solutions, with different trade-offs with respect to centralization and federation, company jurisdiction, ease of use, openness of design and code, and security guarantees provided by the underlying cryptographic protocols. The absence of a standard for secure messaging means that many of the contenders feature bespoke cryptographic protocols with unclear security guarantees [7, 4, 6]. Indeed, these security guarantees are often much weaker than those provided by Signal, which can be seen as a providing a "gold standard" for secure messaging.

1.1 Our Contributions

We provide the first in-depth security analysis of the cryptography used in Threema, focussing on its protocols for securing end-to-end (E2E) and client-to-server (C2S) communications, for user registration, and for backing up users' private keys. We also examine the interactions between these protocols.

Our analysis uncovers seven different attacks against Threema, in three different threat models. One of the attacks

[1]Quoting from [33]: "Threema is 100% Swiss Made, hosts its own servers in Switzerland, and, unlike US services (which are subject to the CLOUD Act, for example), it is fully GDPR-compliant."

was discovered before us by Krebs and disclosed to Threema. It was patched in December 2021 [34]. We include it here for completeness. We produced proof-of-concept implementations for most of the attacks to verify that they work.

We summarise the threat models and attacks next.

In the "external actor" threat model we consider attacks by outsiders with no special access to Threema servers or clients. Here, we found two attacks:

1. An adversary who is able to compromise a single ephemeral value used by a client in the C2S protocol is able to permanently impersonate that client to the server, and through this, access the metadata for all E2E-protected messages. This strongly violates the expected security properties of a well-designed authenticated key exchange protocol.

2. An adversary is able to register the server's public key as a user public key and, by tricking the victim into sending a single carefully crafted message in the E2E protocol, can obtain a value that enables it to again permanently impersonate the victim to the server in the C2S protocol. This is a cross-protocol attack.

In the "compromised Threema" threat model we consider attacks by an adversary who has gained access to Threema servers. Such access should not enable breaking the end-to-end security guarantees. Here, we describe three attacks:

3. Message reordering/deletion attack: due to peculiarities in nonce-handling in the E2E protocol, the adversary is able to arbitrarily and undetectably reorder and/or delete messages sent from any client to any other client.

4. Replay and reflection attacks: again, due to how nonces are handled in the E2E protocol, Threema clients must keep a local database of all nonces used in sending and receiving messages. This database is deleted when a user reinstalls the app or changes devices. After such an action, the adversary can replay old messages. Because of poor key separation, the adversary can also send back *to* user Alice any old messages originally sent *by* Alice.

5. Kompromat attack: during registration, a user's client proves possession of its private key by encrypting a server-selected message to a server-selected public key using the E2E encryption mechanism. This process is invisible to the user. This allows a malicious server to create "Kompromat": a potentially incriminating message that can be delivered at any later time to a target user. This is the attack that had already been discovered and patched [34] and that we rediscovered.

In the "compelled access" threat model, we consider attacks by actors with direct access to a user's device, for example law enforcement officers or border guards. We present two attacks in this setting:

6. A trivial attack exploiting a Threema feature allowing users to easily export a backup of their long-term private key protected under a chosen password. Because of Threema's design, this key can be used to clone a Threema account and, through this, silently access all the user's future messages. In conjunction with a compromised Threema server, all old messages can be accessed too. This attack is only possible when the app and mobile device are both unlocked.

7. A more sophisticated attack exploiting one of Threema's backup mechanisms; this attack does by default not require an unlocked app (but does need an unlocked mobile device). It exploits Threema's use of a "compress-then-encrypt" mechanism when creating the backup in combination with the ability of the adversary to inject chosen strings into the backup file through Threema's use of nicknames. By observing the size of the backup file over many operations with chosen nicknames, the adversary can recover the user's private key, with the same consequences as above.

In totality, our attacks seriously undermine Threema's security claims. All the attacks can be mitigated, but in some cases, a major redesign is needed. We discuss appropriate mitigations after presenting the attacks.

We also draw three wider lessons for developers of secure protocols: using secure libraries for cryptographic primitives does not on its own lead to a secure protocol design; beware of cross-protocol interactions; and adopt a proactive rather than reactive approach to secure protocol design.

1.2 Related Work

An independent early analysis of the Threema client-to-server protocol, dating back to the closed-source Threema version 1.3, was carried out by Ahrens [2], and was the basis for an open-source implementation by Berger [10]. We analyse the current version (version 4.831) of the Threema Android application, and base our analysis of the protocol on the 2020 open-source release of the Threema app [30] and the Threema security whitepaper [31].

Rösler et al. [55] considered the end-to-end security of group chats in various secure instant messaging services, including Threema. They observed the lack of forward secrecy and post-compromise security for Threema group chats, and described two attacks: a replay attack on group messages, and leakage of group membership information to non-members. Both attacks have since been patched by Threema.

Cremers et al. [18] investigated the security of Threema (and other apps) in a black-box post-compromise setting, by cloning a victim device and observing whether the clone retains access to exchanged messages. The authors note that Threema prevents the clone from accessing messages exchanged while the clone is offline. While this result holds in a black-box setting, the weaknesses we uncover in Threema's client-to-server protocol enable even an adversary with limited technical proficiency to intercept messages if given access to the full cloned state of the victim.

Threema itself commissioned two security audits [39, 38]. Both audits focused on the Android and iOS apps for Threema, and do not touch on its cryptographic security beyond confirming that the implementation of the protocol matches the description provided in the Threema whitepaper [31].

Soatok [59] highlighted Threema's lack of forward secrecy and transcript consistency in groups as deficiencies of the app. The lack of transcript consistency is not yet resolved by Threema, and, as [59] argues, can be combined with the lack of robustness of the XSalsa20-Poly1305 AEAD cipher to obtain "Invisible Salamader"-style attacks [19, 3]: an attacker can send attachments in group chats that decrypt to different plaintexts for different chat participants.

1.3 Ethical Considerations

We only used the description of Threema in [31] and the client-side source code to conduct our analysis. We did not attempt any reverse-engineering of the server behaviour other than what could be observed passively through normal client interaction. We performed experiments only with test accounts under our full control; no other user accounts were targeted. We reduced as much as possible the number of messages sent to Threema servers, and we avoided sending malformed messages to servers so as to avoid triggering DoS attacks.

We did purchase a "Threema Gateway" account (with Threema ID `*LYTAAAS`) and registered a public key under that account identical to the Threema server public key (obviously, without knowing the corresponding private key). This step was necessary in order to validate one of our attacks (Section 3.2.2). It did not violate any terms or conditions of the service, so far as we could determine.

We disclosed the results in this paper to Threema on 03.10.2022, proposing a 90-day disclosure period. They acknowledged receipt also on 03.10.2022. On 13.10.2022, we met with Threema representatives in order to discuss mitigations for the attacks we found. On that occasion, we agreed on the details of the coordinated disclosure: a first batch of mitigations would be released in Q4 of 2022, while further mitigations would be released in Q1 of 2023, along with a post on Threema's blog. The first batch of mitigations were included in Threema 5.0 for both Android and iOS[2] on 29.11.2022. To further remediate the issues we reported, Threema included their custom protocol, Ibex, in the same release. The new protocol aims to provide forward secrecy at the E2E layer.[3] They also updated their whitepaper [31] on 15.12.2022 to add discussion of the forward security feature. We have not audited this new protocol. We agreed to publicly disclose our findings on 09.01.2023.

In one of our attacks, we leveraged a vulnerability in a library, Zip4j, used by Threema to create backup zip files. We disclosed our findings to the author of Zip4j on 08.10.2022, proposing a 60-day disclosure period. At the time of writing, the author has yet to acknowledge our email. The considerations above for the Threema disclosure apply here too.

1.4 Paper Structure

In Section 2 we introduce the Threeema architecture and provide a description of Threema's core cryptographic protocols and subsystems. Section 3 presents our attacks, grouped by threat model. We propose mitigations in Section 4. We discuss three wider lessons for secure protocol designers in Section 5.

2 Threema Architecture and Protocols

The design of Threema can be studied as the ensemble of several interoperable cryptographic protocols, all sharing the same user state and cryptographic primitives. The functional core of Threema consists of an end-to-end (E2E) protocol, which employs the long-term cryptographic secrets of the users to encrypt messages, and a client-to-server (C2S) protocol, a secure channel protocol that protects these messages in flight from clients to the Threema servers. Because of the composition of these two protocols, each message is wrapped in two layers of encryption: an inner end-to-end layer, and an outer client-to-server layer. We depict the composition in Fig. 1. Also relevant to our analysis and attacks are the registration protocol, data backup and contact discovery system.

We first describe the cryptographic primitives used by Threema and establish our notation. Then we analyze the E2E protocol, the C2S protocol, and the registration protocol. Finally, we briefly cover the backup mechanisms. For completeness, we also include a description of the contact discovery mechanism in Appendix A.

2.1 Cryptographic Primitives and Notation

The protocols used by Threema rely on standard cryptographic primitives. The main tool used is the `crypto_box` abstraction provided by the Networking and Cryptography Library (NaCl) [11] which consists of a Curve25519 Diffie-Hellman key agreement, followed by encryption with an Authenticated Encryption with Additional Data (AEAD) algorithm. The AEAD algorithm used by NaCl is the nonce-based XSalsa20-Poly1305.

Let $\mathsf{X25519}(\cdot)$ denote the NaCl function that takes a private key and a public key, and outputs a shared secret byte string. In NaCl, this string results from applying the $\mathsf{hsalsa20}$ key derivation function to the byte representation of the shared elliptic curve point and a fixed nonce. Internally, the $\mathsf{X25519}$

[2]https://threema.ch/en/whats-new

[3]https://threema.ch/en/blog/posts/ibex

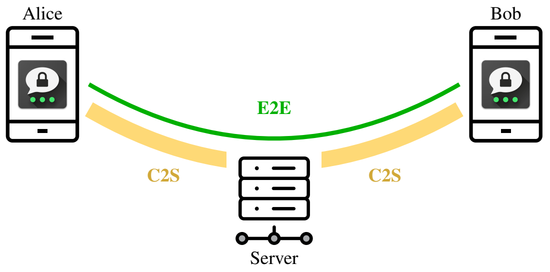


Figure 1: The composition of the E2E and C2S protocol. Each client establishes a secure channel with the server using the C2S protocol (in yellow) to send and receive E2E-encrypted messages from other users, which are relayed via the server (the connection in green).

