ETH zürich

ETH Library

On the Cryptographic Fragility of the Telegram Ecosystem

Conference Paper

Author(s):
von Arx, Theo; Paterson, Kenneth G.

Publication date:
2023-07-10

Permanent link:
https://doi.org/10.3929/ethz-b-000620789

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
https://doi.org/10.1145/3579856.3582811

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

On the Cryptographic Fragility of the Telegram Ecosystem

Theo von Arx
ETH Zurich
Switzerland
theo.vonarx@inf.ethz.ch

Kenneth G. Paterson
ETH Zurich
Switzerland
kenny.paterson@inf.ethz.ch

ABSTRACT

Telegram is a popular messenger with more than 550 million active users per month and with a large ecosystem of different clients. The wide adoption of Telegram by protestors relying on private and secure messaging provides motivation for developing a profound understanding of its cryptographic design and how this influences its security properties. Telegram has its own bespoke transport layer security protocol, MTProto 2.0. This protocol was recently subjected to a detailed study by Albrecht et al. (IEEE S&P 2022). They gave attacks on the protocol and its implementations, along with a security proof for a modified version of the protocol. We complement that study by analysing a range of third-party client implementations of MTProto 2.0. We report practical replay attacks for the Pyrogram, Telethon and GramJS clients, and a more theoretical timing attack against the MadelineProto client. We show how vulnerable third-party clients can affect the security of the entire ecosystem, including official clients. Our analysis reveals that many third-party clients fail to securely implement MTProto 2.0. We discuss the reasons for these failures, focussing on complications in the design of MTProto 2.0 that lead developers to omit security-critical features or to implement the protocol in an insecure manner. We also discuss changes that could be made to MTProto 2.0 to remedy this situation. Overall, our work highlights the cryptographic fragility of the Telegram ecosystem.

1 INTRODUCTION

Messenger services have become an indispensable tool to billions of people. Telegram is one of the most popular messenger services: Telegram reached 500M monthly active users in January 2021 [1] and 550M by October 2021 [2], making it the fifth most popular messenger service globally. Telegram is not only popular for everyday messaging, but is also widely used by activists and demonstrators for group organization [3]: protestors favour Telegram over other messenger apps since Telegram allows both small private as well as large public group chats. Activists have further perceived Telegram's security to be stronger than that of any other messenger. Telegram does not require its users to register with a phone number, and therefore provides enhanced privacy compared to, for example, Signal and WhatsApp, which do require such registration. In short, Telegram's popularity among potentially vulnerable and sensitive democratic actors shows that it is crucial to understand its security.

Of particular importance in this endeavour is Telegram's transport protocol MTProto. This protocol acts as the equivalent of the TLS record protocol and is the only security mechanism protecting messages in transit between Telegram servers and clients. Prior work has shown multiple attacks against an earlier version of MTProto [4, 5]. In a recent paper, Albrecht et al. [6] presented the first in-depth analysis of MTProto 2.0. Their work had two complementary aspects. First they showed four different attacks, two of them at the protocol specification level, and two against specific MTProto 2.0 implementations in official clients. Second, they provided a cryptographic proof of security of a slightly modified version of MTProto 2.0, albeit under previously unstudied cryptographic assumptions. Altogether, [6] has substantially increased our understanding of the security of the MTProto 2.0 protocol.

As opposed to other chat platforms such as WhatsApp, the source code of Telegram's official clients is publicly available (though the server source code is not). Furthermore, Telegram allows and encourages developers to implement and deploy custom clients: the Telegram web pages provide detailed information about the server APIs [7], the MTProto 2.0 protocol [8], and the custom schema used [9]. Consequently, there is a flourishing implementation 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 [10].

Another vital part of the Telegram ecosystem are bots which can interact with other services such as email, YouTube, payments, and games. Therefore, Telegram cannot be regarded as an isolated platform with the sole purpose of exchanging text messages. Rather, it is developing towards being a system that connects multiple, different services and offers users a simple way of interaction. Clearly, this further underlines the importance of having a correctly implemented, secure protocol at its core.

