Breaking Cryptography in the Wild: Threema

Master Thesis

Kien Tuong Truong

September 13, 2022

Advisors: Prof. Dr. Kenneth G. Paterson, Matteo Scarlata

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

In this thesis, we analyze the Threema messaging application. Threema has been proposed by the media as a more secure alternative to messengers such as Whatsapp and Telegram, due to the emphasis on privacy put by the developers. This led Threema to be chosen as the messenger application of choice for the Swiss government and army, as well as various other private companies.

However, to our knowledge, Threema has not received a deep cryptographic analysis yet, despite being open-source and having received two external audits. We show that the Threema cryptographic design is flawed and we present multiple attacks against the protocols that Threema uses in its functioning. More specifically, we present attacks that break the confidentiality of messages between the client and the server, we show that replay and reflection attacks are feasible on the end-to-end level, and finally we show that an attacker can reveal long-term private keys of users in some settings.

Overall, we show that Threema does not provide important security properties that would be expected from a truly secure messaging application, and that Threema is not suitable for high-risk users, such as whistleblowers and political dissidents.

We conclude with an analysis of what, in our opinion, are the causes of the security flaws found in Threema. In particular, we focus on the deployment of new, custom protocols and we warn against careless composition of secure protocols.

Acknowledgements

The following work would not have been possible without the contribution of both of my supervisors, Kenny and Matteo, both of whom have helped me, given me ideas and, overall, pushed me to become a better researcher. Along them, I would like to thank the entirety of the Applied Cryptography group at ETH Zürich, who have been exceptionally welcoming since the first day I began my work with them.

From beyond the Alps, I would like to send my most sincere thanks to my parents, who have given me unshakeable support throughout the years, regardless of the distance between us. I love you, and I hope I can make you proud.

I would also like to thank my friend Giacomo, for the interesting discussions we had every day for the last two years, for the amazing food he cooked and, most importantly, for pushing me to become a better cryptographer.

I would like to thank all of the people that I met on my journey, all the people who, to this day, shine a light on my path to the future, pushing me to always strive for more, while putting a smile on my face. Thank you, in no particular order, Pietro, Arianna, Daniele, Dario, Davide, Gianmarco, Giorgio, Letizia, Massimo, Pasquale and Stefan for being the best group of friends I could have ever asked for.

Thank you to M, without which I would have never even thought about applying to ETH. Thank you to all the people of Tower of Hanoi that taught me how to play CTFs: the skills that I learnt back in Milan were vital for my time here in Zürich and I would have never written a tenth of this thesis if it weren't for the hacking abilities that I acquired.

In no particular order, I will name a few people that have helped making my days brighter while writing this thesis: A and E, for the *benessere*, M for showing me around Zürich and for keeping me company, D, L, and A for playing Counter Strike with me, during the few moments of free time I had.

Contents

Contents **iii**

1 Introduction **1**
- 1.1 The Setting: Messaging Applications 2
- 1.2 The Target: Threema 3
  - 1.2.1 The Security Promises of Threema 4
- 1.3 Related Work 5
- 1.4 Contributions 5

2 Background **7**
- 2.1 Notation Used 7
- 2.2 The Threema Cryptographic Toolbox 8
  - 2.2.1 Authenticated Encryption 9
  - 2.2.2 Curve25519 10
  - 2.2.3 Encryption Without Authentication 11
  - 2.2.4 Key Derivation Functions 12
  - 2.2.5 Compression Algorithms 14
  - 2.2.6 Encodings 15
- 2.3 Transport Layer Security 15
- 2.4 Randomness Sources 15
- 2.5 Secure Messaging 16
  - 2.5.1 Security Properties 17

3 The Threema Cryptographic Design **19**
  - 3.0.1 Methodology of the Analysis 19
- 3.1 The Threema Architecture 20
- 3.2 Registration Protocol 20
- 3.3 End-to-End Protocol 22
  - 3.3.1 Message Types and Serialization 23
  - 3.3.2 Encryption and Dispatch 24

3.4 Client-to-Server Protocol 28
3.4.1 The Handshake Protocol 28
3.4.2 The Transport Protocol 30
3.5 The Group Messaging System 32
3.6 Contact Discovery Protocol 33
3.7 Backup Methods 34
3.7.1 Threema Safe 34
3.7.2 ID Export 37
3.7.3 Data Backup 37
3.8 Threema Web 39
3.9 Application Access Control and Local Encryption 40

4 Attacks **43**
4.1 External Attacker Model 44
4.1.1 Attack 1 (Impersonation by Randomness Failure) . . . 44
4.1.2 Attack 2 (Vouch Box Forgery) 47
4.2 Compromised Threema Model 50
4.2.1 Attack 3 (Message Reordering and Omission) 50
4.2.2 Attack 4 (Replay and Reflection Attacks) 51
4.2.3 Attack 5 (Kompromat Forgery) 52
4.3 Compelled Access Model 53
4.3.1 Attack 6 (Cloning via Threema ID Export) 53
4.3.2 Attack 7 (Private Key Recovery through Compression Side-Channel) 54
4.3.3 Attack 8 (Invisible Salamanders in Group Messaging) 59
4.4 Discussion 59

5 Mitigations **63**
5.1 Mitigating Cross-Protocol Attacks 63
5.2 Preventing Tampering with Metadata 64
5.3 Mitigating Attacks on the C2S Protocol 65
5.4 Implementing Forward Secrecy in the E2E Protocol 66
5.5 Preventing Cloning via Threema ID Export 66
5.6 Mitigating the Compression Side-Channel 67

6 Conclusions **69**
6.1 Revisiting Threema's Security Promises 69
6.2 Analyzing Threema's Security Failures 70
6.3 Conclusions 71

A Generated Public Keys for the Vouch Box Forgery Attack **73**

Bibliography **75**

Chapter 1

Introduction

Cryptography is a field of research that permeates through the daily lives of an enormous share of the world's population. The Transport Layer Security (TLS) protocol protects over 98% of the traffic from Chrome users [63]. Whatsapp, one of the main secure messaging solutions for mobile, has more than two billion active users each month [91]. Privacy concerns, exacerbated by the Snowden revelations, combined with the necessity to follow regulations such as the GDPR, have pushed developers to embed stronger cryptography in their applications. This is the case, for example, of Whatsapp and Facebook Messenger migrating to the Signal protocol [89, 82] or Google doubling their share of traffic that is fully encrypted between 2014 and 2022 [63].

Open-source and provably secure cryptographic implementations such as libsignal [83], libsodium [26], and the Noise Protocol Framework [73] are nowadays publicly available for any developer to use, enabling the creation of more secure applications. Yet, it is not hard to still find software "in the wild" that implement cryptographic primitives and protocols incorrectly: vulnerabilities were found in famous and widely-used products such as Telegram [51, 6], Bridgefy [4, 5] and MEGA [10], and many pitfalls were found in protocols such as TLS [2, 9, 17, 31, 79, 96, 65] , EMV [11] and IPSec [25]. The impact of these attacks can be massive: people, companies, and governments rely on these products, expecting them to provide security in their communications. Not being able to provide such properties in practice opens the possibility of an attacker exploiting such vulnerabilities at scale, potentially endangering millions of people.

Messaging applications take a particular spotlight in this scenario: according to Whatsapp CEO Will Cathcart: "Whatsapp is able to deliver roughly 100 billion messages every day" [19]. This means that every day an enormous amount of personal information and media content is shared between people through these messaging apps, and third parties have large incentives to access the data that passes through those applications (e.g. governments looking to spy on their citizens, advertisers who want to provide targeted campaigns, or malicious parties searching for blackmailing material). It is, however, unreasonable to ask users to understand the cryptographic constructions underlying the applications they use. It is even more unreasonable to ask them to understand whether such constructions are secure. For this reason, people often migrate from one application to the other according to claims made by news outlets and competing applications, rather than a sound security assessment. For example, millions of users migrated from Whatsapp to Telegram due to the former's change in Terms of Service [49, 67], despite the latter being shown to be less secure [51].

In this thesis, we analyze Threema, which has been proposed as an alternative secure messaging application to other mainstream messengers and that has received the attention of the public due to Threema's commitment to anonymity and their focus on security. This aura of security has been further strengthened by the fact that the servers of Threema are based in Switzerland, a nation that is not part of the 5-eyes and 14-eyes intelligence sharing agreements, that is required to follow the GDPR as well as not being subject to the CLOUD act [46]. At the same time, they provide a different cryptographic design than Signal, which has been formally analyzed [35, 21]. This usage of a new and under-analyzed protocol makes it a good "in the wild" target for a cryptographic assessment. We begin with describing, at a high level, the objectives and architecture of a secure messaging application (Chapter 1.1). Then, we describe Threema and the security properties it is claimed to provide (Chapter 1.2).

1.1 The Setting: Messaging Applications

A plethora of messaging applications exists, among which we can include Whatsapp, Facebook Messenger, Signal, iMessage and Telegram, which are used worldwide, but also nation-specific ones such as WeChat and QQ, which are extremely common in China [77] and serve billions of users together [91].

All of them share the common goal of allowing users to send messages to each other over the Internet and mostly rely on the same high-level architecture: users download an application on their mobile devices (the *clients*) and connect to a centralized *server* that manages the messages. More specifically, users add other users to their contact list, often by giving the application access to their phonebook or by manually adding contact information in the app. Then, they are able to write to their contacts by sending a message to the server, who will relay it to the correct receiver. Other applications exist that rely on a different architecture, for example mesh networking messengers such as Bridgefy [50]. However, in this work, we will focus on the centralized setting.

Evidently, the server plays a major role in the communications: in a centralized application, a server is always able to launch a Denial-of-Service (DoS) attack by simply refusing to relay messages, which limits the guarantees that one can ask about the availability of the application. If encryption keys of other users are retrieved through the server, then the latter has the power to execute a man-in-the-middle (MitM) attack on the conversation by replacing the authentic keys with ones of which it knows the corresponding private key. Even in this setting, we want to prevent the server from spying on or interfering with the communication. This is the aim of *end-to-end security*: preserving confidentiality, integrity and authenticity between the sender and the receiver in the presence of a malicious or compromised central server. Unfortunately, there are limits to the end-to-end privacy guarantees: to be able to route messages, the server must have information about the user, such as the time of their last access, the (possibly pseudonymized) social graph of their communications and, often, their contact information. While it is very hard to hide this data from the server, we still wish to protect it, at least, from a network attacker that can analyze and interfere with the traffic from the client to the server. To this goal, messaging applications will often employ some encryption protocol such as TLS to establish a secure channel between the client and the server to prevent any metadata leakage.

1.2 The Target: Threema

Threema is a messaging application that in 2021 has seen a sharp rise in the number of users due to the change in Terms of Service of Whatsapp and now boasts over ten million users for their paid application [37].

It was created in 2012 under the name of “End-to-End Encrypted Messaging Application” or “EEEMA” and later renamed to “Threema” [43]. The app was created by Manuel Kasper, Martin Blatter and Silvan Engeler [94], three Swiss software developers, who later went on to found Threema GmbH, the company that now owns Threema.

The application started as a closed-source application, later publishing the source code in an effort to enable independent security reviews of Threema [41]. Threema GmbH has since released a whitepaper describing their cryptographic design [38]. Its code was also professionally audited twice, once by the Lab for IT security of the Münster University of Applied Sciences and once by Cure53 [39], the latter describing Threema’s code quality and project structure as being “unusually solid”.

This transparency helped Threema foster a large community of users: by 2022 the Threema developers claimed to be serving “more than 10 million users" worldwide, as well as more than 1000 companies that use their enterprise-level messaging app, Threema Work [37]. In response to Whatsapp's change in Terms of Service, Threema has been advertised as a more secure messenger [45], and their success led to the Swiss government and Arma Suisse declaring them as their recommended method for communicating internally [34, 93]. Reportedly, Olaf Scholz, the chancellor of Germany, uses Threema as his main method of communication [78].

1.2.1 The Security Promises of Threema

The developers claim that Threema is a "secure messenger". We now try to describe at a high level what security properties Threema is claimed to have. Later, we will discuss how they relate to the properties provided by other messaging apps. The following are taken from the Threema Cryptographic Whitepaper [38], with preserved wording:

1. *End-to-End Encryption*: Messages sent from one user to the other are encrypted so that anyone who sees the ciphertexts, including the server, cannot decrypt them.
2. *Integrity*: No third party should be able to tamper with or forge messages between two users.
3. *Local Group Handling*: The server does not know which groups exist and which users belong to which group.
4. *Forward Secrecy at the Client-to-Server level*: An external attacker who collects Client-to-Server ciphertexts, and then leaks a long-term key of either the client or the server, cannot decrypt any past message.
5. *User Authentication at the Client-to-Server level*: A user can only log in with a Threema ID if they are in possession of the private key corresponding to that ID.
6. *Repudiability of Messages*: Any recipient can forge a message that will look as if it was generated by the purported sender.
7. *Replay/Reflection Attack Prevention*: A malicious server should not be able to replay or reflect messages to a user without knowing their private key.
8. *Protection of Private Keys in Local Storage*: Private keys should not be accessible by other apps on the same device or by unauthorized users.
9. *Anonymity of Threema Safe Backups*: A Threema Safe backup server cannot tell which backup belongs to which Threema user by looking at the uploaded data.

1.3 Related Work

Security Analysis of Threema To our knowledge, there are no in-depth analyses of the entire cryptographic design of Threema. Threema itself has released a description of some of their cryptographic choices [38], but without formalizing the protocols they created. The two audits cited above concern mostly secure programming within the application, while also discussing cryptographically relevant aspects such as constant-time programming, but without deeply investigating the cryptographic protocols. Rösler et al. [80] compare the security of group chats in Threema with other messaging applications, finding a vulnerability that would allow an attacker to replay messages to a group of users. The issue has been fixed since version 3.14 of the application. A security researcher who goes by the nickname "Soatok", has written a blog post about Threema [90], highlighting some vulnerabilities in Threema. We discuss part of their findings in detail later, when describing the list of vulnerabilities (Chapter 4).

Cross-Protocol Attacks Protocols that are secure independently of each other may not preserve their security when used at the same time. Examples of this emerged in SSL 3.0 [96], which was extended by [65] for TLS 1.2. A cross-protocol attack allowing an adversary to decrypt TLS ciphertext by downgrading connections to SSL 2.0 was presented in [9]. Similar issues were found in TLS application-level protocols [17]. In Threema, we show that a malicious actor can create payloads with different meaning depending on the context, breaking client authentication.

Compression Side-Channel Attacks In 2002, Kelsey [54] described a side-channel created by compression algorithms: by analyzing the length of the ciphertext resulting from compressing and then encrypting a message, an attacker can learn the contents of the latter. Duong and Rizzo used the side-channel to steal cookies from HTTPS sessions, using the fact that the TLS and SPDY protocols compress messages before encrypting them. This same problem is found in Threema, where the client compresses and then encrypts a backup, which it then sends to the server. We use a similar attack method on Threema, in order to leak the long-term private key of users.

1.4 Contributions

We first analyze the code of the Threema Android application and extract a formal abstraction of the protocols used to secure communications both end-to-end and client-to-server (Chapter 3). We describe multiple vulnerabilities that we discovered within the Threema application, both theoretical and practical, analyzing multiple threat models and describing the impact of the attacks for end users (Chapter 4). Namely, we provide a description of the following attacks, as well as including other security considerations about Threema:

1. We show that leaking a single client ephemeral key leads to permanent impersonation by the adversary.
2. We break client authentication in the client-to-server channel by exploiting a cross-protocol vulnerability, using messages from the end-to-end protocol.
3. We show that Threema does not enforce message ordering and that a malicious server is able to reorder messages, as well as replay and reflect messages under certain conditions.
4. We use a compression side-channel to recover the long-term key of a client that is using the cloud backup solution provided by Threema.

Lastly, we describe mitigations for the vulnerabilities found, both in the short term, by non-invasively modifying their protocols, and the long term, suggesting well-analyzed protocols as alternatives (Section 5).

Chapter 2

Background

In this chapter, we present an overview of the notation used throughout this work (Section 2.1) and of the cryptographic primitives used by Threema, along with their properties (Section 2.2). Finally, we define the concept of "secure messaging", as well as desirable properties for a messaging application from a cryptographic standpoint (Section 2.5).

2.1 Notation Used

We use the typical notation utilized in the literature to describe protocols in pseudo-code.

- $\{0,1\}^k$: a bit-string of length k.
- ε: the empty string.
- $\{0, ..., 255\}$: the set of all possible values for a single byte.
- $\{0, ..., 255\}^*$: the set of all possible byte strings.
- $a \oplus b$: the result of the eXclusive OR operation (XOR) between byte strings a and b.
- $|x|$: if x is a (byte-)string, the number of characters (resp. bytes) in the string. The character set will be clear from context. If x is a set, the size of the set.
- $x \leftarrow\!\!\$\ \mathcal{X}$: x is a value sampled uniformly at random from a set $\mathcal{X}$. For the purposes of this thesis, $\mathcal{X}$ will always have finite size and, thus, it is always possible to sample uniformly at random.
- $x \leftarrow y$: x is assigned the same value as y.
- $x \leftarrow \mathcal{A}(y)$: x is assigned the value returned from the *deterministic* algorithm $\mathcal{A}$ with input y.

- $x \leftarrow\!\!\$\ \mathcal{A}(y)$: x is assigned the value returned from the *probabilistic* algorithm $\mathcal{A}$ with input y.
- $s_1 \parallel s_2$: the concatenation of the strings s_1 and s_2, in that order.
- $T[k]$: assuming that T is a table that maps keys to values, the value contained in T corresponding to key k. If there is no such value, then $T[k]$ returns $\perp$. To assign the value v to the key k, we write $T[k] \leftarrow v$.
- $s[i]$: assuming that s is a string or an array, the element of s at position i. We assume that the initial index is 1.

Let G be a group and let g be a generator of G. For the remainder of this work, we will use multiplicative notation, even though all operations are done on an elliptic curve, where additive notation would be more appropriate. We prefer doing so since the multiplicative notation is more common and the choice of notation is not relevant for the purposes of this thesis.

As a convention, we use the letters a and b when referring to *long-term* secret keys and we use A and B as the corresponding public keys ($A = g^a$ and $B = g^b$ respectively). We use the letters x and y when referring to *ephemeral* secret keys and we use X and Y as the corresponding public keys. This last notation will extend to the case where the keys are not strictly ephemeral but where they are reused across multiple runs of the protocol, but in a limited span of time.

We now list the notation we use to describe the Threema protocols:

- When referring to users, we denote them with symbols $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$... or $\mathcal{U}_i$, $i \in \mathbb{N}$.
- The server is denoted with $\mathcal{S}$.
- Each user is assigned an 8-character string by the server, called the *Threema ID*. We denote the Threema ID of user $\mathcal{U}$ as $\mathsf{ID}_{\mathcal{U}}$.
- We denote the function that the server uses to generate Threema IDs as $\mathsf{GenID}()$.[1] This function returns an 8-character string composed of uppercase letters and digits.
- If not using other names, we will refer to the private-public keypair of user $\mathcal{U}$ as $(\mathsf{pk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{U}})$.

2.2 The Threema Cryptographic Toolbox

Threema reuses few primitives to build multiple protocols. We now describe the cryptographic libraries and functions used, highlighting their properties

[1]We do not know how the generation process is done server-side, as we do not have a way to analyze the server code

and interfaces. The primitives for authenticated encryption (Section 2.2.1) and key exchange (Section 2.2.2) come from the usage of NaCl, a cryptographic library created by Bernstein, Lange and Schwabe and described in [14]. NaCl attempts to provide an easy-to-use interface, while at the same time providing high speeds and high security margins. The main abstraction provided by the library is the `crypto_box` function, which combines an elliptic curve Diffie-Hellman key exchange and an authenticated encryption algorithm. We study both of these components separately in the next sections. For a more detailed overview, we refer to [12] which describes how cryptography is implemented in NaCl. Threema uses a pure Java porting of the original C code, provided by the jnacl library. [2]

Furthermore, we study additional primitives used by Threema, namely: password hashing algorithms, key derivation functions and compression methods.