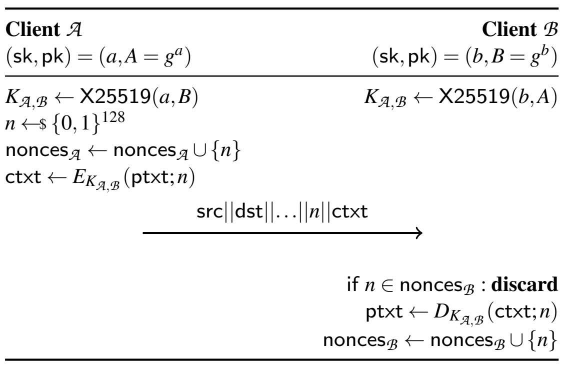

Client $\mathcal{A}$
$(\mathsf{sk}, \mathsf{pk}) = (a, A = g^a)$
Client $\mathcal{B}$
$(\mathsf{sk}, \mathsf{pk}) = (b, B = g^b)$
$K_{\mathcal{A},\mathcal{B}} \leftarrow \mathsf{X25519}(a, B)$
$K_{\mathcal{A},\mathcal{B}} \leftarrow \mathsf{X25519}(b, A)$
$n \leftarrow\!\!\$ \{0,1\}^{128}$
$\mathsf{nonces}_{\mathcal{A}} \leftarrow \mathsf{nonces}_{\mathcal{A}} \cup \{n\}$
$\mathsf{ctxt} \leftarrow E_{K_{\mathcal{A},\mathcal{B}}}(\mathsf{ptxt}; n)$
$\mathsf{src}||\mathsf{dst}||\ldots||n||\mathsf{ctxt}$
if $n \in \mathsf{nonces}_{\mathcal{B}}$: **discard**
$\mathsf{ptxt} \leftarrow D_{K_{\mathcal{A},\mathcal{B}}}(\mathsf{ctxt}; n)$
$\mathsf{nonces}_{\mathcal{B}} \leftarrow \mathsf{nonces}_{\mathcal{B}} \cup \{n\}$

Figure 2: User $\mathcal{A}$ sending ptxt to user $\mathcal{B}$ with the E2E protocol. The set $\mathsf{nonces}_{\mathcal{U}}$ represents the nonce database of user $\mathcal{U}$.

algorithm operates on the Curve25519 elliptic curve: the algorithm uses 32-byte scalars as private keys and computes the scalar multiplication of the chosen base point of the curve g with the private key to obtain a 32-byte public key. We will use multiplicative notation for such operations: e.g. Alice has a secret key a and a public key $A = g^a$ for some generator g. We denote by $(x, X) \leftarrow\!\!\$ \mathsf{KeyGen}()$ the key generation procedure of Curve25519, which outputs a private key x and a public key $X = g^x$. Let K be an AEAD key; we denote by $\mathsf{E}_K(m; n)$ the encryption of message m under key K using nonce n, and by $\mathsf{D}_K(m; n)$ the corresponding decryption operation. If the decryption fails, we assume that the algorithm returns a special value $\perp$. This may happen, for example, if the ciphertext has been tampered with, or if the wrong key was used to decrypt.

We denote by $k \leftarrow \mathsf{KDF}(K, \sigma, \tau)$ a Key Derivation Function taking some key material K, a salt σ, and a label τ as input, and producing key k as output. Threema uses a KDF based on the Blake2b hash function [56].

2.2 Threema End-to-End Protocol

The Threema E2E protocol is concerned with guaranteeing end-to-end security of messages. Every user $\mathcal{U}_i$ of the protocol has an alphanumeric 8-character identity $ID_{\mathcal{U}_i}$ (the *Threema ID*, or simply *ID*), a private key a_i, and a corresponding public key $U_i = g^{u_i}$. Messages sent from user $\mathcal{U}_i$ to user $\mathcal{U}_j$ are encrypted under $K_{\mathcal{U}_i,\mathcal{U}_j} = \mathsf{X25519}(u_i, U_j) = \mathsf{X25519}(u_j, U_i)$, a static Diffie-Hellman key. Note that the same key is derived by user $\mathcal{U}_i$ for $\mathcal{U}_j$ as the one from user $\mathcal{U}_j$ for $\mathcal{U}_i$: the key $K_{\mathcal{U}_i,\mathcal{U}_j}$ is bidirectional.

A message m is first serialized into a byte string, and prefixed with a byte that indicates the message type. The specific serialization method depends on the message itself. For instance, a *text message* would be directly encoded as a byte string representing the UTF-8 string, and prefixed with `0x01`. A random amount of PKCS7 padding, between 1 and 254 bytes, is appended to the message.The resulting plaintext `ptxt` is then encrypted under K using the AEAD algorithm with a random nonce, `nonce`, to obtain a ciphertext `ctxt`.

To create the final message packet, the following information is prepended to the ciphertext: the Threema ID of the sender `src`, and of the receiver `dst`, a random 8-byte message ID `msg-id`, the timestamp at which the message was sent `timestamp`, the nickname of the sender `src-nick`, the nonce `nonce` and an *optional* `metadata` value.

To construct this latter `metadata`, the app takes the message ID, the nickname of the sender and the timestamp, concatenates them, and encrypts the resulting plaintext under $\mathsf{KDF}(K, \text{"mm"}, \text{"3ma-csp"})$, using the same nonce value. Note that since the encryption of the metadata uses a different key, reuse of the same nonce does not lead to loss of security. However, since the presence of the metadata box is optional and it is not cryptographically bound to the ciphertext, it can always be stripped off and its length set to 0 by an adversary.

When sending an E2E message, the associated random nonce is stored in the device's local storage in order to avoid reusing it. If a generated nonce matches a nonce in the local storage, up to five attempts are made to sample a new nonce. When receiving a message, the app checks if the corresponding nonce has already been seen: if so, it rejects the message.If the nonce is otherwise fresh, the app tries to decrypt the message, verifying the MAC tag as part of the process. If successful, the app stores (a hashed representation of) the nonce. This storage of nonces is aimed at preventing trivial replay and reflection attacks (which would otherwise be possible since the key $K_{\mathcal{U}_i,\mathcal{U}_j}$ is bidirectional) and hedges against the possibility of a faulty randomness source. A pictorial representation of the E2E messages can be found in Fig. 3.

Remarkably, the E2E protocol does not provide any forward secrecy, since it only uses the long-term keys of the users, with no ephemeral values being exchanged. The simplicity of the key exchange part of the protocol, relying on a static Diffie-Hellman key exchange, makes it superfluous to study its security in traditional authenticated key exchange models such as eCK [44], ACCE [40] or MSKE [26]. Indeed

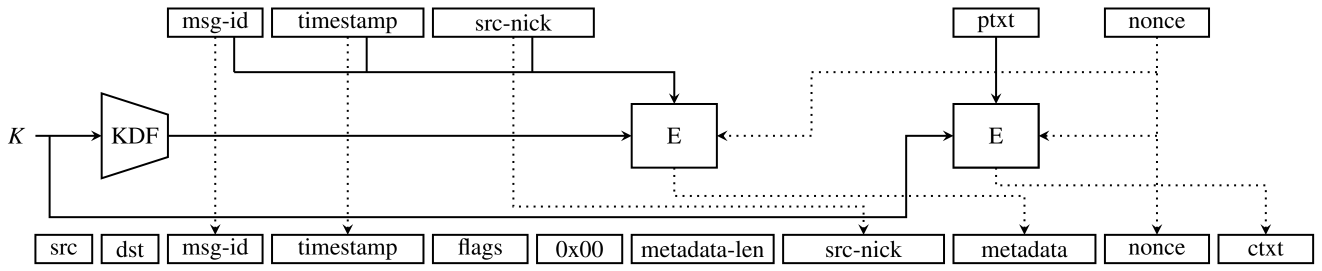


Figure 3: Structure of an E2E encrypted message. The input to the KDF is the key material K. The salt and label are implicitly input into the KDF, taking values "mm" and "3ma-csp", respectively.

the protocol is trivially broken if the adversary has access to a party's secret key material. A formal analysis of the E2E protocol could instead be carried out in a model for Non-Interactive Key Exchange (NIKE), cf. [28], in order to show that the compromise of a user's private key u_i does not affect the security of another user $\mathcal{U}_j$'s long-term private key, nor their keys $K_{\mathcal{U}_j,\mathcal{U}_\ell}$ shared with other, non-compromised users. We leave such an analysis to future work.

2.3 Threema Client-to-Server Protocol

The Threema C2S protocol is concerned with establishing a secure channel between a client and the server to protect messages in flight, shielding metadata and the non-forward secret E2E protocol messages from a network adversary.

Threema's C2S protocol is a novel secure channel protocol, broadly comparable in function and construction to the TLS protocol. It assumes a reliable transport, and can be decomposed as a *handshake subprotocol*, in which the server and the client interact to establish a session key, and a *transport subprotocol*, where the server and the client use the established session key to exchange messages.

The C2S protocol requires that each user connects to the server from only one device at any given moment. If two devices try to connect to the server at the same time, the older connection is dropped in favour of the newer one. Before terminating the connection, an error message is sent to the device, informing it that another device has connected with the same Threema ID.

2.3.1 Handshake Subprotocol

The handshake subprotocol runs between the server $\mathcal{S}$ (with long-term key pair $(s, S = g^s)$) and a client $\mathcal{U}$ (with long-term key pair $(u, U = g^u)$). Fig. 4 gives a pictorial representation of the protocol. We refer to the figure for the numbering of the messages below. We can assume that the server and the client already know each other's public keys: the client knows the server's key as it is pinned in the app and the server receives the client's key at registration time.

