Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Analysis of Telegram Client Security

Semester Project

Theo von Arx

December 21, 2021

Advisor: Prof. Dr. Kenny Paterson

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

Abstract

Telegram is a popular messenger with a large variety of different available clients. We study various third-party client implementations of Telegram's transport layer security protocol MTProto 2.0. First, we present a timing side-channel attack against MadelineProto. This allows an attacker to recover 29 bits of the plaintext with a probability of 2^{-14} in a clean oracle setting. The required assumptions on the knowledge of the server_salt and the session_id as well as practical limitations turn the attack into a mostly theoretical one. Second, we show a replay attack against the popular third-party client libraries Pyrogram, Telethon, and GramJS, which expose missing checks of the message ID. A PitM attacker can significantly change the view of the conversation for the vulnerable receiver. The replay attack is practicable and can be exploited by a malicious Wi-Fi access point. Our attacks show, that many third-party clients fail to securely implement MTProto 2.0 and underline the fragility of MTProto 2.0 implementations wich affects the entire Telegram ecosystem.

Contents

Chapter 1

Introduction

The importance of messenger services has grown lately. One of the most popular messenger services is Telegram which has reached 500M monthly users in January 2021 [22]. Telegram is not only popular for daily messaging, but also for sensitive messaging. Especially for activists of political protests, Telegram is a widely used messenger for group organization [4].

As opposed to other chat platforms such as WhatsApp, Telegram's official clients are open source, meaning that the source code is publicly available. Furthermore, Telegram allows and encourages developers to implement and deploy custom clients. Consequently, there is a flourishing ecosystem around Telegram. The number of available clients and libraries as well as their popularity is hard to estimate, but Telegram already lists 13 clients on the official webpage [29].

Another vital part in Telegram's ecosystem are bots which can interact with other services such as email, YouTube, payments, games. Therefore, Telegram can no more be seen as an isolated platform with the sole purpose of exchanging text messages. Rather, it develops towards a service that connects multiple services and offers users a simple way of interaction. Clearly, this further underlines the urge of a secure and correctly implemented protocol.

Telegram states that it has a "focus on security and speed". Presumably the latter led to the design of the custom protocol *MTProto 2.0* to secure the transport layer [24]. MTProto 2.0 acts as the equivalent to TLS' record protocol and is applied on top of TCP. It is therefore the only layer of security for the transport of messages between the server and the client.

In contrast to WhatsApp or Signal, Telegram is a cloud-based messenger: The default setting is to encrypt a message only between server and client, but to store all messages in plaintext on the server. This allows to download messages from multiple devices simultaneously without having to distribute

private keys. However, we stress that, except in Telegram's secret chats, there is no end-to-end encryption. In fact, users need trust Telegram for a responsible handling of their data. Nevertheless, we expect that MTProto 2.0 provides solid defense against an attacker on the wire in between the Telegram server and the client application.

In a recent paper, [3] showed a cryptographic proof of the security of MTProto 2.0. The proof is based on a slightly modified version of MTProto 2.0 and relies on security assumptions that were not studied before in literature. In addition to that, [3] shows four attacks against official Telegram clients and servers: A replay attack, an attack in the indistinguishability under chosen plaintext attack (IND-CPA) model, a person-in-the-middle (PitM) attack, and a timing side-channel attack enabled by the wrong use of encrypt-and-MAC to decrypt parts of arbitrary ciphertexts.

The vulnerability in official clients together with the diverse and complex network of available Telegram clients and libraries nourished our interest on the implementation of MTProto 2.0 out in the wild.

1.1 Contributions

As a first contribution, we present in Chapter 4 a timing side-channel attack against the PHP library MadelineProto, similar to the one described in [3]. The attack exploits the misuse of the encrypt-and-MAC scheme: MadelineProto does not check the integrity of the plaintext directly after decryption, but processes unauthenticated data. Depending on the input, the message processing time differs significantly. This allows an attacker to learn some parts of the plaintext. We evaluate the time difference, implement the attack in a synthetic setting and evaluate its limitations. We note that the attack is mostly of theoretical interest. For an arbitrary target block m_i, the attacker has to know the previous plaintext block m_{i-1} as well as m_1 which contains the 64-bit values server_salt and session_id. The attack in [3] that enables the attacker to learn m_1 is highly likely no more possible due to server side changes by Telegram. However, server_salt and session_id are not specified to be secret [25] and could potentially be revealed in future implementations.

The second contribution is the replay attack against two Python libraries (Pyrogram, Telethon) and a JavaScript library (GramJS) presented in Chapter 5. The vulnerability arises from the missing checks on the message ID which are designed to ensure that every message is processed exactly once. We present a practicable replay attack for exemplary clients in a real world setting using mitmproxy [6]. To avoid TCP errors and reconnections, we do not inject additional TCP packets but rather replace the TCP payload of an old packet with the one of a previous packet.

The vulnerable Python libraries are quite popular: They have 2.2K resp. 5.7K stars on GitHub and are, according to GitHub, used by 28.9K resp. 20.7K other projects. While GramJS is less popular (300 stars), it is used in one of the official clients (Telegram Web Z). Due to the use of WebSockets over TLS 1.3, Telegram Web Z is not vulnerable to the attack.

During our research, we analysed more clients and libraries for vulnerabilities as described in [3]. However, we did not find any further vulnerabilities in six clients[1].

The vulnerability of several official clients [3] together with the newly discovered vulnerabilities of multiple libraries leads us to a more fundamental question in Chapter 6: How can security be granted in a proliferating ecosystem? We highlight two design choices that complicate a secure implementation: The use of encrypt-and-MAC as well as the complex checks on the message ID. We propose how to significantly simplify those design choices and lower the hurdles for developers.

1.2 Disclosure

We informed the maintainers of the vulnerable libraries about our findings in the week of 20 November 2021. Besides explaining the discovered vulnerabilities, we also proposed fixes and highlight the missed checks.

MadelineProto's developer informed us that as of version 6.0.118 the timing side-channel vulnerability is fixed. Shortly after the release 6.0.118, MadelineProto ruled out the major update 7.0 which is declared mandatory for all MadelineProto users.

The maintainer of GramJS fixed the replay vulnerability as of version 1.11.1 without a further notice to its users. We further informed Telegram's security team about the vulnerability in GramJS which is used in the official client Telegram Web Z. We were informed, that Telegram Web Z uses the latest version of GramJS including the fix of the vulnerability. Telegram awarded a bug bounty for this replay attack against GramJS.

The maintainer of Telethon confirmed the receipt of the disclosure and assured to address the vulnerability as soon as possible. However, by the time of writing, they did neither contact us for further information nor did they fix the vulnerability.

We supported the maintainer of Pyrogram in fixing the vulnerability. However, by the time of writing, the patch has not been merged into the main project.

[1] The clients are: Kotatogram-Desktop, Nicegram, Telegram React, Telegram Web K, Telegram Web Z, Telegram-FOSS, and Unigram.

Chapter 2

Preliminaries

In this chapter, we present the notational conventions and formally introduce the standard definitions used in this work. We closely follow the conventions and definitions of [3].

2.1 Notational conventions

Let $\mathbb{N} = \{1, 2, ...\}$. For $i \in \mathbb{N}$ define $[i]$ as the set $\{1, ..., i\}$. For any string $x \in \{0,1\}^\star$, let $|x|$ denotes its bit-length, $x[i]$ denote its i-th bit for $0 \leq i < |x|$, and $x[a : b] = x[a]...x[b-1]$ for $0 \leq a < b \leq |x|$. Furthermore, let $x[a :] = x[a : |x|]$ and $x[: b] = x[0 : b]$. For two strings $x, y \in \{0,1\}^\star$, we define $x||y$ as their concatenation. In algorithms, let $x \leftarrow v$ denote the assignment of the value v to a variable x.

2.2 Standard definitions

2.2.1 Functional families

A family of functions F specifies algorithm F.Ev, a key set F.Keys, an input set F.In, and an output length F.ol $\in \mathbb{N}$. F.Ev takes a function key fk $\in$ F.Keys and an input $x \in$ F.In to return an output $y \in \{0,1\}^{\mathsf{F.ol}}$. We write $y \leftarrow \mathsf{F.Ev}(\mathsf{fk}, x)$. The key length of F is F.kl $\in \mathbb{N}$ if $\mathsf{F.keys} = \{0,1\}^{\mathsf{F.kl}}$.

2.2.2 Block ciphers

Let E be a function family. We say that E is a block cipher if $\mathsf{E.In} = \{0,1\}^{\mathsf{E.ol}}$, and if E specifies (in addition to E.Ev) an inverse algorithm E.Inv: $\{0,1\}^{\mathsf{E.ol}} \rightarrow$ E.In such that $\mathsf{E.Inv}(ek, \mathsf{E.Ev}(ek, x)) = x$ for all $ek \in$ E.Keys and all $x \in$ E.In. We refer to E.ol as the block length of E. Our pictures use E and E^{-1} as a shorthand for $\mathsf{E.Ev}(ek, \cdot)$ and $\mathsf{E.Inv}(\mathsf{ek}, \cdot)$, respectively.

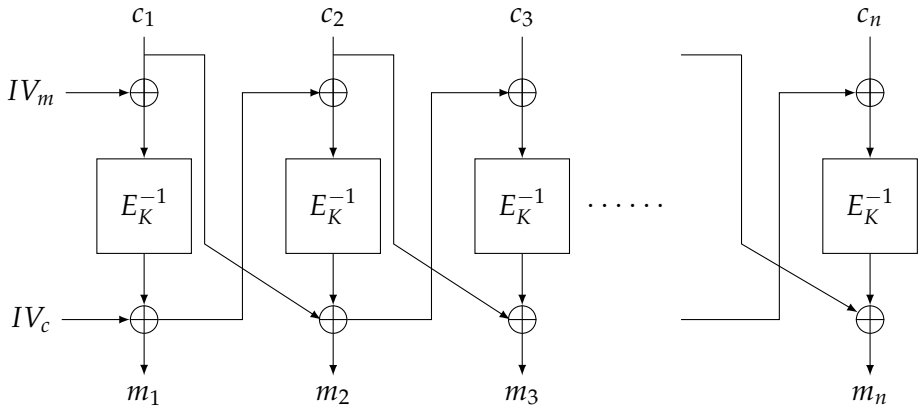

Figure 2.1: IGE decryption with $c_0 = IV_c$ and $m_0 = IV_m$.

2.2.3 IGE block cipher mode of operation

Let E be a block cipher. Let the Infinite Garble Extension (IGE) mode of operation be defined with encryption and decryption as in Algorithms 1 and 2, respectively. A visualization of the decryption can be seen in Fig. 2.1. The inputs to the algorithms are the secret key K, the initialization vectors (IVs) m_0 and c_0, and the plaintext m, respectively the ciphertext c.

We require that the plaintext m and the ciphertext c have a size divisible by the block length E.ol. For a bit string x we write $x = x_1||...||x_n$ such that $\forall i\, |x_i| = \mathsf{E.ol}$ to indicate the different blocks of x. Here, we implicitly set $n = \frac{|x|}{\mathsf{E.ol}}$.