2.2.1 Authenticated Encryption

To provide authenticated encryption, Threema relies on the nonce-based XSalsa20-Poly1305 ciphersuite, composed of the XSalsa20 stream cipher and the Poly1305 one-time authenticator. XSalsa20-Poly1305 is the composition proposed by Bernstein in [12]. The ciphersuite provides the following interface:

- $c \leftarrow \mathsf{AEAD.Enc}(k, m; n)$: given a 32 byte key k, a message m and a 24 byte nonce n, compute the ciphertext c.
- $m \leftarrow \mathsf{AEAD.Dec}(k, c; n)$: given a 32 byte key k, a ciphertext c and a 24 byte nonce n, compute the message m corresponding to the ciphertext or $\perp$, if the decryption fails. Assume that $\perp$ is different from any value returned from a successful decryption.

We denote the nonce space as $\mathcal{N} = \{0,1\}^{192}$.

On a high level, the encryption algorithm works by combining the key and the nonce in the Salsa20 streaming function[3] to obtain a keystream, part of which is used as key for the authenticator, while the rest is used to encrypt the plaintext. More specifically, the second part of the keystream is XOR-ed with the plaintext to obtain a ciphertext, which is then fed into the Poly1305 authentication algorithm, yielding a 16 byte authentication tag. When decrypting, the tag is checked, ensuring integrity, after which the ciphertext is decrypted by XOR-ing the keystream again.

When using XSalsa20-Poly1305 nonces must be carefully handled: reusing the same nonce to encrypt different messages under the same key may lead

[2] https://github.com/neilalexander/jnacl
[3] See Section 7 of [12] for further details

to loss of confidentiality and of integrity. This is because using the same key and nonce leads to generating the same keystream. This means, on the one hand, that one can XOR the two ciphertexts together to obtain the XOR of the plaintexts, as the keystream will cancel out. On the other hand, this also means that the authentication key is the same for both ciphertexts. Due to the properties of the Poly1305 authenticator, this allows for a key-recovery attack on the authentication key, which allows the attacker to forge new tags under the same key-nonce pair. Thus, one must take into consideration the possibility of a collision when choosing to sample nonces uniformly at random, due to the birthday bound. It is often more convenient to initialize a counter with a value C and increment it by 1 for each message encrypted. Unless the counter wraps around and assuming that there exists a method to reliably preserve state in the application, this ensures that no nonce is used twice. We denote with $\mathsf{ctr} \leftarrow \mathsf{Ctr.Init}(C)$ the creation of a new counter ctr with a starting value C. In Threema, C is a 16 byte value called the *cookie* which is set as the most significant part of the nonce. The least significant part is initialized to the big-endian representation of the number 1. We denote the generation of a new nonce n from the counter as $n \leftarrow \mathsf{ctr.Next}()$, where the current value of the nonce is returned and the counter is increased by 1.

We highlight some interesting properties of XSalsa20-Poly1305. First, assuming that the Salsa20 expansion function[4] is a PRF and that Poly1305 is ε-almost-Δ-universal (for some suitable parameters ε and Δ), then the composition is provably AE-secure. A proof for ChaCha20-Poly1305 was given by Degabriele et al. [24], which can be slightly modified to obtain a valid proof for XSalsa20-Poly1305. Secondly, since XSalsa20 is a stream cipher, encryption preserves information about the length of the plaintext m: the resulting ciphertext c will satisfy $|c| = |m| + |\tau|$, where τ is the authentication tag for Poly1305, whose length is 16 bytes. This last fact enables our attack in Section 4.3.2, since it allows us to infer the length of the plaintext from the length of the ciphertext.

2.2.2 Curve25519

Curve25519 was proposed by Bernstein as an elliptic curve to enable fast and secure Diffie-Hellman computations [13]. Curve25519 is a Montgomery curve defined over the prime field $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$ and operates only on the x-coordinate of points by use of the Montgomery ladder algorithm. This allows to have both public keys and private keys of 32 bytes.

On a high level, the interface provided by Curve25519 is the following:

[4]As specified in [12], Section 7

- $(x, X) \leftarrow\!\!\$\ \mathsf{KeyGen}()$: generate a private key x and the corresponding public key X.
- $K \leftarrow \mathsf{DH}(x, Y)$: given a private key x and a public key Y, compute the shared Diffie-Hellman value K. More precisely, given a scalar x and the x-coordinate Y of a point on the curve P_Y, compute the scalar multiplication $K' \leftarrow [x]P_Y$. Finally return K, the 32 byte big-endian representation of K'. This function is also called the X25519 function in [13].

Let g denote the base point for Curve25519 defined in [12] (namely, the point with $x = 9$): we can compute a public key X for the private key x by computing $X \leftarrow \mathsf{DH}(x, g)$. Before being multiplied, x is passed through a clamping step, to ensure that the private key is large enough and that it is a multiple of 8, the cofactor of the curve.[5] For conciseness, we will abbreviate the DH step to $X = g^x$. Due to the preprocessing, this abbreviation is a slight abuse of notation. However, since no part of our analysis of Threema relies on the internals of Curve25519 operations, it is sufficient to consider the entire $\mathsf{DH}(\cdot, \cdot)$ operation as a standard Diffie-Hellman key exchange.

2.2.3 Encryption Without Authentication

In some instances, Threema does not use authenticated encryption, opting to use encryption without an authentication mechanism. For example, this is done when saving data locally on Android devices, where access control is sufficient to prevent other applications or users from accessing and tampering with the stored data. Depending on the context, one of the following two options is used:

- XSalsa20: using only the stream cipher, without the authentication.
- AES-CBC [32]: the Cipher Block Chaining mode of operation applied using the AES block cipher.

Due to the lack of authentication, neither encryption scheme is secure against a chosen ciphertext attack. In particular, both options are malleable: since the XSalsa20 keystream is XOR-ed directly to the plaintext, flipping a bit in the ciphertext also flips the same bit in the plaintext. For CBC, flipping a bit in a ciphertext block also flips the same bit in the next plaintext block, while mangling the current plaintext block.

XSalsa20 provides the same interface as XSalsa20-Poly1305. On the other hand, AES-CBC requires an IV in its operation, which we assume to be

[5]This is necessary because using a multiple of the cofactor as a private key in the X25519 function ensures that, if the public key of the other party received lies in a small subgroup, the result of a Diffie-Hellman computation will be $\mathcal{O}$, the point at infinity, leaking no information about the private key.

generated internally in the encryption function and is part of the outputted ciphertext. Thus, the following will be the interfaces provided:

- $c \leftarrow \mathsf{XSalsa20.Enc}(k, m; n)$: given a 32 byte key k, a message m and a 24 byte nonce n, compute the ciphertext c.
- $m \leftarrow \mathsf{XSalsa20.Dec}(k, c; n)$: given a 32 byte key k, a ciphertext c and a 24 byte nonce n, compute the message m corresponding to the ciphertext. Note that this decryption cannot fail, since no integrity check is done.
- $c \leftarrow \mathsf{AES\text{-}CBC.Enc}(k, m)$: given a 32 byte key k, a message m, compute the ciphertext c. Internally, this generates a 16 byte IV and outputs it as the first block of the ciphertext.
- $m \leftarrow \mathsf{AES\text{-}CBC.Dec}(k, c)$: given a 32 byte key k and a ciphertext c, compute the message m corresponding to the ciphertext. Note that this decryption cannot fail, since no integrity check is done.

2.2.4 Key Derivation Functions

A key derivation function (KDF) is used to obtain a cryptographic key starting from another secret value, for example the output of a Diffie-Hellman key exchange or a password. The objective of a KDF is twofold: a KDF can be used to extract from a value unsuitable for immediate cryptographic usage (e.g. a password) something that can be used for cryptographic purposes, such as a fixed-length key for an encryption scheme. On the other hand, a KDF can also be used in combination with a set of *labels* to generate multiple keys from some initial high-entropy secret value. In [66, Sec. 13.5.1], Menezes et al. have described the importance of using different keys in different modes of operation, a piece of cryptographic good practice that is commonly called the "Key Separation Principle". Using a KDF with a set of labels helps achieve this principle in practice. Formally, this requires that an adversary that does not know the initial secret value cannot distinguish outputs of the KDF from random bit strings, even with knowledge of the labels used. A formal treatment of the security of KDFs is given by Krawczyk in [58].

We now describe the different KDFs that are used by Threema. When deriving a key using a KDF, Threema takes as input either a password or another cryptographic key. In the first instance, standardized implementations are used, whereas to derive keys from other keys, Threema employs a keyed hash function, calling the construction the "Threema KDF".

The Threema KDF

To derive a key from another cryptographic key, Threema uses the BLAKE2b hash function [8]. The interface used by Threema is the following:

- $K \leftarrow \mathsf{BLAKE2b}(P, \sigma, \tau)$: given a password P, a salt σ and a label τ, compute a 32 byte value K to be used as cryptographic material.

This method is only used in one location in the code, namely to derive a key for the encryption of the metadata box from the key used to encrypt the plaintext (see Section 3.3). In that instance, the values of the salt σ and the label τ are, respectively, the fixed strings "mm" and "3ma-csp".

Password-based KDFs

In many places, Threema uses a user-defined password for cryptographic purposes, such as encrypting data. This has the advantage that a password is often more memorable than a string of random bytes, but, on the other hand, has much lower entropy. To prevent brute-forcing, dictionary attacks, and rainbow table-based attacks, a specialized password hashing algorithm should be used, in combination with a random byte string called *salt*.

For Threema, scrypt [72] is the main algorithm of choice, superseding their previous choice of PBKDF2 [69]. The former is often preferred, since it is memory-hard [7] and, thus, hinders brute-force attacks on specialized hardware. Where PBKDF2 was used, the Threema application tries to update the key material in order to use scrypt.

The scrypt algorithm is called with the following fixed parameters: $n = 65536$ (the cost parameter), $r = 8$ (the block size), $p = 1$ (the parallelization parameter). Since the parameters n, r, p are the same throughout the application, we do not include them in the abstraction.

Similarly, the PBKDF2 algorithm is called with fixed parameters: 100000 for the number of iterations and "HmacSHA256" for the Pseudo-Random Function (PRF) used.

In addition to those fixed parameters and for both algorithms, the caller must provide a password P, a salt σ and the length of the output in bytes λ.

- $K \leftarrow \mathsf{scrypt}(P, \sigma, \lambda)$: given a password P and a salt σ, return a value K of length λ bytes to be used as cryptographic material using the scrypt algorithm.

- $K \leftarrow \mathsf{PBKDF2}(P, \sigma, \lambda)$: given a password P and a salt σ, return a value K of length λ bytes to be used as cryptographic material using the PBKDF2 algorithm.

The cryptographic material K can be safely used as a symmetric key for authenticated encryption, assuming that the password P has sufficient entropy.

2.2.5 Compression Algorithms

A compression algorithm is used on a document in order to decrease its size by removing redundancy. Threema uses multiple methods to compress data, but they all fundamentally rely on the DEFLATE algorithm [27]. On a high level, the DEFLATE algorithm uses a sliding window and, whenever a sequence of bytes appears twice within the window, the one that occurs later is replaced with a backreference to the first occurrence. Afterwards, the algorithm uses Huffman coding to replace symbols with a more efficient representation.

Threema uses two different compression algorithms for different purposes: for cloud backups, it uses the gzip algorithm provided by the Java standard library, while for local data backups it uses an external library, called Zip4j [62], to create an encrypted ZIP archive, following the AE-2 WinZip standard [23]. Both the gzip and the zip file formats rely on the DEFLATE algorithm, with the difference that zip compresses the files separately, rather than employing solid compression (i.e. concatenating the files together and then passing them through DEFLATE). This difference does not matter for our purposes, since whenever gzip is used, there is only one file that is compressed.

We give an overview of the AE-2 WinZip standard. The user sets a password P for the archive. Then, each file to be inserted into the archive is compressed separately by using the DEFLATE algorithm. For each file f, a random 16 byte salt is sampled[6] and utilized to derive a file encryption key $K_{\text{enc},f}$ and a file authentication key $K_{\text{mac},f}$ from the password P using PBKDF2. The file is encrypted under $K_{\text{enc},f}$ using AES-CTR with null IV. The ciphertext is then passed through HMAC-SHA1 under key $K_{\text{mac},f}$ in order to derive an authentication tag. Finally, the salt, the ciphertext and the tag are appended to the archive and the algorithm proceeds to the next file. We note that file names are not encrypted in the zip metadata and thus will be readable even without a password.

We only require the interface of the $\mathsf{gzip}(\cdot)$ function and its inverse.

- $m' \leftarrow \mathsf{gzip}(m)$: given a message m (byte string), return its compressed version m' using the DEFLATE algorithm.

- $m \leftarrow \mathsf{gunzip}(m')$: given a compressed message m', return its uncompressed version m.

[6]This is assuming AES-256 is used, since different encryption algorithms cause different lengths to be used for the salt

2.2.6 Encodings

Threema uses multiple encodings for data, especially when representing cryptographic keys. The three encodings used are hex encoding, base32 encoding and base64 encoding, a description of which can be found in [52].[7]

We use the following interface:

- $m' \leftarrow \mathsf{hex.Encode}(m)$: given a message m (byte string), return its hex-encoded version.
- $m \leftarrow \mathsf{hex.Decode}(m')$: given a hex-encoded message m', return a byte string representing the decoded message.
- $m' \leftarrow \mathsf{b64.Encode}(m)$: given a message m (byte string), return its base64-encoded version.
- $m \leftarrow \mathsf{b64.Decode}(m')$: given a base64-encoded message m', return a byte string representing the decoded message.
- $m' \leftarrow \mathsf{b32.Encode}(m)$: given a message m (byte string), return its base32-encoded version.
- $m \leftarrow \mathsf{b32.Decode}(m')$: given a base32-encoded message m', return a byte string representing the decoded message.

For short, a value x has a base64 string representation denoted by x_{b64}, which is implicitly encoded and decoded as needed. Similarly, its hex string representation is denoted by x_{hex} and the base32 string representation is denoted by x_{b32}.

2.3 Transport Layer Security

Transport Layer Security (TLS) is a protocol used to protect communications over TCP by establishing a secure channel between two parties. Its latest version, TLS 1.3, is standardized in [75]. TLS has been extensively analyzed in the literature throughout its history [15, 70, 59, 28] and is now reliably used by billions of people.

Threema uses TLS to protect most, but not all, of their communications, employing techniques such as certificate pinning within the application to further prevent Man-in-the-Middle attacks.

2.4 Randomness Sources

Most cryptographic applications require a reliable and unpredictable source of random bits. This is used, for example, to sample cryptographic keys,

[7]Hex encoding is called Base16 in the document.

nonces or IVs in a way that is close as possible as sampling them uniformly at random from the space of all their possible values. The Android and iOS operating systems provide developers with functions that return bytes sampled from a secure randomness source, which we usually call *system randomness*.

In our analysis, we will mostly abstract away all explicit usage of a randomness source. For example, the key generations described in Section 2.2.2 implicitly use secure randomness. We assume that for all cryptographic purposes, Threema samples by means of a secure randomness source. Whenever we need to explicitly model usage of bits from secure randomness, we use the following interface:

- $s \leftarrow\!\!\$ \; \mathsf{SecureRandom}(\lambda)$: sample λ bits from system randomness and return them.

2.5 Secure Messaging

Threema is but a single fish in a sea of competing messenger applications. Each of them provides different properties and has a different cryptographic design. The question we now turn our attention to is: which properties should we strive for in a truly secure messaging application? Against which sort of adversaries should we protect?

Confidentiality, integrity, and authenticity appear to be the baseline for security properties: a messenger cannot be considered secure if a third party, or even the server itself, can observe messages in plaintext. Nor should it be able to alter the conversation by injecting new messages or modifying messages in flight. Furthermore, an attacker should not be able to claim another user's identity as their own. However, we argue that these properties, while essential, are not sufficient.

On the one hand, a passive attacker still has the possibility to store all messages in transit, waiting for the opportunity to decrypt them when they manage to obtain the secret key by some means. This directly implies that keys must be periodically changed, in order to reduce the impact that a single key leakage would have. On the other hand, we would also like to ensure that if a single key is leaked to a passive adversary, then the protocol can "heal" and provide confidentiality and integrity after some time, by injecting randomness in the exchange. These two properties are commonly denominated "forward secrecy" and "post-compromise security", respectively. The term "perfect forward secrecy" was first coined by Günther in [47] and has since been a property that protocols such as Off-the-Records Messaging (OTR) [16], and later Signal [84] have tried to achieve. Post-compromise security, on the other hand, has been formalized in [22]. Signal manages to provably provide both properties, as shown in [21].

One might ask whether providing such strong properties may hinder usability. For example, it may render desirable features impossible to implement due to security constraints, making the developer unable to meet the user's needs. Additionally, it can be argued that leaking session keys or long-term keys requires a very strong attacker and is a feat that is unlikely to happen in day-to-day communications. However, we note that Signal provably provides both properties without affecting in any noticeable way the user experience. This sets a baseline for any other messenger application, at least from a security standpoint. For this reason, we argue that a secure messenger should strive to obtain at least what is provided by Signal and, if possible, even more.

Threat Models While an external attacker is the main concern when trying to protect communications, we stress that all the properties cited above must hold even in the case where the server has been compromised. Indeed, as the server often acts as the message router and as a directory for public keys, it would be a very strong assumption to believe the server to be a trusted entity. Additionally, the companies that run large-scale messaging applications are also susceptible to subpoenas, search warrants, and espionage. A truly secure messaging protocol must account for this attack point in its design. It is unclear if Threema assumes a compromised server to be within their threat model.

2.5.1 Security Properties

When designing a new protocol it is of key relevance to assess which properties the future and potential users will desire. This is a non-trivial task which presents multiple problems. Ermoshina et al. [33] highlight two problems for secure messaging app developers. First, there is a disconnect between users and developers in understanding which are the needs of the users, since the former do not necessarily understand the security provided by a protocol, while the latter may often have to speculate which properties will be required by the users. Second, there exist high-risk users, such as whistleblowers and political dissidents, which have different needs from low-risk users in terms of the security properties required. We argue that the same issues apply to Threema, as their threat model is unclear, meaning that the application might not be suitable for high-risk users. On the one hand, if their application is not meant for high-risk users, they are not explicitly stating so in any of their documentation. On the other hand, if the Threema developers believe their app to be used for high-risk users, we believe there to be a mismatch between the security properties that Threema explicitely provides, described in Section 1.2.1, and the ones that have been shown to be needed by such users.

In their Systematization of Knowledge paper, Unger et al. [95] give an overview of properties that a secure messaging application may have. Among the properties that they highlight, we stress that Threema does not claim to provide and, indeed, does not provide a few important features. First, forward secrecy in Threema is only guaranteed at the client-to-server level, rather than at the end-to-end, meaning that more trust is required towards the central messaging server than usual. Second, it does not enforce any property on ordering, neither causal (i.e. messages should be visualized to the user in the order in which they were sent) nor global (i.e. all participants in a group communication should see messages in the same order). Lastly, it does not mention post-compromise security.

Signal and, more recently, Whatsapp manage to provide all of the aforementioned properties in their design. In particular, most of their cryptographic design revolves around the *double ratchet algorithm*, which creates a chain of KDF steps and of Diffie-Hellman key exchanges in order to securely generate new keys for each message that is sent. This ensures that the impact of a single key leaking is minimal. Threema, on the other hand, relies on a different cryptographic design which we will describe in the next chapter, which lacks relevant security properties.