The client first generates an ephemeral key pair $(x, X = g^x)$ and a 16-byte value called the *client cookie*, and sends both to the server (M 1). After receiving the client's ephemeral public key and the client cookie, the server generates its own ephemeral key pair $(y, Y = g^y)$ and a 16-byte value called the *server cookie*. It then computes the first shared symmetric key of the exchange $K_1 = \mathsf{X25519}(s, X)$, mixing the long-term key of the server with the ephemeral key of the client, and uses it to encrypt its ephemeral public key concatenated with the client cookie, using a random nonce n. The server sends the resulting ciphertext, along with its own cookie (M 2).

The client and server cookies prevent replay attacks and are used to initialize client and server *counters*. For each counter, the most significant 16 bytes of the counter are set to the respective cookie, while the lower 8 bytes function as a monotonically increasing value. Subsequent AEAD nonces for the exchanged messages are drawn from these counters.

When the client receives the message, it recomputes K_1 and checks if the ciphertext decrypts correctly, and if the client cookie in the plaintext corresponds to the one that the client previously generated, aborting the protocol otherwise. At this point, the client and the server derive two additional keys: K, obtained by combining the two ephemeral key pairs (x, X) and (y, Y), and is used in the transport subprotocol as the *session key*, and K_2, obtained by combining the two long-term key pairs (u, U) and (s, S). The long-term symmetric key K_2 is used by the client to create the so-called *vouch box* $\mathsf{vouch} \leftarrow E_{K_2}(X; n')$ using a random nonce n'. The vouch box is used from the server to authenticate the client, and to bind the client's ephemeral key X to the client's identity.

The vouch box is concatenated with $ID_{\mathcal{U}}$ and the server cookie $C_{\mathcal{S}}$, and encrypted using the session key K (M 3). The server decrypts the message with K and tries to decrypt the vouch box with K_2. If the decryption fails or the value contained in the vouch box does not correspond to the client's ephemeral key, the server will abort the protocol. Otherwise, the server sends a final confirmation message composed of 16 zero bytes, encrypted with the session key K (M 4).

Taken as a standalone protocol, the C2S handshake could be formally analysed using, for example, the extended Canetti-Krawczyk security model [44] or the MSKE security model of [26] that was used in [14, 15] to analyse Signal. We will (informally) use these models when relating the attacks in

Section 3 to the formal security properties they violate.

2.3.2 Transport Subprotocol

In the transport subprotocol, the client and the server exchange messages, encrypted using the session key K. The client and server nonces are derived from the same (respective) counters inherited from the handshake subprotocol, ensuring that handshake messages cannot be used as transport messages (and *vice versa*). If a decryption fails, the entire connection is dropped. This prevents message reordering and message deletion by a network adversary, since either attack would result in a party decrypting with a nonce different from the correct one, and then the AEAD integrity guarantees would ensure that an error is generated on decryption.

When the connection is dropped, the client attempts to reconnect, with up to five reconnection attempts being made. If these all fail, the app will display an error to the user and will require a restart to be able to connect to the server. If two devices are actively trying to connect to the server, they will keep interfering with each other until one of them relinquishes the connection. This is relevant for any attacker that wants to attempt an impersonation attack, since using all five attempts will alert the user that another device is trying to connect to the server with the same Threema ID.

While the connection is alive, the server will send messages that are meant for the user encrypted under the session key K. The client replies with an ACK in response to each message. If the client does not send an ACK, the server will try to send the message again when the client later reconnects.

2.4 Registration Protocol

In order to create a new account and register it with the Threema server, each user generates a Curve25519 key pair (a,A). The entropy for the generation of the key pair is taken from the system randomness, as well as random user input which is processed and mixed with the former to obtain the 32-byte private key a, from which $A = g^a$ is derived. The client later runs a registration protocol with the server to prove that they know a. This is done via a challenge-response exchange run over a TLS-protected connection: the client sends A to the server. Then, the server replies with an ephemeral public key X and a message m. The client computes $K = \mathsf{X25519}(a,X)$ and then encrypts m using K and a fixed nonce (“`createIdentity response.`”) with the AEAD scheme to create a response. If this response decrypts correctly and yields m as plaintext, the server accepts the registration and then issues a new Threema ID, storing it along with the public key of the user in the Threema database.

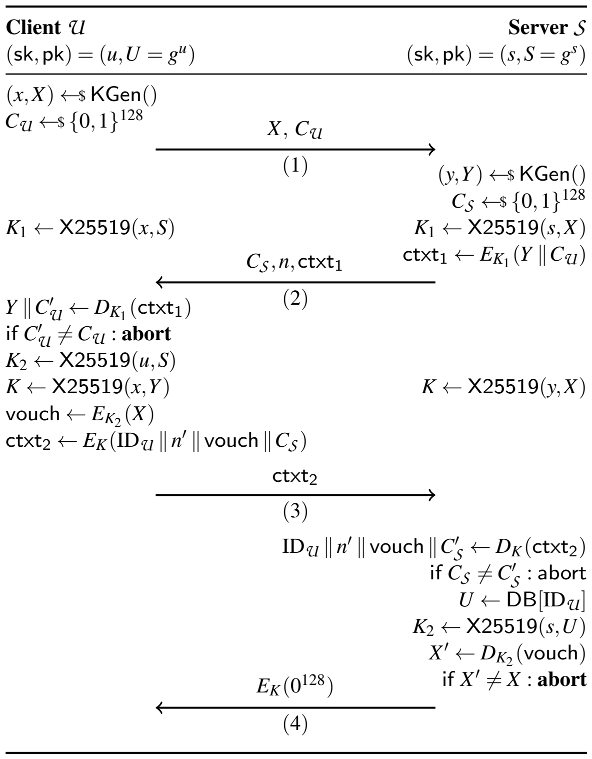

Client $\mathcal{U}$
$(\mathsf{sk},\mathsf{pk}) = (u, U = g^u)$
Server $\mathcal{S}$
$(\mathsf{sk},\mathsf{pk}) = (s, S = g^s)$
$(x,X) \leftarrow\!\$\ \mathsf{KGen}()$
$C_{\mathcal{U}} \leftarrow\!\$\ \{0,1\}^{128}$
$X,\ C_{\mathcal{U}}$
(1)
$(y,Y) \leftarrow\!\$\ \mathsf{KGen}()$
$C_{\mathcal{S}} \leftarrow\!\$\ \{0,1\}^{128}$
$K_1 \leftarrow \mathsf{X25519}(x,S)$
$K_1 \leftarrow \mathsf{X25519}(s,X)$
$\mathsf{ctxt}_1 \leftarrow E_{K_1}(Y \,\|\, C_{\mathcal{U}})$
$C_{\mathcal{S}}, n, \mathsf{ctxt}_1$
(2)
$Y \,\|\, C'_{\mathcal{U}} \leftarrow D_{K_1}(\mathsf{ctxt}_1)$
$\text{if } C'_{\mathcal{U}} \neq C_{\mathcal{U}}$: **abort**
$K_2 \leftarrow \mathsf{X25519}(u,S)$
$K \leftarrow \mathsf{X25519}(x,Y)$
$K \leftarrow \mathsf{X25519}(y,X)$
$\mathsf{vouch} \leftarrow E_{K_2}(X)$
$\mathsf{ctxt}_2 \leftarrow E_K(\mathrm{ID}_{\mathcal{U}} \,\|\, n' \,\|\, \mathsf{vouch} \,\|\, C_{\mathcal{S}})$
ctxt_2
(3)
$\mathrm{ID}_{\mathcal{U}} \,\|\, n' \,\|\, \mathsf{vouch} \,\|\, C'_{\mathcal{S}} \leftarrow D_K(\mathsf{ctxt}_2)$
$\text{if } C_{\mathcal{S}} \neq C'_{\mathcal{S}}$: abort
$U \leftarrow \mathsf{DB}[\mathrm{ID}_{\mathcal{U}}]$
$K_2 \leftarrow \mathsf{X25519}(s,U)$
$X' \leftarrow D_{K_2}(\mathsf{vouch})$
$\text{if } X' \neq X$: **abort**
$E_K(0^{128})$
(4)

Figure 4: The Threema C2S Protocol (Handshake Subprotocol). For the sake of brevity, nonces are omitted from this diagram. The table DB is an abstraction for the database that maps Threema IDs to public keys.

2.5 Threema Safe

Threema allows a user to activate, at any point in time, the Threema Safe feature. This allows the user to backup their long-term private key, their nickname, the list of their contacts and additional account information on the Threema servers. Note that the backup does not store any message history.

When the Threema Safe is first set up, the user chooses a password for their backups. A minimum length of 8 characters is enforced, and the password is checked against an offline list of weak passwords. The app derives two 32-byte strings, the Backup ID BID and the Backup Key BK, using scrypt with the password as input and the Threema ID of the user as salt. For every backup, the app creates a JSON payload containing all the data to be backed up and serializes it into a byte string, which is then compressed using gzip. Finally, the app encrypts this compressed JSON string with XSalsa20-Poly1305 under BK with a random nonce. The resulting ciphertext is sent to the server over a TLS connection.

The server will store ciphertexts indexed by BID, supposedly without any other information related to the user. In the Threema whitepaper [31], the developers claim that this process is designed to make it impossible for the Threema Safe server to distinguish which backup comes from whom by looking at the uploaded data. We note, however, that when a backup is created the app is likely to have an active C2S protocol session, which reveals the Threema ID of the user. By linking the two connections coming from the same IP, the server can learn the identity associated with a given backup.

When the user initiates the process to restore a backup, the app requests the password and the Threema ID from the user. Then, it derives BID and BK from the password, and retrieves the ciphertext corresponding to BID from the server. Finally, the ciphertext is integrity-checked and decrypted, and the user data is deserialized and restored.