The vulnerabilities in MTProto 2.0 in various official clients highlighted in [6] suggest that third-party implementations of MTProto 2.0 might also contain security flaws. It is plausible that these flaws might be more severe than those presented in [6], given the unregulated nature of the ecosystem and the well-documented propensity of non-expert developers to make mistakes when implementing complex cryptographic protocols. Moreover, the diverse and complex collection of available Telegram clients and libraries potentially opens new attack vectors: vulnerable third-party clients could be used to harm users of official clients. Consequently, the ease with which secure implementations of MTProto 2.0 can be built is essential to the security of the entire ecosystem.

1.1 Contributions

In this paper, we analyse the security of the Telegram client ecosystem, focussing on different client implementations of MTProto 2.0. Specifically, we present multiple attacks against third-party Telegram client implementations of MTProto 2.0. We show how vulnerabilities in third-party clients affect the security of the entire ecosystem. While the attacks are not particularly novel from a technical perspective, their presence is surprising. We explain how

ACM ASIACCS 2023, 2023, Melbourne, Australia
2023.

the design choices in MTProto 2.0 make implementations prone to errors and open these implementations up to known attack vectors.

Our first contribution, presented in Section 4, is a replay attack against two Python libraries (Pyrogram, Telethon) and a JavaScript library (GramJS). The vulnerabilities arise from those implementations failing to implement a set of checks on the message ID field that are required in the Telegram documentation for client developers. These checks are designed to ensure that every message is processed exactly once. We describe a practical replay attack for exemplary clients in a real-world setting. The vulnerable Python libraries Pyrogram and Telethon are quite popular: by the time of writing they have 2.4K and 6.2K stars on GitHub, respectively, and are, according to GitHub, used by 39.2K and 25.9K other projects. While GramJS is less popular (379 stars), it is used in an official client (Telegram Web Z). Due to its use of WebSockets over TLS 1.3, Telegram Web Z was not vulnerable to the replay attack.

As a second contribution, we present in Section 5 a timing side-channel attack against MTProto 2.0 decryption in the PHP library MadelineProto. The attack is similar to the one described in [6], but differs in its details to match our specific target. It exploits the way in which the encrypt-and-MAC scheme employed by MTProto 2.0 is implemented in MadelineProto: the code does not check the integrity of the plaintext directly after decryption, but first performs sanity checks on the unauthenticated plaintext and only then checks integrity. Notably, this approach ignores advice aimed at developers in the Telegram security guidelines [11].

Depending on the input, the message processing time differs significantly. This allows an attacker to learn some parts of the plaintext. We evaluate the timing differences that arise, implement the attack in a synthetic setting, and evaluate the attack's limitations. We consider the attack to be 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. As noted in [6], it is indicated in the Telegram protocol description [12] that these values are not intended to be treated as sensitive values, and it is possible they could be revealed in future implementations. On the other hand, the attack in [6] that enables the attacker to learn them by attacking the MTProto 2.0 key exchange protocol is likely not to be possible any longer due to server-side changes made by Telegram.

We discuss how these vulnerabilities in third-party clients affect the security of the entire Telegram ecosystem: security issues arise because of the flexibility with which Telegram can be used, and these issues go beyond pure messaging applications.

As a third contribution, in Section 6 we address a more fundamental question: how can security be guaranteed in a proliferating ecosystem? This question is prompted by the vulnerabilities in several official clients noted in [6] and the new ones we report here. The replay and reordering attacks are examples of fundamental, yet easy-to-defend-against attacks on a protocol level. The reported issues do therefore collectively indicate that Telegram's MTProto 2.0 protocol is hard to implement in a secure manner. Specifically, we highlight two design choices that hinder secure implementation: the use of encrypt-and-MAC and the complex checks on the message ID field. We suggest how to simplify these design choices and lower the implementation hurdles for developers.