Chapter 3

The Threema Cryptographic Design

Threema presents multiple protocols necessary for its functioning. We start with an overview of the architecture of Threma, giving a bird's eye view of the servers and how the application is meant to be used. Then, we discuss the cryptographic design. At its core, we find the messaging protocol, which is responsible for encrypting messages end-to-end, securely sending them to the server, and decrypting them at the other end. This protocol is split into two parts, an end-to-end protocol that uses the long-term keys of the users (Section 3.3), and a client-to-server protocol that establishes a secure channel between the user and the server (Section 3.4). Additionally, a registration protocol is necessary for users to create accounts and to record their public key with the Threema server (Section 3.2). The contact discovery system allows user to discover which of their contacts is also using Threema (Section 3.6). Furthermore, Threema allows the user to backup their personal data in three different ways, which we analyze separately (Section 3.7). Finally, we briefly describe the solution that Threema uses for multi-device support, called by its developers "Threema Web" (Section 3.8).

3.0.1 Methodology of the Analysis

In order to analyze the architecture of the Threema application, we mainly analyzed the source code of the mobile application, which has been published by Threema on Github. We mainly analyzed the version of the Android mobile operating system.[1]

The cryptographic description for the client was abstracted from the Java code of the application. Since we did not have access to any server code, we inferred its functionality from the replies to the client's messages. In our analysis, we tried to assume as little as possible of the server, which means that its behaviour may differ from what would be expected from a honest

[1]https://github.com/threema-ch/threema-android

server. Thus, in our analysis we must consider the possibility of the server acting maliciously.

To test the results of our analysis we referred to a previous description of the Threema protocol written by Jan Ahrens [3], which is limited to the client-to-server protocol, and the Threema whitepaper [38], which describes the architecture at a high level. Additionally, we wrote various scripts in Python that simulated specific client functionalities in order to test the server's behaviour.

3.1 The Threema Architecture

The high-level architecture of Threema is depicted in Fig. 3.1. Each user has their own mobile device with the Threema application installed on it. Throughout its functioning, this device connects to multiple Threema servers, each of which serves a different purpose. The *directory server* contains the list of all Threema users and is used at registration time to register a new public key, as well as to receive other users' public keys when contacting them. The directory server exposes a REST API to the public, served over HTTPS. The *chat server* is used to send and receive messages. A custom client-to-server protocol is used by clients to securely connect to this server over TCP. The *Threema Safe server* is used to upload and retrieve backups created with the cloud backup feature of Threema (Section 3.7.1). Finally, the *media server* is used to upload media messages meant for other users, in encrypted form. The connection with the media server also happens over HTTPS.

From now on we will often refer to a "server" without explicitly describing which server we mean, since it is not relevant and the server we refer to will be clear through context. Note that usually a single abstract server is concretely composed of multiple physical servers, among which the load is shared and that can communicate with each other. Since this aspect is irrelevant for our work, we will only speak of a single server, even when multiple servers are involved.

3.2 Registration Protocol

Whenever a user C first creates an account, they are prompted to move their finger around the screen for a few seconds. This process is used to harness entropy from the user, by regularly collecting the (x,y) position of the finger along with the timestamp, and hashing all the positions and timestamps collected using SHA-256. The result is a 32 byte string called "seed". Let $\mathsf{SeedGen}()$ be the procedure that generates the seed. Another random 32 byte string is sampled from system randomness and is XOR-ed together

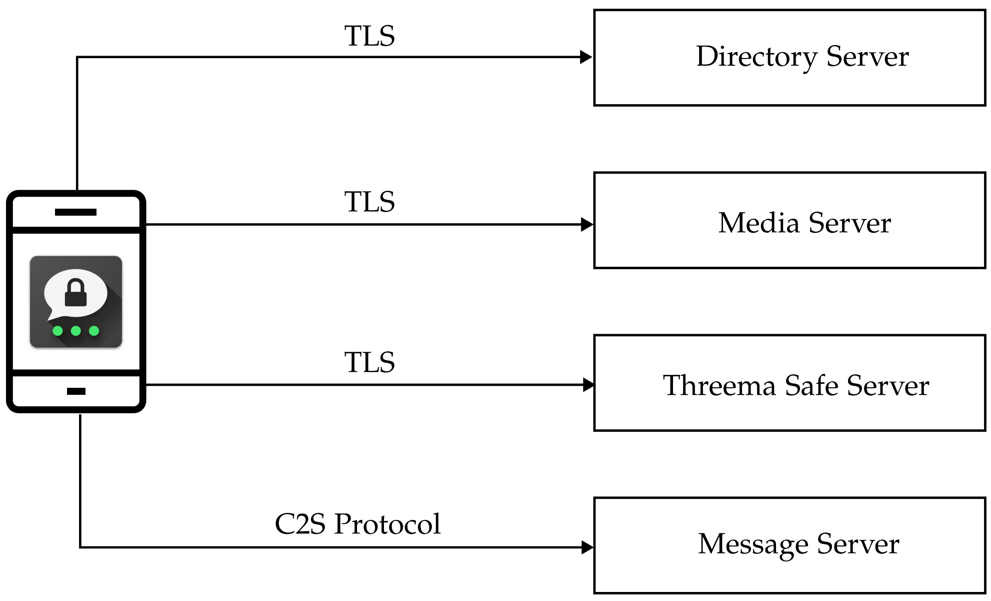


Figure 3.1: The high-level architecture of Threema.

with the seed to obtain the private key a. The corresponding public key $A = g^a$ is then computed from the private key and sent to the server, along with the license that proves that $\mathcal{C}$ has bought the app. The license can be either bought from standard digital distribution services such as the Google Play Store and the Apple App Store, or from Threema directly, through their website. We note that whenever a license is bought, the user's personal data is linked to the license code, as it is required for completing the purchase. Whenever the license code is sent to the server during registration, it would be possible for Threema to match a Threema account to a particular identity, breaking pseudonymity.

Algorithm 1 shows the key generation process in pseudocode.

Algorithm 1 Threema Long-term Key Generation (LongTermKeyGen)

```
1: procedure LongTermKeyGen( )
2:     $s \leftarrow\!\$ \mathsf{SeedGen}()$
3:     $s' \leftarrow\!\$ \mathsf{SecureRandom}(256)$
4:     $a \leftarrow s \oplus s'$
5:     $A \leftarrow g^a$
6:     return $(a, A)$
```

Afterwards, a challenge-response protocol is run between the user and the server in order to register the newly generated public key. The protocol is run over a TLS-protected connection, with the support of certificate pinning on the client to avoid MitM attacks. The server samples an ephemeral keypair (y, Y) and a random byte string $\widetilde{m}$. Afterwards, it prepends a `0xff` byte to $\widetilde{m}$ to obtain the challenge value chall, which is then encoded in base 64. The reason for prepending the `0xff` byte is to patch a vulnerability that we will discuss in Section 4.2.3. The ephemeral public key Y and the encoded challenge $\mathsf{chall}_{\mathsf{b64}}$ are both sent to the client. To prove possession of the private key, the client derives a key from the long-term private key a and the given ephemeral key Y: $K_{\text{reg}} \leftarrow \mathsf{DH}(a, Y)$. This key is then used to encrypt chall with the fixed nonce "`createIdentity response.`", sending the corresponding ciphertext to the server. If the ciphertext decrypts correctly, and the plaintext corresponds to the original challenge chall, then the protocol succeeds. Finally, the server samples a new Threema ID and returns it to the user. Internally, it will also map the public key to the Threema ID in its own database. If the protocol fails at any point, the client can retry by sampling a new key pair and restarting the protocol. Figure 3.2 shows the flow of the registration protocol.

Assuming that the server has not been compromised, the only way by which the client can pass the verification with non-negligible probability is by knowing either the private key a or the private key y. Since (y, Y) is randomly sampled by the server during the protocol and discarded immediately after the protocol has run[2], the latter case is unlikely to happen.

Multiple accounts can be created under the same license, however Threema limits the number of Threema IDs that can be generated.

3.3 End-to-End Protocol

We now discuss the cryptographic core of the encrypted messaging provided by Threema. The messaging protocol itself is the result of the composition between two protocols: an end-to-end protocol, whose responsibility is to provide confidentiality and authenticity between two users against any third party, and a client-to-server protocol, which establishes a secure channel between the user and the server, in order to hide metadata from network attackers. In practice, this means that packets are encrypted twice: the actual message is encrypted and placed within an end-to-end packet, which is itself re-encrypted and placed within a client-to-server packet. We begin the analysis by describing the former protocol, the End-to-End protocol (or E2E Protocol).

Assume Alice (denoted $\mathcal{A}$) and Bob (denoted $\mathcal{B}$) want to communicate and that each of them has the authentic long-term keypair of the other party. We denote their keypairs as (a, A) and (b, B), respectively. Without loss of

[2]This is inferred by the fact that registering multiple times in a short timespan leads to different keys being used.

Client $\mathcal{A}$		**Server** $\mathcal{S}$
$(a, A) \leftarrow\!\!\$ \ \mathsf{LongTermKeyGen}()$	$\xrightarrow{A}$	$(y, Y) \leftarrow\!\!\$ \ \mathsf{KeyGen}()$
		$\mathsf{chall} \leftarrow\!\!\$ \ \{s \in \{0, ..., 255\}^* : \lvert s \rvert > 32\}$
		$\boxed{\mathsf{chall} \leftarrow \mathtt{0xff} \parallel \mathsf{chall}}$
	$\xleftarrow{Y_{\mathrm{b64}} \parallel \mathsf{chall}_{\mathrm{b64}}}$	
$\boxed{\textbf{if } \mathsf{chall}[1] \neq \mathtt{0xff} \textbf{ then abort}}$		
$K_{\mathrm{reg}} \leftarrow \mathsf{DH}(a, Y)$		$K_{\mathrm{reg}} \leftarrow \mathsf{DH}(y, A)$
$c \leftarrow \mathsf{AEAD.Enc}(K_{\mathrm{reg}}, \mathsf{chall}; n)$	$\xrightarrow{c_{\mathrm{b64}}}$	$\mathsf{chall}' \leftarrow \mathsf{AEAD.Dec}(K_{\mathrm{reg}}, c; n)$
		$\textbf{if } \mathsf{chall}' \neq \mathsf{chall} \textbf{ then abort}$
	$\xleftarrow{\mathsf{ID}_{\mathcal{C}}}$	$\mathsf{ID}_{\mathcal{C}} \leftarrow \mathsf{GenID}()$

Figure 3.2: The Threema Registration protocol. $\boxed{\text{Boxed statements}}$ come from a patch present only on Android versions $\geq$ 4.62 and iOS versions $\geq$ 4.6.14. The nonce n is the fixed string "createIdentity response.". Not shown in the picture is the license-checking protocol, where the client sends its license to the server as proof of purchase.

generality, we assume here that Alice is the sender, Bob is the receiver and that Alice wants to send a message m.

3.3.1 Message Types and Serialization

In Threema, a message can be of many types. To distinguish between types during serialization, a one-byte value is assigned to each of them (the *type byte*). We now list a selection of the message types, along with their type byte (in parenthesis) and a brief description on how they are serialized.

- *Text message* (`0x01`): The message content is serialized as a byte string representing UTF-8 characters.
- *Image/Video/Audio/File message* (`0x02`, `0x13`, `0x14`, `0x17`, respectively): The media is encrypted using XSalsa20-Poly1305 under a random key K and a random nonce n. The encrypted media is sent to the media server, which returns a *blob ID*. To serialize the message, the app concatenates the blob ID, n and K.
- *Delivery Receipt message* (`0x80`): Given a list of message IDs of messages to be ACK-ed, the receiver concatenates the messages IDs and sets the resulting string as the serialized version.
- *Typing Indicator message* (`0x90`): The serialized message consists entirely of a single byte, representing a boolean. If the value is 1, the

user has started typing, while 0 means that the user has stopped typing.

Before encrypting, each message is serialized, the type byte is prepended and a random amount of PKCS7 padding (as described in [53], Section 10.3) is appended. The amount of padding is chosen randomly by sampling an integer in the $[1, 254]$ range from system randomness. We will refer to the concatenation of the type byte, the serialized message and the padding as the *plaintext* (`ptxt` for short).

3.3.2 Encryption and Dispatch

In the E2E protocol, the plaintext is encrypted using the long-term keys of the users. In our example, Alice will derive $K_{\mathcal{A},\mathcal{B}} \leftarrow \mathsf{DH}(a, B)$ as the encryption key. This key is then used to encrypt the plaintext using the XSalsa20-Poly1305 AEAD, under a randomly sampled nonce n. We call the resulting of this encryption the *ciphertext* (`ctxt` for short).

To create the packet that will be sent to the server, additional metadata has to be added. We list here all the values contained in the packet, putting in parenthesis the shorthand notation that we use for Figure 3.3:

- Threema ID of the sender (`src`).
- Threema ID of the receiver (`dst`).
- A random 8 byte message ID (`msg-id`).
- The little-endian representation of the time at which the message was sent (`timestamp`).
- A set of single-bit flags that encode various characteristicss of the message, such as whether a message should be ACK-ed or whether it is a VoIP message. This field is irrelevant for our analysis. (`flags`).
- A zero byte.
- The little-endian representation of the length of the metadata box (See below, `metadata-len`).
- The nickname of the sender (`src-nickname`).
- The metadata box itself (See below, `metadata`).
- The nonce n used to encrypt the message (`nonce`).
- The ciphertext (`ctxt`).

All the metadata listed above is inserted in the packet in plaintext, without any integrity mechanism to prevent tampering by an adversary. Threema prevents this, to a small extent, by including an optional value called the

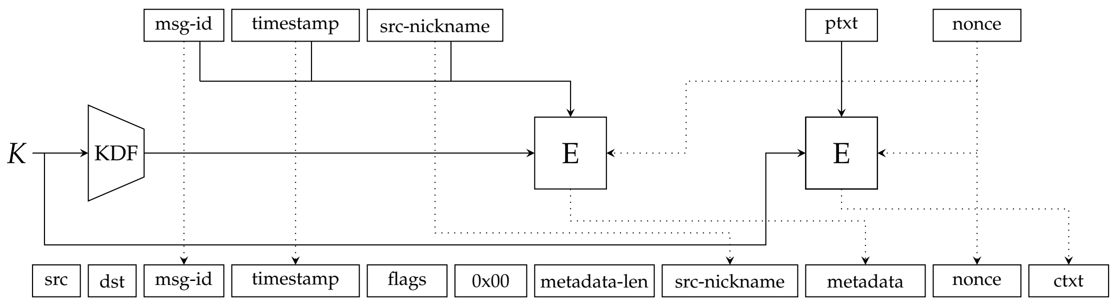


Figure 3.3: Structure of an E2E encrypted message. The input to the KDF is the key material. The salt and label are implicitly input into the KDF, taking values "mm" and "3ma-csp", respectively.

metadata box. This value is constructed by concatenating the message ID, the timestamp and the nickname of the sender and encrypting it using XSalsa20-Poly1305, under the same nonce n that was used for the plaintext and under the key $K^{\text{meta}}_{\mathcal{A},\mathcal{B}} \leftarrow \mathsf{BLAKE2b}(K_{\mathcal{A},\mathcal{B}}, \text{"mm"}, \text{"3ma-csp"})$. Whenever a message is decrypted, the data contained in the metadata box will take priority over the data outside of it. This harnesses the integrity protection provided by the AEAD scheme, which, at first, seems to prevent an attacker from tampering with the values of the message ID, timestamp and nickname. However, as the presence of the metadata box is optional, an attacker can set the length of the metadata box (`metadata-len`) to 0, and remove the metadata box itself, which in turn removes all integrity protection of the values cited earlier.

We remark that, since this protocol only uses long-term keys, it cannot provide forward secrecy at the end-to-end level. Threema attempts to mitigate this by providing forward secrecy at the client-to-server level, an unsatisfying solution given that the server always sees the end-to-end packets and can thus be considered a target by any adversary that wants to bypass forward secrecy.

We summarise the encryption process pictorially in Figure 3.3.

Retrieving Long-Term Keys

Retrieving the authentic long-term key of another user is a non-trivial task for a messaging application. This is due to the fact that users must be able to add new contacts even if the other person is offline, to avoid impacting the user experience, thus disallowing for interactive protocols. This leads to the necessity of having a trusted server that holds all long-term public keys, and can thus be queried when a user wants to contact another person for the first time. In Threema, this role is taken on by the directory server.

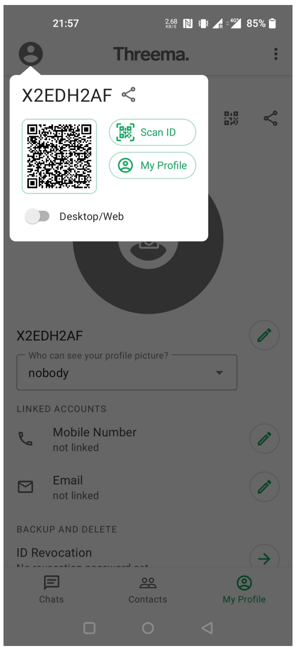
When a user (the *sender*) wants to send a message to another user (the *receiver*), they must know the Threema ID of the latter. The sender client will fetch the public key of the receiver by sending a GET request to the server over HTTPS, including the Threema ID of the receiver in the URL. The server will reply with a JSON message containing the base64 encoding of the receiver's public key if the user exist, or returning a 404 (Not Found) HTTP error if no corresponding user was found. The key of the receiver is then saved on the device storage, in order to be used in the future.

There is no guarantee, however, that the key returned from the server is the authentic key of the receiver. A compromised server might decide to return a different key, to which it knows the corresponding private key. If this is done, then the server would be able to decrypt the communications, effectively obtaining a Man-in-the-Middle position.

To combat this, Threema uses an out-of-band technique to check if the keys that the server has distributed are correct, and assigns an indicator of the trust level to each contact. This indicator is called the *verification level* by Threema. We list the possible values of the verification level:

1. *Level 1 (Red)*: The public key has been fetched from the Threema server because a message has been received from this contact for the first time or because the Threema ID of this contact was inserted manually. The public key of the contact cannot be fully trusted.
2. *Level 2 (Orange)*: The public key has been fetched by using the contact's email or phone number. Threema claims that this increases the confidence in the authenticity of the public key, since the user can be sure that if the server is honest, then they are talking to the owner of that email or phone number. However, this is still vulnerable to a malicious server providing non-authentic keys.
3. *Level 3 (Green)*: The public key has been verified out-of-band. Assuming no long-term key compromise has happened, the user can trust the public key.

The out-of-band verification mechanism is based on users scanning each other's QR codes, which are contained in the app. We show in Fig. 3.4 the QR code for a Threema user. Each QR code contains the Threema ID of the user, as well as their public key. When the QR code is scanned, the public key is checked against the key provided by the server by sending a HTTPS request. If they match, the contact is automatically given a verification level of 3. If there is a mismatch, the device will alert the user that there was an error and that the QR code scanned was incorrect. We note that the wording of the error does not mention the possibility that the key provided by the server is incorrect, which may lead users to ignore the error even if the server is acting maliciously. We also note that this method requires the client who scanned a QR code to also fetch the key from the server,



Who can see your profile picture?
nobody

Mobile Number
not linked

Email
not linked

Figure 3.4: A screenshot of the Threema app for Android, with the QR code necessary to verify the public key of the user

which forces the device to be connected to the internet when running the out-of-band verification.

Nonce Handling

As stated earlier in Section 2.2.1, repeating the value of a nonce for the encryption of two different plaintexts using XSalsa20-Poly1305 leads to a loss of security, with the attacker gaining the ability of learning the XOR of the plaintexts and of forging new valid ciphertexts for the same key-nonce pair. In order to safely generate nonces, Threema stores all nonces, both for outbound messages and for inbound messages. Whenever a new nonce n is generated, its value is checked against all previously stored values, in order to check its freshness. If the nonce has been seen before, a new nonce is generated in up to five total attempts, after which the application returns an error. This hedges against the possibility of the system randomness being faulty and avoids the nonce-reuse vulnerability.

More specifically, instead of storing nonces, the application stores a keyed hash of the nonces, using HMAC-SHA256 [56] and the Threema ID of the user as key. This provides unlinkability between the nonce databases of multiple users, since the same nonce will lead to different entries on different devices. On the other hand, this increases the storage cost: a nonce is 24 bytes while the result of HMAC is 32 bytes in size.