A backup is scheduled to happen every day or, in case the previous backup failed to be uploaded, as soon as the app is restarted. This happens even if access to the app is protected by a PIN or biometric login.

2.6 Other Backup Methods

Threema provides two additional methods to save personal data: the first one is called a *Threema ID export* and the second one is called a *data backup*.

The Threema ID export requires that the user provide a password, which is used to derive a 32-byte symmetric key by using the PBKDF2 algorithm with a random 8-byte salt. This key is used to encrypt the ID of the user, as well as their long-term private key using XSalsa20 with a zero nonce (but with no integrity protection). The result is base32-encoded and given to the user in order to be saved outside of the app.

A data backup consists of an encrypted zip containing various files. The `identity` file stores the Threema ID and the long-term key of the user; the `contact` file stores Threema IDs, nicknames, and public keys of contacts in CSV form, and for each chat, a file named with the Threema ID of the correspondent stores the chat history. The zip is encrypted following the WinZip AE-2 specification [16]. This essentially means that the user-provided password is used to derive a master key. For each file, a pair consisting of an encryption key and an authentication key are derived from the master key. A different salt value is used for each key derivation. Then, the file is compressed, encrypted with AES-CTR and authenticated using HMAC-SHA1.

We note that, because WinZip does not hide the file names, this encrypted zip leaks information about with whom the user has been communicating. Due to the length of each file being known, an attacker may also infer the length of the conversation from the encrypted zip.

3 Attacks on Threema

3.1 Threat Models

We consider three threat models: first, an external actor who can monitor communications between a device running the app and the Threema servers; second, an actor that has compromised the Threema servers; third, an actor that can take control of the device for a short period of time. Figure 5 visually depicts each of these threat models.

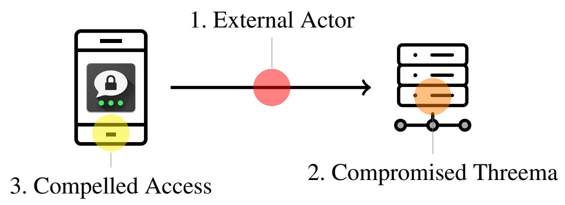


Figure 5: In the threat models we consider, the attacker can have access to network communication (1.), the Threema servers (2.) or the victim device itself (3.).

The first of these models corresponds to the standard adversary for secure network protocols, often called the Dolev-Yao adversary. We refer to it as the *external actor* threat model.

Concerning the second model, any reasonable security analysis must consider the possibility that the messaging server is malicious: national security agencies and hacking groups are tasked with penetrating servers like those run by Threema in order to gain persistent access to concentrated amounts of sensitive data. It is then reasonable to ask what security guarantees remain for users in the event of such a breach. This is especially so for a system that claims to provide end-to-end security. We refer to this model as the *compromised Threema* threat model.

We call the third model the *compelled access* threat model. It is relevant, for example, in the case of border searches of electronic devices, when protesters are detained by police forces and searched for incriminating evidence, or in the setting of intimate partner violence [27]. We may subdivide this setting into two sub-cases: one where the app is locked, and the other where it is unlocked (we will always assume the device itself is unlocked).

We next introduce our attacks, grouped by threat model.

3.2 External Actor Threat Model

3.2.1 Attack 1 (C2S Ephemeral Key Compromise)

In this section we show various problems with the structure of the handshake protocol, which lead to multiple *permanent* impersonation attacks. More specifically, if the client generates a weak ephemeral key that the attacker can discover, then it can use information from the compromised session to create new sessions with the server in which they impersonate the victim and can tamper with the flow of messages.

Suppose that an attacker learns the ephemeral secret key of a victim user x in a C2S handshake session. This capability is commonly referred to in formal security models as *ephemeral key reveal*, and it can arise in practice from a randomness failure. Further suppose that the attacker has the transcript of a complete handshake, gained, for example, by passively intercepting a completed session. As highlighted in Figure 6, the attacker then has all the material necessary to derive K_1 and decrypt message (3) from the C2S protocol run, from which it can recover the value of vouch.

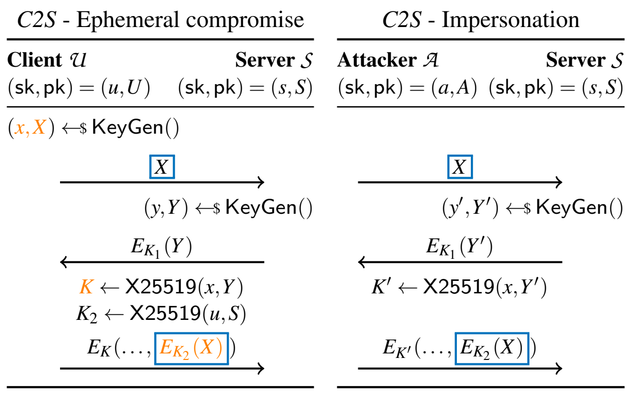
Now the attacker initiates a new run of the C2S protocol, playing the role of the victim user. In this run it replays the ephemeral public value X. A new session key K known to the attacker is generated (the server should send a fresh Y in its message (2) but the attacker still knows x). The attacker can then reuse vouch when constructing its response in message (3). The reason is that vouch, while constructed under a key K_2 not known to the attacker, does not contain any fresh value from the server, only X (which the attacker has replayed). In addition, while vouch depends on the nonce n', we have verified that the server does not detect replays of this value. Hence the server will accept the attacker's message (3) and the new handshake session will complete successfully.

Since vouch can be reused repeatedly, this attack (involving the loss of just a single ephemeral value) allows the attacker to permanently impersonate the victim user to the server, enabling them access to all E2E-encrypted messages meant for the victim. This, in turn, reveals all the metadata information in all communications sent to the victim user, allowing the attacker to learn with whom they have been communicating, as well as all the message timestamps.

Most importantly, this attack also leads to a break of the forward secrecy property that the C2S protocol is intended to provide. Indeed, after the attack, the adversary can see all E2E-encrypted messages; since these only use long-term keys for encryption, full decryption of those messages becomes possible if the victim's long-term key is ever later revealed. In addition, the attacker gains the opportunity to selectively drop messages from the conversation by deciding which messages to ACK to the server. Recall that, if a message is ACK-ed by the attacker, it will not be received by the victim when they next connect to the server, virtually deleting that message from the conversation on the victim's end.

We stress that, as long as the attacker and the victim do not connect to the server at the same time, this attack is not detectable by the client. If they were to connect at the same time, the attacker would know this is the case, since they would receive an error message from the server. At this point, the attacker may relinquish the connection to the victim, in order to avoid detection.

The main weakness that this attack highlights is that vouch does not contain any fresh value that would prevent its replayability. In order to prevent this class of attacks, the vouch box should not only depend on the ephemeral key chosen by the client, but also on a value chosen by the server. For instance, the server cookie could be included in vouch rather than being outside it. This attack also shows that, for the C2S protocol, ephemeral keys are as important to protect as long-



C2S - Ephemeral compromise
Client $\mathcal{U}$
Server $\mathcal{S}$
$(\mathsf{sk},\mathsf{pk}) = (u,U)$
$(\mathsf{sk},\mathsf{pk}) = (s,S)$
$(x,X) \leftarrow\!\!\$\ \mathsf{KeyGen}()$
$\boxed{X}$
$(y,Y) \leftarrow\!\!\$\ \mathsf{KeyGen}()$
$E_{K_1}(Y)$
$K \leftarrow \mathsf{X25519}(x,Y)$
$K_2 \leftarrow \mathsf{X25519}(u,S)$
$E_K(\dots,\boxed{E_{K_2}(X)})$
C2S - Impersonation
Attacker $\mathcal{A}$
Server $\mathcal{S}$
$(\mathsf{sk},\mathsf{pk}) = (a,A)$
$(\mathsf{sk},\mathsf{pk}) = (s,S)$
$\boxed{X}$
$(y',Y') \leftarrow\!\!\$\ \mathsf{KeyGen}()$
$E_{K_1}(Y')$
$K' \leftarrow \mathsf{X25519}(x,Y')$
$E_{K'}(\dots,\boxed{E_{K_2}(X)})$

Figure 6: Attack 1: two C2S sessions. The attacker compromises an honest session by revealing the ephemeral private key x, from which they can compute the session key K and obtain the vouch box $E_{K_2}(X)$ (in orange, Left side). The values are replayed by the attacker in a new session (blue boxes, Right side). Note the lack of *session independence*.

term keys. This is a very strong assumption to have to make in order to guarantee security for a key exchange protocol.

We note that a similar vulnerability was present in the Off-The-Record (OTR) protocol [52]. The authors of [52] highlight the consequent lack of the *session independence* property in OTR: "the exposure of ephemeral session-specific secrets should have no bearing on the security of other sessions". Threema's C2S protocol suffers from the same issue.

Non-ephemeral Ephemerals: An additional problem with the C2S protocol lies in how the ephemeral keys are handled client-side: if the app is never restarted, then the same ephemeral key is used by the client for up to seven days before generating a new one. Leaking such a re-used ephemeral key then allows an attacker to also impersonate the server to the user by forging message (2) using the server's public key and the compromised ephemeral key (this is known as an *ephemeral Key Compromise Impersonation* attack in the literature). In turn, this implies that the attacker has the power to also read the outbound E2E-encrypted messages of the victim that are meant for the real Threema server. This reveals additional metadata, allows the attacker to drop arbitrary messages and allows a network adversary to execute attacks that would otherwise be in the "compromised Threema" threat model, such as the ones described in Sections 3.3.1 and 3.3.2. If ephemeral keys were to be freshly generated for each handshake, this problem would be confined to the single compromised session. However, by reusing the same ephemeral key for a long time, the effect of this attack is heavily amplified.