Algorithm 1 IGE[E].Enc(K, m_0, c_0, m)

1: **for** $i = 1, ..., n$ **do**
2: $\quad c_i \leftarrow E_K(m_i \oplus c_{i-1}) \oplus m_{i-1}$
3: **return** $c_1||...||c_n$

Algorithm 2 IGE[E].Dec(K, m_0, c_0, c)

1: **for** $i = 1, ..., n$ **do**
2: $\quad m_i \leftarrow E_K^{-1}(c_i \oplus m_{i-1}) \oplus c_{i-1}$
3: **return** $m_1||...||m_n$

We further define the AES-256-IGE symmetric encryption: We let E be the Advanced Encryption Standard (AES) block cipher with block length 128 bit as defined in [9]. Then let AES-256-IGE describe the symmetric encryption and decryption as defined in Algorithms 1 and 2.

2.3 Attack scenario

For all of our attacks, we consider the active person-in-the-middle (PitM) scenario: The attacker is able to arbitrarily drop, reorder and inject TCP messages. The attacker only sees the encrypted MTProto 2.0 packets but can compose arbitrary (potentially invalid) ciphertexts and send those to the client.

This scenario is indeed realistic: First, TCP does not provide any security guarantees against malicious modification of TCP packets. Second, the requirements can easily be fulfilled in a real world setting, e.g., by seducing the victim to use a malicious Wi-Fi access point [19].

The attacker's goal is twofold: For the timing side-channel attack in Chapter 4, the attacker's goal is to learn some bits of the target plaintext. For the replay attack in Chapter 5, the attacker wants to alter the meaning of a conversation for at least one participant.

Additional assumptions on the attacker's knowledge are discussed in Chapters 4 and 5, respectively.

Chapter 3

Description of the symmetric part of MTProto 2.0

In this chapter, we describe Telegram's protocol MTProto 2.0. Because all of our attacks target messages that are not end-to-end encrypted but only encrypted between the server and the client, we focus on the symmetric part of MTProto 2.0 as described in [25]. The symmetric part of MTProto 2.0 is Telegram's equivalent to TLS' record protocol. We refer to [25] for a detailed description of the asymmetric part.

MTProto 2.0 aims to guarantee security for the transport layer. The reliable transport protocol TCP is specified for transport. While the unreliable transport protocol UDP is listed as another option, it is neither further specified nor does it seem to be used in any implementation [26]. We stress, that TCP (as well as UDP) cannot provide any security guarantees and therefore – without any further defense mechanisms such as TLS 1.3 or MTProto 2.0 – the payload can be arbitrarily manipulated. Furthermore, MTProto 2.0 specifies optional transport obfuscation which mainly aims to bypass censorship and does not increase the security on a cryptographic level. There are more options available for transport such as HTTPS or WebSockets over TLS which involve another layer of encryption. However, the security of MTProto 2.0 cannot rely on that.

In Section 3.1 we describe the symmetric encryption of MTProto 2.0. Section 3.2 discusses the checks that a client needs to perform upon receiving a new message.

3.1 Encryption

The payload p of a MTProto 2.0 message consists of the fields described in Table 3.1. The server_salt and the session_id in the first block are identifiers that are valid for multiple messages in a given time period respectively in the

Table 3.1: MTProto payload format [3]. The horizontal lines mark the boundaries of the 128 bit blocks.

Field	Type	Description
server_salt	int64	Server-generated random number valid in a given time period.
session_id	int64	Client-generated random identifier of a session under the same auth_key.
msg_id	int64	Time-dependent identifier of a message within a session. Approximately equal to Unix time multipliey by 2^{32}.
msg_seq_no	int32	Message sequence number.
msg_length	int32	Length of msg_data in bytes.
msg_data	bytes	Actual body of the message.
padding	bytes	12 - 1024B of random padding.

same session. The second block contains metadata with validity limited to the given message. Finally, the remaining blocks contain the actual message data and random padding with size between 12 B to 1004 B.

For a given plaintext m with blocks $m_1||m_2||...||m_n$, we denote the values obtained by parsing m into the fields defined in Table 3.1 as $\mathsf{server\_salt}(m_1)$, $\mathsf{session\_id}(m_1)$, $\mathsf{msg\_id}(m_2)$, $\mathsf{msg\_seq\_no}(m_2)$, $\mathsf{msg\_length}(m_2)$, as well as the remaining $\mathsf{msg\_data}(m_3||...||m_n)$ and $\mathsf{padding}(m_3||...||m_n)$, respectively.

After the asymmetric key establishment, the server and the client have established a common secret: the auth_key. It is used to derive the auth_key_id, the msg_key, and the final ciphertext c. The encryption is visualised in Fig. 3.1 and explained in the following lines. Let $x = 0$ for messages sent by the client and $x = 64$ for messages sent by the server.

The hash of auth_key is computed as

$$\mathsf{auth\_key\_id} := \mathsf{SHA\text{-}1}(\mathsf{auth\_key})[96 : 160] \tag{3.1}$$

and used to uniquely identify an authorization key for both the server and the client.

The message authentication code (MAC) of the payload p is computed as

$$\mathsf{msg\_key} := \mathsf{SHA\text{-}256}(\mathsf{auth\_key}[704 + x : 960 + x]||p)[64 : 192] \tag{3.2}$$

and allows the receiver to verify that the sent plaintext was not tampered with. The auth_key_id and the msg_key are sent in plain as external headers. Together with the auth_key, the msg_key is used as an input to the key

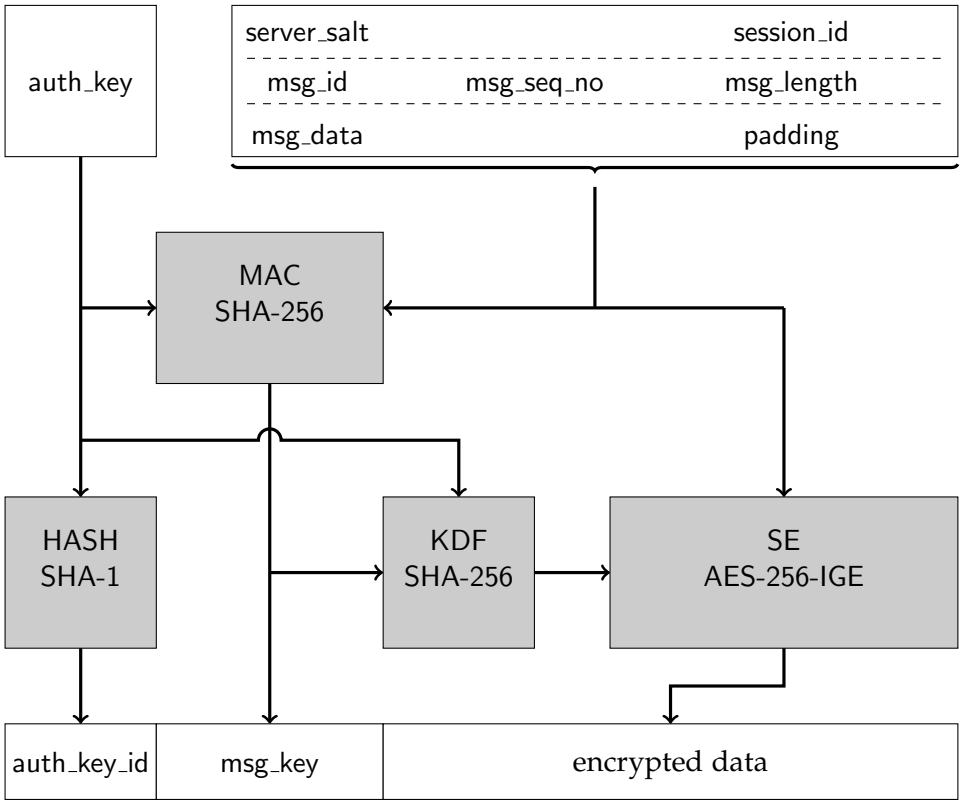

Figure 3.1: Overview of message processing in MTProto 2.0. Note that only parts of auth_key are used in MAC and KDF. This figure is a modified copy from [3].

derivation function (KDF) to compute the key and the IV for the symmetric encryption (SE):

$$A := \text{SHA-256}(\text{msg\_key}||\text{auth\_key}[x : 288 + x]) \tag{3.3}$$

$$B := \text{SHA-256}(\text{auth\_key}[320 + x : 608 + x]||\text{msg\_key}) \tag{3.4}$$

$$\text{key} := A[0 : 64]||B[64 : 192]||A[192 : 256] \tag{3.5}$$

$$\text{iv} := B[0 : 64]||A[64 : 192]||B[192 : 256] \tag{3.6}$$

Finally, the encrypted data c is computed using AES-256 in IGE mode:

$$c := \text{AES-256-IGE}(\text{key}, \text{iv}, p)\,. \tag{3.7}$$

3.2 Required checks on metadata

When receiving a message, the client has to check the validity of various metadata fields. In the following, we discuss the checks that are relevant with respect to our attacks.

The client has to perform the following checks [27]:

(C1) Directly after decryption, the client must check that msg_key be equal to the SHA-256 hash of the plaintext. To prevent timing side-channel attacks, this check has to be done independently of potential previous errors.

(C2) The client must check that msg_length be not bigger than the total size of the plaintext. The size of the padding is computed as the difference between the total size of the plaintext and msg_length and has to be within the range from 12 B to 1024 B.

The msg_length has to be divisible by four and non-negative.

(C3) The client must check that the session_id in the decrypted message be equal to the one of an active session.

(C4) The client must check the validity of msg_id:

(C4.1) The client must check that msg_id be odd.

(C4.2) The client must store the msg_id of the last N received messages. Here, the value of N is not specified[1] [27]. The client must check that an incoming msg_id be not smaller than all N stored message IDs and that msg_id be not already stored.

(C4.3) Furthermore, the client must ignore msg_id values which are more than 30 seconds in the future or more than 300 seconds in the past.

In case of a failure, the client has to discard the message and should close and re-establish the TCP connection to the server.

[1] Official implementation use different values: Telegram Desktop [28] uses $N = 400$, TDLib [21] uses $N = 2000$.

Chapter 4

Timing side-channel attack

In this chapter, we describe a timing side-channel attack similar to the one in [3], Section IV.

In Section 4.1 we recapitulate the general attack idea which is then applied to the PHP library MadelineProto in Section 4.2. We describe the missing checks in MadelineProto in Section 4.2.1 and show in Section 4.2.2 that there is a statistically measurable timing difference between a length check failure and a MAC failure. In Section 4.2.3 we present an attack in a clean oracle setting and discuss its limitations and the success probability in Section 4.2.4. Finally, we list other inspected clients that are not vulnerable to the timing side-channel attack in Section 4.3.

4.1 Attack idea

For a given ciphertext $c_1||...||c_n$ corresponding to a (unknown) plaintext $m_1||...||m_n$ and an arbitrary target block m_i with $2 \leq i \leq n$, the attacker's goal is to learn some bits of m_i. For a successful exploit of the timing side-channel attack, we need additional assumptions on the knowledge of the attacker:

The attack described in Section F of [3] allows an attacker to learn the server_salt and the session_id contained in m_1. Even though this is hard and likely no more possible due to server-side changes by Telegram, there might be a successful attack against m_1 in the future. Furthermore, server_salt as well as session_id are not specified to be secret values and may be revealed in plain in future implementations of MTProto 2.0. Hence, we assume that the adversary knows m_1.