This mechanism also provides an important security property to the E2E protocol: by checking nonces of inbound messages, the client can reject all messages that contain an already-seen nonce. This automatically prevents replay and reflection attacks on the E2E protocol. We note that, however, this is not the most efficient way to defend against those attacks. For example, the chain of key derivations in the Signal protocol, among other things, ensures ordered delivery [87, Sec. 2.6] as well as replay protection without having to increase local storage and without having to run a database search for every message received. Another option that prevents reflection attacks is to derive two different keys from the shared secret by use of a KDF. Each key would be used for one direction of conversation, which effectively separates keys and is a technique that is widely employed in TLS [75, Sec. 7.1]. Clearly, in the Threema cryptographic design the security of the system against replay and reflection attacks entirely depends on the nonce database, since deleting it would result in replay and reflection attacks being viable again.

3.4 Client-to-Server Protocol

In order to disallow network adversaries from accessing the metadata in the E2E messages, as well as preventing them from tampering with communications, a secure channel must be established between the client and the server. In Threema, this channel is created by use of the Client-to-Server protocol (C2S Protocol), which can be visualized as the composition of two subprotocols:

- An authenticated key-exchange (AKE) protocol, which establishes an ephemeral session key between the client and the server, as well as authenticating each party to the other. We call this the *handshake protocol*.
- A *transport protocol*, which uses the session key established earlier to encrypt and decrypt messages between the client and the server.

We analyze each subprotocol separately in the following sections.

3.4.1 The Handshake Protocol

The handshake protocol is executed after the user has registered, which means that, at the time of the protocol execution, the server $\mathcal{S}$ has a database which contains the long-term key of the user that has initiated the protocol. We abstract this by defining a table $T_{\text{users}}[\cdot]$ that maps Threema IDs to long-term public keys. We also assume that every user knows the long-term key of the server, since it is embedded within the application.

We remarked in Section 3.3 that the E2E protocol does not provide forward secrecy: the C2S protocol is Threema's attempt at providing it at the client-to-server level, disincentivizing a network attacker from storing messages encrypted with long-term values. To do so, the handshake protocol must establish a session key involving ephemeral keys, using the long-term keys to authenticate the parties to each other.

We now describe the handshake protocol in detail. In our description, we will refer to Fig. 3.5 for the message numbering. Assume user $\mathcal{A}$, with long-term keypair (a, A), wants to connect to the server $\mathcal{S}$, with long-term keypair (s, S). The client first generates an ephemeral keypair $(x, X = g^x)$ and a 16 byte value called the *client cookie*, and sends both values to the server (Message 1). The purpose of the client cookie is to both provide freshness to the exchange and to be later used to initialize the nonce counters for the transport protocol. After receiving the client's ephemeral public key and the client cookie, the server generates its own ephemeral keypair $(y, Y = g^y)$ and a 16 byte value called the *server cookie*. Then, it computes the first shared symmetric key of the exchange $K_1 = \mathsf{DH}(s, X)$, mixing the long-term key of the server with the ephemeral key of the client, and uses it to encrypt its ephemeral public key Y concatenated with the client cookie, using a random nonce n. The server then sends the resulting ciphertext, along with their own cookie (Message 2). When the client receives the message, it recomputes K_1 and checks if the ciphertext decrypts correctly. If so, the server also checks if the client cookie in the plaintext corresponds to the one that the client previously generated, aborting the protocol if there is a mismatch or if decryption failed. At this point, both the client and the server derive two additional keys: K, which is formed from the combination of the two ephemeral keys and is used in the transport protocol as the session key, and K_2, which is the combination of the two long-term keys. Both also initialize two counters: a counter for messages from the client to the server $\mathsf{ctr}_{\mathcal{A}}$ (the *client counter*) and one for messages from the server to the client $\mathsf{ctr}_{\mathcal{S}}$ (the *server counter*). The long-term symmetric key K_2 is used by the client to create the so called *vouch box* $\mathsf{vouch} \leftarrow \mathsf{AEAD.Enc}(K_2, X; n')$ using a random nonce n'. This vouch box acts as the authentication mechanism between the client and the server since it relies on knowledge of the client long-term private key and binds the client ephemeral key to the client's identity. The vouch box is concatenated with $\mathsf{ID}_{\mathcal{A}}$ and the server cookie $C_{\mathcal{S}}$, and then encrypted using the session key using the first nonce from the client counter $\mathsf{ctr}_{\mathcal{A}}$ (Message 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. If the check succeeds, the server sends a final confirmation message composed of 16 zero bytes, encrypted with the session key and the first nonce from the server counter $\mathsf{ctr}_{\mathcal{S}}$ (Message 4).

When the handshake is finished, the protocol yields the session key K, which will be used by the transport protocol.

3.4.2 The Transport Protocol

In the transport phase, the client and the server exchange messages, encrypted using the session key K established earlier and using nonces from the client counter ctr_C (for messages sent from the client to the server) and the server counter ctr_S (for messages sent from the server to the client).

If a decryption fails, the entire connection is dropped and a new handshake is started. This prevents, for example, message reordering and message removal by a network adversary, since either attack would require the client or the server decrypting with a nonce different from the expected one.

Another instance in which the connection is dropped, this time by the server, is when multiple devices try to connect to the server with the same Threema ID. In this case, the older connection is dropped by the server, alerting the dropped client that another device has connected. In fact, Threema expects every Threema ID to login only from one device at a time and does not provide multi-device support, except for its own Threema Web solution (Section 3.8).

The behaviour of a client when the connection is dropped is to attempt the connection again. Whenever the connection is dropped due to another device connecting, up to 5 attempts are done for reconnecting, after which the application 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 5 attempts will alert the user that another device is trying to connect to the server with the same Threema ID.

After a connection is established, the connection is kept alive until either party decides to close the connection. In order to maintain the connection and to check whether the connection has been dropped at the other end, the client sends an echo request to the server every 180 seconds. If the server does not reply within 20 seconds, the connection is deemed dead and a new connection is attempted.

While the connection is alive, the server will send messages that are meant for the user encrypted under the session key. Whenever the client receives a message, it replies with an ACK in response to that specific message. If the client does not send an ACK, the server will try to send the message again when the client reconnects to the server. If the ACK has been received by the server, the message will not be sent again by the server and we expect the server to delete the message.

Client $\mathcal{A}$		**Server** $\mathcal{S}$
$(\mathsf{sk}, \mathsf{pk}) = (a, A = g^a)$		$(\mathsf{sk}, \mathsf{pk}) = (s, S = g^s)$
$(x, X) \leftarrow\$ \mathsf{KGen}()$		
$C_{\mathcal{A}} \leftarrow\$ \{0,1\}^{128}$		
	$\xrightarrow[(1)]{X,\ C_{\mathcal{A}}}$	
		$(y, Y) \leftarrow\$ \mathsf{KGen}()$
		$C_{\mathcal{S}} \leftarrow\$ \{0,1\}^{128}$
$K_1 \leftarrow \mathsf{DH}(x, S)$		$K_1 \leftarrow \mathsf{DH}(s, X)$
		$n \leftarrow\$ \{0,1\}^{192}$
		$\mathsf{ctxt}_1 \leftarrow \mathsf{AEAD.Enc}(K_1, Y \parallel C_{\mathcal{A}}; n)$
	$\xleftarrow[(2)]{C_{\mathcal{S}}, n, \mathsf{ctxt}_1}$	
$Y \parallel C'_{\mathcal{A}} \leftarrow \mathsf{AEAD.Dec}(K_1, \mathsf{ctxt}_1; n)$		
if $C'_{\mathcal{A}} \neq C_{\mathcal{A}}$ **then abort**		
$K_2 \leftarrow \mathsf{DH}(a, S)$		
$K \leftarrow \mathsf{DH}(x, Y)$		$K \leftarrow \mathsf{DH}(y, X)$
$\mathsf{ctr}_{\mathcal{A}} \leftarrow \mathsf{Ctr.Init}(C_{\mathcal{A}})$		$\mathsf{ctr}_{\mathcal{S}} \leftarrow \mathsf{Ctr.Init}(C_{\mathcal{S}})$
$n' \leftarrow\$ \{0,1\}^{192}$		
$\mathsf{vouch} \leftarrow \mathsf{AEAD.Enc}(K_2, X; n')$		
$n_{1,\mathcal{A}} \leftarrow \mathsf{ctr}_{\mathcal{A}}.\mathsf{Next}()$		
$\mathsf{ctxt}_2 \leftarrow \mathsf{AEAD.Enc}(K, \mathrm{ID}_{\mathcal{A}} \parallel n' \parallel \mathsf{vouch} \parallel C_{\mathcal{S}}; n_{1,\mathcal{A}})$		
	$\xrightarrow[(3)]{\mathsf{ctxt}_2}$	
		$\mathrm{ID}_{\mathcal{A}} \parallel n' \parallel \mathsf{vouch} \parallel C'_{\mathcal{S}} \leftarrow \mathsf{AEAD.Dec}(K, \mathsf{ctxt}_2; n_{1,\mathcal{A}})$
		if $C_{\mathcal{S}} \neq C'_{\mathcal{S}}$ **then abort**
		$A \leftarrow T_{\text{users}}[\mathrm{ID}_{\mathcal{A}}]$
		$K_2 \leftarrow \mathsf{DH}(s, A)$
		$X' \leftarrow \mathsf{AEAD.Dec}(K_2, \mathsf{vouch}; n')$
		if $X' \neq X$ **then abort**
		$n_{1,\mathcal{S}} \leftarrow \mathsf{ctr}_{\mathcal{S}}.\mathsf{Next}()$
	$\xleftarrow[(4)]{\mathsf{AEAD.Enc}(K, 0^{128}; n_{1,\mathcal{S}})}$	

Figure 3.5: The Threema Handshake Protocol. Counter-derived nonces are implicitly re-derived on the receiving end and are not transmitted.

3.5 The Group Messaging System

Threema provides users with the possibility of creating groups, similarly to other messaging applications: when a message is sent to the group, it is received by every group member. Threema's implementation of group messaging is based on client-side fan-out. Assume a group of k users $\mathcal{G} = \{\mathcal{U}_1, ..., \mathcal{U}_k\}$: whenever a user wants to send a message m to the other users, they encrypt the message against the public key of every other user, using the E2E protocol.

We give a more technical explanation of the group messaging protocol. Groups are handled exclusively on the clients, and the server is oblivious to their organization. Without loss of generality, assume that a group is created by user $\mathcal{U}_1$ and that the group includes all other users in $\mathcal{G}$: $\mathcal{U}_1$ randomly samples a group ID and creates a message which includes the group ID, the identity of the creator, the name of the group, the list of all the members, and the creation date. This message is then sent to all other users in $\mathcal{G}$ in order to inform them of the group creation. Afterwards, all members will have the same initial view of the group membership due to the information contained in the message. Whenever there is the need for an update, such as when removing or adding a member, an updating message is sent by $\mathcal{U}_1$ to all other members, who will, in turn, update their view of the group.

When sending a message, the process is similar to the standard E2E protocol, except that it is encrypted against the public key of each user currently in the group and that the plaintext also includes information about the group to which the message belongs. More specifically, for each one-on-one message type, there exists a group messaging counterpart, which also includes the creator of the group and the group ID. Assume that $\mathcal{U}_i$ wants to send a message m to the group: for each user $u \in \{\mathcal{U}_j | j \neq i\}$, the message m is encrypted using the key $K_{\mathcal{U}_i,u} \leftarrow \mathsf{DH}(\mathsf{sk}_{\mathcal{U}_i}, \mathsf{pk}_u)$ and a random nonce, sampled independently for each recipient. All the messages are then sent to the Threema server for dispatching.

While the update mechanism is enough to ensure that all users have the same view of the group membership, there is no mechanism that ensures that all users see the same messages in the same order, a property known as *transcript consistency*. In fact, a user may modify their client to send the message only to a subset of users in the group, without the other members being able to notice. This is in contrast with Signal and Whatsapp, where the protocol enforces transcript consistency among all users in a group.

3.6 Contact Discovery Protocol

Even though Threema can be used as an anonymous messaging app, users have the possibility of linking their Threema account with an email or a telephone number, which allows the user to be found via that contact information.

Whenever a user wants to find which of their contacts is using Threema, they can run what we call the *contact discovery protocol*. The protocol consists of a simple set intersection, in which the user collects all the emails and phone numbers in their phone book, hashes them and sends the hashes to the Threema server.[3] The server will then check which of those hashes are also present in the Threema database, meaning that the contact information corresponding to the hash belongs to a user of Threema. The server then collects all the hashes found in the database, including the corresponding identity and public key, and sends the information back to the client.

This is essentially the same system used by Signal [86] and Whatsapp, the latter sending information in plaintext rather than hashing it [48]. In [48], Hagen et al. raise privacy concerns in these types of system, since the server will have a way of knowing the contacts of each person, including those that may not even be aware of the app itself. Hashing phone numbers is also insufficient, since they carry little entropy and can thus be bruteforced. The impact for the privacy of the end users is that a server acting maliciously could reconstruct their social graph, de-anonymizing them and their contacts [20]. Signal tries to minimize the impact of this leakage by running the contact matching protocol inside an Intel SGX enclave, which minimizes the possibility of the server learning the results of the computation [86] and by using privacy-preserving protocols [85]. This decreases the trust assumptions needed with respect to the server and increases confidence in the service provided by Signal.

In order to prevent abuse of the system, the user must authenticate to the server in order to obtain a string called a "match token". This token can be later used during the matching protocol to show that the user has previously authenticated. The authentication protocol is the same that is used for the registration protocol (Section 3.2), except that the nonce is now randomized rather than being fixed.

[3]More precisely: the hash is implemented by using HMAC with one of two possible keys, one of which is used for emails and one for phone numbers. We are not aware of any reason behind this implementation.

3.7 Backup Methods

In Threema there are three methods by which a user can backup their data and, most importantly, their account, so that it can be later restored on another device, or on the same device after having reset it.

1. *Threema Safe*: the cloud-based solution by Threema that enables users to upload a backup to the Threema server, containing their private key and information about their contacts. The user can later retrieve and restore the backup with only the knowledge of a user-chosen passphrase and of their Threema ID. This method does not include chat history in the backup.
2. *Data Backup*: a method that allows the user to export all of their chat history, their contacts, as well as the information necessary to recover the account. This method creates an encrypted zip file that the user can choose to store in another location, such as a computer or a cloud storage service.
3. *ID Export*: this method exports only the private key of the user, allowing the account to be easily restored on another device. This method creates a string of text that can be imported into a new Threema installation and only requires the user to remember a passphrase of their choice.

3.7.1 Threema Safe

When first creating an account, the user is prompted to activate Threema Safe, in order to regularly backup their account information to the Threema server. If a user wishes to do so, they may decide to initially skip the backup process and opt-in at a later time by visiting the Backups section of the application. To begin the protocol, the application prompts the user for a passphrase, which is then checked against a local file of known weak passphrases. If the passphrase is found to be weak, the user is asked to confirm whether they wish to continue with that passphrase or if they want to change it.[4] After receiving a passphrase, the application builds the first backup and sends it to the server.

To build a backup, the application collects the following information to build a JSON string:

- General information about the device, such as the language and the version of Android that is being run.

[4]Note that the file of weak passphrases is fairly short, given that it is stored locally in the application. This implies that many weak passwords found in common weak password lists will not be found in the one provided by Threema.

- The private key of the user, encoded in base64.
- The current nickname of the user.
- Their phone number and email, if the user decided to link them to their Threema account.
- A list of their contacts, containing for each contact:
  - Their Threema ID.
  - The time at which they were added as a contact.
  - Their verification level.
  - Their nickname.
  - Other flags necessary for features of the messaging app: whether the contact is verified in Threema Work, whether they send typing indicators and read receipts, whether the contact has been hidden in the application by the user and whether the contact is private.
  - Their public key, only if the contact has been verified out-of-band (Level 3 for the verification level).
- A list of groups to which the user belongs.
- A list of distribution lists[5] to which the user belongs.
- A map of the settings chosen by the user for the functioning of the app.

The app then uses scrypt with the chosen passphrase as input and the identity of the user as a salt to generate two 32 byte values. The first value is the *Backup ID* and will be used to retrieve the backup without revealing the identity of the user to the server, while the second value will be used as the encryption key K_{bkp} for the backup. The JSON is compressed with gzip and encrypted using the XSalsa20-Poly1305 AEAD using the key K_{bkp} and a random nonce. The nonce and the AEAD ciphertext are then uploaded to the Threema server, accompanied by the Backup ID.

To restore a backup, the user is prompted for their Threema ID and the passphrase that was previously chosen. The application then re-derives the Backup ID and the encryption key, using the former to recover the backup and the latter to decrypt it. If the wrong passphrase was provided, then with high probability the corresponding backup ID will be incorrect and will not match any backup in the system.

[5] A distribution list is the method that Threema uses to create a broadcast channel, where one user writes to many users at the same time. This differs from a group as it is unidirectional and receivers do not know who else is in the distribution list.

We describe the backup process in pseudocode in Algorithms 2 and 3. The backup creation algorithm assumes that the JSON described above has already been created by the application and is given to the procedure as a string bkp.

Algorithm 2 The backup procedure for Threema Safe

Input: A password (P), the Threema ID of the user (ID) and the JSON backup (bkp)
Output: None

```
1: procedure ThreemaSafeBackupCreate(P, ID, bkp)
2:     $\text{bkpID} \parallel K_{\text{bkp}} \leftarrow \mathsf{scrypt}(\mathrm{P}, \mathrm{ID}, 64)$    // $|bkpID| = |K_{bkp}| = 32$
3:     $\text{bkp}_{\text{gzip}} \leftarrow \mathsf{gzip}(\text{bkp})$
4:     $n \leftarrow\$ \mathcal{N}$
5:     $\text{bkp}_{\text{enc}} \leftarrow \mathsf{AEAD.Enc}(K_{\text{bkp}}, \text{bkp}_{\text{gzip}}; n)$
6:     $\mathsf{SendBackup}(\text{bkpID}, n \parallel \text{bkp}_{\text{enc}})$    // Send the backup to the Threema server
```

Algorithm 3 The backup restoring procedure for Threema Safe

Input: A password (P) and the Threema ID of the user (ID)
Output: None

```
1: procedure ThreemaSafeBackupRestore(P, ID)
2:     $\text{bkpID} \parallel K_{\text{bkp}} \leftarrow \mathsf{scrypt}(\mathrm{P}, \mathrm{ID}, 64)$    // $|bkpID| = |K_{bkp}| = 32$
3:     $\text{res} \leftarrow \mathsf{RetrieveBackup}(\text{bkpID})$    // Retrieve backup from the server
4:     if res $= \perp$ then abort
5:     $n \parallel \text{bkp}_{\text{enc}} \leftarrow \text{res}$
6:     $\text{bkp}_{\text{gzip}} \leftarrow \mathsf{AEAD.Dec}(K_{\text{bkp}}, \text{bkp}_{\text{enc}}; n)$
7:     $\text{bkp} \leftarrow \mathsf{gunzip}(\text{bkp}_{\text{gzip}})$
8:     $\mathsf{RestoreFromJSON}(\text{bkp})$    // Restore backup to local device
```

We make a few observations about the Threema Safe backup protocol. First, the backup data is uploaded to the Threema server via HTTPS, not enabling a network adversary to recover the backup itself. Second, the backup itself has no explicit connection to a given Threema ID, as it is identified by the pseudorandom backup ID. However, the Threema server would often still be able to link a backup to a Threema ID by observing the connections to the messaging server (using the C2S protocol) and checking if the same IP address has sent a backup. In that case, the server may also attempt an offline brute-forcing attack, though the attack would still be hindered by the memory requirements of scrypt. Third, although an attacker may not be able to recover the backup itself, the length of the backup will still be visible through the TLS connection. Fourth, the client has a retry behaviour whenever a backup fails. Usually, when Threema Safe is activated, a first backup is started and backups are then regularly scheduled to be executed every 24 hours. If, however, the backup fails for some reason, then the client will attempt another backup as soon as the application is restarted. This happens even if the application is behind a PIN or biometric login (but does not happen if the entire application is protected with a passphrase, since the passphrase protection locally encrypts the private key and must then be unlocked to trigger a backup, see Section 3.9). These last two observations are useful for our attack of Section 4.3.2, since they theoretically allow for a weaker adversary to break the security of Threema.