Telegram presents yet another example of the challenges of ensuring security in an open ecosystem. In this sense, the situation is broadly analogous to what we have seen over the last decade in the spheres of mobile banking [13], SSL/TLS [14–16], certificate validation [17], and (at a much lower level) primality testing [18]. Indeed, the problem of building a secure communications channel is an ancient one that is by now well-understood at a theoretical level by the research community. Yet it is a largely open problem in the cryptographic community – interpreted broadly, to include both researchers and developers – to find ways of ensuring that this theoretical understanding is properly translated into practice. The problems arise at multiple levels: at the lowest level of ensuring that strong cryptographic algorithms are used, to the middle level of ensuring that these algorithms are combined in suitable ways to build more complex systems like secure channels, to the highest level, of ensuring that a protocol specification is simple enough that it can be securely implemented in an ecosystem populated with developers without cryptographic expertise.

1.2 Ethical Principles

We experimentally verified the feasibility of the replay attack using Telegram's test servers. We have neither interacted with Telegram's production servers nor were any users involved in our experiments.

We informed the maintainers of the vulnerable libraries about our findings in the week of 20 November 2021 by e-mail, proposing a standard 90-day disclosure window. In each case, as well as explaining the discovered vulnerabilities, we also recommended fixes and highlighted the missing checks.

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

The maintainer of GramJS fixed the replay vulnerability inversion 1.11.1, but without giving any notice of the issue to users of GramJS. We also informed Telegram's security team about the vulnerability in GramJS, since it 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 for the vulnerability we disclosed. Telegram awarded a bug bounty.

The maintainer of Telethon confirmed receipt of our disclosure e-mail. Since the vulnerability was not fixed by the end of the 90-day disclosure window, we offered our help in another e-mail and informed the maintainer that we would publicly disclose the vulnerability. As we did not receive an answer, we filed a public GitHub issue in which we described the attack. At the time of writing, the issue has been partially addressed but remains open [19].

The maintainer of Pyrogram deployed the security fixes in version 1.3. The vulnerability is mentioned in the release notes and all users are strongly encouraged to update.

During our research, we analysed six more clients and libraries for similar vulnerabilities: Kotatogram-Desktop, Nicegram, Telegram React, Telegram Web K, Telegram Web Z, Telegram-FOSS, and Unigram. However, we did not find any vulnerabilities in these clients.

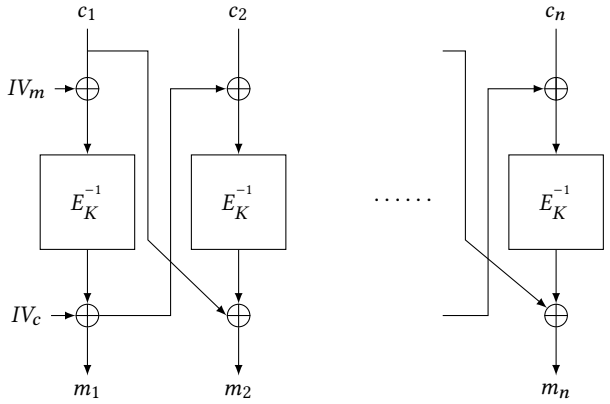

Figure 1: IGE decryption with $c_0 = IV_c$ and $m_0 = IV_m$.

2 PRELIMINARIES

We closely follow the conventions and definitions of [6], but removing complexity that is unnecessary for our presentation.

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.

2.1 IGE block cipher mode of operation

Let $\mathsf{E} : \{0,1\}^k \times \{0,1\}^b \rightarrow \{0,1\}^b$ be a block cipher with k-bit key and b-bit block. We write $E_K(P)$ to denote the block cipher operation on b-bit plaintext block P with k-bit key K. 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. 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 b. For a bit string x we write $x = x_1||...||x_n$ such that $\forall i\ |x_i| = b$ to indicate the different blocks of x. Here, we implicitly set $n = \frac{|x|}{b}$.

Algorithm 1 $\mathsf{IGE[E].Enc}(K, m_0, c_0, m)$

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 $\mathsf{IGE[E].Dec}(K, m_0, c_0, c)$

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 $b = 128$ and key length $k = 256$ as defined in [20]. Then let AES-256-IGE describe the symmetric encryption and decryption as defined in Algorithms 1 and 2.