We further assume that for the target block m_i the attacker knows the previous plaintext block m_{i-1}. To motivate this, assume that m_{i-1} = "Today's

password" and m_i = "is SECRET". In general, this assumption is considered to be realistic.

The attacker then forges a ciphertext $c_1||c^\star$ with $c^\star = c_i \oplus m_{i-1} \oplus m_1$. The decryption of this ciphertext is $m_1||m^\star$ s.t.

$$m^\star = E_K^{-1}(c^\star \oplus m_1) \oplus c_1 \tag{4.1}$$
$$= E_K^{-1}(c_i \oplus m_{i-1}) \oplus c_1 \tag{4.2}$$
$$= m_i \oplus c_{i-1} \oplus c_1\,. \tag{4.3}$$

Consequently, if there is a side-channel that allows an attacker to learn some bits of the second block, then the attacker can learn the corresponding bits of m_i by using Eq. (4.3).

4.2 MadelineProto

MadelineProto is a PHP library that implements a MTProto 2.0 client. The library is officially listed on Telegram's webpage as an exemplary implementation [26]. The library can be used for multiple purposes including voice over IP (VoIP) webradio, downloading files and controlling a server [14].

4.2.1 Message processing

When receiving a packet, MadelineProto processes it as follows (c.f. Listing 4.1):

1. Check whether received auth_key_id matches the computed one.
2. Reduce the ciphertext s.t. its size is a multiple of 16 B.
3. Decrypt the ciphertext.
4. Check the session_id according to (C3).
5. Check the msg_id mostly following (C4), see Listing A.1 for more details.
6. Check the validity of the packet length according to (C2). Here, the padding size is computed as the difference between the length of the ciphertext and msg_length.
7. Check the integrity of the decrypted data (C1) by comparing the received msg_key with the computed one.

The reduction of the ciphertext to a multiple of 16 B by removing at most 15 B is a leftover from the implementation of a previous version of MTProto[1].

[1]This leads to an attack in the indistinguishability under chosen ciphertext attack (IND-CCA) model in which the attacker can forge valid ciphertexts! While theoretically interesting, the given scenario does not allow a practical attack.

Listing 4.1: Message processing in MadelineProto [12]. Modified for readability. $seq_no and $message_data_length corresponds to msg_seq_no and msg_length respectively.

```
public function readMessage(): \Generator {
  # [...]
  $auth_key_id = yield $buffer->bufferRead(8);
  # [...]
  if ($auth_key_id === $shared->getTempAuthKey()->getID()) {
    # [...]
    $encrypted_data = yield $buffer->bufferRead($payload_length -
        24);
    $protocol_padding = \strlen($encrypted_data) % 16;
    if ($protocol_padding) {
        $encrypted_data = \substr($encrypted_data, 0, -
            $protocol_padding);
    }
    $decrypted_data = Crypt::igeDecrypt($encrypted_data, $aes_key
        , $aes_iv);
    # [...]
    $message_id = \substr($decrypted_data, 16, 8);
    $connection->msgIdHandler->checkMessageId($message_id, ['
        outgoing' => false, 'container' => false]);
    $seq_no = \unpack('V', \substr($decrypted_data, 24, 4))[1];

    $message_data_length = \unpack('V', \substr($decrypted_data,
        28, 4))[1];
    if ($message_data_length > \strlen($decrypted_data)) {
        throw new \SecurityException('message_data_length is too
            big');
    }
    if (\strlen($decrypted_data)-32-$message_data_length < 12) {
        throw new \SecurityException('padding is too small');
    }
    if (\strlen($decrypted_data)-32-$message_data_length > 1024){
        throw new \SecurityException('padding is too big');
    }
    if ($message_data_length < 0) {
        throw new \SecurityException('message_data_length not
            positive');
    }
    if ($message_data_length % 4 != 0) {
        throw new \SecurityException('message_data_length not
            divisible by 4');
    }
    $message_data = \substr($decrypted_data, 32,
        $message_data_length);
    if ($message_key != \substr(\hash('sha256', \substr($shared->
        getTempAuthKey()->getAuthKey(), 96, 32).$decrypted_data,
        true), 8, 16)) {
        throw new \SecurityException('msg_key mismatch');
    }
  }
  # [...]
  return true;
}
```

The restriction of ciphertexts being a multiple of 16 B arises from the used block cipher with block size 16 B. However, instead of this reduction, the client could directly reject a malformed message since it was tampered[2].

The operations between decryption and the integrity check are done on unauthenticated data. Thus, an attacker can supply a forged ciphertext with a valid auth_key_id and session_id which will be processed until a failure occurs. Consequently, if the attacker can differentiate between different failure types, the attacker can learn some bits of the targeted message.

As we show in the next section, the differences between a failure in a msg_id, msg_length and a msg_key are indeed measurable. Not only does the processing time differ for different failure types, but the TCP connection is re-established as well. However failure does not force a re-establishment of the MTProto session and the server_salt and the session_id stay the same. Hence, all sent forged ciphertexts during a single attack against the target m_i will be decrypted and further processed. This makes it easier for an attacker to measure the processing time for a forged ciphertext as we discuss below.

4.2.2 Practical timing experiments

By measuring the response time, an attacker can estimate the time it takes to process a message. To verify the existence of the timing difference between the msg_id, the msg_length and the msg_key check, we measured the message processing time in a simulated environment (see Listing B.1): We modified the program to be synchronous and created a clean interface, i.e., the messages are not sent over the network but passed as arguments. We conducted the experiments on an Intel i7-6500U processor running Linux-libre 5.10.72 at 2.5 GHz with turboboost and hyper threading both disabled.

The results are visualized in Fig. 4.1. The time difference between msg_length and msg_id check is due to the additional SHA-256 computation in case of a passing msg_length check. The size of this time difference linearly depends on the payload size which is passed to SHA-256. Even though the msg_id checks are evaluated first, the processing time is larger in case of a msg_id failure. The reason is, that msg_id failures are logged which involves comparably slow operations.

At the beginning, the attacker can forge a ciphertext consisting of only $c_1||c^\star$ which will not pass the msg_length check as there is no padding. Therefore, the attacker does not have to distinguish between msg_key and msg_id failure, but only between msg_length and msg_id failure.

There is one significant limitation: we do not assume that the time difference between different length checks (e.g., that the padding size is bigger

[2]This is a special case where the early abortion and the skipped computation of msg_key do not allow an attacker to learn new information about the plaintext.

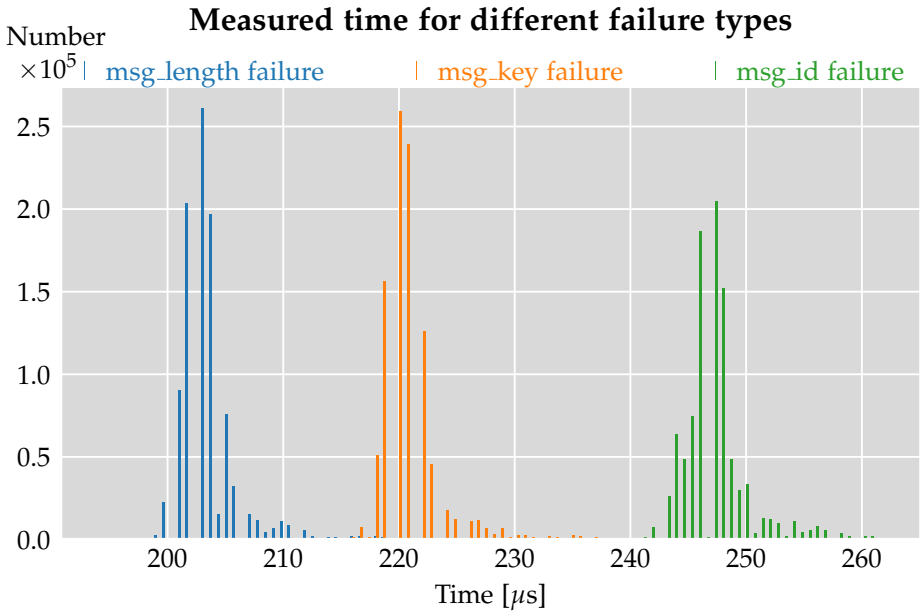

Figure 4.1: Timing difference between msg_id, msg_length and msg_key checks, measured under ideal conditions with a modified version of [12]. Packet size: 2048 B.

than 12 B and that the padding size is smaller than 1024 B) is measurable. The reason is that between two such checks no computationally intensive operations are involved.

As shown in Table 4.1, the means of the timing difference between failures of the msg_id, length and the integrity check are in the range of 15 µs to 17 µs each. This is enough for a remote attacker to detect the failure type over the network [2].

Table 4.1: Statistics of processing time.

Error type	# samples	Mean [µs]	Median [µs]	St. dev [µs]
msg_length	996057	204.010400	203.132629	4.313188
msg_key	993465	221.408019	221.014023	4.290814
msg_id	967952	247.270518	247.001648	2.834639

4.2.3 Attack in a clean oracle model

As a proof of concept, we show the attack in a clean oracle setting, i.e., we assume that the attacker can distinguish between the three different error types with a success probability of 100 %. The pseudocode of the attack can

Algorithm 3 Timing side-channel attack against MadelineProto.

procedure $\mathcal{A}^{\mathsf{O}}(i, m_1, m_{i-1}, \mathsf{payload})$
  $\mathsf{auth\_key\_id} \leftarrow \mathsf{payload}[0:8]$; $\mathsf{msg\_key} \leftarrow \mathsf{payload}[8:24]$
  $c_1, \ldots, c_n \leftarrow \mathsf{payload}[24:]$ ▷ $\forall j : ||c_j|| = 16\,\mathrm{B}$
  $c^\star \leftarrow c_i \oplus m_{i-1} \oplus m_1$
  $\tilde{c} \leftarrow \mathsf{auth\_key\_id}|\mathsf{msg\_key}|c_1|c^\star$; $l \leftarrow 0$

  ▷ Increment l linearly until msg_key failure or reaching of the limit
  repeat
    $c' \leftarrow \tilde{c}|\mathsf{randomBytes}(l)$
    if $\mathsf{len}(c') > 2^{22}$ **then**
      Log("The 10 MSBs of $m^\star$ or the 2 LSBs of $m^\star$ are nonzero")
      return $\perp$
    ans $\leftarrow \mathsf{O}(c')$
    if ans = MSG_ID_FAIL **then**
      Log("msg_id ($m^\star$) is too big or its LSB is nonzero"); **return** $\perp$
    if ans = LENGTH_FAIL **then**
      $l \leftarrow l + 1008$
  until ans = INTEGRITY_FAIL
  $l \leftarrow l - 1008$ ▷ Set l to the max value s.t. the padding is too small

  lo $\leftarrow 0$; hi $\leftarrow 1008/16$ ▷ Binary search to get lowest msg_key failure
  while lo $\neq$ hi **do**
    mid $= \lceil \frac{\text{lo}+\text{hi}}{2} \rceil$
    $c' = \tilde{c}|\mathsf{randomBytes}(16 \cdot \text{mid})$
    ans $\leftarrow \mathsf{O}(c')$
    if ans = LENGTH_FAIL **then**
      lo $\leftarrow$ mid
    else if ans = INTEGRITY_FAIL **then**
      hi $\leftarrow$ mid-1
  $l \leftarrow l + 16 \cdot \text{mid}$
  if ans = LENGTH_FAIL **then**
    $l \leftarrow l + 16$ ▷ Add one block to get an integrity check failure

  $c' \leftarrow \tilde{c}|\mathsf{randomBytes}(l + 1008)$
  ans $\leftarrow \mathsf{O}(c')$
  if ans $=$ LENGTH_FAIL **then**
    $l^\star \leftarrow l - 17$
  else
    $l^\star \leftarrow l - 12$
  $m^\star \leftarrow l^\star \oplus (c_{i-1} \oplus c_1)$[96:128] ▷ Compute the guess
  return $m^\star$ ▷ Only guess length field

be seen in Algorithm 3. We successfully implemented the attack in Python.