Finally, there is similar reuse of ephemeral keys on the server side. We noticed that whenever the same ephemeral key X is used multiple times by a user within a short span of time, the server will use the same ephemeral key Y as well. In our experiments, we managed to prolong the lifetime of the server ephemeral key for up to 2 months and 14 days (at the time of writing) by repeatedly connecting with the same ephemeral key every hour. We hypothesise that the server caches ephemeral keys in a structure similar to a key-value store, where the client's ephemeral key X is mapped to a server ephemeral key y and where a Least Recently Used policy is applied. If a key is not used for a period of time longer than a certain threshold (approximately 24 hours) it is evicted from the cache. An attacker is thus able to force the same session key to be used for a long time by regularly connecting to the server with the same value X, thus refreshing the matching value y stored in the cache.

Assuming that our hypothesis of an ephemeral key cache holds, compromising said cache would allow the impersonation of multiple users to Threema for an indefinite period of time, as long as the attacker has recorded C2S handshakes involving the revealed ephemeral keys. In fact, it suffices for the attacker to replay message (1), inducing the server to reuse a previously revealed key y. As the attacker can compute the session key, they can create a new message (3), which contains an old vouch box, thus completing the handshake. This makes the hypothesised cache a weak point in the cryptographic design of Threema; it therefore requires careful protection.

In order to test all these weaknesses, we created a Python script which, given a valid transcript of a C2S handshake and the client ephemeral key used, can impersonate the user to the server. The script then shows all the metadata of incoming messages and can drop arbitrary messages from the conversation.

3.2.2 Attack 2 (Vouch Box Forgery)

An alternative route to achieve the same goal as the previous attack is to somehow forge a completely new vouch box containing an ephemeral key X for which the adversary knows the private value x. This is feasible due to a cross-protocol interaction between the E2E and the C2S protocols. The consequence is that the attacker again gains the ability to impersonate the victim for an indefinite amount of time.

The attack requires two steps. In the first step, the attacker has to convince the victim that their long-term public key corresponds to the public key S of the server used in the C2S protocol. In the second step, the attacker, now acting as a Threema user, has to convince the victim user to send a specifically crafted text message to the attacker in the E2E protocol. Under normal circumstances, these actions should not lead to an attack: if the victim user sends a message encrypted against the server's public key, then the attacker should not be able to decrypt it, nor use it for other purposes. Furthermore, sending an E2E message should not compromise security of the C2S protocol.

We will show two different methods to implement the first step of the attack. In the first method, we use part of the infrastructure of Threema to create a new account which has the server's public key S as its own public key. In the second method, we leverage a vulnerability in a dependent library used by the Android version of Threema.

General idea: Let (u,U) be the long-term key pair of the victim. Assume that the first step of the attack has already been executed, meaning that, whenever the victim wants to send a message to the attacker, they will use their private key u and the server's public key S to encrypt their message. Assume also that the attacker manages to find a private key x such that the corresponding public key $X = g^x$ has the following form: a `0x01` byte, followed by a string σ composed of 30 printable UTF-8 characters, followed by another `0x01` byte.

The attacker's objective is to forge a vouch box vouch for this key X that would allow the attacker to authenticate as the victim in the C2S protocol. If the victim sends σ as a text message to the attacker in the E2E protocol (for example, via social engineering, e.g. sending a special code to win a prize). This would result in the victim deriving a key $K^* = \mathsf{X25519}(u,S)$ and encrypting the message. Since σ is sent as a text message, the ciphertext will include `0x01` as the first byte, followed by the string σ and then PKCS7 padding at the end. The attacker hopes to obtain a ciphertext $c = \mathsf{E}_{K^*}(\mathtt{0x01} \| \sigma \| \mathtt{0x01}; n) = \mathsf{E}_{K^*}(X; n)$ (for some nonce n). This has probability $1/254$ of working, since this is the probability of obtaining `0x01` as a PKCS7 padding with Threema's choice of padding method. We note that K^* here is actually equal to K_2, the key that would be derived by the victim during the C2S protocol and used in creating a vouch box. Because of this, by construction, the ciphertext c that was created and sent to the attacker is thus a valid vouch box vouch for X. Since the attacker knows the corresponding x, the attacker can now use (x,X) and vouch to authenticate to the Threema server as the victim.

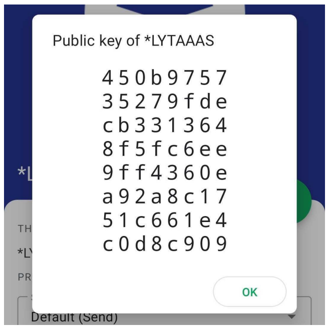
Finding a good X: Creating a suitable key X is feasible, simply by randomly sampling x and computing $X = g^x$ until X has the desired form. We have empirically estimated the probability of each trial being successful in finding a good X as being $\approx 2^{-51}$. Thus, after 2^{51} trials, the success probability would be 0.63. We implemented this approach, using some batch inversion optimisations to reduce the cost of coordinate conversions.

We found a key of the correct form after $2^{50.0}$ trials. In total, this required approximately 8100 core-days across a variety of computational clusters available to us. The results are shown in Figure 7. We used this key in our experimental environment to demonstrate that the attack works in practice.

Claiming the server's public key: We now describe two methods allowing an attacker to claim S as their long-term public key, the first step in the attack.

Our first method exploits the fact that Threema provides a paid API, the "Threema Gateway", for companies to interface with the Threema messaging system automatically. When registering a new account, the user is required to provide a

```
privkey: UErBPgAAAAAAMAAz
         bWEtMi0yMjEyMTkt
         MDMtMDAyMAA=
 pubkey: AXU5ajbfkydqauCk
         1tS7Xt+DMdebVzot
         zIE73KFSQQE=
```



Public key of *LYTAAAS

```
450b9757
35279fde
cb331364
8f5fc6ee
9ff4360e
a92a8c17
51c661e4
c0d8c909
```

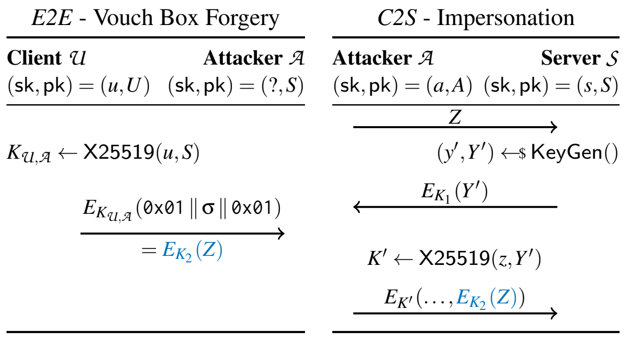
Figure 7: Attack 2 in practice: on the left, a suitable keypair (base64 encoded). The public key bytes 1 to 31, also encoded in the QR code, all consist of printable UTF-8 characters. On the right, the `*LYTAAAS` Threema gateway account (since revoked), with the hijacked public key of the server. User $\mathcal{U}$ sending the contents of the QR to `*LYTAAAS` as a message will allow `*LYTAAAS` to authenticate to Threema as the $\mathcal{U}$.

public key. The gateway, however, does not check (1) if the user has the corresponding private key and (2) that the key does not correspond to the server's public key S.[4] This allows the creation of an account which has the public key of the server as its own. We created such a Threema account under the Threema ID "`*LYTAAAS`".[5]

Our second method uses a vulnerability in one of the dependencies that Threema has on the Android version of the app for creating data backups (described in Section 2.6). The library used by Threema to create zip files is called Zip4j [45]. This library possesses a bug where the MAC is not checked when decrypting the zip file, if certain conditions are met. This is the case whenever a Threema backup is restored, meaning that any tampering with the zip would not be detected. In the context of Threema, this allows an attacker that has write access to the encrypted zip to modify the contacts file within it, allowing it to overwrite their public key with the server's. For example, a user might choose to save their zip file in shared storage (e.g. a cloud service, or a folder on a computer which the attacker can access), believing it to be both encrypted and integrity-protected. While this second method does not exploit a vulnerability in Threema *per se*, a more robust design would have prevented this bug from being escalated to client impersonation in the C2S protocol.

[4]Note that performing this check properly is non-trivial: we also successfully registered keys $T \neq S$ such that $T = S \cdot P$, where P is a Curve25519 point of order 8. When computing a Diffie-Hellman share using the private key a and the public key T, due to clamping $T^a = P^a \cdot S^a = S^a$, thus claiming T is equivalent for our attack to claiming S.

[5] "Lose Your Threema Account As A Service"



E2E - Vouch Box Forgery
C2S - Impersonation
Client $\mathcal{U}$
Attacker $\mathcal{A}$
Attacker $\mathcal{A}$
Server $\mathcal{S}$
$(\mathsf{sk},\mathsf{pk}) = (u,U)$
$(\mathsf{sk},\mathsf{pk}) = (?,S)$
$(\mathsf{sk},\mathsf{pk}) = (a,A)$
$(\mathsf{sk},\mathsf{pk}) = (s,S)$
Z
$K_{\mathcal{U},\mathcal{A}} \leftarrow \mathsf{X25519}(u,S)$
$(y',Y') \leftarrow\!\$\ \mathsf{KeyGen}()$
$E_{K_1}(Y')$
$E_{K_{\mathcal{U},\mathcal{A}}}(\texttt{0x01} \| \sigma \| \texttt{0x01})$
$= E_{K_2}(Z)$
$K' \leftarrow \mathsf{X25519}(z,Y')$
$E_{K'}(\ldots, E_{K_2}(Z))$

Figure 8: Attack 2: a cross-protocol interaction of an E2E and a C2S session. The attacker claims the public key of the server, and knows a keypair of the form $(z, Z = \texttt{0x01} \| \sigma \| \texttt{0x01})$. They convinced the victim $\mathcal{U}$ to send σ to them as a E2E text message (in blue, Left side). The attacker can now start a session of the C2S protocol (Right side) where they use the "ephemeral" keypair (z,Z) and the corresponding vouch box $E_{K_2}(Z)$ (in blue) in order to authenticate as $\mathcal{U}$ to the server.