2.2 Attack scenario and targets

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 adversary can easily achieve become a PitM, e.g., by seducing victims to use a malicious Wi-Fi access point [21].

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

We selected to inspect the most popular clients and libraries (cf. Tables 1 and 2) for the mentioned vulnerabilities. The analysis was done by manually looking at the source code and by exploiting found vulnerabilities in a simulated environment.

Table 1: 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	Library
Kotatogram-Desktop	542	Telegram Desktop	-
Nicegram	297	Telegram-iOS	-
Telegram React	1.8K	-	TDLib
Telegram Web K*	449	-	-
Telegram Web Z*	1.1K	-	GramJS
Telegram-FOSS	1.6K	Telegram (Android)	-
Unigram	2K	-	TDLib

Table 2: 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 official library (marked with "*") is TDLib.

Name	★	Used by	Language
GramJS	379	580	JavaScript
MadelineProto	1.9K	167	PHP
Pyrogram	2.4K	39.2K	Python
TDLib*	4.5k	-	C++
Telethon	6.2K	25.9K	Python

3 DESCRIPTION OF THE SYMMETRIC PART OF MTPROTO 2.0

We have chosen to focus messages in *cloud chats* which are the default setting in Telegram and are, according to Telegram's founder Pavel Durov, "designed for the majority of users" [22]. End-to-end encrypted *secret chats* are, in contrast to cloud chats, not available in all official clients [23].

The messages in cloud chats 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 [12]. The symmetric part of MTProto 2.0 is Telegram's equivalent to the TLS record protocol. We refer to [12] for a description of the asymmetric part.

MTProto 2.0 aims to guarantee security for application layer messages. The reliable transport protocol TCP is specified for transport of MTProto 2.0. 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 implementations [8]. We stress, that TCP (as well as UDP) cannot provide any security guarantees and therefore – without any further defence mechanisms such as TLS 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 the presence of such additional layers since they are only optional or available in certain settings.

3.1 MTProto 2.0 Encryption

The payload p of a MTProto 2.0 message consists of the fields described in Table 3. 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 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 1024 B.

A plaintext m with blocks $m_1||m_2||...||m_n$ is parsed into the fields defined in Table 3, and denoted by server_salt(m_1), session_id(m_1), msg_id(m_2), msg_seq_no(m_2), msg_length(m_2), as well as the remaining msg_data($m_3||...||m_n$) and 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. 2 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

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

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

$$h := \text{SHA-256}(\text{auth\_key}[704 + x : 960 + x]||p) \tag{2}$$

$$\text{msg\_key} := h[64 : 192] \tag{3}$$

Table 3: MTProto payload format [6]. 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 multiplied 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.

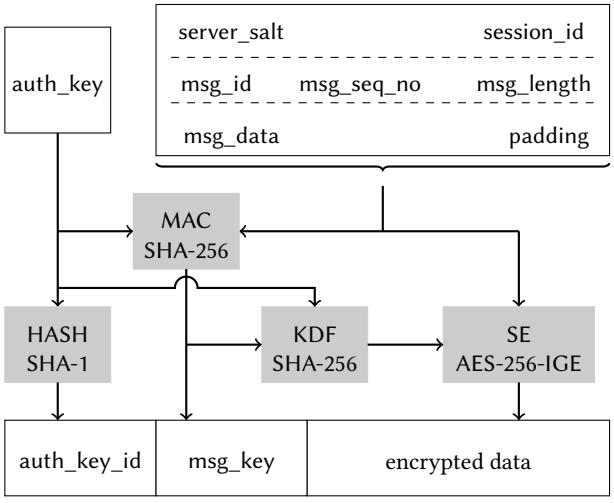

Figure 2: Overview of message processing in MTProto 2.0. Note that only parts of auth_key are used in MAC and KDF. This figure is a modified copy from [6].