By varying the number l of the random bytes, the attacker can trigger different errors for a forged payload $\mathsf{auth\_key\_id}||\mathsf{msg\_id}||c_1||c^\star||\mathsf{randomBytes}(l)$. This will then be decrypted to $m_1||m^\star||m'_3||...||m'_n$ where m_1 contains the valid server_salt and session_id, $m^\star$ is as in Eq. (4.3) and $m'_3, ..., m'_n$ are garbled blocks with $n = \lfloor \frac{l}{16} \rfloor + 2$. The key point is, that the MadelineProto client interprets $m^\star$ as msg_id, msg_seq_no, and msg_length. The same holds for the blocks $m'_3, ..., m'_n$ that are interpreted as msg_data and padding. The MadelineProto client computes the size of the padding as $|\mathsf{msg\_data}(m'_3||...||m'_n)| + |\mathsf{padding}(m'_3||...||m'_n)| - \mathsf{msg\_length}(m^\star) = l - \mathsf{msg\_length}(m^\star)$.

The idea is to find the smallest value for l, such that the msg_key check fails. This will give the attacker an information about the value $\mathsf{msg\_length}(m^\star)$ and allows to infer the corresponding bits of the target block m_i using Eq. (4.3).

Since MadelineProto reduces the size of the ciphertext to be a multiple of 16 B, an attacker increases l by multiples of 16 B. Between a msg_length check failure due to a too small padding and a msg_length due to a too big padding is a window of 1012 B. This allows the attacker to first increase l linearly by $1008\,\mathrm{B} = 16 \cdot 63\,\mathrm{B}$ while being sure that the window of a msg_key failure is not missed. We stress that a binary search is not possible due to the indistinguishability of a too small and a too big padding. In a binary search, an attacker would not know when to increase and when to decrease l.

Once there is a msg_id check failure for a given l, we know that the padding size is between 12 B to 1024 B, hence the following inequation holds:

$$12 \leq l - \mathsf{msg\_length}(m^\star) \leq 1024\,. \tag{4.4}$$

Now, given that there is a lower ($l - 1008$) and an upper limit (l) for the minimal size that triggers a msg_key check failure, the attacker can now use a binary search to find the smallest value l^- which is a multiple of 16 B and triggers a msg_key check failure. We then know that

$$12 \leq l^- - \mathsf{msg\_length}(m^\star) \quad < 12 + 16 \tag{4.5}$$

$$0 \leq l^- - \mathsf{msg\_length}(m^\star) - 12 < 16\,. \tag{4.6}$$

At this point, the attacker can correctly guess all but the four least significant bits (LSBs) of $\mathsf{msg\_length}(m^\star)$. However, there is a trick to learn the fourth LSB: The attacker queries the oracle with $l^- + 1008$ random bytes. If the answer is a msg_key check failure, we have

$$l^- + 1008 - \mathsf{msg\_length}(m^\star) \leq 1024 \tag{4.7}$$

and hence

$$12 \leq l^- - \mathsf{msg\_length}(m^\star) \leq 16 \tag{4.8}$$

$$0 \leq l^- - \mathsf{msg\_length}(m^\star) - 12 \leq 4\,. \tag{4.9}$$

Otherwise, in case of a msg_length check error, we get

$$l^- + 1008 - \mathsf{msg\_length}(m^\star) > 1024 \tag{4.10}$$

and hence

$$16 < l^- - \mathsf{msg\_length}(m^\star) < 12 + 16 \tag{4.11}$$

$$4 < l^- - \mathsf{msg\_length}(m^\star) - 16 < 12\,. \tag{4.12}$$

In other words: The value of the fourth LSB of $l^- - \mathsf{msg\_length}(m^\star) - x$ with either $x = 12$ or $x = 16$ is fixed, and the attacker successfully learns it. Since the attacker knows l^- and x, the attacker can transform this knowledge to the 29 most significant bits (MSBs) of $\mathsf{msg\_length}(m^\star)$ and finally to the corresponding bits of m_i.

The number of needed queries is the sum of queries in the linear phase and in the binary search phase. It amounts to approximately $\frac{\mathsf{msg\_length}(m^\star)}{1008} + \log_2(63) \approx \frac{\mathsf{msg\_length}(m^\star)}{1008} + 6$. The measured number of queries for different values of $\mathsf{msg\_length}(m^\star)$ is visualized in Fig. 4.2. Clearly, it matches the expected characteristic: For $\mathsf{msg\_length}(m^\star) \leq 2^{10}$ the number of queries is dominated by the binary search. For larger $\mathsf{msg\_length}(m^\star)$, the number of queries grows linearly because it is dominated by the linear search. Note that the number of queries of the algorithm only depends on the value of $\mathsf{msg\_length}(m^\star)$.

On average over all possible $m^\star$, s.t. $\mathsf{msg\_length}(m^\star) \leq 2^{22}$, $m^\star$ is 2^{21}. Hence, the attacker needs $\approx 2^{11}$ queries on average in the clean oracle setting.

4.2.4 Limitations

Several conditions need to hold for a successful attack. There are two types of limitations: First, the attacker needs to have the possibility to trigger a msg_key check failure. This is not the case, if the message is already rejected due to an invalid msg_id ($m^\star$) or if msg_length ($m^\star$) is not divisible by four. Then there is a practical limitation: If $\mathsf{msg\_length}(m^\star) > 2^{22}$, then the attack does not finish in reasonable time. Furthermore, the attacker needs to send a message in the order of 2^{22} B which can trigger OS exception due to too large memory allocation and lead to a crash of the client.

To summarize, the attack is successful in the following cases:

1. $\mathsf{msg\_id}(m^\star)$ is smaller than the maximum limit of approximately 2^{63}.
2. $\mathsf{msg\_id}(m^\star)$ has odd parity.

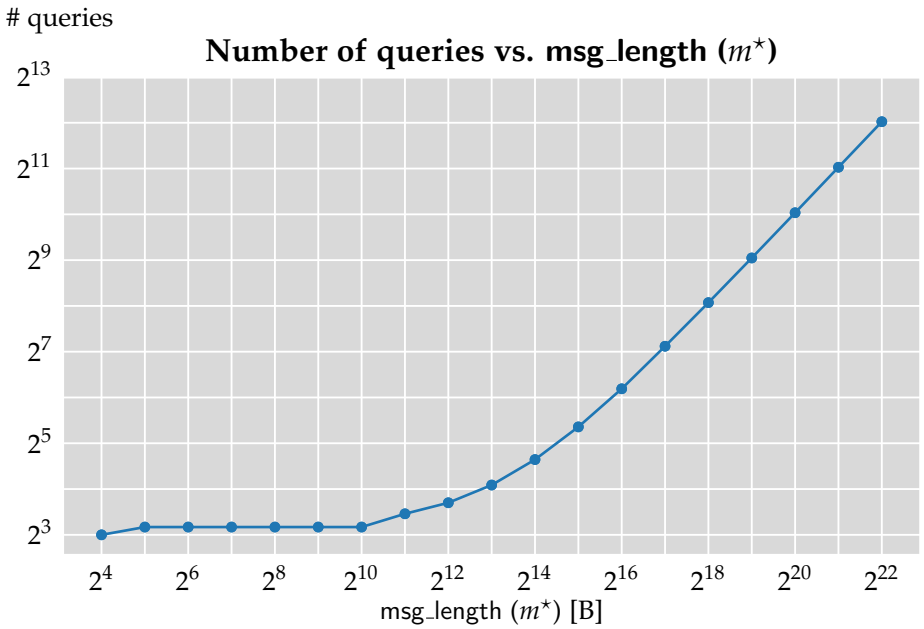

Figure 4.2: Number of queries vs. msg_length ($m^\star$)

3. msg_length($m^\star$) is smaller than 2^{22}.
4. msg_length($m^\star$) is divisible by four.

If $i = 2$, then $m^\star = m_2$ and all conditions are fulfilled. Hence, an attacker can find the true message length up to the last three LSBs with a success probability of 100 % in a clean oracle setting. The length of a message is no more obfuscated.

If $i \neq 2$, this reduces the success probability to $\approx 2^{-1} \cdot 2^{-1} \cdot 2^{-10} \cdot 2^{-2} = 2^{-14}$ assuming a clean distinction between the three failure conditions. Note that the attack can be carried out for every block of a message with independent success probability. Thus, an attacker can recover 29 bits from one in every 2^{14} blocks on average.

In a real-world setting, we have to take into account, that the correctness of the oracle queries is probabilistic. We assume that the processing times for a msg_length and a msg_key check failure are random variables that follow normal distributions, i.e., $X_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$ with mean μ_j and variance σ_j^2 as in Table 4.1 for $j = 1, 2$, respectively. Let $D = \frac{\mu_1 + \mu_2}{2}$, then for a measured time x the attacker guesses

$$x = \begin{cases} X_1, & \text{if } x \leq D \\ X_2, & \text{if } x > D \end{cases} \tag{4.13}$$

or in terms of the oracle answer:

$$\text{ans} = \begin{cases} \text{msg\_length error}, & \text{if } x \leq D \\ \text{msg\_key error}, & \text{if } x > D\,. \end{cases} \tag{4.14}$$

The error for a wrong guess in every case can be computed as

$$P(x > D | x = X_1) = Q\left(\frac{D - \mu_1}{\sigma_1}\right) \tag{4.15}$$

$$P(x \leq D | x = X_2) = 1 - Q\left(\frac{D - \mu_2}{\sigma_2}\right), \tag{4.16}$$

where $Q(\cdot)$ denotes the tail distribution function of the standard normal distribution.