3.3 Compromised Threema Threat Model

3.3.1 Attack 3 (Message Reordering and Deletion)

The lack of an ordering mechanism in the E2E protocol combined with the lack of authentication of the metadata in that protocol allows the server to carry out trivial reordering attacks on E2E messages. This means that the server can change the order in which the messages are delivered to the receiver, thereby potentially changing the semantics of the conversation. Furthermore, the server can withhold messages for as long as it pleases and send them at a later time with an updated timestamp, in order to evade detection.

In addition to the above, we noticed that incoming messages are shown in the order in which they are received by the Threema app, rather than being ordered by timestamp. Moreover, the Threema app does not display seconds to the user in the message information. This means that, even if an attacker could not tamper with the metadata, it would still be able to reorder messages within any one-minute window.

3.3.2 Attack 4 (Message Replay and Reflection)

In Section 2.2 we explained how Threema tries to prevent replay and reflection attacks on the E2E protocol by storing nonces of both outgoing and incoming messages. Fundamentally, this requires that the nonce database is always kept updated and is never deleted in the client app. Unfortunately, this cannot be guaranteed whenever the app is reinstalled or when the user changes device: in either case[6], the nonce database is reset and a compromised Threema server can replay messages that were received by the victim in the past or

[6]Note that on iOS, migrating the app to a new device using *Quick Start* would preserve the database.

reflect messages that the victim has sent.

To make this attack more effective, we note that the Threema server can easily tell when a victim is likely to have reset their nonce database, as long as the victim has decided to use Threema Safe as their backup option. (This is a reasonable assumption, since Threema Safe is the default option and the user is strongly encouraged to use it at registration time.) Whenever a user restores a backup from Threema Safe, it will connect to the server with the C2S protocol shortly after, most likely with the same IP address. Thus, not only can the server learn the identity associated with a given backup, but it will also know that, with high probability, the user's nonce database has been deleted, making that user a target for the aforementioned attacks.

To test Attacks 3 and 4, we created a simulated Threema server in Python to which clients can connect. The server provides control over the order of the messages and their inclusion in the conversation, and allows the attacker to save the messages in order to be later replayed or reflected. We experimentally verified all the behaviours observed above.

3.3.3 Attack 5 (Kompromat)

Recall that, whenever a user $\mathcal{A}$ tries to register to Threema, it must prove to the server that it owns its public key by use of a challenge-response protocol. In that protocol, the user combines their private key a with a public key provided by the server $X = g^x$ and encrypts a message m chosen by the server. When the contact discovery protocol is run (see Appendix A), the same challenge-response protocol is executed in order to obtain the match token. The encryption method used is identical to that in the E2E protocol.

In either case, the server does not necessarily have to own the private key x corresponding to the provided ephemeral public key, and they in fact can put the long-term public key of any other Threema user $\mathcal{B}$ instead. This means that $\mathcal{A}$ will be encrypting the challenge with the same key that they would use to communicate with $\mathcal{B}$ in the E2E protocol. Hence the resulting ciphertext is a valid E2E message that can be sent to either $\mathcal{A}$ or $\mathcal{B}$ (and which can be made to appear to come from the other party). Each ciphertext can be sent only once per user, since the receiver will always store the relevant nonce after processing the ciphertext, preventing another ciphertext with the same nonce from being processed.

This flaw was fixed in version 4.6.14 for iOS and 4.62 for Android in December 2021 [34]. The app now requires that the message m provided by the server has a type byte equal to `0xff`, which does not correspond to any valid message type, thus preventing it from being accepted as a valid message. Nonetheless, this attack provides another example of a dangerous cross-protocol interaction, in this case between the registration protocol and the E2E protocol. While the fix does effectively prevent this attack, it does not tackle the other cross-protocol attacks that we present. In Section 4 we discuss mitigations that offer full protection.

3.3.4 Additional Notes

From the point of view of an attacker located at the Threema server, the E2E-encrypted communications have no forward secrecy whatsoever. This is because the forward secrecy property is only provided at the client-to-server level rather than the end-to-end level. This leaves the user completely vulnerable in the event of a long-term key compromise in the "compromised Threema" threat model.

This is explicitly acknowledged in the Threema whitepaper [31], where it is stated that "The risk of eavesdropping on any path through the Internet between the sender and the server [...] is orders of magnitude greater than the risk of eavesdropping on the server itself". While true in practice, a messaging app cannot claim to have "maximum security" [35] unless it is also secure against strong attackers that may compromise the Threema server. In fact, such claims give the illusion that the server merely acts as a message router, which cannot read messages and has no gain in storing the messages that it sees. In a messaging app, compromising the server should not give away significant data to the attacker, but, due to the lack of forward secrecy on the E2E level, this expectation is not met by Threema.

Our attacks also show that the weak notion of forward secrecy achieved by the Threema design may be insufficient: by using one of our attacks on the C2S protocol, the adversary gains access to E2E-encrypted messages, which use long-term keys only. Furthermore, the reuse of ephemeral keys further weakens the guarantees given by forward secrecy, since an attacker gains access to messages up to a week in the past. Comparing this design to Signal's, we note that in the latter each E2E message is encrypted with a different key, providing a much stronger and fine-grained notion of forward secrecy.

3.4 Compelled Access Threat Model

3.4.1 Attack 6 (Cloning via Threema ID Export)

Among the backup methods provided, Threema allows the user to export the Threema ID (as explained in Section 2.6). However, this method exposes the long-term secret key of the user, and does not require any form of authentication, allowing a compelled access attacker who has access to the unlocked phone and app to backup the private key with a password of their choice and later decrypt it on their own device.

This is an intentional feature in Threema, but one which we deem to be a grave security danger: all security in Threema relies on maintaining the secrecy of long-term private keys, and this attack allows an adversary to recover such keys by having access to the phone and app for a few seconds.

While similar mechanisms are present in other messengers, their security impacts are much smaller. For instance, Signal allows users to transfer their accounts to a new device,

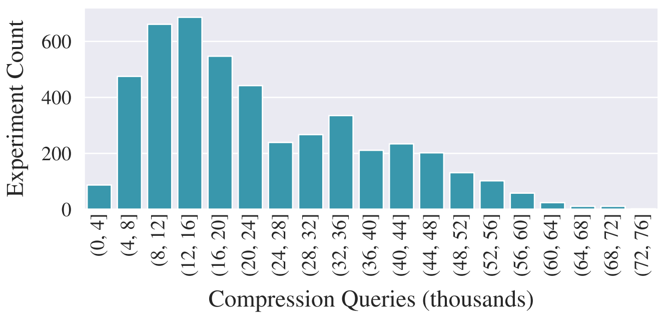


Figure 9: The distribution of the number of compression side-channel oracle queries in our successful key-recovery experiments. Each "query" is a backup attempt by the Threema app.

but it requires physical proximity of the new and the old devices, and disables the old device permanently after such a transfer is completed. This is in contrast to Threema, where a Threema ID export is undetectable on the victim device after it happened and, due to the lack of forward secrecy and post-compromise security, irreversibly forfeits all security.

The Threema app does provide some optional locking mechanisms. For example, the app can be locked with a PIN, a passphrase or by using biometric data, thus preventing this trivial attack.

3.4.2 Attack 7 (Compression Side-Channel)

While the app locking mechanism does prevent a compelled access attacker from accessing the user's data via the user interface, it does not prevent background processes from being executed. In particular, the app will attempt to make a new Threema Safe backup whenever the application is launched if the last successful backup is more than one day old. Moreover, incoming messages will be processed and may affect the contents of the backup. We exploit these facts in our final attack to recover a victim user's private key.

Core idea: Recall that a Threema Safe backup contains the private key of the user, and a list of the user's contacts. Before being sent to the server for storage, the backup is *compressed and then encrypted*. It is well-known that this combination of processing steps can be vulnerable to attack, especially if the attacker can control part of the plaintext before compression [42, 54]. Specifically, the length of the ciphertext may leak information about how much plaintext compression has been achieved, and this in turn can leak information about the degree of overlap between the attacker-controlled and the unknown parts of the plaintext. In Threema Safe, the XSalsa20 stream cipher is used for encryption, so a ciphertext leaks the exact length of the plaintext. Furthermore, if a user $\mathcal{U}$ sends a message to victim $\mathcal{V}$, then $\mathcal{U}$'s Threema ID and chosen nickname are added to the contact list of $\mathcal{V}$ even if $\mathcal{V}$'s app is locked. This provides a means for an attacker to inject chosen plaintext into the backup of a victim. In this way, the conditions needed to build a compression-based side-channel attack are fulfilled. We provide a more detailed description of the attack in Appendix B.

Proof-of-Concept: We created a custom Threema server and instrumented the client code in order to redirect requests to our server rather than the actual Threema Safe server. This allows us to both see the length of the backups and to make backups fail by returning an HTTP error code. To automatically induce a new backup, we use Android debugging tools which can force an app to stop and restart. In order to measure the number of queries needed to complete the attack at scale, we also reimplemented the client backup mechanism, and simulated the attack locally. Figure 9 summarizes our experimental results. We successfully recovered the user's private key in 4.7k out of 10k experiments; in these successful runs, we required a median of 19.4k backup attempts (mean 23.4k). In our PoC, each attempt requires around 2s, implying a total running time of around 11 hours for the attack.

The backup is normally transmitted to the server on a TLS connection, but with modern cipher suites (AES-GCM or ChaCha20-Poly1305), TLS does not hide the length of its payloads. So we consider the attack to remain viable even if the attacker can only see TLS-protected traffic as in our compelled access threat model.

Impact: Recovering the private key leads to a complete loss of security: after doing so the attacker can impersonate the user in any action. Considering a combined attack setting of compelled access *and* a compromised Threema server, we obtain an attack that can decrypt all past communications. This is due to the lack of forward secrecy on the E2E level.