3.7.2 ID Export

The most lightweight method to backup data in Threema is the ID export, which allows the user to backup their Threema ID and their long-term private key by using a passphrase. The result is a string that the user can save or print, optionally in QR form, and restore at a later moment with only the knowledge of the passphrase.

The application first prompts the user for a passphrase, which is then used to derive a symmetric key by use of the PBKDF2 algorithm, using a random 8-byte salt. The symmetric key is then used to encrypt the concatenation of the Threema ID and the long-term private key by using XSalsa20 with a zero nonce. The resulting byte string is base32-encoded and split into sets of 4 characters for better readability.

We show the pseudocode for the ID export process in Algorithms 4 and 5.

Because the choice of passphrase is free, an adversary that has control of the device for a few minutes can create an export to which they know the passphrase. This would allow them to clone the entire account, since they would have control of the long-term private key. We discuss this further in Section 4.3.1 and provide mitigations for this issue in Section 5.5.

3.7.3 Data Backup

As neither the Threema Safe method nor the ID export saves the chat history, Threema provides an additional backup method that allows a user to export the most of their local application storage. This includes all the information backed up with Threema Safe as well as chat history and the public keys of all contacts (recall that Threema Safe only stores public keys of verified contacts). The result of the backup is a single encrypted zip file.

Algorithm 4 The backup procedure for an ID Export

Input: A password (P), the Threema ID of the user (ID), the long-term private key of the user sk

Output: The backup string

```
1:  procedure IDExportCreate(P, ID, sk)
2:      ptxt ← ID || sk
3:      $\sigma \leftarrow\$ \{0,1\}^{64}$
4:      $K_{bkp} \leftarrow \mathsf{PBKDF2}(P, \sigma, 32)$
5:      $\mathsf{ctxt} \leftarrow \sigma \,\|\, \mathsf{XSalsa20.Enc}(K_{bkp}, \mathsf{ptxt}; 0^{192})$
6:      $\mathsf{out} \leftarrow \varepsilon$                    // Initialize out to the empty string
7:      for $i \leftarrow 1$ to $|\mathsf{ctxt}_{b32}|$ do
8:          $\mathsf{out} \leftarrow \mathsf{out} \,\|\, \mathsf{ctxt}_{b32}[i]$
9:          if $i \mod 4 = 0$ then                 // Every four characters, add a hyphen
10:             $\mathsf{out} \leftarrow \mathsf{out} \,\|\, \text{"}-\text{"}$
11:     return out
```

Algorithm 5 The procedure to restore an ID Export

Input: A password (P), the exported ID backup (bkp)

Output: The Threema ID (ID) and the the long-term secret key sk

```
1:  procedure IDExportRestore(P, bkp)
2:      $\mathsf{bkpdec} \leftarrow \varepsilon$                 // Initialize bkpdec to the empty string
3:      for $i \leftarrow 1$ to $|\mathsf{bkp}|$ do
4:          if $i \mod 5 \neq 0$ then                           // Skip the hyphens
5:              $\mathsf{bkpdec} \leftarrow \mathsf{bkpdec} \,\|\, \mathsf{bkp}[i]$
6:      $\mathsf{bkpdec} \leftarrow \mathsf{b32.Decode}(\mathsf{bkpdec})$
7:      $\sigma \,\|\, \mathsf{ctxt} \leftarrow \mathsf{bkpdec}$                    // $|\sigma| = 8$
8:      $K_{bkp} \leftarrow \mathsf{PBKDF2}(P, \sigma, 32)$
9:      $\mathsf{ptxt} \leftarrow \mathsf{XSalsa20.Dec}(K_{bkp}, \mathsf{ctxt}; 0^{192})$
10:     $\mathsf{ID} \,\|\, \mathsf{sk} \leftarrow \mathsf{ptxt}$                       // $|ID| = 8$
11:     return $(\mathsf{ID}, \mathsf{sk})$
```

To build the zip file, Threema follows the AE-2 WinZip standard [23], using AES-256 as the base block cipher. We refer to Section 2.2.5 for an in-depth explanation of the AE-2 standard.

The backup includes the following files:

- A *settings file*, containing basic application settings.
- A *identity file*, for the private key of the user, exported in the same format as the backup method of Section 3.7.2 and using the same password as the zip file.
- A *contacts file*, which is a CSV containing all information about the contacts, including their long-term public key.
- For each contact, a CSV file for the *conversation chat history* with that contact. Each file is named `message_<Threema ID>.csv`, where `<Threema ID>` is replaced by the actual ID of the contact.
- A *groups file*, which is a CSV containing all information about the groups to which the user belongs.
- For each group, a CSV file for the *group chat history* in that group. Each file is named `group_message_<Group ID>.csv`, where `<Group ID>` is replaced by the actual ID of the group.
- A backup of all the distribution lists.
- A backup of all ballots and votes in ballots.

We note that, due to the way the zip format is built, all the file names are accessible without the necessity of a password. This directly implies that an adversary that has access to the encrypted zip file can see who the user has as their contacts, due to the presence of the conversation chat history files. Furthermore, this backup method suffers from the same deficiency as the data backup: an adversary that has access to the device for a few minutes can simply export all the user data and thus clone the account. The same mitigations described above apply to this backup method.

3.8 Threema Web

As discussed earlier, Threema only allows one device to be connected to the messaging server at any given time, for each Threema ID. To improve user experience, however, Threema has provided the possibility to add other devices that would be linked to a single "main" device, which would be, in turn, connected to the messaging server. This solution is called *Threema Web*, which is accessible via a web browser or via a desktop application. For the rest of the discussion we will be focusing on the web browser setting, although everything will apply similarly to the desktop application.

Threema Web relies on the phone being connected to the messaging server and relaying messages to the browser or the desktop device, allowing a user to receive and send messages from any device. The relaying is done in a peer-to-peer fashion, protected by a layer of DTLS [76] and another custom protocol provided by the SaltyRTC library [81]. According to the SaltyRTC documentation, the reason for using two encryption protocols is that SaltyRTC "[...] is able to protect the clients' signalling data even in case the underlying TLS encryption [...] has been completely broken" [81].

We now describe, at a high-level, the architecture of Threema Web. The three types of entities found are the main device, the signalling server and the secondary devices. To connect devices on a peer-to-peer (P2P) level, it is necessary to bypass common network functionalities such as NAT, which normally prevents two devices from connecting directly to each other. In order to solve this problem, Threema uses a signalling server to which the single devices connect, exchanging handshake messages through the server. This handshake establishes a session key that is used to create a secure P2P channel between the web client and the main device and through which the messaging of the Threema application can be relayed. We refer to the Threema cryptography whitepaper for a more detailed explanation of the protocol [38].

We hope that future work will analyze Threema Web more in depth, possibly analyzing the interactions between Threema Web and the other Threema protocols. In particular, SaltyRTC uses a design which is very similar to the one used by Threema, which opens up the possibility of unwanted cross-protocol interactions between the application and the library.

3.9 Application Access Control and Local Encryption

Messaging applications must take into consideration the threat posed by an adversary that has control of the device. While the access control provided by the device may seem enough, this would not include common scenarios where the user forgets their phone in an unlocked state or where they are handing their unlocked device to another person. For these scenarios, it is convenient to include an additional access control mechanism, such as a PIN, passphrase or a biometric login.

Threema provides two different types of protections. The first one is a simple UI lock, which does not cryptographically protect the device and is meant for preventing an "occasional" adversary from accessing the application data. This UI lock uses either a PIN, a passphrase or a biometric login to protect the application by showing a login screen before the main view of the application. In Section 4.3.2 we show that this protection is insufficient against an adversary that has control of the device. The second is a stronger cryptographic protection that involves the use of a passphrase, with the objective of encrypting the entire local storage of the application. During the initial setup of the application, a 32 byte AES key is randomly sampled by the application, which will be used to encrypt the local storage of the application. We will call this key the *master key*. In order to protect the master key, a user-chosen passphrase is used to derive a 32 byte secret by use of scrypt (or PBKDF2, in previous versions of the application) with a random salt. The resulting secret is XOR-ed with the master key in order to encrypt it. The resulting encrypted key is stored in the application local storage. Naturally, without the protection given by the passphrase, an attacker with the ability to read the device storage could also read the unprotected master key, making any encryption essentially useless.

To encrypt the application data, Threema uses AES-CBC which, as discussed in Section 2.2.3, does not offer any integrity. This is done by either directly encrypting files with the standard Java cryptography library or by encrypting the SQLite database [92] using the SQLCipher library [64]. To protect the data against tampering, Threema entirely relies on the access control provided by the operating system, which ensures that all data stored by Threema is made inaccessible to other applications.

Chapter 4

Attacks

In the previous chapter, we analyzed the choices that the Threema developers have made in creating their protocol. In this chapter, we show that some of these choices are suboptimal and, in some cases, lead to both theoretical and practical attacks. We consider three models:

- An external threat actor that has access to the network and can interact with the victim. The attacker will often have some additional power that we will state while discussing the attacks. We call this the *external threat actor* model.
- A malicious actor that compromises the Threema server, gaining access to the contents of the server as well as being able to act as the server in protocol runs. We call this model the *compromised Threema* model.
- An agency that takes control of the unlocked device for a short amount of time. This might be the case with border searches or when protesters are arrested by police forces and searched for incriminating evidence. Assuming that the device cannot be withheld for a long time and cannot be invasively tampered with, the application should still provide a certain amount of security. We call this model the *compelled access* model.

We stress that any reasonable security analysis must consider the possibility that the messaging server is acting maliciously. There are many incentives for national security agencies, even ones outside Switzerland, to exert control over the users of Threema. As an example of a similar case, in 2020 it was discovered that the CIA secretly owned the Swiss company Crypto AG, which sold backdoored machines to other national governments [71].

While not all the attacks that we present are practical, theoretical attacks are also relevant when discussing the security of a real-world software. First, what might be an attack of theoretical interest at first, has the possibility to escalate into a greater security danger after a few improvements. Second, threat models differ between users: our compelled access model makes strong assumptions about the situations in which an attacker is able to achieve a break but, given the possible implications of an attack, this might still not be acceptable for certain individuals, such as whistleblowers and investigative journalists.

We now describe in detail the attacks that we found on the Threema application, grouped by the threat model considered.

4.1 External Attacker Model

4.1.1 Attack 1 (Impersonation by Randomness Failure)

In this attack we show that a randomness failure on the client in the C2S protocol can lead to a *permanent impersonation* attack. More specifically, if the client generates a weak ephemeral key which an attacker can discover, then the attacker can use information from the compromised session to create new sessions with the server where they impersonate the victim. This allows the attacker to discover the metadata attached to the messages intended for the victim: namely the ID and the nickname of the sender, and the timestamp of the message. In addition to this, the attacker can selectively suppress messages intended for the victim, preventing them from ever being received. Furthermore, the attacker can impersonate the server to the user as long as the user keeps using the same ephemeral key, which for the Threema Android application is up to one week. In our analysis, we refer to Fig. 3.5 for the numbering of the messages in the C2S protocol.

Attack Assumptions and Adversarial Capabilities

We assume that the attacker has the power to reveal the ephemeral secret key x of the client, an adversarial ability which is commonly found in formal security models for authenticated key-exchange protocols [18, 60]. We further assume that the attacker has the transcript of a complete handshake. This can be trivially done by any network adversary that can see and subsequently store all the messages of the C2S protocol run between the victim and the server.

Attack Description and Impact

Since the attacker has both the transcript and the ephemeral private key of the client, they have all the material necessary to derive K_1 and decrypt message (3) of the C2S handshake, from which they can recover the value of vouch. Because the attacker knows the ephemeral key, they can execute a new instance of the C2S protocol where the same ephemeral key is used, derive the new session key, and the replay the vouch box, encrypted under the new session key. Since the vouch box is valid for the ephemeral key used and contains no way for the server to check its freshness, it will be accepted by the server, making the handshake complete successfully.

This allows the attacker to impersonate the user while communicating with the server, allowing the adversary to obtain access to all the end-to-end encrypted messages meant for the victim. This, in turn, reveals all metadata in all communications directed towards the victim, allowing the attacker to learn with whom they have been communicating, as well as all the message timestamps. Most importantly, this attack sidesteps the forward-secrecy property that the C2S protocol provides with respect to an external attacker. Indeed, after the attack, the adversary sees E2E-encrypted messages, which only use long-term keys for the encryption, as described in Section 3.3. Thus, while this attack in itself does not provide the attacker with the ability of reading the inner plaintext, it provides the possibility of storing E2E encrypted messages that could be decrypted in the future if the adversary manages to reveal the long-term keys of the user.

In addition to this, the attacker gains the ablity to selectively drop messages from the conversation by deciding which messages to ACK to the server. As discussed in Section 3.4, if a message in the C2S protocol 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 will know this is the case, since they will receive the error message from the server telling them that another device has connected. At this point, the attacker would relinquish the connection to the victim, in order to avoid detection.

The main weakness that is highlighted by this attack 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 on a value decided by the server. For instance, the server cookie could be included within the vouch box, rather than being outside it. This attack also shows that, due to the way the protocol is constructed, an ephemeral key becomes as important to protect as a long-term key, which is a very non-standard assumption for any key-exchange protocol. Most notably, a similar vulnerability was present in the Off-The-Record (OTR) protocol and was discussed by Raimondo et al. in [74]. The authors highlight the consequent lack of *session independence* property in OTR's protocol: "the exposure of ephemeral session-specific secrets should have no bearing on the security of other sessions", a property that

Threema also lacks due to this weakness.

Client-Side Ephemeral Key Misuse An additional problem with the C2S protocol lies in how the ephemeral keys are handled client-side: if the application is never restarted, then the same ephemeral key is used for up to seven days before generating a new one. Leaking an ephemeral key allows an attacker to impersonate the server to the user by forging message (2) using the server's public key and the compromised ephemeral key x. In turn, this implies that the attacker has the power to also see the outbound E2E-encrypted messages of the victim that are meant for the actual Threema server. This reveals additional metadata, allows the adversary to drop arbitrary messages, and allows other attacks in the "compromised Threema" model such as the ones described in Sections 4.2.1 and 4.2.2. This is possible since, unlike other AKE protocols like in the Noise Protocol Framework [73], the session key does not depend on the long-term keys but uniquely on the ephemeral keys. In Threema, 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 amplified.

Server-Side Ephemeral Key Misuse Finally, we highlight a further problem with a similar reuse of ephemeral key on the server side. Whenever the same ephemeral public key is used multiple times by the user within a short span of time, the server will use the same ephemeral key as well. By running experiments, we noticed that we could prolong the lifetime of the server ephemeral by regularly running the C2S protocol with the same client ephemeral key. In our observations, we prolonged the lifetime of the server ephemeral key for multiple months at the time of writing, and we conjecture that an attacker could possibly prolong it indefinitely. We hypothesize that the server caches ephemeral keys in a structure similar to a key-value store, where the client's ephemeral key is mapped to a server ephemeral key and where a Least Recently Used policy is applied. Whenever a client presents an ephemeral public key that is found in the cache, the server will use the corresponding server ephemeral key. On the other hand, 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 and, if the same client ephemeral key is presented again, the server will generate a new ephemeral keypair. Because of this design, an attacker is able to force the same session key to be used for an indefinite amount of time by regularly connecting to the server with the same key, thus refreshing the value in the cache. This has various implications. First, if an adversary manages to reveal a session key, rather than an ephemeral key, they are still able to impersonate the client to the server indefinitely, since the attacker can force the server to never choose a different Diffie-Hellman share and the compromised session key will continue to be valid. Second, assuming that the hypothesis of an ephemeral key cache holds, compromising said cache leads to impersonation of multiple users for an indefinite amount of time, as long as the attacker has recorded the handshakes involving the revealed ephemeral keys. This would make the hypothesized cache a weak point in the cryptographic design of Threema and requires careful protection an security measures from the server.

4.1.2 Attack 2 (Vouch Box Forgery)

An alternative route to the previous attack is to forge a completely new vouch box for a public key C to which the attacker knows the corresponding private key c. We show that this is theoretically feasible due to a cross-protocol interaction between the E2E protocol and the C2S protocol. In this attack, the adversary acts as a Threema user and has to first convince the victim that the server's public key is their own long-term public key and then has to convince the user to send the attacker a specially crafted message. We leave open the possibility that the first condition can be satisfied in different ways. In our particular instantiation of the attack, we leverage a vulnerability in a dependency used by the Android version of Threema. Similarly to the previous attack, this method allows an adversary to impersonate a user to the server for an indefinite amount of time.

Attack Assumptions and Adversarial Capabilities

Assume that an adversary can claim the server's public key for the C2S protocol (S, in Fig. 3.5) as their own long-term public key to the victim. This means that whenever the user sends a message to the attacker, they will use their own private key a and the server's public key S, which the victim believes to belong to the adversary.

Assume that the attacker manages to find a private key x such that the corresponding public key X has the following form: a `0x01` byte, followed by a string σ composed of 30 printable ASCII characters, followed by another `0x01` byte. The latter assumption is reasonable if we consider the distribution of public keys to be close to uniformly random, which approximately holds since the public keys are x-coordinates of random points on Curve25519. We discuss this point in Section 4.1.2 in more detail.

In order for an attacker to claim the server's public key as their own, we assume that the attacker has access to the user's data backup (Section 3.7.3) and knows the list of the victim's contacts. We note that these assumptions arise from our particular instantiation of the attack. In general, there might be other ways for the attacker to claim the server's public key as their own. For example, the API that Threema provides to integrate other applications and clients with the Threema messaging architecture advises taking public keys from other users' QR codes. If a public key contained in a scanned QR code is not checked against the directory server, an attacker could claim any public key as their own.

Attack Description and Impact

The objective of the attacker is to convince the victim to send a message that, other than being and E2E message, doubles as a valid vouch box, containing the aforementioned public key X to which the attacker knows the corresponding private key x. This means that the message that the attacker receives should be the AEAD encryption $\mathsf{AEAD.Enc}(K, X; n)$, with $K \leftarrow \mathsf{DH}(a, S)$ and n being a random nonce. By comparing the structure of a vouch box with the one of an E2E-encrypted message, this means that the victim must send a message to an account which has the server's public key as their own long-term key and that X has to begin with the message type byte `0x01` and end with valid PKCS7 padding.

Let (a, A) be the long-term keypair of the victim and assume that the attacker has successfully managed to claim the server's public key as their own long-term key. If the attacker manages to convince the victim to send σ (the ASCII part of the public key X) as a text message to the attacker, this would result in the victim deriving a key $K = \mathsf{DH}(a, S)$ and encrypting the message using the AEAD. Since σ is sent as a text message, the plaintext will include `0x01` as the type byte, followed by the string σ and the PKCS7 padding at the end. The attacker hopes to obtain a ciphertext c equal to $\mathsf{AEAD.Enc}(K, \texttt{0x01} \| \sigma \| \texttt{0x01}; n)$ (for some value n), which has probability $1/254$ to occur due to the requirement of obtaining `0x01` as a PKCS7 padding. Because the adversary has claimed the server's public key as their own, K is the same as K_2, the key that would be derived by the victim during the C2S protocol to create the vouch box. Then, by construction, the ciphertext c that was created and sent to the attacker is thus a valid vouch box for X, which the attacker can now use to authenticate to the Threema server.