The error probabilities can be reduced by repeating every query t times and averaging over all measurements. Then we have the random variables $S_j = \frac{1}{t}\sum_{k=1^n} X_j^{(k)}$ where $X_j^{(k)}$ denotes the k-th query. Consequently, $S_j \sim \mathcal{N}(\mu_j, \frac{\sigma_j^2}{t})$ and therefore the error probabilities for a guess on an averaged processing time s reduce to

$$p_1^{(t)} = P(s > D | s = S_1) = Q\left(\frac{(D - \mu_1)\sqrt{t}}{\sigma_1}\right) \tag{4.17}$$

$$p_2^{(t)} = P(s \leq D | s = S_2) = 1 - Q\left(\frac{(D - \mu_2)\sqrt{t}}{\sigma_2}\right). \tag{4.18}$$

Let q be the number of guesses that the attacker needs in a clean oracle setting. Let $p_- = \min(1 - p_1^{(t)}, 1 - p_2^{(t)})$ be the minimal success probability for a single query that is repeated t times. Then the probability that all guesses are correct when each packet is resent t times is $p > p_-^q$. In a real-world setting, we have to multiply the success probabilities above with p.

When requiring $p > \alpha$, we can compute t using $p_- > \sqrt[q]{p}$ for a fixed q. Table 4.2 shows the value of m for exemplary values of msg_length ($m^\star$) respectively q with fixed $p = 0.5$.

4.3 Analysis of other forks and variants of official clients

The fact that three official Telegram clients were vulnerable to a timing side-channel attack [3], drew our attention to have a look at third party clients and libraries. Tables 4.3 and 4.4 show the analysed clients and libraries. Apart from MadelineProto, none of the examined implementations of MTProto 2.0 is vulnerable to a timing side-channel attack.

Table 4.2: Minimal values of t for fixed $\alpha = 0.5$ for various q.

msg_length$(m^\star)$	$q \approx$	$p_- >$	t
2^{14}	2^4	0.9781414659457593	1
2^{15}	2^5	0.9789033166108467	2
2^{19}	2^9	0.9997613555358401	3
2^{22}	2^{12}	0.9998334326346597	4

Table 4.3: Analysed clients. "★" denotes the number of stars on GitHub and is a rough indicator for the popularity. Clients marked with "*" are officially supported by Telegram.

Name	★	Upstream	Used library
Kotatogram-Desktop	422	Telegram Desktop	-
Nicegram	281	Telegram-iOS	-
Telegram React	1.7K	-	TDLib
Telegram Web K*	314	-	-
Telegram Web Z*	357	-	GramJS
Telegram-FOSS	1.4K	Telegram (Android)	-
Unigram	1.8K	-	TDLib

Table 4.4: Analysed libraries. "★" denotes the number of stars on GitHub and is a rough indicator of the popularity. "Used by" indicates the number of projects on GitHub which depend on the library. For TDLib this number is not available. The only officially supported library (marked with "*") is TDLib.

Name	★	Used by	Language
GramJS	282	440	JavaScript
MadelineProto	140	1.7K	PHP
Pyrogram	2.2K	29.3K	Python
TDLib*	4.2k	-	C++
Telethon	5.7K	20.9K	Python

Chapter 5

Replay attack

In this chapter, we discuss the replay attack vulnerability found in three third-party libraries. Section 5.1 discusses the missing checks that enable the replay attack. Then, in Section 5.2 we show how to launch the attack in a real world setting. Finally, in Section 5.3 we discuss that reordering attacks are not prevented by MTProto 2.0 and highlight the potential implications of a reordering attack.

5.1 Description of the vulnerability

In a *replay attack*, an attacker can resend certain messages and let the receiver believe that both messages originate from the sender. While the attacker cannot read the messages, it is a simple yet powerful attack to modify conversations.

To prevent against replay attacks in MTProto 2.0, receivers must perform the check (C4.2) discussed in Section 3.2. Namely, the receiver has to ensure that no two messages with the same msg_id are processed. During our analysis we discovered that the following third-party libraries miss the relevant checks: The Python libraries Pyrogram [7] and Telethon [10], as well as the JavaScript library GramJS [15]. The relevant code snippets can be seen in Listings 5.1 to 5.3. Pyrogram only checks that the msg_id is odd. Telethon and GramJS do not even check that. Furthermore, Telethon and GramJS both include comments that hint at the missing checks.

While Pyrogram and Telethon appear to be independent projects, GramJS' core is completely based on Telethon. The relevant lines in the code only differ in syntax. Table 4.4 shows the popularity of the libraries on GitHub. Even though GramJS is not that popular, it is used by one of Telegram's official web clients, Telegram Web Z [30].

An attacker can perform a replay attack against a client using one of those

Listing 5.1: `mtproto.py`. Message processing in Pyrogram [8]. Modified for readability.

```
def unpack(b: BytesIO, session_id: bytes, auth_key: bytes,
    auth_key_id: bytes) -> Message:
    # [...]
    data = BytesIO(aes.ige256_decrypt(b.read(), aes_key, aes_iv))
    # [...]
    message = Message.read(data)
    # [...]
    # https://core.telegram.org/mtproto/security_guidelines#
        checking-msg-id
    assert message.msg_id % 2 != 0
    return message
```

Listing 5.2: `mtprotostate.py`. Message processing in Telethon [11]. Modified for readability.

```
def decrypt_message_data(self, body):
    # TODO Check salt, session_id and sequence_number
    # [...]
    body = AES.decrypt_ige(body[24:], aes_key, aes_iv)
    # [...]
    reader = BinaryReader(body)
    reader.read_long()  # remote_salt
    if reader.read_long() != self.id:
        raise SecurityError('Wrong session ID')
    remote_msg_id = reader.read_long()
    remote_sequence = reader.read_int()
    reader.read_int()  # msg_len
    obj = reader.tgread_object()
    return TLMessage(remote_msg_id, remote_sequence, obj)
```

Listing 5.3: `MTProtoState.ts`. Message processing in GramJS [16]. Modified for readability.

```
async decryptMessageData(body: Buffer) {
    // [...]
    // TODO Check salt,sessionId, and sequenceNumber
    const keyId = helpers.readBigIntFromBuffer(body.slice(0, 8));
    // [...]
    body = new IGE(key, iv).decryptIge(body.slice(24));
    // [...]
    const reader = new BinaryReader(body);
    reader.readLong(); // removeSalt
    const serverId = reader.readLong();
    if (serverId !== this.id) {
        // throw new SecurityError('Wrong session ID');
    }
    const remoteMsgId = reader.readLong();
    const remoteSequence = reader.readInt();
    reader.readInt(); // msgLen
    // [...]
    const obj = reader.tgReadObject();
    return new TLMessage(remoteMsgId, remoteSequence, obj);
}
```

libraries: The attacker records an encrypted message from the server to the client and replays it at a later point in time. Both messages will appear valid to the victim.

5.2 Attack implementation

To experimentally verify the presence of the vulnerability, we implemented clients using the libraries above (c.f. Listings C.1 to C.3). To exploit the attack we configure the clients to route all traffic to a local proxy server. For the proxy server, we use mitmproxy [6] with the add-on shown in Listing C.4 to easily record and replay specific TCP packets. Instead of injecting additional TCP packets we replace the content of every second TCP packet containing a text message with the previous one. This facilitates the attack since we neither have to update all TCP sequence numbers, nor do we need to handle additional acknowledgement packets.

Figure 5.1 illustrates the attack: The sender Alice sends two different messages which arrive correctly at the Telegram server. The Telegram server decrypts, re-encrypts and forwards the messages to the proxy to which Bob is connected to. The malicious proxy is run by the attacker Mallory. Mallory records the first message (✉) and replaces the TCP payload of the second packet (✉) with the recorded message. Hence, Bob receives two times the same message.

Figure 5.1: Overview of the replay attack. The symbols "✉" and "✉" denote two different messages.

For a sample run, the views of a sender Alice and a receiver Bob negotiating a meeting time differ as shown in Table 5.1. While Alice is late herself according to her second message, Bob experiences an increasingly aggressive tone of voice and does not learn that Alice is late. Clearly, the attacker Mallory is able to alter the meaning of the conversation.

The attack is successful against all of three tested libraries. However, the attack does not apply to Telegram Web Z. The reason is, that Telegram Web Z uses WebSockets over TLS 1.3 for the transport layer. Hence, MTProto 2.0 is run on top of TLS 1.3. While the implementation of MTProto 2.0 cannot prevent replay attacks, this is – luckily – done by TLS 1.3.

Table 5.1: Example: different views of the sender Alice and the receiver Bob.

Alice	Bob
Hurry up!	Hurry up!
Sorry, I'm late myself.	Hurry up!
Where are you?	Where are you?
I'm still at home.	Where are you?

We stress that the attack only applies for messages sent from the Telegram server to a vulnerable client. The attacker cannot replay messages sent by a vulnerable client since the Telegram servers correctly defend against replay attacks. Furthermore, the attack may not apply to all clients using one of the vulnerable libraries: Besides the use of WebSockets over TLS 1.3, the attack may also be mitigated on the application layer.

The attack is even more powerful, when a vulnerable library is used to implement the control of, e.g., a server. Telminal [17] is such a program based on Telethon. Instead of sending commands over a Secure Shell (SSH), the user sends the commands over Telegram messages. If the user for example sends the command to remove the first entry of a database, the attacker can flush the entire database by repeatedly replaying the command.

Other interesting settings include message forwarding from Telegram to WhatsApp and vice versa [1] (based on GramJS), cryptocurrency trading [5] (based on GramJS), as well as bots that broadcast a received message [18] (based on Pyrogram).

5.3 A note on reordering attacks

In a *reordering attack*, the attacker aims to modify the order of the received messages. The attack is similar to the replay attack: Record and hold back the first message, let the second message pass and finally release the withheld message. Again, the meaning of a conversation can be significantly altered.

We tested this attack against Pyrogram, Telethon, and GramJS. All of them are vulnerable to the reordering attack as well. The used add-on for mitmproxy is shown in Listing C.5.

A reordering attack is generally considered a serious weakness and similar protocols such as TLS or the Signal messaging protocol successfully defend against this type of attack. However, this vulnerability is *not* a violation of the security guarantees specified by [27]: Unless $N = 1$, the check (C4.2) on the msg_id does not force message IDs to be strictly monotonously increasing and therefore allow that messages are processed out of order [3]. Official Telegram clients only prevent reordering attacks at the application level [3].

Chapter 6

Discussion

In this work, we present several attacks against the implementation of MTProto 2.0 in third-party Telegram clients. The main contribution are two types of attack: A timing side-channel attack and a replay attack. In this chapter we discuss our results. We first show the strengths and limitation of the two attacks. Then we contextualize to the broader question of establishing security in an ecosystem with many actors and developers in Section 6.1. Finally, we propose further research problems in Section 6.2.

In the timing side-channel attack against MadelineProto we show how the attacker can learn 29 bits of an arbitrary plaintext block under certain conditions. We show how to practically implement the attack and give an algorithm with the asymptotically optimal number of queries. This attack is mostly of theoretical interest due to hardly achievable assumptions of the knowledge of the server_salt and the session_id in m_1. The large average number of queries (and therefore the large runtime as well as the large size of the forged packets) of the attack reduces the practicability further. However, the power of attacks grow with time: New potential discoveries can make unrealistic attacks possible in the future. Furthermore, the values server_salt and session_id are not specified to be secret [25]. Hence, the two values may be revealed in a future implementation.