4 Mitigations

Preventing Cross-Protocol Attacks: The attacks in Sections 3.2.2 (Vouch Box Forgery) and 3.3.3 (Kompromat Forgery) exploit the fact that the same "X25519-then-Encrypt" paradigm is used multiple times for different purposes but with the same keys. For example, the E2E protocol uses it for exchanging messages between users, while the C2S protocol uses it to authenticate the user to the server. By using payloads created in one protocol in another, the adversary gains capabilities that they would not have when viewing the protocols separately. A secure application should cryptographically compartmentalise its protocols, taking a very conservative approach when using the same cryptographic material in different protocols (if doing so at all). A common way to prevent this class of attack is to use proper *key separation*: the X25519 key material will then only be used to derive context-dependent keys using a KDF with proper labels. We stress that key material that is used to derive other keys should not be used itself for other purposes, since it is important to ensure that leaking one of the keys during its usage does not affect any other key, i.e. the keys should remain effectively *independent* of each other.

Protecting Metadata: The simplest way to prevent the attacks of Sections 3.3.2 (Replay and Reflection Attacks) and 3.3.1 (Message Reordering) is to protect the integrity of the metadata contained in the E2E packet. This can be simply achieved by including the metadata in the (authenticated but not encrypted) "additional data" field of the AEAD scheme. This modification has zero message overhead and supersedes the usage of the metadata box, eliminating the need for an additional ciphertext and tag. Protecting the metadata in this way would prevent an adversary from swapping the source and destination of the message and thus prevent reflection attacks. To prevent message reordering attacks, the app could use a per-recipient counter for the AEAD nonces in the E2E protocol. This would also allow the recipient to detect lost or adversarially deleted messages, but would still leave the problem of how to avoid nonce repetitions when the Threema app is reinstalled or when the user changes device. We suggest that the app should re-run the registration protocol in this event, establishing a fresh key pair (a, A).

Strengthening the C2S Protocol: The least invasive modification to the C2S protocol that directly prevents our attack of Section 3.2.2 would be to include the server cookie inside the vouch box. This ensures that, as long as the server picks a fresh cookie every time the time protocol is run, the vouch box cannot be trivially replayed. Note that, while Threema claims their protocol to be optimal in terms of round-trips [31, p. 10], there are protocols which provide the same properties with less message overhead. An example is the IK protocol of the Noise Protocol Framework [50], which allows for 0-RTT encrypted communications between the client and the server and provides authentication and forward secrecy. Furthermore, it prevents the attack of Section 3.2.1 (Impersonation by Randomness Failure). The Noise protocols have been analyzed and proven secure [22, 23], and Noise protocols have seen practical deployment in Wireguard [20] and Whatsapp [51]. We, therefore, recommend that Threema replaces its bespoke C2S protocol with the IK protocol.

Forward Secrecy at the E2E level: Here there is no simple fix for Threema: to ensure forward secrecy at the E2E level a protocol must use ephemeral keys, which the current E2E protocol is not designed to handle. We, however, disagree with Threema's statement that "providing reliable Forward Secrecy on the end-to-end layer is difficult" [31, p .14]. We recommend that Threema adopt the Signal protocol, whose cryptographic API is available through the libsignal library [46]. The Signal protocol has received extensive security analysis [29, 14, 43, 15] and provides forward secrecy and post-compromise security at the end-to-end layer. We note that Whatsapp has transitioned towards using the Signal protocol in the past [47], showing that such a change is feasible even for a messaging app with a huge user base.

Preventing Cloning via Threema ID Export: The ID export feature should be protected, possibly by requiring the user to provide a passphrase chosen at registration time, or at least by enforcing the use of the phone OS's authentication mechanisms such as biometrics and pin codes. It is up to Threema to strike a fair balance between usability and security, but we would err on the side of the latter.

Mitigating the Compression Side-Channel: Threema should avoid using compression. To avoid an explosion in server-side storage requirements, a binary serialization format such as CBOR [13] could be used instead of JSON.

5 Conclusions

Threema is marketed as a secure app, suitable for both individuals and companies that want to communicate privately. It is used by government departments and high-profile politicians. On the other hand, the seven attacks we have presented highlight several fundamental weaknesses in the design of Threema. Indeed, the Threema protocols lack basic properties that are nowadays considered *de rigeur* for a messenger app to be regarded as secure: forward secrecy with respect to a malicious server, and protection against replay, reflection, and reordering attacks. We believe that the cryptographic core of Threema has basic design flaws that need to be addressed in order to meet the security expectations of its users and to restore parity between Threema's security claims and what it actually delivers.

Stepping back from the details of our analysis of Threema, we believe that our work presents (at least) three useful lessons for developers when deploying complex cryptographic protocols:

1. *Using modern, secure libraries for cryptographic primitives does not on its own lead to a secure protocol design*: It is possible to misuse libraries such as NaCl and libsignal when building a more complex protocol and developers must be wary not to be lulled into a false sense of security. A pertinent example is the C2S protocol in Threema, which has many issues with ephemeral keys. Another recent example is Bridgefy's mis-integration of libsignal [6]. To their credit, Threema's developers did (largely) avoid "rolling their own crypto" but this advice should be extended to "don't roll your own cryptographic protocol". Of course this advice is only useful if good alternatives are available. In the particular case of the C2S protocol, Threema could have adopted the Noise IK protocol or just used TLS.

2. *Beware of cross-protocol interactions*: A standalone protocol may appear to be secure, but cross-protocol interactions can undermine those security guarantees, as we have shown in Threema with the vouch box and Kompromat attacks. So developers should be careful not to

introduce such vulnerabilities. Every message should be cryptographically bound to its protocol and should then be rejected whenever it is used in a different protocol. This can be done by strictly following the key separation principle – use different keys for different purposes. Notably, a recent analysis of the secure messaging protocol Matrix also exploited cross-protocol interaction to violate the confidentiality of messages [5].

3. *Proactive, not reactive security*: Our inability to find an attack on a protocol does not imply it is secure: new attacks could be found at any moment and known attacks only get stronger over time if left unaddressed. Often, software releases follow a *design-release-break-patch* process (a *reactive* approach [49]), finding a fix for attacks as they appear. This can be inconvenient for users and may also lead to the need to maintain backwards compatibility, opening the way to possible downgrade attacks [41, 12, 1, 9]. Developers should adopt a *proactive* approach, where the protocol is formally analysed during the design stage. Ideally, different forms of analysis would be carried out, e.g. using symbolic approaches or by producing computational security proofs. While this cannot exclude every attack possibility, due to limitations in scope of the analysed model or due to using an incorrect protocol formalisation, it can help improve confidence in the protocol's security. Furthermore, protocols should be designed in such a way that this formal analysis can be more easily carried out. In the present setting, because the C2S protocol uses its session key directly in the handshake (to derive the metadata key), it is impossible to prove session key security, a standard property targeted in security proofs for key exchange protocols.

None of these lessons is fundamentally new, but given the results of our analysis of Threema, and the recent works in the field of secure protocol analysis [7, 6, 5], they apparently bear repeating.

References

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 5–17. ACM, 2015.

[2] Jan Ahrens. Threema protocol analysis. https://blog.jan-ahrens.eu/files/threema-protocol-analysis.pdf, March 2014.

[3] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. *IACR Cryptol. ePrint Arch.*, page 1456, 2020.

[4] Martin R Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking Bridgefy. In *Cryptographers' Track at the RSA Conference*, pages 375–398. Springer, 2021.

[5] Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable cryptographic vulnerabilities in Matrix. https://nebuchadnezzar-megolm.github.io, September 2022.

[6] Martin R Albrecht, Raphael Eikenberg, and Kenneth G Paterson. Breaking Bridgefy, again: Adopting libsignal is not enough. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[7] Martin R Albrecht, Lenka Mareková, Kenneth G Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. In *43rd IEEE Symposium on Security and Privacy (IEEE S&P 2022)*, 2022.

[8] AppBrain. Threema - Android app on AppBrain. https://www.appbrain.com/app/threema/ch.threema.app, October 2022.

[9] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS using sslv2. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 689–706. USENIX Association, 2016.

[10] J. Berger. openMittsu. https://github.com/blizzard4591/openMittsu, September 2016.

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

[12] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 535–552. IEEE Computer Society, 2015.

[13] Carsten Bormann and Paul E. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949, December 2020.

[14] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 451–466. IEEE, 2017.

[15] 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.

[16] WinZip Computing. AES Encryption Information: Encryption Specification AE-1 and AE-2. https://www.appbrain.com/app/threema/ch.threema.app, 2009.

[17] Graeme Connell. Technology deep dive: Building a faster ORAM layer for enclaves. https://signal.org/blog/building-faster-oram, August 2022.

[18] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1481–1495, New York, NY, USA, October 2020. Association for Computing Machinery.

[19] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast Message Franking: From Invisible Salamanders to Encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, Lecture Notes in Computer Science, pages 155–186, Cham, 2018. Springer International Publishing.

[20] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[21] Benjamin Dowling, Felix Günther, and Alexandre Poirrier. Continuous authentication in secure messaging. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part II*, volume 13555 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2022.

[22] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the WireGuard protocol. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2018.

[23] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part I*, volume 12110 of *Lecture Notes in Computer Science*, pages 341–373. Springer, 2020.

[24] Kit Eaton. These Apps Promise to Encrypt Your Smartphone Communications. https://www.nytimes.com/2016/03/24/technology/personaltech/encryption-by-app-adds-security-to-smartphones.html, March 2016.

[25] Corin Faife. Swiss Army drops WhatsApp for homegrown messaging service, citing privacy concerns. https://www.theverge.com/2022/1/7/22871881/swiss-army-whatsapp-messaging-threema-privacy-concerns-us-jurisdiction, January 2022.

[26] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1193–1204. ACM, 2014.

[27] Diana Freed, Jackeline Palmer, Diana Elizabeth Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "A Stalker's Paradise": How intimate partner abusers exploit technology. In Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox, editors, *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, page 667. ACM, 2018.