The last limitation to discuss is how an attacker can claim S as their long-term public key. To do so, we exploit a vulnerability in one of the dependencies that Threema uses on the Android version of the app for creating data backups (described in Section 3.7.3). The library used by Threema to create zip files is Zip4j [62], which possesses a bug where, under certain conditions, the Message Authentication Code (MAC) would not be checked when decrypting the zip file. We discovered that these conditions are fulfilled whenever a Threema backup is restored, meaning that any modification of the zip would not be detected. In the context of Threema, this allows an attacker that has access to the encrypted zip to modify the contacts file within it, allowing them to claim the server's public key as theirs. For example, a user might choose to save their zip in some shared storage (e.g. a cloud service, or a folder on a computer which the attacker can access), believing it to be stored securely.

More specifically, the attacker can leverage the malleability of AES-CTR, which is used to encrypt the single files, to flip single selected bits in the plaintext by flipping the same bits in the ciphertext. This means that an adversary with knowledge of the plaintext can change it so that, after decryption, the message will be modified to resemble a target plaintext of the attacker's choice. The message authentication code, if properly checked, should prevent this malleability property from being exploited.

We note that, however, this attack requires that the attacker has knowledge of which users are in the contact list of the victim, since the zip can only be "blindly" modified after compression and encryption. While this is technically not a Threema vulnerability, we stress that a more robust design would have prevented this from being escalated to client impersonation in the C2S protocol. Furthermore, we cannot exclude the possibility that an alternative method exists for an adversary to claim the server's long-term key as their own, for example targeting different client implementations.

We created a Python tool that given a zip file and a target file can change the zip in order to replace a file with the target file.

Generating Valid Public Keys

Given that there are 95 printable ASCII characters among the 256 possible values of a single byte, we evaluate the probability of a random byte string to encode valid ASCII to be $\left(\frac{95}{256}\right)^{30}$, which, when multiplied by the probability of obtaining two `0x01` bytes, yields a probability of obtaining a valid public key of approximately $\approx 2^{-58.9}$. To decrease the complexity of the key search, we initially tried to relax the validity condition by requiring the middle section to consist of 30 valid UTF-8 characters, rather than ASCII. According to the formulae in [1], we can compute the probability of a random byte string of 30 characters to encode a valid UTF-8 string to be around $\approx 2^{-24.9}$. Combined with the requirement of the `0x01` bytes, this yields a probability of obtaining a valid public key of $\approx 2^{-40.9}$. Unfortunately, all keys that we tried to generate in this manner turned out to be impossible to copy and paste reliably, since some characters, such as ASCII control characters, would be removed when the message was pasted in the chat box.

In order to look for keys, we have created a Rust script that generates many private keys and checks if the corresponding public key matches the desired pattern. When looking for UTF-8 strings rather than ASCII strings, we found 12 valid public keys. The valid public keys, as well as the corresponding private keys, can be found in Appendix A. Unfortunately, all the generated keys contain ASCII control characters which make the public keys unsuitable for copy-pasting. We ran the same script in order to find public keys containing only printable ASCII in their middle section without success yet. We expect that, given more time and computational resources, a suitable key could be found.

4.2 Compromised Threema Model

For the following three attacks, we assume that the Threema servers have been completely compromised. This means that the adversary has compromised the long-term keys of the server S as well as their TLS keys, allowing the adversary to impersonate the server to any user that tries to connect. The expectation for a truly secure messenger is that, even in this setting, the server is not able to read nor interfere with end-to-end communications except for denying communications entirely. We present three attacks that allow an attacker to change the semantics of the conversations between honest users. Attack 5 has been discovered and patched in version 4.6.14 for iOS and 4.62 for Android, prior to our analysis [40]. We choose to include it regardless in order to highlight what we believe to be a lack of structural soundness in the Threema cryptographic design.

4.2.1 Attack 3 (Message Reordering and Omission)

Strict message ordering is an important property in a messaging application: the context in which messages are sent is important for the semantics of an end-to-end conversation. Rearranging messages, possibly in an adversarial way, has important consequences on the actual meaning that is conveyed. We show that Threema lacks the capacity to enforce message ordering at the E2E level, enforcing it uniquely at the C2S level. This directly implies that, if the server is ever compromised by an adversary, then no ordered delivery property can be guaranteed.

Attack Description and Impact

In the structure of an E2E packet (in Section 3.3), there is no inherent integrity-protected mechanism for enforcing order, such as use of a monotonically increasing counter. The timestamp in the message is not strictly protected from tampering, since the integrity-protected metadata box is not mandatory. Thus, a malicious server can run a trivial reordering attack by storing messages that are received from the client via the C2S channel and decide the order with which to forward them to the intended recipient. If necessary, the server can overwrite the timestamp with a plausible value, in order to evade detection. Furthermore, the adversary can omit messages when forwarding them, effectively eliminating those messages from the conversation.

The behaviour of the Threema application is that incoming messages are shown in the order in which they are received, rather than ordering by timestamp. This means that an attacker that is unable to tamper with the metadata would still be able to reorder messages with the granularity of a minute, since the Threema app does not show the seconds in the message information. Thus, protecting the timestamp from tampering and relying on the fact that the user will visually detect message reordering, is an insufficient protection and there is a need for additional mechanisms to enforce ordering.

We note that this attack is not feasible by a network attacker without control of the server or without being able to impersonate the server. In fact, as explained in Section 3.4, the C2S protocol encrypts messages using nonces derived from a counter, which disallow any reordering since doing so would lead to the (honest) server decrypting with the wrong nonce, leading to a decryption error with overwhelming probability and thus to the termination of the C2S connection. The same argument applies to message omission. In Attack 1, however, we described how leaking a client ephemeral key can lead to an attacker being able to impersonate the server for up to one week, allowing for this attack to be executed by a network attacker.

4.2.2 Attack 4 (Replay and Reflection Attacks)

In Section 3.3 we explained how Threema tries to prevent replay and reflection attacks by storing nonces of both outgoing and incoming messages. Fundamentally, this requires that the nonce database is always kept updated and is never deleted. We show this cannot be guaranteed in some instances, leaving the user vulnerable to replay and reflection attacks. Such attacks, like the previous one, allow an attacker to change the semantics of the end-to-end conversation between two parties.

Attack Description and Impact

Whenever the app is reinstalled or when the user changes device, the data in the application can be restored by using the backup methods discussed in Section 3.7. We note, however, that none of the backup methods include the nonce database, making it impossible to restore it. As a consequence, whenever the app is reinstalled, the nonce database is irreversibly lost and, when installing the app on a different device, there is no user-friendly method to transfer the nonce database. Whenever the database is reset, a compromised Threema server is able to replay messages that were received by the victim in the past or reflect messages that the victim has sent.

The underlying weakness that this attack shows is again the lack of integrity-protected metadata that would specify the ordering of messages, preventing a message to be delivered twice, as well as specifying the source and the destination.

To make this attack more effective, we note that the Threema server can easily notice when a user has most likely 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. After a user restores a backup from Threema Safe, they will connect to the server with the C2S protocol, most likely with the same IP address. Thus, not only can the server learn the identity associated with a given address, but they will also know that their nonce database has been deleted, making them a target for the aforementioned attacks.

4.2.3 Attack 5 (Kompromat Forgery)

We present an attack that involves a cross-protocol interaction between the registration protocol and the E2E protocol. This attack uses the registration protocol to forge an arbitrary message that is unwittingly encrypted and authenticated by the victim and that can be sent to a target user as a fresh message. This attack has been patched in Threema version 4.6.14 for iOS and 4.62 for Android [40], so we initially present the attack for the unpatched version of the application in order to discuss the weakness of the protocol against cross-protocol attacks. While the patch does address the specific attack, it does not address the more general problem, which is shown by our other attacks.

Attack Description and Impact

Recall from Section 3.2 that, whenever a user $\mathcal{A}$ tries to register to Threema, they must prove to the server that they own their public key. This is done in the form of a challenge-response protocol where 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, under a fixed nonce n. The main insight of the attack is that the server does not have to own the private key x corresponding to the provided ephemeral public key X. The adversary can then put the long-term public key $B = g^b$ of any other Threema user $\mathcal{B}$ in place of the challenge key X. This means that $\mathcal{A}$ will be encrypting the challenge with $K \leftarrow \mathsf{DH}(a, B)$: the same key that they would use to communicate with $\mathcal{B}$. If the challenge has the correct shape of a text message (i.e. starting with `0x01` and ending with valid PKCS7 padding), the resulting encryption can be used as a forged message to be sent to either $\mathcal{A}$ or $\mathcal{B}$.

There are a few caveats with the attack in the described form: first, as the attack happens during registration, the server has no knowledge of the identity of the targeted victim. This means that the server must commit to the public key B to use and to the message m for which forge a ciphertext. The message can only be sent once to each of $\mathcal{A}$ and $\mathcal{B}$, due to the nonce-based replay prevention described in Section 3.3.

Enhanced Attack If a user enables the optional contact discovery feature, this attack can be further enhanced: recall that each time the client wants to run the contact discovery protocol, they must prove ownership of the private key by using the same contact discovery protocol used for registration, except for the fact that the nonce is now randomized rather than being fixed. If activated, the contact discovery is run once every 24 hours, allowing a malicious server to forge a new message per day, with the possibility of changing the public key and the message used, making this the most flexible version possible of the attack. Since the nonce is always randomly selected, each new encryption will be accepted by the victim (either $\mathcal{A}$ or $\mathcal{B}$) as a fresh message with overwhelming probability. The practical consequence of multiple forgeries is that, for example, group chats can be forged and that VoIP calls can be faked. This is because VoIP communications use a shared key established in an initial *signalling* phase between the two parties, which is run over the E2E protocol. If the attacker can forge signaling messages, the victim $\mathcal{A}$ will believe that they are establishing a VoIP key with $\mathcal{B}$ even though the actual communication will be established with the attacker.

Vulnerability Patch and Discussion Both the initial attack and the enhanced version have been fixed in Threema version 4.6.14 for iOS and 4.62 for Android [40] by requiring that the message chall provided by the server start with `0xff`, enforcing separation between message types and thus preventing it from being recognized as any other type of valid message. We highlight the patch in the description of the registration protocol in Fig. 3.2. Nonetheless, this is 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 Chapter 5 we discuss mitigations that would better protect against these attacks.

4.3 Compelled Access Model

4.3.1 Attack 6 (Cloning via Threema ID Export)

The following attack allows the adversary to reveal the long-term private key of the victim, assuming that the adversary has control of the unlocked phone and if no locking mechanism is set in Threema (we note that this is the default behaviour for Threema and the locking mechanism must be explicitly activated in the settings). As described before, the application provides the possibility of exporting the Threema ID (as explained in Section 3.7.2). However, this method accepts any password as valid, virtually allowing an attacker to encrypt the private key with any password and decrypt it on their own device.

This is an intentional feature in Threema, which we deem to be a grave security danger: all the security in Threema relies on the knowledge of the long-term private key, and this attack allows an adversary to recover it by having access to the unlocked phone for a few minutes.

While similar mechanisms are present in other messengers, their impact is much smaller: in Signal and Whatsapp, other devices can be linked to the main device, allowing them to receive the same messages as the main device, as well as accessing the history of messages [88]. However, this is not only visible to the owner of the main device, but it is also reversible: by invalidating the linked device, it will not receive new messages anymore. This is in contrast with Threema, where losing the private key irreversibly forfeits all security.

4.3.2 Attack 7 (Private Key Recovery through Compression Side-Channel)

We now present an attack that also requires an unlocked phone to be executed. However, in contrast to the previous attack, the following attack can also work in the case where the user used an access control mechanism to protect the Threema application. Recall from Section 3.7.1 that a Threema Safe backup contains the private key of the user, which allows it to be retrieved when the user wants to set up their Threema account on a new device. However, before being sent to the server for storage, the backup is compressed and then encrypted. A well-known attack against this compress-then-encrypt paradigm is the CRIME attack due to Rizzo and Duong [79], which harnesses the fact that, if the attacker can control part of the plaintext before compression, then it can also use that power to leak contents of the plaintext through the length of the resulting compressed string. As discussed in Section 2.2.1, XSalsa20 is a stream cipher, which allows an adversary to infer the length of the plaintext from the length of the ciphertext. Furthermore, TLS often does not hide the length of the payload, which, as we conjecture, makes the attack viable even by an attacker that cannot see the ciphertext directly but can see TLS-protected network traffic.

Our attack can recover most of the private key contained in the backup after 23k backup attempts from the user on average, which allows the remaining part to be retrieved using offline techniques.

We stress that leaking the private key leads to a complete loss of security: after doing so the attacker can impersonate the user in any action. The attacker can send messages on behalf of the user, they can covertly receive a copy of messages intended for the user using the same methods described in Attack 1. Furthermore, by incorporating the compromised Threema threat model, this leads to an attacker being able to decrypt all past communications due to the lack of forward secrecy on the E2E level.

Attack Description and Impact

We start by assuming that the attacker has managed to obtain the victim's unlocked device. We may also assume that the victim might be employing an access control mechanism such as one of the methods described in Section 3.9, which would prevent the attacker to directly gain access to the unencrypted messages or to send messages on behalf of the victim.

The structure of the JSON backup is described in Section 3.7.1. In Fig. 4.1 we show an example JSON, with the private key redacted and fake data added.

The attacker can change the contents of the backup by simply sending a message to the victim, containing a specific nickname within the E2E packet. Since nicknames are handled client-side, this will automatically change the nickname of the attacker in the victim's local data model. The objective of the attacker 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. If the attacker's nickname and the private key are sufficiently close together, they will both fall within the same sliding window of the zipping 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 induces the compression algorithm into finding redundancy between the nickname and the private key, creating the compression side channel.

The attacker starts by setting their username to their guess for the base64-encoded key, along with the canary string "privatekey" at the start and the "=" canary at the end of the nickname. The first one corresponds to the name of the field that the attacker wants to leak, whereas the second one is the final character of the base64-encoded string. We can be sure that the equal sign will always be present since the length of the key (32 bytes) is not divisible by 3, which requires padding with the base64 padding character, the equal sign. In our experiments, both canary strings were necessary in order to exploit the vulnerability, although we cannot explain why the second canary is necessary. By following techniques similar to the ones described in the presentation of the CRIME attack, the attacker can leak one or two characters at a time from the private key.

More specifically, the attacker begins by guessing one character at a time from the base64 alphabet and then tries to induce the client into creating a new backup. For example, the attacker can wait until a backup is scheduled and make the backup fail. Then, the attacker can leverage the client's retry behaviour and induce a backup every time the app is restarted. We abstract this process by devising an oracle that can be queried with a JSON backup returns the length of the corresponding encrypted gzip.

After querying the oracle with all 64 characters, the attacker checks which character induces the shortest compressed ciphertext. Sometimes the shortest ciphertext corresponds to only one character, at which point the attacker knows that to be the correct guess. However, most times, multiple characters can induce a ciphertext of minimal length, at which point the attacker can retry with two-character combinations. After guessing either the correct single character or two-character combination, the attacker updates their guess by retaining those characters and prepending a new guess for the next character, until the entire key is leaked. One thing to note, however, is that the size of the nickname field is limited to 32 characters. This implies that one needs to "slide" our guess as the attack goes on.

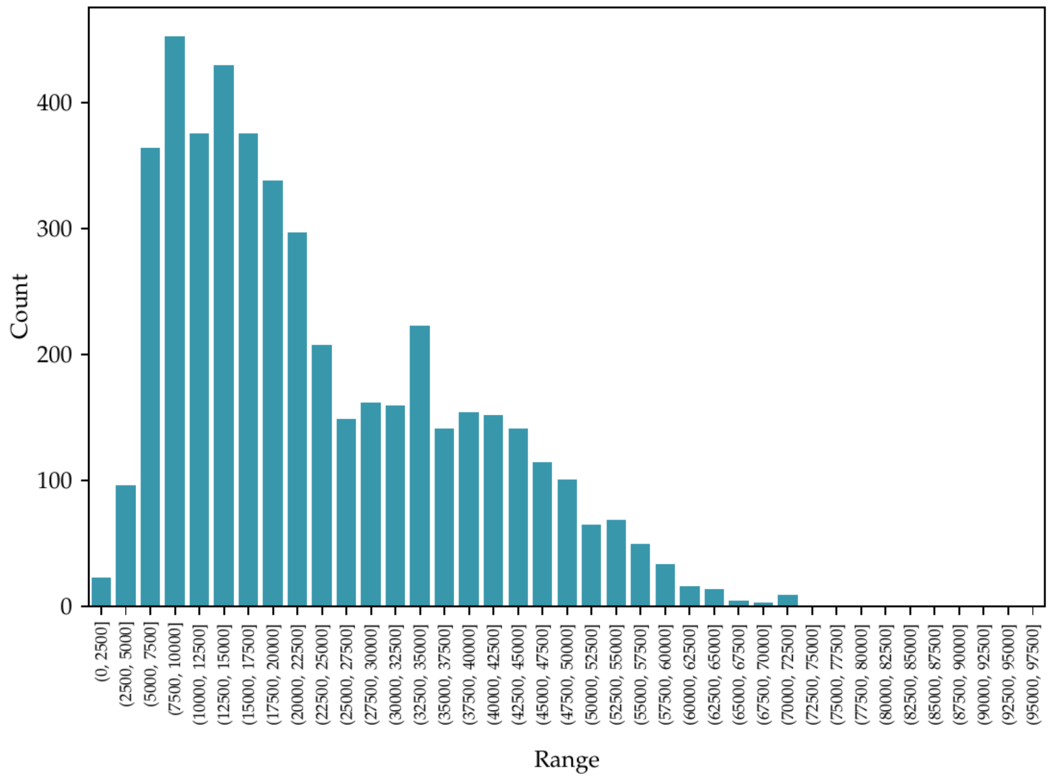
We have made some tweaks that increase the probability of success in our setting: we leak keys from the last byte to the first one. The reason is twofold: first, we already know the last character to be an equal sign, which means that this method is feasible and, second, the last character of the encoded key represents a sextet whose last two bits are set to 0 due to padding. This means that there are only 16 possible characters to search for when looking for the last byte.

We give a concrete example which highlights the most important steps.

1. The attacker begins by querying `privatekeya=` to the oracle. Then they query `privatekeyb=` and so on.
2. The attacker discovers that the letter `s` induces the shortest ciphertext. This means that the key ends with `s=`. The next set of queries will then be `privatekeyas=`, `privatekeybs=`, ..., `privatekey9s=`, `privatekey+s=`, `privatekey/s=`
3. The attacker cannot find a single character that induces the shortest ciphertext. The characters `a` and `Z` both induce the same length, which is minimal among all queries. The attacker then proceeds to query two-character sequences `privatekeyaas=`, ..., `privatekey/as=`, `privatekeyaZs=`, ..., `privatekey/Zs=`.

4. The last set of queries reveals that `+a` is the correct two-character sequence, at which point the attacker knows that the key ends with `+as=`.
5. The attack proceeds similarly, until the 32 character limit is reached. Assume that the key known at this point is `2H/lXdIp5aMV0ZhR+I+as=`. Including the initial canary, this turns out to be 32 characters in length. The next queries will "slide over" and will not include the last character of the key. The attacker proceeds to query `privatekeya2H/lXdIp5aMV0ZhR+I+as`, `privatekeyb2H/lXdIp5aMV0ZhR+I+as` and so on.
6. The attack continues until the attacker has collected a sufficient amount of information on the private key.

Figure 4.2: The distribution of number of queries in our successful experiments(Over 10000 experiments). On the x-axis, we grouped the number of queries in ranges of width 2500. On the y-axis, the number of experiments that fall within that range.



Our simple algorithm manages to reach a success rate of 47.27% over 10000 simulations. In our simulations, the number of oracle queries required was 23382 in the average case, 90176 in the worst case, and 1920 in the best case. We expect that a more complex algorithm would be able to have a better success rate, at the expense of requiring more queries. We attempted to execute a binary search on the characters, by using 32 users with different guesses for the nickname, with inconclusive results due to the unreliability of the side-channel. In Fig. 4.2 we show the distribution of the number of queries among all our 10000 experiments. We did not include the 5273 experiments that did not succeed.