On the other hand, the replay attack against Pyrogram, Telethon, and GramJS is practicable and can be exploited by running a malicious Wi-Fi access point. The attack is also powerful and lets an attacker significantly alter the view of a conversation for the participant that uses a vulnerable client. In addition to obtaining additional copies of received messages, the receiver can also not detect the deletion of messages. Yet, not all clients that use a vulnerable library can be attacked by a replaying messages. Besides additional defense on the transport layer security, such as the use of TLS 1.3 in Telegram Web Z, clients may also defend on the application layer: Messages come with a time stamp set by the Telegram server. Because this time stamp

is contained in the ciphertext, it cannot be tampered with. Some clients then use this time stamp to store the message in a data structure, that allows only one message with a given time stamp. Hence, a replayed message is not displayed twice. Nevertheless, the security of a library should not depend on requirements that are not specified.

6.1 Security in a proliferating ecosystem

The found vulnerabilities in third-party clients and libraries together with the ones discussed in [3] suggest a more far-reaching question: how can security be guaranteed in an environment of various independent implementations?

The origin seems to be two-fold: On the one hand, Telegram is developer-friendly and encourages developers to implement their own clients and bots [23]. This openness attracts developers without a cryptographic background. On the other hand, the custom protocol MTProto 2.0 does not lower the hurdles enough for a secure implementation.

The first problem is partially addressed by introducing the cryptographic library TDLib in 2018 [20]. While the strong recommendation of TDLib is reasonable, not all developers will use it. Although TDLib can be integrated with various programming languages including Python, the popularity of the Python libraries Pyrogram and Telethon indicate that developers tend to use a library written in the same language as the rest of the code. An officially supported and thoroughly tested Python library could partially mitigate the problem. Otherwise, the specifications and security guidelines need to be more precise and understandable for non-cryptographers. One possibility is to provide pseudocode for the correct implementation of MTProto 2.0.

The second problem is not addressed yet. In contrary, design choices such as the relatively complex checks on the message ID could be simplified: Requiring the message IDs to increment by one for every new message is easier to implement and even improves the security as reordering attacks and deletions are directly prevented.

Similarly, it is the design choice of encrypt-and-MAC that opens the door for bad implementations of the decryption and potential timing side-channels. The usage of Encrypt-than-MAC would significantly lower the potential for a timing side-channel vulnerability because the MAC is computed on the ciphertext. Here, the natural order for a receiver is to first check the MAC and only then decrypt the ciphertext. Consequently, the probability that the receiver applies further checks on unauthenticated data is much lower.

More fundamentally, the justification of the use of a custom protocol is questionable. Telegram mentions reliability for weak mobile connection and speed for cryptographically processing of large files as the reason for a MTProto 2.0 [24]. However, even the official client Telegram Web Z uses TLS 1.3 on top of MTProto 2.0. While the best security of both protocols may be achieved, the performance is limited by the slower protocol. In contrast to MTProto 2.0, TLS 1.3 is well-studied in literature and many state-of-the-art libraries for various languages exist.

However, one argument for MTProto 2.0 lies in the root of trust: When designing and deploying their own protocol, Telegram can carefully choose the root of trust and does not have to rely on the trust of dozens to hundreds of root certficate authorities (CAs) as TLS 1.3 does. But, this argument is drastically weakened by the reliance on secure transport of the client software itself to the user which will be most likely secured by TLS.

6.2 Future work

In our analysis, we focused on the symmetric part of the encryption of cloud chats. With its large ecosystem and the broad variety of applications, there is a lot of interesting work open. Future work includes the research on private end-to-end encrypted chats, bots, and control messages.

Another pressing topic is the one discussed above: How to systematically improve the security of Telegram clients? Designing a test suite or a verification tool are just two possibilities to address this question.

Finally, Telegram's reasoning for the decision to use their custom protocol should be examined: extensive measurements of the reliability and performance of MTProto 2.0 would help to clear up the question of advantages of MTProto 2.0 over TLS 1.3.

Bibliography

[1] Siddiqui Affan and Rashid Pathiyil. WhatsGram. Yet another user-bot for Whatsapp. https://github.com/WhatsGram/WhatsGram, April 2021. [Online; accessed 29-November-2021].

[2] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[3] Martin Albrecht, Lenka Marekova, Kenny Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. In *IEEE S&P 2022*, July 2021.

[4] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: the case of Hong Kong. *CoRR*, abs/2105.14869, 2021.

[5] Steven Almeroth. telegram-signals. https://github.com/stav/telegram-signals, November 2021. [Online; accessed 29-November-2021].

[6] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 7.0].

[7] Dan "delivrance". Pyrogram. Telegram MTProto API framework for Python. https://github.com/pyrogram/pyrogram, 2017–. [Online; accessed 26-November-2021].

[8] Dan "delivrance". mtproto.py. https://github.com/pyrogram/pyrogram/blob/34b6002c689273d7233ca1a0976da009a3aafe09/pyrogram/crypto/mtproto.py#L52, June 2021. [Online; accessed 7-December-2021].

[9] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced Encryption Standard (AES), 2001-11-26 2001.

[10] "LonamiWebs". Telethon. https://github.com/LonamiWebs/Telethon, 2016–. [Online; accessed 26-November-2021].

[11] "LonamiWebs". mtprotostate.py. https://github.com/LonamiWebs/Telethon/blob/f9643bf7376a5953da2050a5361c9b465f7ee7d9/telethon/network/mtprotostate.py#L133, February 2021. [Online; accessed 7-December-2021].

[12] Daniil Gentili. Madelineproto – ReadLoop.php. https://github.com/danog/MadelineProto/blob/1389b24751fa3f06ba783888c4eee7b1c42dea84/src/danog/MadelineProto/Loop/Connection/ReadLoop.php#L106, October 2020. [Online; accessed 18-November-2021].

[13] Daniil Gentili. Madelineproto – MsgIdHandler64.php. https://github.com/danog/MadelineProto/blob/1389b24751fa3f06ba783888c4eee7b1c42dea84/src/danog/MadelineProto/MTProtoSession/MsgIdHandler/MsgIdHandler64.php#L50, February 2021. [Online; accessed 18-November-2021].

[14] Daniil Gentili. Madelineproto – Readme.md. https://github.com/danog/MadelineProto/blob/5969ebe783692c8c7aa1b38d380489954a540f66/README.md, December 2021. [Online; accessed 25-December-2021].

[15] GramJS. Gramjs. NodeJS/Browser MTProto API Telegram client library. https://github.com/gram-js/gramjs, 2019–. [Online; accessed 24-November-2021].

[16] GramJS. MTProtoState.ts. https://github.com/gram-js/gramjs/blob/7474e57e1f5e392ce9750871db1ca78bf3fcc453/gramjs/network/MTProtoState.ts#L190, September 2021. [Online; accessed 8-December-2021].

[17] Mohammadreza Jafari. telminal. A terminal in Telegram! https://github.com/fristhon/telminal, October 2021. [Online; accessed 29-November-2021].

[18] Fayas Noushad, Nikhil Eashy, and "MrBotDeveloper". BroadcastBot. https://github.com/nacbots/BroadcastBot, September 2021. [Online; accessed 29-November-2021].

[19] Nissy Sombatruang, M. Angela Sasse, and Michelle Baddeley. Why do people use unsecure public Wi-Fi? An investigation of behaviour and factors driving decisions. In *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust*, STAST '16, page 61–72, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Telegram. TDLib -– Build your own Telegram. https://telegram.org/blog/tdlib, January 2018. [Online; accessed 2-December-2021].

[21] Telegram. Telegram Desktop, mtproto_received_ids_manager.h. https://github.com/telegramdesktop/tdesktop/blob/9308615361c77d983bac458e48196646b0660c3b/Telegram/SourceFiles/mtproto/details/mtproto_received_ids_manager.h#L15, November 2019. [Online; accessed 25-November-2021].

[22] Telegram. 500 milion users. https://t.me/durov/147, 2021. [Online; accessed 24-September-2021].

[23] Telegram. API. https://web.archive.org/web/20211127010953/https://core.telegram.org/api, November 2021. [Online; accessed 2-December-2021].

[24] Telegram. FAQ for the technically inclined. https://web.archive.org/web/20211115225615/https://core.telegram.org/techfaq, December 2021. [Online; accessed 7-December-2021].

[25] Telegram. Mobile protocol: Detailed description. https://web.archive.org/web/20211016013637/https://core.telegram.org/mtproto/description/, October 2021. [Online; accessed 25-November-2021].

[26] Telegram. MTProto mobile protocol. https://web.archive.org/web/20211213201047/https://core.telegram.org/mtproto, December 2021. [Online; accessed 15-December-2021].

[27] Telegram. Security guidelines for client developers. https://web.archive.org/web/20211028151304/https://core.telegram.org/mtproto/security_guidelines, October 2021. [Online; accessed 25-November-2021].

[28] Telegram. TDLib, mtproto_received_ids_manager.h. https://github.com/tdlib/td/blob/2725f7c58a2e1c33f25b8306eeeb6ca8b2a41247/td/mtproto/AuthData.h#L287, August 2021. [Online; accessed 25-November-2021].

[29] Telegram. Telegram applications. https://web.archive.org/web/20211201125716/https://telegram.org/apps, December 2021. [Online; accessed 2-December-2021].

[30] Alexander Zinchuk. Telegram Web Z. https://github.com/Ajaxy/telegram-tt, April 2021. [Online; accessed 26-November-2021].

Appendix A

MadelineProto code

Listing A.1: msg_id check in MadelineProto [13]. Modified for readability.

```
<?php
    public function checkMessageId($newMessageId, array $aargs):
        void
    {
        $newMessageId = \is_integer($newMessageId) ?
            $newMessageId : Tools::unpackSignedLong($newMessageId
            );
        $minMessageId = (\time() + $this->session->time_delta -
            300) << 32;
        if ($newMessageId < $minMessageId) {
            $this->session->API->logger->logger('Given message id
                 ('.$newMessageId.') is too old compared to the
                min value ('.$minMessageId.').', \danog\
                MadelineProto\Logger::WARNING);
        }
        $maxMessageId = (\time() + $this->session->time_delta +
            30) << 32;
        if ($newMessageId > $maxMessageId) {
            throw new \danog\MadelineProto\Exception('Given
                message id ('.$newMessageId.') is too new
                compared to the max value ('.$maxMessageId.').
                Consider syncing your date.');
        }
        if ($aargs['outgoing']) {
    # ...
        } else {
            if (!($newMessageId % 2)) {
                throw new \danog\MadelineProto\Exception('message
                     id mod 4 != 1 or 3');
            }
            $key = $this->maxIncomingId;
            if ($aargs['container']) {
        # ...
            } else {
                if ($newMessageId <= $key) {
```

```
24                      $this->session->API->logger->logger('Given
                            message id ('.$newMessageId.') is lower
                            than or equal to the current limit ('.
                            $key.'). Consider syncing your date.', \
                            danog\MadelineProto\Logger::NOTICE);
25                  }
26              }
27              $this->cleanup(true);
28              $this->maxIncomingId = $newMessageId;
29          }
30      }
```