[28] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, volume 7778 of *Lecture Notes in Computer Science*, pages 254–271. Springer, 2013.

[29] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How secure is textsecure? In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 457–472. IEEE, 2016.

[30] Threema GmbH. GitHub - threema-ch/threema-android: Threema App for Android., December 2020.

[31] Threema GmbH. Cryptography whitepaper. https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf, November 2021.

[32] Threema GmbH. Why Threema Instead of WhatsApp? https://threema.ch/en/work/blog/posts/why-threema-instead-of-whatsapp, May 2021.

[33] Threema GmbH. About – Threema. https://threema.ch/en/about, 2022.

[34] Threema GmbH. Threema changelog. https://threema.ch/en/versionhistory, 2022.

[35] Threema GmbH. Threema provides maximum security and comprehensive privacy protection. https://threema.ch/en/security, 2022.

[36] Threema GmbH. Threema.OnPrem. https://threema.ch/en/onprem, 2022.

[37] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are US: large-scale abuse of contact discovery in mobile messengers. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[38] M. Heiderich, M. Wege, and C. Kean. Pentest- and audit-report Threema mobile apps. https://threema.ch/press-files/2_documentation/security_audit_report_threema_2020.pdf, October 2020.

[39] Fabian Ising, Damian Poddebniak, and Sebastian Schinzel. Security audit report Threema 2019. https://threema.ch/press-files/2_documentation/security_audit_report_threema_2019.pdf, March 2019.

[40] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293. Springer, 2012.

[41] Tibor Jager, Kenneth G. Paterson, and Juraj Somorovsky. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[42] John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2002.

[43] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 435–450. IEEE, 2017.

[44] Brian A. LaMacchia, Kristin E. Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Provable Security, First International Conference, ProvSec 2007, Wollongong, Australia, November 1-2, 2007, Proceedings*, volume 4784 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2007.

[45] Srikanth Lingala. Zip4j. https://github.com/srikanth-lingala/zip4j.

[46] Signal Messenger LLC. libsignal. https://github.com/signalapp/libsignal.

[47] Moxie Marlinspike. WhatsApp's Signal protocol integration is now complete. https://signal.org/blog/whatsapp-complete, April 2016.

[48] Moxie Marlinspike. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery, September 2017.

[49] Kenneth G. Paterson and Thyla van der Merwe. Reactive and proactive standardisation of TLS. In Lidong Chen, David A. McGrew, and Chris J. Mitchell, editors, *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*, volume 10074 of *Lecture Notes in Computer Science*, pages 160–186. Springer, 2016.

[50] Trevor Perrin. The Noise Protocol Framework. http://www.noiseprotocol.org/noise.html, July 2018.

[51] Meta Platforms. WhatsApp encryption overview. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, November 2021.

[52] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Secure off-the-record messaging. In Vijay Atluri, Sabrina De Capitani di Vimercati, and Roger Dingledine, editors, *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, WPES 2005, Alexandria, VA, USA, November 7, 2005*, pages 81–89. ACM, 2005.

[53] Johannes Ritter. Was Russland stört, überzeugt Olaf Scholz. https://m.faz.net/aktuell/wirtschaft/unternehmen/threema-was-russland-stoert-ueberzeugt-olaf-scholz-18248712.amp.html, January 2022.

[54] Juliano Rizzo and Thai Duong. The CRIME attack. https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU, 2012.

[55] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 415–429. IEEE, 2018.

[56] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693, November 2015.

[57] Michel Schreiner and Thomas Hess. Examining the role of privacy in virtual migration: The case of WhatsApp and Threema. In *21st Americas Conference on Information Systems, AMCIS 2015, Puerto Rico, August 13-15, 2015*. Association for Information Systems, 2015.

[58] Jochen Siegle. Schutz beim Chatten: Threema wird noch sicherer. https://www.nzz.ch/digital/whatsapp-alternative-threema-wird-noch-sicherer-ld.1498576, July 2019.

[59] Soatok. Threema: Three Strikes, You're Out. https://soatok.blog/2021/11/05/threema-three-strikes-youre-out/, November 2021.

[60] SRF. Threema setzt sich durch - Schweizer Armee verbietet Whatsapp und Co. https://www.srf.ch/news/schweiz/threema-setzt-sich-durch-schweizer-armee-verbietet-whatsapp-und-co, January 2022.

[61] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptol.*, 12(1):1–28, 1999.

[62] WhatsApp. WhatsApp Help Center - Answering your questions about WhatsApp's January 2021 Privacy Policy update. https://faq.whatsapp.com/general/security-and-privacy/answering-your-questions-about-whatsapps-privacy-policy/?lang=en, January 2021.

A Contact Discovery Protocol

If the user gives permission to do so, Threema can run a protocol to check which of the people in the user's phone's contact list are using Threema. To do so, it must compare the user's contact list with the list of all registered Threema users, while at the same time preserving both the privacy of the user and the confidentiality of the list of registered Threema users.

Threema does this by hashing all the contacts of the user and sending the hashes to the Threema server over TLS. The server will reply with the hashes that correspond to users that have previously registered with Threema. The hash algorithm is implemented using HMAC-SHA256 with a fixed key. This is done to "ensure that hashes generated by Threema are unique and do not match those of any other app" [31]. This is similar to the process used by apps such as Whatsapp, Telegram and Signal [37]. Such protocols tend to raise privacy concerns [37] due to the information leaked to the server, including information about people who might not even be aware of the app's existence. Hashing the information is insufficient, since, for example, phone numbers carry little entropy and can be brute-forced. This allows the service provider to build a partial social graph for an entire population. Signal tries to provide stronger guarantees by implementing their contact discovery protocol in an Intel SGX enclave [48], minimizing the information leak due to memory access patterns, and using techniques such as Oblivious RAM to minimize the information leaked to the server [17].

To prevent abuse of the system, anyone that wants to run the contact discovery protocol must first obtain a *match token* by authenticating to the Threema server. To obtain one, the client must pass a challenge-response protocol which is essentially the same as the one done for the registration process, except for the fact that the nonce is now randomly chosen by the client, rather than being fixed.

B Compression Side Channel

From a high-level perspective, the attack leverages the fact that nicknames are handled client-side: when a message is received, the nickname contained in the metadata is set as the current nickname of the sender. Since nicknames are contained in the backup, this gives the attacker the ability to partially change the contents of the victim's backup by sending a message to the victim. The objective of the attacker

```
{
    "info": { ... },
    "user": {
        "privatekey": <private key>,
        "nickname": "user_nickname",
        "links": []
    },
    "contacts": [
        {
            "identity": "ABCDEFGH",
            "nickname": "other_user_nickname",
            ...
        },
        ...
    ],
    "groups": [],
    "distributionlists": [],
    "settings": { ... }
}
```

Figure 10: Example of a JSON backup, with the private key redacted.

is to leverage this partial control in order to force a backlink to be created from the attacker's nickname in the `contacts` field to the private key in the `user` field. We depict an example of a JSON backup, with only the relevant information present, in Fig. 10. If the attacker's nickname and the private key are sufficiently close together, they will both fall within the same sliding window of the compression algorithm. This ensures that there is the possibility of a backlink being created. However, in order to increase the chances of such a backlink being created, we include a string called a *canary* which we know is already included in the JSON backup, just before the string that we want to leak. This tricks the compression algorithm into finding redundancy between the nickname and the private key, creating the compression side channel that we need.

For our experiments, we used "privatekey" as a canary for the beginning of the private key and "=" for the end of the private key. The equal sign always appears as the last base64 character of the key due to padding of the private key during encoding, since the key is 32 bytes long and must be padded to a multiple of 6 bits. We noticed that both canary strings were necessary. The reasons for this are opaque to us. Between the two canary strings, the attacker needs to place, in order, its guess for the next character of the key, and the currently known characters of the key (initially none, thus an empty string). If the guess is correct, it should result in a shorter Threema Safe ciphertext. If the shortest ciphertext was induced by only one character among all guesses, then the currently known characters of the key are updated to include the current guess. Otherwise, two-character guesses can be attempted. This method effectively tries to guess characters of the base64 encoding of the private key from the last one to the first one.

In Fig. 9, we show the number of compression side-channel oracle queries needed to extract 31 characters of the private key, over 10,000 experiments; of these attempts 4727 were successful and 5273 failed to recover key material. Over the successful experiments, the median number of queries needed was 19,382 and the mean was 23,382. We remark that "failure" here means that our *automated* exploit did not successfully retrieve the key. If a targeted attack were to be run, one could manually guide the process. In fact, when a failure happens it is often because the exploit begins revealing characters of other parts of the backup, which can be easily spotted as the exploit is running.

To obtain the required queries, it suffices for an attacker to forcibly close the app and start it again. This induces the app into retrying to upload the backup. The forced closure can be done by either using debugging tools or by simulating user input on the unlocked phone. Then, in the setting of a compromised Threema Safe server, it is sufficient for the server to reply with an HTTP error code. Otherwise, a network attacker can forcibly close the connection, making the upload fail. Both of these techniques ensure that a new upload will be attempted at the next restart. At the pace of one restart every two seconds, an attacker can recover 31 base64 characters of the private key in a median time of about 11 hours (in a successful case).

Not all the bytes of the private key have to be recovered, since algorithms such as the van Oorschot-Wiener algorithm [61] can be used to recover the remaining bits. Indeed, this creates a trade-off between the number of queries to the oracle and the amount of subsequent, offline work required by the attacker. Since the offline work does not require access to the device anymore, it is convenient to execute just the sufficient number of queries required to make the offline attack feasible. For example, the van Oorschot-Wiener algorithm has time complexity $O(2^{n/2})$, where n is the size of the set to be searched for the desired value, and it is parallelisable. If we recover 31 base64 characters via the compression side channel (as above), we are left with 13 characters to recover, equivalent to a search space of size 2^{78} after base64 decoding. This makes the offline part of the attack perfectly feasible.