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

$$A := \text{SHA-256}(\text{msg\_key}||\text{auth\_key}[x : 288 + x]) \tag{4}$$

$$B := \text{SHA-256}(\text{auth\_key}[320 + x : 608 + x]||\text{msg\_key}) \tag{5}$$

$$\text{key} := A[0 : 64]||B[64 : 192]||A[192 : 256] \tag{6}$$

$$\text{iv} := B[0 : 64]||A[64 : 192]||B[192 : 256] \tag{7}$$

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

$$c := \text{AES-256-IGE}(\text{key}, \text{iv}, p)\,. \tag{8}$$

3.2 Required checks on metadata

When receiving a message, the client has to perform the following checks, according to [11]:

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

(C2) The client must check that msg_length is 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 is 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 is odd.

(C4.2) The client must store the msg_id of the last N received messages. Here, the value of N is not specified [11]. [1] The client must check that an incoming msg_id is not smaller than all N stored message IDs and that msg_id is 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.

4 REPLAY ATTACK

4.1 Description of the vulnerability

In a *replay attack*, an attacker can resend certain messages and fool the receiver into believing that both messages originate from the sender. While the attacker cannot read the messages, it is a simple yet powerful attack to modify conversations. By leveraging patterns in communication and stereotypical message lengths (only partially hidden by variable-length padding), the attacker can decide which messages to target, improving the effectiveness of such an attack [26]. Here, we omit these fingerprinting steps and focus on demonstrating the general feasibility of a replay attack.

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 [27] and Telethon [28], as well as the JavaScript library GramJS [29]. The relevant code snippets can be seen in Listings 1 to 3. Pyrogram only checks that the msg_id is odd. Telethon and GramJS do not even check this. Furthermore, Telethon and GramJS both include comments that hint at the missing checks.

While Pyrogram and Telethon appear to be independent projects, the core of GramJS is fully based on Telethon. The relevant lines in the code only differ in syntax. Table 2 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 [33].

[1] Official implementation use different values: Telegram Desktop [24] uses $N = 400$, TDLib [25] uses $N = 2000$.

Listing 1: `mtproto.py`. Message processing in Pyrogram [30]. Modified for readability.

```python
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 2: `mtprotostate.py`. Message processing in Telethon [31]. Modified for readability.

```python
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 3: `MTProtoState.ts`. Message processing in GramJS [32]. Modified for readability.

```javascript
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);
}
```

An attacker can perform a replay attack against a client using any one of these three 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.

4.2 Attack implementation

To experimentally verify the presence of the vulnerability, we implemented clients using the libraries above (c.f. Listings 5 to 7 in Appendix A). To exploit the attack we configure the clients to route all traffic to a local proxy server. For the proxy server, we use mitmproxy [34] with the add-on shown in Listing 8 to easily record and

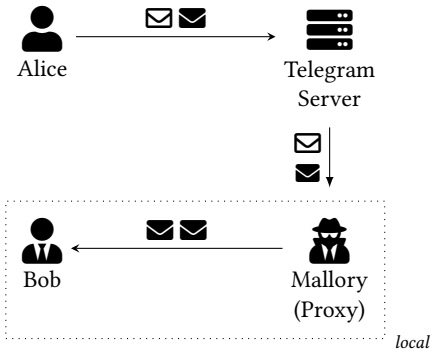

Figure 3: Overview of the replay attack. The symbols "✉" and "✉" denote two different messages.

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 3 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. 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 the same message twice.

The attack was successful against all three of the 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.

Not all clients which use a vulnerable library are automatically vulnerable to the replay attack. As messages come with an integrity-protected time-stamp that is set by the Telegram server, some clients then use this time-stamp to store the message in an internal 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 must not depend on unspecified requirements.

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.

4.3 Wider impacts of replay attacks

The attack is even more powerful when a vulnerable library is used to implement the control of, e.g., a server. Telminal [35] 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, e.g., sends the command to remove the first entry of a database, the attacker can flush the entire database by repeatedly replaying the command. While Telminal is currently a niche application (only 20 stars on GitHub), it illustrates a potential future attack vector.