Appendix B

Implementation of timing side-channel attack

B.1 Timing experiment code

Listing B.1: madeline_timing.php.

```php
<?php

require_once("vendor/autoload.php");

use danog\MadelineProto\Stream\Common\BufferedRawStream;
use danog\MadelineProto\Connection;
use danog\MadelineProto\DataCenterConnection;
use danog\MadelineProto\MTProto\AuthKey;
use danog\MadelineProto\MTProto\PermAuthKey;
use danog\MadelineProto\MTProto\TempAuthKey;
use danog\MadelineProto\MTProtoTools\Crypt;
use danog\MadelineProto\Tools;
use danog\MadelineProto;
use danog\MadelineProto\Settings;

class Buffer
{
  public ?string $data = Null;
  public int $offset = 0;
  public function bufferWrite(string $data)
  {
    $this->data .= $data;
  }

  public function bufferRead(int $length)
  {
    $result = \substr($this->data, $this->offset, $length);
    $this->offset += $length;
    return $result;
  }
}
```

```
32
33 # Taken from https://github.com/danog/MadelineProto/blob/1389
       b24751fa3f06ba783888c4eee7b1c42dea84/src/danog/MadelineProto/
       MTProtoSession/MsgIdHandler/MsgIdHandler64.php#L50,
34 # Logging slightly changed by Theo von Arx
35 class msgIDHandler
36 {
37
38
39   /**
40    * Maximum incoming ID.
41    *
42    * @var int
43    */
44   private $maxIncomingId = 0;
45   /**
46    * Maximum outgoing ID.
47    *
48    * @var int
49    */
50   private $maxOutgoingId = 0;
51   /**
52    * Check validity of given message ID.
53    *
54    * @param string $newMessageId New message ID
55    * @param array  $aargs        Params
56    *
57    * @return void
58    */
59   public function checkMessageId($newMessageId, array $aargs):
       void
60   {
61     $newMessageId = \is_integer($newMessageId) ? $newMessageId :
         Tools::unpackSignedLong($newMessageId);
62     /* $minMessageId = (\time() + $this->session->time_delta -
         300) << 32; */ // Old
63     $minMessageId = (\time() - 300) << 32; // Theo von Arx
64     if ($newMessageId < $minMessageId) {
65       $this->logger->logger('Given message id ('.$newMessageId.')
             is too old compared to the min value ('.$minMessageId.
           ').', \danog\MadelineProto\Logger::WARNING);
66     }
67     /* $maxMessageId = (\time() + $this->session->time_delta +
         30) << 32; */
68     $maxMessageId = (\time() + 30) << 32; // Theo von Arx
69     if ($newMessageId > $maxMessageId) {
70       throw new \danog\MadelineProto\Exception('Given message id
           ('.$newMessageId.') is too new compared to the max
           value ('.$maxMessageId.'). Consider syncing your date.'
           );
71     }
72     if ($aargs['outgoing']) {
73       if ($newMessageId % 4) {
```

```
74          throw new \danog\MadelineProto\Exception('Given message
                id ('.$newMessageId.') is not divisible by 4.
                Consider syncing your date.');
75        }
76        if ($newMessageId <= $this->maxOutgoingId) {
77          throw new \danog\MadelineProto\Exception('Given message
                id ('.$newMessageId.') is lower than or equal to the
                current limit ('.$this->maxOutgoingId.'). Consider
                syncing your date.');
78        }
79        $this->cleanup(false);
80        $this->maxOutgoingId = $newMessageId;
81      } else {
82        if (!($newMessageId % 2)) {
83          throw new \danog\MadelineProto\Exception('message id mod
                4 != 1 or 3');
84        }
85        $key = $this->maxIncomingId;
86        if ($aargs['container']) {
87          if ($newMessageId >= $key) {
88            $this->logger->logger('Given message id ('.
                  $newMessageId.') is bigger than or equal to the
                  current limit ('.$key.'). Consider syncing your
                  date.', \danog\MadelineProto\Logger::NOTICE);
89          }
90        } else {
91          if ($newMessageId <= $key) {
92            $this->logger->logger('Given message id ('.
                  $newMessageId.') is lower than or equal to the
                  current limit ('.$key.'). Consider syncing your
                  date.', \danog\MadelineProto\Logger::NOTICE);
93          }
94        }
95        /* $this->cleanup(true); */ // Theo von Arx
96          $this->maxIncomingId = $newMessageId;
97      }
98    }
99
100   /**
101    * Generate message ID.
102    *
103    * @return string
104    */
105   public function generateMessageId($offset=0): string
106   {
107     # see https://github.com/LonamiWebs/Telethon/blob/4
          b16183d2bbe80cbf4dabdb266a8015c5bf975cc/telethon/network/
          mtprotostate.py#L172
108     $messageId = ((\time()) << 32) + 3;
109     $nanoseconds = exec('date +%N')<<2;
110     $messageId = $messageId | $nanoseconds;
111     $messageId += $offset;
112     /* if ($messageId <= $this->maxOutgoingId) { */
113       /* $messageId = $this->maxOutgoingId + 4; */
```

```
114     /* } */
115     return (\danog\MadelineProto\Magic::$BIG_ENDIAN ? \strrev(\
            pack('q', $messageId)) : \pack('q', $messageId));
116   }
117 }
118
119 # Mostly copied from https://github.com/danog/MadelineProto/blob
      /6bf767f61435e11b624c7e68f09d7fb04d4d84e1/src/danog/
      MadelineProto/Loop/Connection/WriteLoop.php#L310-L318
120 # Adapted sha256 computation to match direction server -> client.
121 function prepareMessage($shared, $connection, $message_data,
      $message_data_length, $message_id, $seq_no) {
122
123   $plaintext = $shared->getTempAuthKey()->getServerSalt().
        $connection->session_id.$message_id.\pack('VV', $seq_no,
        $message_data_length).$message_data;
124   $padding = \danog\MadelineProto\Tools::posmod(-\strlen(
        $plaintext), 16);
125   if ($padding < 12) {
126     $padding += 16;
127   }
128   $padding = \danog\MadelineProto\Tools::random($padding);
129
130   $message_key = \substr(\hash('sha256', \substr($shared->
        getTempAuthKey()->getAuthKey(), 96, 32).$plaintext.$padding
        , true), 8, 16);
131   list($aes_key, $aes_iv) = Crypt::aesCalculate($message_key,
        $shared->getTempAuthKey()->getAuthKey(), false);
132   $message = $shared->getTempAuthKey()->getID().$message_key.
        Crypt::igeEncrypt($plaintext.$padding, $aes_key, $aes_iv);
133
134   return $message;
135 }
136
137 /*
138  * $buffer:
139  *   auth_key_id    : 8
140  *   message_key    : 16
141  *   payload (inc padding)
142  * */
143 function processMessage($buffer, $payload_length, $shared,
      $connection)
144 {
145
146   /* if ($payload_length === 4) { */
147   /*     $payload = \danog\MadelineProto\Tools::unpackSignedInt(
        yield $buffer->bufferRead(4)); */
148   /*     $API->logger->logger("Received {$payload} from DC ".
        $datacenter, Logger::ULTRA_VERBOSE); */
149   /*     return $payload; */
150   /* } */
151   /* $connection->reading(true); */ // Theo von Arx
152     try {
153       $auth_key_id = $buffer->bufferRead(8);
```

```
154       // [Theo von Arx:] This wont be received
155       if ($auth_key_id === "\0\0\0\0\0\0\0\0") {
156         /*      $message_id = yield $buffer->bufferRead(8); */
157         /*      //if (!\in_array($message_id, [\1, \0])) { */
158         /*      $connection->msgIdHandler->checkMessageId(
              $message_id, ['outgoing' => false, 'container' =>
              false]); */
159         /*      //} */
160         /*      $message_length = \unpack('V', yield $buffer->
              bufferRead(4))[1]; */
161         /*      $message_data = yield $buffer->bufferRead(
              $message_length); */
162         /*      $left = $payload_length - $message_length - 4 - 8
               - 8; */
163         /*      if ($left) { */
164         /*          $API->logger->logger('Padded unencrypted
              message', Logger::ULTRA_VERBOSE); */
165         /*          if ($left < (-$message_length & 15)) { */
166         /*              $API->logger->logger('Protocol padded
              unencrypted message', Logger::ULTRA_VERBOSE); */
167         /*          } */
168         /*          yield $buffer->bufferRead($left); */
169         /*      } */
170       } elseif ($auth_key_id === $shared->getTempAuthKey()->getID
            ()) {
171         $message_key = $buffer->bufferRead(16);
172         list($aes_key, $aes_iv) = Crypt::aesCalculate(
              $message_key, $shared->getTempAuthKey()->getAuthKey()
              , false);
173         $encrypted_data = $buffer->bufferRead($payload_length -
              24);
174         $protocol_padding = \strlen($encrypted_data) % 16;
175         if ($protocol_padding) {
176           $encrypted_data = \substr($encrypted_data, 0, -
                $protocol_padding);
177         }
178         $decrypted_data = Crypt::igeDecrypt($encrypted_data,
              $aes_key, $aes_iv);
179       /*
180         $server_salt = substr($decrypted_data, 0, 8);
181         if ($server_salt != $shared->getTempAuthKey()->
              getServerSalt()) {
182         $API->logger->logger('WARNING: Server salt mismatch (my
              server salt '.$shared->getTempAuthKey()->
              getServerSalt().' is not equal to server server salt
              '.$server_salt.').', Logger::WARNING);
183         }
184       */
185         $session_id = \substr($decrypted_data, 8, 8);
186         if ($session_id != $connection->session_id) {
187           $API->logger->logger("Session ID mismatch", Logger::
                FATAL_ERROR);
188           $connection->resetSession();
189           throw new NothingInTheSocketException();
```

```
190         }
191         $message_id = \substr($decrypted_data, 16, 8);
192         $connection->msgIdHandler->checkMessageId($message_id, ['
                outgoing' => false, 'container' => false]);
193         $seq_no = \unpack('V', \substr($decrypted_data, 24, 4))
                [1];
194         $message_data_length = \unpack('V', \substr(
                $decrypted_data, 28, 4))[1];
195         if ($message_data_length > \strlen($decrypted_data)) {
196           throw new \danog\MadelineProto\SecurityException('
                message_data_length is too big');
197         }
198         if (\strlen($decrypted_data) - 32 - $message_data_length
                < 12) {
199           throw new \danog\MadelineProto\SecurityException('
                padding is too small');
200         }
201         if (\strlen($decrypted_data) - 32 - $message_data_length
                > 1024) {
202           throw new \danog\MadelineProto\SecurityException('
                padding is too big');
203         }
204         if ($message_data_length < 0) {
205           throw new \danog\MadelineProto\SecurityException('
                message_data_length not positive');
206         }
207         if ($message_data_length % 4 != 0) {
208           throw new \danog\MadelineProto\SecurityException('
                message_data_length not divisible by 4');
209         }
210         $message_data = \substr($decrypted_data, 32,
                $message_data_length);
211
212         if ($message_key != \substr(\hash('sha256', \substr(
                $shared->getTempAuthKey()->getAuthKey(), 96, 32).
                $decrypted_data, true), 8, 16)) {
213           throw new \danog\MadelineProto\SecurityException('
                msg_key mismatch');
214         }
215       } else {
216         $API->logger->logger('Got unknown auth_key id', Logger::
                ERROR);
217         return -404;
218       }
219       /* [$deserialized, $sideEffects] = $API->getTL()->
              deserialize($message_data, ['type' => '', 'connection'
              => $connection]); */
220       /* if (isset($API->referenceDatabase)) { */
221       /*     $API->referenceDatabase->reset(); */
222       /* } */
223       /* $message = new IncomingMessage($deserialized,
              $message_id); */
224       /* if (isset($seq_no)) { */
225       /*     $message->setSeqNo($seq_no); */
```

```
226       /* } */
227       /* if ($sideEffects) { */
228       /*     $message->setSideEffects($sideEffects); */
229       /* } */
230       /* $connection->new_incoming[$message_id] = $connection->
              incoming_messages[$message_id] = $message; */
231       /* $API->logger->logger('Received payload from DC '.
              $datacenter, Logger::ULTRA_VERBOSE); */
232     } finally {
233       /* $connection->reading(false); */
234     }
235   return true;
236 }
237
238 $shared = new \danog\MadelineProto\DataCenterConnection();
239 $connection = new \danog\MadelineProto\Connection();
240 $connection->session_id = random_bytes(8);
241 $connection->msgIdHandler = new msgIdHandler();
242 $settings_logger = new \danog\MadelineProto\Settings\Logger();
243 $connection->msgIdHandler->logger = new \danog\MadelineProto\
      Logger($settings_logger);
244
245 $key = random_bytes(256);
246 $salt = random_bytes(8);
247
248 $PermAuthKey = new \danog\MadelineProto\MTProto\PermAuthKey();
249 $PermAuthKey->setAuthKey($key);
250 $PermAuthKey->setServerSalt($salt);
251
252 $TempAuthKey = new \danog\MadelineProto\MTProto\TempAuthKey();
253 $TempAuthKey->setAuthKey($key);
254 $TempAuthKey->setServerSalt($salt);
255 $shared->setAuthKey($PermAuthKey, false);
256 $shared->setAuthKey($TempAuthKey, true);
257
258
259
260 $filename = "data/" . date("Y-m-d-H-i-s", time() + 1 * 60 * 60) .
      ".csv";
261 echo $filename . "\n";
262
263 $n_runs = 10**0;
264 $n_packets = 10**7;
265
266 // save the column headers
267 $data = array(array('N runs', 'message data length', 'Time msg_id
      check', 'Time length check', 'Time padding check'));
268
269   $message_data_length = 2**11;
270 for($i_packet = 0; $i_packet < $n_packets; ++$i_packet) {
271   $message_data = random_bytes($message_data_length);
272   $message_id = $connection->msgIdHandler->generateMessageId(1);
273   $seq_no = random_int(0, 2**32 - 1);
274
```

```
275   # Measure the msg_id check
276   $buffers = array();
277   $encrypted_data = prepareMessage($shared, $connection,
          $message_data, $message_data_length, $message_id, $seq_no);
278   $payload_length = \strlen($encrypted_data);
279   for($i_run = 0; $i_run < $n_runs; ++$i_run) {
280     $buffer = new Buffer();
281     $buffer->bufferWrite($encrypted_data);
282     array_push($buffers, $buffer);
283
284   }
285
286   $start_msg_id = microtime(true);
287   for($i_run = 0; $i_run < $n_runs; ++$i_run) {
288     try {
289       processMessage($buffers[$i_run], $payload_length, $shared,
              $connection);
290     } catch (\danog\MadelineProto\Exception $e) {
291       /* print_r($e->getMessage()); */
292     }
293   }
294   $end_msg_id = microtime(true);
295
296   $connection->msgIdHandler = new msgIdHandler();
297   $connection->msgIdHandler->logger = new \danog\MadelineProto\
          Logger($settings_logger);
298
299   /* print_r("\nMSG_ID: DONE\n"); */
300
301   /* # Measure length checks */
302   $buffers = array();
303   $payload_length = \strlen($encrypted_data);
304   for($i_run = 0; $i_run < $n_runs; ++$i_run) {
305     $message_id = $connection->msgIdHandler->generateMessageId();
306     $encrypted_data = prepareMessage($shared, $connection,
            $message_data, $message_data_length + 1024, $message_id,
            $seq_no);
307     $buffer = new Buffer();
308     $buffer->bufferWrite($encrypted_data);
309     array_push($buffers, $buffer);
310
311   }
312
313   $start_length = microtime(true);
314   for($i_run = 0; $i_run < $n_runs; ++$i_run) {
315     try {
316       processMessage($buffers[$i_run], $payload_length, $shared,
              $connection);
317     } catch (\danog\MadelineProto\SecurityException $e) {
318       /* assert($e->getMessage() == "padding is too big"); */
319       /* print_r($e->getMessage()); */
320     }
321   }
322   $end_length = microtime(true);
```

```
323
324   $connection->msgIdHandler = new msgIdHandler();
325   $connection->msgIdHandler->logger = new \danog\MadelineProto\
          Logger($settings_logger);
326
327   /* # Measure padding check */
328   $buffers_padding = array();
329   for($i_run = 0; $i_run < $n_runs; ++$i_run) {
330     $message_id = $connection->msgIdHandler->generateMessageId();
331     $encrypted_data = prepareMessage($shared, $connection,
            $message_data, $message_data_length, $message_id, $seq_no
            );
332     $payload_length = \strlen($encrypted_data);
333     $last_byte = \substr($encrypted_data, \strlen($encrypted_data
            ), \strlen($encrypted_data) - 1);
334     do {
335       $new_byte = random_bytes(1);
336     } while ($last_byte == $new_byte);
337     $new_encrypted_data = \substr($encrypted_data, 0, \strlen(
            $encrypted_data) - 1) . $new_byte;
338     $buffer = new Buffer();
339     $buffer->bufferWrite($new_encrypted_data);
340
341     array_push($buffers_padding, $buffer);
342
343   }
344
345   $start_padding = microtime(true);
346   for($i_run = 0; $i_run < $n_runs; ++$i_run) {
347     try {
348       processMessage($buffers_padding[$i_run], $payload_length,
              $shared, $connection);
349     } catch (\danog\MadelineProto\SecurityException $e) {
350       /* assert($e->getMessage() == "msg_key mismatch"); */
351       /* print_r($e->getMessage()); */
352     }
353   }
354   $end_padding = microtime(true);
355
356   $delta_msg_id = ($end_msg_id - $start_msg_id)/$n_runs;
357   $delta_padding = ($end_padding - $start_padding)/$n_runs;
358   $delta_length = ($end_length - $start_length)/$n_runs;
359   assert($delta_padding > $delta_length);
360   array_push($data, array($n_runs, $message_data_length,
          $delta_msg_id, $delta_length, $delta_padding));
361 }
362
363 # Save to file
364 $file = fopen($filename, 'w'); # Write new
365
366
367 /* // save each row of the data */
368 foreach ($data as $row) {
369   fputcsv($file, $row);
```

```
370 }
371 /* echo "measured"; */
372
373 /* // Close the file */
374 fclose($file);
```