As previously described, to obtain this number of queries, it suffices for an attacker to forcibly close the application and start it again, which induces the application into retrying the upload of the backup. This can be done by either using the debugging tools or by simulating user input on the unlocked phone. Whenever an upload attempt happens, a network attacker can make it fail by intercepting the network traffic and dropping the associated packets, allowing a new upload to be attempted at the next restart.

Not all bytes of the key have to be recovered, since there are offline techniques such as Pollard's Kangaroo algorithm or the parallelized Van Orschoot-Wiener that can be used to recover the remaining bits [68]. Indeed, this creates a trade-off between the number of queries to the oracle and the amount of offline work required by the attacker. Since the offline work can be parallelized and does not require access to the device anymore, it is convenient to execute just the sufficient amount of queries required to make the offline attack feasible. For example, the complexity of Pollard's Kangaroo algorithm is $\mathcal{O}(2^{n/2})$, where n is the size of the set to be searched for the discrete logarithm, while for Van Orschoot-Wiener the complexity is $\mathcal{O}(\frac{1}{L}\sqrt{\frac{\pi 2^n}{2}})$, where L is the number of parallel threads used. For example, if we recover 31 base64 characters, we are left with 13 characters, equivalent to 78 bits after decoding. This means that the runtime of the offline step will be close to $2^{78/2} = 2^{39}$ with Pollard's algorithm, which is within feasibility. At the pace of one restart every two seconds, an attacker can recover 31 base64 characters of the private key in 50 hours in the worst case of our experiments and 1 hour in the best case.

As a proof-of-concept for the attack, 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 uploads fail by returning any HTTP error code. To automatically induce a new backup, we use the Android debugging tools which can force an application to stop (with the `adb shell am force-stop ch.threema.app` command) and restart (with the `adb shell monkey -p ch.threema.app 1` command). We conjecture that this attack could be executed by a network attacker with control of the unlocked device as well, although we have not tested this in practice. This is because TLS-encrypted traffic usually does not hide the length of the payloads and because the adversary can drop arbitrary packets, making the upload fail.

4.3.3 Attack 8 (Invisible Salamanders in Group Messaging)

Due to the cryptographic design of the group messaging system described in Section 3.5, it is possible for a user to send different messages to different people. This lack of transcript consistency can be used by a malicious user to misdirect other users in a group by sending them a different message with respect to the other members of the group. This is a problem by itself, since it can be used to deceive users, but it also leads to another attack that exploits the lack of robustness of XSalsa20-Poly1305. This attack was first described by the cryptography researcher nicknamed Soatok in one of their blog posts [90].

Attack Description and Impact

Dodis et al. [29] have shown that it is possible to generate an AES-GCM ciphertext that decrypts to two different plaintexts when using two different keys. Later Len et al. [61] have shown that this same property holds for XSalsa20-Poly1305. Recall the way media messages are sent in Threema: the media is first encrypted using XSalsa20-Poly1305 with a random key, it is then uploaded to the media server, who returns a blob ID. The user then sends the blob ID, as well as the key, to the other user, encrypted using the E2E protocol. By using the aforementioned techniques, an attacker can create a ciphertext c which will decrypt to one of two different media files, depending on which among two keys K_1, K_2 is used. Then, the attacker uploads the ciphertext to the server, which will reply with a blob ID. The attacker now sends an E2E-encrypted payload containing the blob ID and one of the keys, depending on which media file the attacker wants the user to see.

4.4 Discussion

As we discussed during the description of the E2E protocol (Section 3.3), from the point of view of the Threema server, forward secrecy does not hold in end-to-end communications. This is because the forward secrecy property is only provided at the client-to-server level rather than the end-to-end level. If the Threema server were to be compromised, this would mean that the attacker could store all messages passing through the Threema server and decrypt them at a later time, after having compromised the user's long-term private key.

This is explicitly acknowledged in the Threema security whitepaper, citing 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 technically true in practice, a messaging app cannot claim to have "maximum security" [42] 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. If that were to be true, then it should also hold when using Threema OnPrem [44], the on-premise solution for Threema, where compromise might be easier and where the owner can always see E2E-encrypted messages in transit. In a messaging app, compromising the server should not reveal a significant amount of information to the attacker, but, due to the lack of forward secrecy on the E2E level, this expectation is not met for Threema.

Our attacks also show that this weak notion of forward security may be insufficient: by harnessing one of our attacks on the custom C2S protocol, the adversary gains access to E2E-encrypted messages, which are encrypted with long-term keys. Furthermore, the reuse of ephemeral keys for a long period of time further weakens the guarantees of 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 message is encrypted with a different key, providing a much stronger notion of forward secrecy.

```
{
    "info": {
        "version": 1,
        "device": "4.64A\/en_US"
    },
    "user": {
        "privatekey": <base64-encoded private key>,
        "nickname": "user_nickname",
        "links": []
    },
    "contacts": [
        {
            "identity": "ABCDEFGH",
            "createdAt": 1647620663019,
            "verification": 0,
            "workVerified": false,
            "nickname": "other_user_nickname",
            "hidden": false,
            "typingIndicators": 0,
            "readReceipts": 0,
            "private": false
        },
    ],
    "groups": [],
    "distributionlists": [],
    "settings": {
        "syncContacts": true,
        "blockUnknown": false,
        "sendTyping": true,
        "readReceipts": true,
        "threemaCalls": true,
        "locationPreviews": false,
        "relayThreemaCalls": false,
        "disableScreenshots": false,
        "incognitoKeyboard": false,
        "blockedContacts": [],
        "syncExcludedIds": [],
        "recentEmojis": []
    }
}
```

Figure 4.1: Example of a JSON backup, with the private key redacted

Chapter 5

Mitigations

We now present an overall discussion on what we believe to be the underlying issues with the different parts of the Threema cryptographic design, attempting to find multiple solutions. We initially present simple solutions that would directly prevent our attacks but that do not tackle the issue themselves, since these solutions are often easier to implement in the short term and thus useful to protect user data in the immediate future. We, however, also present solutions that require a greater engineering effort but that would solve what we deem to be the weaknesses in the protocols. In the long term, we hope that Threema will adopt these stronger mitigations in order to provide better security guarantees for their users.

We discuss mitigations by tackling each issue separately: we begin by discussing mitigations against cross protocol attacks (Section 5.1), then we discuss how to protect the integrity of the metadata (Section 5.2), we find solutions to the issues found in the C2S protocol (Section 5.3), we discuss solutions to harden the application against cloning via the ID export feature (Section 5.5), and, finally, we discuss how to add forward secrecy to the E2E protocol (Section 5.4).

5.1 Mitigating Cross-Protocol Attacks

The attacks in Sections 4.1.2 (Vouch Box Forgery) and 4.2.3 (Kompromat Attack) exploited the fact that the same "DH-then-Encrypt" paradigm is used multiple times and for different purposes. For example, the E2E protocol uses it for authenticated encryption among users, while the C2S protocol and the registration protocol use it to authenticate the user to the server. In these instances, one of the parties can be fooled into deriving and using a key that is also used in a different context. By using payloads from a protocol in a different protocol, the adversary gains capabilities that they would not possess when viewing the protocols separately. Furthermore, if unmitigated, these cross-protocol attacks create user-unfriendly and deeply unintuitive situations, such as in the vouch box forgery attack, where sending an E2E message manages to compromise client authentication. A secure application must thus compartmentalize its protocols, taking a conservative approach when using the same cryptographic material in different protocols.

A common way to prevent these attacks is to treat the result of the Diffie-Hellman computation as raw key material, not to be used directly as a key, but from which the actual key is to be derived by using a key derivation function (Section 2.2.4). When deriving a key, different labels should be used depending on the context where the key will be used e.g. "c2s-authentication" and "e2e-encryption". By using a suitable KDF, the probability of deriving the same key in two different contexts is negligible, preventing the attacks we described. Differently from Threema's usage of a KDF (for example, in the E2E protocol to encrypt the metadata box) a key that is used to derive another key should not be used for another purpose. This is in accordance to the key separation principle: suppose, in the current Threema implementation of key derivation in the E2E protocol, that the key $K_{\mathcal{A},\mathcal{B}}$ is compromised, then the metadata key $K_{\mathcal{A},\mathcal{B}}$ is compromised as well. On the other hand, suppose that we used the result of the Diffie-Hellman computation as input to the KDF, using two different labels: compromising one of the dervied keys does not directly lead to compromising the other derived keys.

As a concrete example, we propose the usage of the HMAC-based KDF (HKDF) proposed by Krawczyk in [58] and standardized in RFC 5869 [57], since it is widely used in protocols such as TLS.

This solution, however, encounters a backwards compatibility problem: since older clients will not run the additional key derivation step, they will not be able to decrypt and encrypt messages correctly. Since the Kompromat attack has already received a fix, we aim at finding a short-term solution for the vouch box forgery attack as well, which allows patched clients to communicate with unpatched clients while preventing the attack at the same time. For example, we can require clients to pad E2E messages to at least strictly more than 32 bytes. This would ensure that these messages cannot be used in lieu of a vouch box, since the latter is always 32 bytes in length. We give another mitigation of the Vouch Box Forgery attack in Section 5.3.

5.2 Preventing Tampering with Metadata

The simplest way to prevent attacks 4.2.2 (Replay and Reflection Attacks) and 4.2.1 (Message Reordering) is to protect the integrity of the metadata contained in the E2E packet. In contrast with the usage of the metadata box, this should include the source and destination as well, which are necessary to prevent reflection attacks. The most efficient way to do this is to use the full AEAD construction, by adding all the metadata as the "additional data" part. In the case of XSalsa20-Poly1305 this would mean that the metadata is included in the computation of the Poly1305 tag, but is not encrypted. This modification, thus, comes at zero message overhead and supersedes the usage of the metadata box, eliminating both the need for an additional ciphertext and tag, and the need of an additional key derivation to compute the metadata key. Protecting the metadata in this way prevents any malicious entity from swapping the source and destination of the message and thus prevents any reflection attacks. This has also the added advantage of not requiring the nonce storage used by Threema, significantly decreasing the storage requirements of the application and requiring less database accesses when receiving and sending messages.

To completely prevent replay attacks, message reordering, and omission, the application could include a monotonically increasing counter with every message and the receiving client must display messages to the user in the order given by the counter and check for missing messages. This counter must be included in the additional data of the AEAD in order to prevent tampering.

A more immediate mitigation consists in enforcing the presence of a metadata box, which would at least protect the timestamp and allow the client to enforce message ordering by timestamp. This is, however, insufficient to protect against reflection attacks, which would require to add the source and destination to the metadata box.

5.3 Mitigating Attacks on the C2S Protocol

The least invasive modification to the protocol that directly prevents our attack of Section 4.1.1 (Impersonation by Randomness Failure), as well as mitigating the attack of Section 4.1.2 (Vouch Box Forgery), is to include the server cookie inside the vouch box. This ensures that, as long as the server picks new cookies every time, the vouch box cannot be trivially replayed. Furthermore, since the server cookie often contains non-printable characters, it makes the vouch box forgery attack less likely to succeed, since some characters cannot be copy-pasted by the victim. Even if the server cookie consisted entirely of printable characters, the forgery has to happen while the C2S protocol is running and a valid forged vouch box would be valid only during that single handshake, mitigating the impact of the attack.

Nonetheless, we argue that these issues can be fixed while improving the C2S protocol itself: while Threema claims the protocol to be optimal in term of round-trips [38, p. 10], there are protocols which provide the same desired properties with less message overhead. An example is the IK protocol of the Noise Framework [73], which works assuming that the client knows the server long-term public key and *vice versa*, similarly to the current Threema C2S protocol. The IK protocol allows for 0-RTT encrypted communications between the client and the server and provides authentication and forward secrecy. Furthermore, the Noise Framework has been formally analyzed and proved to be secure by Dowling et al. [30].

Another option would be to use TLS [75], which is widely deployed and supported. To decrease the communication overhead, a 0-RTT resumption mode can be used.

5.4 Implementing Forward Secrecy in the E2E Protocol

In order to provide forward secrecy, a new key has to be periodically negotiated between the users. This ensures that, even though a shared key is leaked to an attacker at some point, the previous keys will not be affected.

Unfortunately, we believe there to be no short-term and easy to implement solution 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. Any solution would require a re-design of the E2E protocol. We, however, disagree with the statement by Threema that "providing reliable Forward Secrecy on the end-to-end layer is difficult" [38]. In fact, we believe that the best course of action is to adopt the Signal protocol, whose cryptographic API is available through the libsignal library [83]. The Signal protocol has received extensive security analysis throughout the years [35, 22, 55] and provides forward secrecy and post-compromise security at the end-to-end layer. Adopting the Signal protocol would allow Threema to implement forward secrecy without impacting the user experience. On this point, we stress that Whatsapp itself has transitioned towards using the Signal protocol in the past [89], showing that such a change is feasible even for a messaging app with an enormous user base such as Whatsapp.

5.5 Preventing Cloning via Threema ID Export

The ID export feature, while convenient for users, poses a great security risk for users of the application. All security in Threema hinges on the security of the long-term key, and allowing a user to choose an arbitrary passphrase when creating an ID export gives an attacker the possibility of creating a backup with a known passphrase, effectively handing the long-term key to the attacker.

Solutions to this problem are, for example, forcing the choice of the passphrase at the time of the creation of the account, which disables the possibility for an adversary to choose a passphrase when requesting an ID export. This, however, comes at the cost of usability, since forgetting the passphrase would block the user from creating new backups, as well as recovering old ones. A more robust solution would be to require the user to input the PIN of their phone or to go through a biometric login procedure.

In general, we believe that Threema should enforce better access control in the application, leveraging the possibility of using biometrics to provide a convenient method for users to access sensitive settings.

5.6 Mitigating the Compression Side-Channel

Given that every user is encouraged to activate the Threema Safe feature, compressing the backups appears to be a sensible decision, since it allows Threema to save on the storage requirements. However, we have showed in Section 4.3.2 that this can be leveraged by an attacker to reveal long-term private keys. In order to prevent the attack at its root, it is necessary for Threema to completely avoid compression. However, this might not be feasible, given the storage requirements. An alternative mitigation, that does not remove the vulnerability but makes it harder to exploit is to pad the length of the backup to a multiple of n characters, for a suitable value of n (for example $n = 256$). This makes changes in the compression harder to infer from the ciphertext length, thus mitigating the vulnerability. However, it is known that CRIME-style attacks can still work with block ciphers, as long as the attacker can find a “tipping point” where an incompressible character overflows into an additional block [36], which means that the vulnerability is still present, albeit harder to exploit.

Chapter 6

Conclusions

We conclude by discussing the security promises of Threema, in light of the attacks we found (Section 6.1). We then discuss, in a more general fashion, *why* Threema fails to deliver their security promises despite their usage of secure primitives and libraries (Section 6.2).

6.1 Revisiting Threema's Security Promises

In Section 1.2.1 we discussed the promises that Threema tries to deliver with their application. We now analyze them again, in order to see which ones hold and which ones do not, in light of the attacks described in Chapter 4. We begin with the ones that hold: promises number 3 (Local group handling), 6 (Repudiability of messages), 8 (Protection of private keys in local storage), and 9 (Anonymity of Threema Safe Backups) are not contraddicted by any of our attacks. Most of the other promises, however, are affected by our attacks: revealing the long-term private key of the user, for example by using our Threema Safe compression-side channel, immediately breaks all other promises.[1] In a more targeted manner, property 5 (User Authentication at the Client-to-Server level) is broken by either the "Impersonation by randomness failure" attack or by the vouch box forgery attack. Property 4 (Forward Secrecy at the Client-to-Server level) is indirectly broken by the same attacks, through the retry behaviour of the server when delivering messages to the user. Property 7 (Replay/Reflection Attack Prevention) can be broken whenever the nonce database is reset, as described in Section 4.2.2.

[1]Technically, some of the properties are worded in a way that does not make this a break: the properties hold as long as the attacker does not have the private key of the user. Clearly, one would prefer to exclude the possibility of the attacker revealing the long-term private key of the user.

Most importantly, we stress that even if the application delivered on all the security promises stated, the security level might be insufficient for high-risk users. For example, not providing forward secrecy at the E2E level puts users at risk of being surveilled by a compromised Threema server. The possibility of cloning the application quickly is also another issue with Threema, as we discussed in Section 4.3.1. Noticeably, the application does not claim to guarantee any sort of end-to-end ordering of message delivery and we show in Section 4.2.1 that the application indeed does not provide it.

6.2 Analyzing Threema's Security Failures

We now try to analyze, in our opinion, why Threema fails to deliver on its security promises despite using modern cryptographic primitives that make it harder for the developer to introduce vulnerabilities. For example, their usage of a secure AEAD such as XSalsa20-Poly1305, combined with a modern elliptic curve such as Curve25519 would seem to put Threema in a good position to deliver a secure product. However, we have shown that Threema still presents various security vulnerabilities. We claim that the way Threema fails is two-fold: *custom protocols* that lack important security properties in their design, and *cross-protocol interactions* that create vulnerabilities that do not exist when considering protocols separately.

The former class of problems showcase how hard it is to create secure protocols despite using secure basic primitives. The C2S protocol in Threema is a clear example: the usage of `crypto_box` from the NaCl library prevents trivial attacks, but cannot protect the application from a flawed design. When building applications that are meant to provide security, relying on the cryptographic community is of utmost importance: developers should try to rely upon protocols that have been already analyzed by the cryptographic community and that have provable security guarantees. Nonetheless, despite using secure and well-tested protocols, it is still possible to create security issues. This is the second class of problems that we found in Threema: composing secure protocols does not directly lead to a secure application. We have shown that this can and should be mitigated by separating the cryptographic material used by each protocol, in accordance with the key separation principle. What Threema shows is that, despite being widely known as a piece of folklore in the cryptographic community, the key separation principle evidently fails to penetrate even in projects that try to use state-of-the-art cryptography.

Clearly, when building a secure application that involves cryptography, there is the necessity of allowing cryptographers and security researchers to analyze the protocols and code. For this reason, we commend Threema for open-sourcing their code to aid with auditing their application. On the other hand, we highlight that, despite the developers' efforts in fostering external auditing of the application, neither of the two official audits has focused on the cryptographic design, even though Threema is an application whose main value is derived from encrypted messaging.

6.3 Conclusions

We have analyzed the Threema messaging application and cryptographic design, highlighting shortcomings in their security and providing attacks against the Threema messaging protocol. The attacks that we provide show the pitfalls of implementing and deploying a protocol that is not well-analyzed despite usage of secure and modern cryptographic primitives. We finally provided mitigations and discussion on how future development should proceed.

Appendix A

Generated Public Keys for the Vouch Box Forgery Attack

All values of public and private keys are base64-encoded.