Other interesting settings include message forwarding from Telegram to WhatsApp and vice versa [36] (based on GramJS), cryptocurrency trading [37] (based on GramJS), as well as bots that broadcast a received message [38] (based on Pyrogram). In the broadcast setting, the (vulnerable) bot automatically wraps the received message content into a new message and relays it via the Telegram server to multiple receivers. An attacker can thus intercept and replay the messages towards the broadcast bot which will then in turn forward all messages including the replayed ones. Since all messages appear to be new, all final receivers will accept and process them normally, independently of whether the receivers use a vulnerable library or not. In particular, there is no cryptographic defence that allows them to recognize that an attacker has replayed the original message.

While not tested in practice, these examples show that the mere possibility of interaction of a vulnerable client with other clients translates the security risk of a vulnerable third-party client to the entire Telegram ecosystem, including official clients. Moreover that, there is a potential reputation damage for the entire ecosystem coming from the vulnerability of popular libraries and clients. Telegram consequently has a strong interest in the security of third-party clients.

4.4 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 9.

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 [11]: unless $N = 1$, the check (C4.2) on the msg_id does not force message IDs to be strictly monotonically increasing and therefore messages can be processed out of order [6]. The reordering vulnerability is inherent to the cryptographic specification, and is only defended against at the application level in official clients [6].

While Telegram could argue that reordering attacks are not a concern because official clients are effectively not vulnerable to them, the argument does not hold for library implementations. The implementer of the library simply cannot know how their library will be used by applications. Hence, assuming the library provides a reasonable amount of flexibility to its users, no general security guarantees that rely on the application can be given. The reordering attack should therefore be defended against by MTProto 2.0 itself.

5 TIMING SIDE-CHANNEL ATTACK ON MADELINEPROTO

The fact that three official Telegram clients were found to be vulnerable to a timing side-channel attack in [6] led us to suspect that some third-party clients and libraries might be vulnerable to a similar attack. In fact, aside from MadelineProto, none of the

implementations of MTProto 2.0 listed in Tables 1 and 2 were found to be vulnerable to a timing side-channel attack.

In the remainder of this section, we focus on MadelineProto. We first recapitulate the core idea of the attack from [6], and then explain in depth how we adapt this attack to MadelineProto.

5.1 Attack idea

For a given ciphertext $c_1||...||c_n$ corresponding to an 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. Specifically, the attacker needs to know both m_1 and m_{i-1}.

Recall that in MTProto 2.0, m_1 contains the server_salt and session_id fields. While these values were not intended to be secrets, they are not sent in plaintext at any point in the protocol. The more complex attack on the MTProto 2.0 key exchange protocol described in Section F of [6] allows an attacker to learn server_salt and session_id. Even though this attack is hard and likely not possible any longer due to server-side changes by Telegram, there might be a successful attack against m_1 in the future. An additional argument in favour of assuming that m_1 is known is that the security of a secure channel protocol like MTProto 2.0 should not rest on maintaining the confidentiality of a specific plaintext block: the design goal of such a protocol should be to protect all of the plaintext data all of the time.

The requirement for the attacker to know m_{i-1}, the plaintext block preceding the target block, is a weaker one. This is because general plaintext blocks often contain stereotypical values. Consider for example the situation where m_{i-1} = "The password" and m_i = "is SECRET". In general, we consider this assumption to be realistic.

The attacker then creates a two-block ciphertext $c_1||c^\star$ with $c^\star = c_i \oplus m_{i-1} \oplus m_1$. This ciphertext decrypts to $m_1||m^\star$ where:

$$m^\star = E_K^{-1}(c^\star \oplus m_1) \oplus c_1 \tag{9}$$

$$= E_K^{-1}(c_i \oplus m_{i-1}) \oplus c_1 \tag{10}$$

$$= m_i \oplus c_{i-1} \oplus c_1 \,. \tag{11}$$

Consequently, if there is a side-channel that allows an attacker to learn some bits of the second block (containing $m^\star$), then the attacker can learn the corresponding bits of m_i by using Eq. (11). The key insight from [6] is that this second block is interpreted as containing the packet length field, and that field may be sanity checked prior to the MTProto 2.0 MAC being verified. The success or failure of the sanity checking may be visible through timing behaviour of the client, so leaking some information about $m^\star$, and hence about m_i.