Appendix C

Implementation of replay and reordering attacks

C.1 Client implementations

The Listings C.1 to C.3 show how to implement simple clients using the different libraries. All clients have the same behaviour: For every incoming message, they print the received text. Additionally, the clients connect to the Telegram server over a HTTP or SOCKS5 proxy running on `localhost` port 8080.

C.1.1 Pyrogram

Listing C.1: `pyrogram_client.py`: a simple Pyrogram receiver. The use of the proxy must be specified in the `config.ini` file.

```
from pyrogram import Client, filters

app = Client("test_account", test_mode=True)

@app.on_message(filters.text)
def print_message(client, message):
    print(message.text)

if __name__ == '__main__':
    app.run()
```

C.1.2 Telethon

Listing C.2: `telethon_client.py`: a simple Telethon receiver.

```
from telethon import TelegramClient, events

api_id = 123456
api_hash = 'your_hash_here'
```

```
proxy = ("http", '127.0.0.1', 8080)

with TelegramClient('test', api_id, api_hash, proxy=proxy) as
    client:
    @client.on(events.NewMessage(chats="me"))
    async def handler(event):
        print(event.message.message)

    client.run_until_disconnected()
```

C.1.3 GramJS

Listing C.3: gramJS_client.js: a simple GramJS receiver.

```
const { TelegramClient } = require('telegram')
const { StringSession } = require('telegram/sessions')
const {NewMessage} = require('telegram/events')

const apiId = 123456 // Change to your API ID
const apiHash = '' // Insert your API hash
const stringSession = new StringSession('');

function eventPrint(event) {
    // Everytime you receive a mesage, print it
    console.log(event.message.text);
}

const client = new TelegramClient(stringSession, apiId, apiHash,
    {
        useWSS: false,
        proxy: {
            ip: "127.0.0.1", // Proxy host IP
            port: 8080, // Proxy port
            MTProxy: false, // Use SOCKS
            socksType: 5, // Use SOCKS5
            timeout: 2 // Timeout (in seconds) for connection,

        }
})

client.addEventHandler(eventPrint, new NewMessage({}));
client.connect();
```

C.2 Mitmproxy add-ons

Listing C.4 shows how to replay text messages using mitmproxy. To run the attack, execute

```
mitmproxy -s [replay,reorder]_addon.py [--mode socks5]
```

where socks5 is only needed for the attack against GramJS.

C.2.1 Replay attack

Listing C.4: `replay_addon.py`

```
from mitmproxy import ctx

class Replayer:
    def __init__(self):
        self.saved = None

    def tcp_message(self, flow):
        message = flow.messages[-1]
        message_len = len(str(message))

        ctx.log.info(str(message_len))
        if 700 < message_len < 1000:  # Only save text messages
            if self.saved is None:
                ctx.log.info("SAVE packet")
                self.saved = message.content
            else:
                ctx.log.info("LOAD packet")
                message.content = self.saved
                self.saved = None

addons = [
    Replayer()
]
```

C.2.2 Reordering attack

Listing C.5: `reorder_addon.py`

```
"""
Addon for mitmproxy that reorders packets.

Usage:
    mitmproxy -s reorder_addon.py
    mitmdump -s reorder_addon.py
"""

from mitmproxy import ctx

class Reorder:
    def __init__(self):
        self.packets = []
        self.next = 0
        self.basic_message = None

    def tcp_message(self, flow):
        message = flow.messages[-1]
        message_len = len(str(message))
        ctx.log.info(str(message_len))

```

```
22          if 700 < message_len < 1000: # Only deal with text
                messages
23              if self.basic_message == None:
24                  ctx.log.info("SAVE basic message")
25                  self.basic_message = message.content
26                  return
27
28              # Store the first four packets. Replace them with the
                    basic_message
29              if 0 <= len(self.packets) < 4:
30                  ctx.log.info("SAVE packet")
31                  self.packets += [message.content]
32                  message.content = self.basic_message
33
34              # Only reorder the first 4 packets after
                    basic_message
35              elif self.next < 12:
36                  ctx.log.info("LOAD packet")
37                  message.content = self.packets[self.next % len(
                        self.packets)]
38                  self.next += 3
39
40
41
42
43  addons = [
44      Reorder()
45  ]
```