1. Public key `2OPjOxQAAAAAdGhyZWVtYS1wb2lzb25rZXktMDA2MgA=`

   Private key `AWp0EDRfRUpmXF0i3pXemFIqcdeKLExnFkxhWngYGAE=`
2. Public key: `2Pbsjg8AAAAAM21hLXZvdWNoLWJveC1rZXktMDA1MgA=`

   Private key: `ASt3EEdQcH0vChjMovCRho/NmBZUd2JeQS1odC3LowE=`
3. Public key: `0IYNsR0AAAAAM21hLXZvdWNoLWJveC1rZXktMDA1NwA=`

   Private key: `AeqCjFkJUylXSCYlVURXC2zusqNQDVR0xIUwFxwQMwE=`
4. Public key: `kAp92SAAAAAAM21hLXZvdWNoLWJveC1rZXktMDExMwA=`

   Private key: `AQdLCEx+2pgKVx99ax8GxLge54SuSQMH0qUeYTFHLgE=`
5. Public key: `4M2PaSMAAAAAM21hLXZvdWNoLWJveC1rZXktMDEyNAA=`

   Private key: `AR8OQRAbZMSqG1LDrG58QF9qFRZwSBkIcmEVAVglYwE=`
6. Public key: `4OEok1AAAAAAM21hLXZvdWNoLWJveC1rZXktMDA2OQA=`

   Private key: `ATPbgj0vWDlWX8qLb+yUntKKV393NHkpTm51TlkAEQE=`
7. Public key: `IOKxmlYAAAAAM21hLXZvdWNoLWJveC1rZXktMDAxNgA=`

   Private key: `AXw6BhIZTO6+gRA83adfb1JtGWBsY2UlcVDCqnVxaAE=`
8. Public key: `wDqIXloAAAAAM21hLXZvdWNoLWJveC1rZXktMDA4OQA=`

   Private key: `ASMcE2dRSjZqBlRuHTZhK0l857KnXkjQuh0yTwU7dAE=`
9. Public key: `0L4FnFsAAAAAM21hLXZvdWNoLWJveC1rZXktMDA0NwA=`

   Private key: `AUYpCjgNRQsTRDoXH1V5L8WjSlRpaEUUeygeDjAAHgE=`

10. Public key: `y0iy8VwAAAAAM21hLXZvdWNoLWJveC1rZXktMDAyMwA=`

    Private key: `AX0DFF5mMwV2MCdMwqo+1ZDEov02sa8GGG8WPiVBBwE=`
11. Public key: `QEJ7CGEAAAAAM21hLXZvdWNoLWJveC1rZXktMDAxNwA=`

    Private key: `AWbQqyoD8LW0tGZfHDp/JQ3YlRwA2Y80Yw1qFGM1AwE=`
12. Public key: `AMP7H2oAAAAAM21hLXZvdWNoLWJveC1rZXktMDExMgA=`

    Private key: `AQVoEsaUbAfQjk53Nk9F0wAMAyQp0gs9w5seN3V0agE=`

Bibliography

[1] achille hui (https://math.stackexchange.com/users/59379/achille-hui). *Probability that random byte array is a valid UTF-8 string?* Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/751707 (version: 2022-07-06). eprint: `https://math.stackexchange.com/q/751707`. URL: `https://math.stackexchange.com/q/751707`.

[2] 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: *Commun. ACM* 62.1 (2019), pp. 106–114. DOI: `10.1145/3292035`. URL: `https://doi.org/10.1145/3292035`.

[3] Jan Ahrens. "Threema protocol analysis". In: (2014). URL: `https://blog.jan-ahrens.eu/files/threema-protocol-analysis.pdf` (visited on 09/11/2022).

[4] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. "Mesh Messaging in Large-scale Protests: Breaking Bridgefy". In: *IACR Cryptol. ePrint Arch.* (2021), p. 214. URL: `https://eprint.iacr.org/2021/214`.

[5] Martin R. Albrecht, Rafael Eikenberg, and Kenneth G. Paterson. "Breaking Bridgefy, again: Adopting libsignal is not enough". In: (2022). URL: `https://eikendev.github.io/breaking-bridgefy-again/`.

[6] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. "Four Attacks and a Proof for Telegram". In: (2022). URL: `https://mtpsym.github.io/`.

[7] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. "Scrypt is Maximally Memory-Hard". In: *IACR Cryptol. ePrint Arch.* (2016), p. 989. URL: `http://eprint.iacr.org/2016/989`.

[8] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: simpler, smaller, fast as MD5". In: *IACR Cryptol. ePrint Arch.* (2013), p. 322. URL: http://eprint.iacr.org/2013/322.

[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: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 689–706. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram.

[10] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. "MEGA: Malleable Encryption Gone Awry". In: (2022). URL: https://mega-awry.io/.

[11] David A. Basin, Ralf Sasse, and Jorge Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *CoRR* abs/2006.08249 (2020). arXiv: 2006.08249. URL: https://arxiv.org/abs/2006.08249.

[12] Daniel J Bernstein. "Cryptography in NaCl". In: *Networking and Cryptography library* 3.385 (2009), p. 62. URL: https://cr.yp.to/highspeed/naclcrypto-20090310.pdf.

[13] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, pp. 207–228. DOI: 10.1007/11745853_14. URL: https://doi.org/10.1007/11745853%5C_14.

[14] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. *NaCl: Networking and Cryptography library*. URL: https://nacl.cr.yp.to/ (visited on 07/21/2022).

[15] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. "Proving the TLS Handshake Secure (as it is)". In: *IACR Cryptol. ePrint Arch.* (2014), p. 182. URL: http://eprint.iacr.org/2014/182.

[16] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. "Off-the-record communication, or, why not to use PGP". In: *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*. Ed. by Vijay Atluri, Paul F. Syverson, and Sabrina De Capitani di Vimercati. ACM, 2004, pp. 77–84. DOI: 10.

1145/1029179.1029200. URL: https://doi.org/10.1145/1029179.1029200.

[17] Marcus Brinkmann, Christian Dresen, Robert Merget, Damian Poddebniak, Jens Müller, Juraj Somorovsky, Jörg Schwenk, and Sebastian Schinzel. "ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 4293–4310. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/brinkmann.

[18] Ran Canetti and Hugo Krawczyk. "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels". In: *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes in Computer Science. Springer, 2001, pp. 453–474. DOI: 10.1007/3-540-44987-6_28. URL: https://doi.org/10.1007/3-540-44987-6%5C_28.

[19] Will Cathcart. *This year we've all relied on messaging more than ever to keep up with our loved ones and get business done. We are proud that @WhatsApp is able to deliver roughly 100B messages every day and we're excited about the road ahead.* Oct. 29, 2020. URL: https://twitter.com/wcathcart/status/1321949078381453314.

[20] Yao Cheng, Lingyun Ying, Sibei Jiao, Purui Su, and Dengguo Feng. "Bind your phone number with caution: automated user profiling through address book matching on smartphone". In: *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*. Ed. by Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng. ACM, 2013, pp. 335–340. DOI: 10.1145/2484313.2484356. URL: https://doi.org/10.1145/2484313.2484356.

[21] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. "A Formal Security Analysis of the Signal Messaging Protocol". In: *IACR Cryptol. ePrint Arch.* (2016), p. 1013. URL: http://eprint.iacr.org/2016/1013.

[22] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. "On Post-Compromise Security". In: *IACR Cryptol. ePrint Arch.* (2016), p. 221. URL: http://eprint.iacr.org/2016/221.

[23] WinZip Computing. *AES Encryption Information: Encryption Specification AE-1 and AE-2*. en. 2009. URL: https://www.winzip.com/en/support/aes-encryption/ (visited on 07/04/2022).

[24] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G. Paterson. "The Security of ChaCha20-Poly1305 in the Multi-User Setting". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 1981–2003. DOI: 10.1145/3460120.3484814. URL: https://doi.org/10.1145/3460120.3484814.

[25] Jean Paul Degabriele and Kenneth G. Paterson. "Attacking the IPsec Standards in Encryption-only Configurations". In: *IACR Cryptol. ePrint Arch.* (2007), p. 125. URL: http://eprint.iacr.org/2007/125.

[26] Frank Denis. *Libsodium*. URL: https://github.com/jedisct1/libsodium.

[27] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. May 1996. DOI: 10.17487/RFC1951. URL: https://www.rfc-editor.org/info/rfc1951.

[28] Denis Diemert and Tibor Jager. "On the Tight Security of TLS 1.3: Theoretically Sound Cryptographic Parameters for Real-World Deployments". In: *J. Cryptol.* 34.3 (2021), p. 30. DOI: 10.1007/s00145-021-09388-x. URL: https://doi.org/10.1007/s00145-021-09388-x.

[29] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. "Fast Message Franking: From Invisible Salamanders to Encryptment". In: *IACR Cryptol. ePrint Arch.* (2019), p. 16. URL: https://eprint.iacr.org/2019/016.

[30] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. "Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework". In: *IACR Cryptol. ePrint Arch.* (2019), p. 436. URL: https://eprint.iacr.org/2019/436.

[31] Thai Duong and Juliano Rizzo. *Here Come The XOR Ninjas*. 2011. URL: http://www.hpcc.ecs.soton.ac.uk/dan/talks/bullrun/Beast.pdf.

[32] William F. Ehrsam, Carl H. W. Meyer, John L. Smith, and Walter L. Tuchman. *Message verification and transmission error detection by block chaining*. 1976.

[33] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. "Can Johnny Build a Protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols". In: (2017). URL: https://www.ndss-symposium.org/wp-content/uploads/2018/03/eurousec2017_16_Ermoshina_paper.pdf.

[34] Corin Faife. *Swiss Army drops WhatsApp for homegrown messaging service, citing privacy concerns*. en. Jan. 2022. URL: https://www.theverge.com/2022/1/7/22871881/swiss-army-whatsapp-messaging-threema-privacy-concerns-us-jurisdiction (visited on 05/16/2022).

[35] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. "How Secure is TextSecure?" In: *IACR Cryptol. ePrint Arch.* (2014), p. 904. URL: http://eprint.iacr.org/2014/904.

[36] Yoel Gluck, Neal Harris, and Angelo Prado. "BREACH: Reviving the CRIME Attack". In: (2013). URL: https://www.breachattack.com/resources/BREACH%5C%20-%5C%20SSL,%5C%20gone%5C%20in%5C%2030%5C%20seconds.pdf.

[37] Threema GmbH. *About - Threema*. May 2022. URL: https://web.archive.org/web/20220425195930/https://threema.ch/en/about.

[38] Threema GmbH. *Cryptography Whitepaper*. en. 2021. URL: https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf (visited on 05/19/2022).

[39] Threema GmbH. *How does Threema audit its code?* URL: https://threema.ch/en/faq/code_audit (visited on 07/13/2022).

[40] Threema GmbH. *Threema Changelog*. en. 2021. URL: https://threema.ch/en/versionhistory (visited on 07/11/2022).

[41] Threema GmbH. *Threema Goes Open Source, Welcomes New Partner*. Sept. 3, 2020. URL: https://threema.ch/en/blog/posts/open-source-and-new-partner (visited on 07/13/2022).

[42] Threema GmbH. *Threema Website, Section on Security*. en. 2022. URL: https://threema.ch/en/security (visited on 07/06/2022).

[43] Threema GmbH. *Threema's Success Story: From the Company's Founding to Today*. URL: https://threema.ch/press-files/1_press_info/press_threema_story_en.pdf (visited on 07/13/2022).

[44] Threema GmbH. *Website for Threema, OnPrem edition*. en. 2022. URL: https://threema.ch/en/onprem (visited on 07/06/2022).

[45] Threema GmbH. *Why Threema Instead of WhatsApp?* en. 2021. URL: https://threema.ch/en/work/blog/posts/why-threema-instead-of-whatsapp (visited on 05/17/2022).

[46] Threema GmbH. *Why Threema Instead of WhatsApp?* en. 2022. URL: https://threema.ch/en/why-threema (visited on 09/12/2022).

[47] Christoph G. Günther. "An Identity-Based Key-Exchange Protocol". In: *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*. Ed. by Jean-Jacques Quisquater and Joos Vandewalle. Vol. 434. Lecture Notes in Computer Science. Springer, 1989, pp. 29–37. DOI: 10.1007/3-540-46885-4_5. URL: https://doi.org/10.1007/3-540-46885-4%5C_5.

[48] 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: *IACR Cryptol. ePrint Arch.* (2020), p. 1119. URL: https://eprint.iacr.org/2020/1119.

[49] Alex Hern. “WhatsApp loses millions of users after terms update”. In: (Jan. 24, 2021). URL: https://www.theguardian.com/technology/2021/jan/24/whatsapp-loses-millions-of-users-after-terms-update (visited on 07/26/2022).

[50] Bridgefy Inc. *Offline Messages App & SDK — Bridgefy*. URL: https://bridgefy.me/ (visited on 07/28/2022).

[51] Jakob Jakobsen and Claudio Orlandi. “On the CCA (in)security of MTProto”. In: *IACR Cryptol. ePrint Arch.* (2015), p. 1177. URL: http://eprint.iacr.org/2015/1177.

[52] Simon Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. Oct. 2006. DOI: 10.17487/RFC4648. URL: https://www.rfc-editor.org/info/rfc4648.

[53] Burt Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. Mar. 1998. DOI: 10.17487/RFC2315. URL: https://www.rfc-editor.org/info/rfc2315.

[54] John Kelsey. “Compression and Information Leakage of Plaintext”. In: *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*. Ed. by Joan Daemen and Vincent Rijmen. Vol. 2365. Lecture Notes in Computer Science. Springer, 2002, pp. 263–276. DOI: 10.1007/3-540-45661-9_21. URL: https://doi.org/10.1007/3-540-45661-9%5C_21.

[55] 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*. IEEE, 2017, pp. 435–450. DOI: 10.1109/EuroSP.2017.38. URL: https://doi.org/10.1109/EuroSP.2017.38.

[56] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. DOI: 10.17487/RFC2104. URL: https://www.rfc-editor.org/info/rfc2104.

[57] Dr. Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://www.rfc-editor.org/info/rfc5869.

[58] Hugo Krawczyk. “Cryptographic Extraction and Key Derivation: The HKDF Scheme”. In: *IACR Cryptol. ePrint Arch.* (2010), p. 264. URL: http://eprint.iacr.org/2010/264.

[59] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis". In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. Lecture Notes in Computer Science. Springer, 2013, pp. 429–448. DOI: 10.1007/978-3-642-40041-4_24. URL: https://doi.org/10.1007/978-3-642-40041-4%5C_24.

[60] Brian A. LaMacchia, Kristin E. Lauter, and Anton Mityagin. "Stronger Security of Authenticated Key Exchange". In: *IACR Cryptol. ePrint Arch.* (2006), p. 73. URL: http://eprint.iacr.org/2006/073.

[61] Julia Len, Paul Grubbs, and Thomas Ristenpart. "Partitioning Oracle Attacks". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 195–212. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/len.

[62] Srikanth Lingala. *Zip4j*. en. URL: https://github.com/srikanth-lingala/zip4j.

[63] Google LLC. *HTTPS encryption on the web*. URL: https://transparencyreport.google.com/https/overview (visited on 07/26/2022).

[64] Zetetic LLC. *SQLCipher*. URL: https://www.zetetic.net/sqlcipher/ (visited on 08/07/2022).

[65] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. "A cross-protocol attack on the TLS protocol". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 62–72. DOI: 10.1145/2382196.2382206. URL: https://doi.org/10.1145/2382196.2382206.

[66] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001. URL: http://www.cacr.math.uwaterloo.ca/hac/.

[67] Telegram Messenger. *Telegram surpassed 500 million active users. 25 million new users joined in the last 72 hours: 38% came from Asia, 27% from Europe, 21% from Latin America and 8% from MENA*. URL: https://twitter.com/telegram/status/1349014065931284480?s=20%5C&t=rs-ga_MRH6p1yT2kZAyNaA (visited on 07/26/2022).

[68] Gabrielle De Micheli and Nadia Heninger. "Recovering cryptographic keys from partial information, by example". In: *IACR Cryptol. ePrint Arch.* (2020), p. 1506. URL: https://eprint.iacr.org/2020/1506.

[69] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC 8018. Jan. 2017. DOI: 10.17487/RFC8018. URL: https://www.rfc-editor.org/info/rfc8018.

[70] Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. "A Modular Security Analysis of the TLS Handshake Protocol". In: *IACR Cryptol. ePrint Arch.* (2008), p. 236. URL: http://eprint.iacr.org/2008/236.

[71] BBC News. 2020. URL: https://www.bbc.com/news/world-europe-51467536 (visited on 07/09/2022).

[72] Colin Percival and Simon Josefsson. *The scrypt Password-Based Key Derivation Function*. RFC 7914. Aug. 2016. DOI: 10.17487/RFC7914. URL: https://www.rfc-editor.org/info/rfc7914.

[73] Trevor Perrin. *The Noise Protocol Framework*. July 11, 2018. URL: http://www.noiseprotocol.org/noise.html (visited on 07/18/2022).

[74] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. "Secure off-the-record messaging". In: *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, WPES 2005, Alexandria, VA, USA, November 7, 2005*. Ed. by Vijay Atluri, Sabrina De Capitani di Vimercati, and Roger Dingledine. ACM, 2005, pp. 81–89. DOI: 10.1145/1102199.1102216. URL: https://doi.org/10.1145/1102199.1102216.

[75] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[76] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147. Apr. 2022. DOI: 10.17487/RFC9147. URL: https://www.rfc-editor.org/info/rfc9147.

[77] iiMedia Research. *Number of monthly active users (MAU) of the leading messaging apps in China as of December 2021 (in millions)*. Mar. 24, 2022. URL: https://www.statista.com/statistics/1062449/china-leading-messaging-apps-monthly-active-users/ (visited on 07/28/2022).

[78] Johannes Ritter. *Was Russland stört, überzeugt Olaf Scholz*. de. Aug. 17, 2022. URL: https://www.faz.net/aktuell/wirtschaft/unternehmen/threema-was-russland-stoert-ueberzeugt-olaf-scholz-18248712.html (visited on 09/12/2022).

[79] Juliano Rizzo and Thai Duong. *The CRIME Attack*. 2012. URL: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1d134dff_1_222.

[80] 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*. IEEE, 2018, pp. 415–429. DOI: 10.1109/EuroSP.2018.00036. URL: https://doi.org/10.1109/EuroSP.2018.00036.

[81] SaltyRTC. *SaltyRTC – End-to-End-Encrypted Signalling*. URL: https://saltyrtc.org/ (visited on 08/02/2022).

[82] Signal. *Facebook Messenger deploys Signal Protocol for end-to-end encryption*. July 8, 2016. URL: https://signal.org/blog/whatsapp-complete/ (visited on 07/28/2022).

[83] Signal. *Libsignal*. URL: https://github.com/signalapp/libsignal.

[84] Signal. *Technical information*. URL: https://signal.org/docs/ (visited on 07/30/2022).

[85] Signal. *Technology Deep Dive: Building a Faster ORAM Layer for Enclaves*. URL: https://signal.org/blog/building-faster-oram (visited on 09/11/2022).

[86] Signal. *Technology preview: Private contact discovery for Signal*. URL: https://signal.org/blog/private-contact-discovery/ (visited on 08/01/2022).

[87] Signal. *The Double Ratchet Algorithm*. URL: https://signal.org/docs/specifications/doubleratchet/ (visited on 08/03/2022).

[88] Signal. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*. URL: https://signal.org/docs/specifications/sesame/ (visited on 09/12/2022).

[89] Signal. *WhatsApp's Signal Protocol integration is now complete*. Apr. 4, 2016. URL: https://signal.org/blog/whatsapp-complete/ (visited on 07/28/2022).

[90] Soatok. *Threema: Three Strikes, You're Out*. Nov. 2021. URL: https://soatok.blog/2021/11/05/threema-three-strikes-youre-out (visited on 07/19/2022).

[91] We Are Social, Hootsuite, and DataReportal. *Most popular global mobile messenger apps as of January 2022, based on number of monthly active users (in millions)*. Jan. 26, 2022. URL: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/ (visited on 07/27/2022).

[92] SQLite. *SQLite Home Page*. URL: https://www.sqlite.org/index.html (visited on 08/07/2022).

[93] SRF. *Threema setzt sich durch - Schweizer Armee verbietet Whatsapp und Co.* de. Jan. 2022. URL: https://www.srf.ch/news/schweiz/threema-setzt-sich-durch-schweizer-armee-verbietet-whatsapp-und-co (visited on 05/17/2022).

[94] Hakan Tanriverdi. *Der Schlossherr*. Feb. 28, 2014. URL: https://www.freitag.de/autoren/der-freitag/der-schlossherr (visited on 07/13/2022).

[95] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. “SoK: Secure Messaging”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 232–249. DOI: 10.1109/SP.2015.22. URL: https://doi.org/10.1109/SP.2015.22.

[96] David Wagner and Bruce Schneier. “Analysis of the SSL 3.0 Protocol”. In: *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*. WOEC’96. Oakland, California: USENIX Association, 1996, p. 4.