5.2 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 [8]. The library can be used for multiple purposes including voice over IP (VoIP) webradio, downloading files and controlling a server [39].

5.2.1 Message processing. When receiving a packet, MadelineProto processes it as follows (c.f. Listing 4):

(1) Check whether received auth_key_id matches the computed one.
(2) Reduce the ciphertext such that 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 [40].
(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. [2] The restriction of ciphertexts being a multiple of 16 B arises from the use of AES with block size 16 B. However, instead of this reduction, the client could directly reject a malformed message since it must have been tampered with.[3]

The operations between decryption at step 3 and the integrity check at step 7 in MadelineProto are carried out 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, it can tell whether the checks on the packet length field (C2) executed at step 6 have passed or failed. Since the packet length field is contained in the second plaintext block, and this can be replaced by $m^\star = m_i \oplus c_{i-1} \oplus c_1$ in the attack, the success or failure of these checks may allow the attacker to learn some bits of the targeted message m_i.

As we show in the next section, the difference between a failure in a msg_id, msg_length and a msg_key are indeed observable by measuring the client's processing time. 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, so the keys, the server_salt and the session_id stay the same. Hence, many forged ciphertexts can be sent in a single attack against a single target ciphertext block c_i, and these ciphertexts will all be decrypted and further processed in the same way. So, many decryption trials can be executed against a single target c_i, allowing noise in the timing measurements to be averaged out and the timing signal to be amplified.

In turn, this makes it easier for an attacker to determine the cause of failure, and hence determine whether checks on the packet length field have passed or failed. Finally, by executing enough trials, this makes it theoretically possible to recover some bits of m_i in a reliable way. This contrasts with the more common situation in such timing attacks (e.g. for Lucky 13 [42]) where each error leads to a session termination and keys being thrown away – in such a situation, each target ciphertext block can only be tested once, requiring the introduction of an additional assumption, namely that the same plaintext target m_i is repeated across many sessions.

[2] This leads to a trivial attack in the indistinguishability under chosen ciphertext attack (IND-CCA) model! The adversary can trivially extend a challenge ciphertext with random bits of its choice to make a new cipheretext, and then submit it for decryption. While theoretically interesting, this does not lead to a practical attack against MTProto 2.0.

[3] 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.

Listing 4: Message processing in MadelineProto [41]. Modified for readability. `$seq_no` and `$message_data_length` correspond 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');
    }
  }
  # [...]
}
```

5.2.2 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 differences between failures of the msg_id, the msg_length and the msg_key checks, we measured the message processing time in a simulated environment: 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.

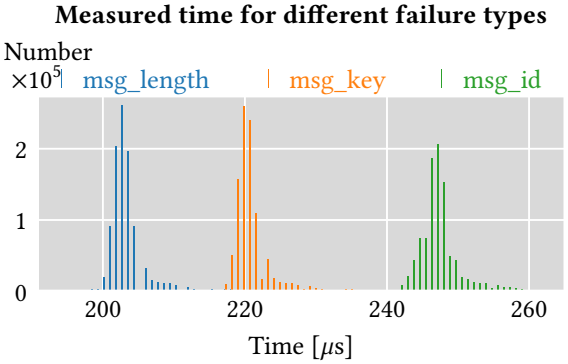

Figure 4: Timings of failures of msg_id, msg_length and msg_key checks, measured under ideal conditions. Packet size: 2048 B.

Table 4: Statistics of processing time measured in µs

Error type	# samples	Mean	Median	St. dev
msg_length	996057	204.010	203.133	4.313
msg_key	993465	221.408	221.014	4.291
msg_id	967952	247.271	247.002	2.835

Our results are visualized in Fig. 4 and key statistics are shown in Table 4. The time difference arising between failures of the msg_length and msg_id checks is due to the additional SHA-256 computation in the 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 relatively 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 always have to distinguish between msg_key and msg_id failures, but only between msg_length and msg_id failures.

There is one significant limitation, however: we cannot assume that the timing differences between failures of the different length checks (e.g., that the padding size is bigger than 12 B or that the padding size is smaller than 1024 B) are measurable. The reason is that no computationally intensive operations are involved between two such checks. This complicates our attack slightly.

5.2.3 Attack in a clean oracle model. As a proof of concept, we now describe how an attacker can exploit timing differences described above to recover part of the target plaintext block m_i. To simplify, we present an attack that works in a "clean oracle" setting, i.e., we assume that the attacker can perfectly distinguish between the three different error types. The pseudocode of the attack is given in Algorithm 3. We successfully implemented the attack in Python.

Algorithm 3 Timing side-channel attack against MadelineProto.

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

▷ Increment l until msg_key failure or reaching the limit
  repeat
    $c' \leftarrow \tilde{c}||\mathsf{randomBytes}(l)$
    if $\mathsf{len}(c') > 2^{22}$ **then**
      Log("The 10 MSBs or the 2 LSBs of $m^\star$ are nonzero")
      return $\perp$
    $\mathrm{ans} \leftarrow \mathsf{O}(c')$
    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 $= \lceil \frac{\mathrm{lo+hi}}{2} \rceil$
    $c' = \tilde{c}||\mathsf{randomBytes}(16 \cdot \mathrm{mid})$
    $\mathrm{ans} \leftarrow \mathsf{O}(c')$
    if ans = LENGTH_FAIL **then**
      lo $\leftarrow$ mid
    else if ans = INTEGRITY_FAIL **then**
      hi $\leftarrow$ mid-1
  $l \leftarrow l + 16 \cdot \mathrm{mid}$
  if ans = LENGTH_FAIL **then**
    $l \leftarrow l + 16$ ▷ Add a block to get an integrity check failure

  $c' \leftarrow \tilde{c}||\mathsf{randomBytes}(l + 1008)$
  $\mathrm{ans} \leftarrow \mathsf{O}(c')$
  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

The attack takes as input a target block index i, the known plaintext blocks m_1, m_{i-1} and the target payload payload consisting of auth_key_id, msg_key and ciphertext blocks $c_1, \ldots, c_n$.

The core idea of the attack is to vary the number l of random bytes that are appended to a partial payload of the form $\tilde{c} = \mathsf{auth\_key\_id}||\mathsf{msg\_key}||c_1||c^\star$, where $c^\star = c_i \oplus m_{i-1} \oplus m_1$ as before.

Such payloads 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. (11) 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 containing msg_id, msg_seq_no, and msg_length, while blocks $m'_3, ..., m'_n$ are interpreted as msg_data and padding. The MadelineProto client computes the size of the padding as $|\mathsf{msg\_data}(m'_3||...||m'_n)| + |\mathsf{padding}(m'_3||...||m'_n)| - \mathsf{msg\_length}(m^\star)$ which is equal to $l - \mathsf{msg\_length}(m^\star)$.

By using such a payload with varying l, the attacker can trigger different errors. The idea is to find the smallest value for l, such that the msg_key check fails. This will give the attacker information about the value $\mathsf{msg\_length}(m^\star)$ and allows the corresponding bits of the target block m_i to be inferred using Eq. (11).

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

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

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

Since there is a lower limit (of $l - 1008$) and an upper limit (of l) for the minimal size that triggers a msg_key check failure, the attacker can now switch to using a binary search to find the smallest value l^- which is a multiple of 16 B and which triggers a msg_key check failure. Once this l^- is found, we know that

$$12 \leq l^- - \mathsf{msg\_length}(m^\star) < 12 + 16 \tag{13}$$

$$i.e. \quad 0 \leq l^- - \mathsf{msg\_length}(m^\star) - 12 < 16\,. \tag{14}$$

At this point, the attacker can correctly recover 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

$$l^- + 1008 - \mathsf{msg\_length}(m^\star) \leq 1024 \tag{15}$$

$$\text{and hence} \quad 0 \leq l^- - \mathsf{msg\_length}(m^\star) - 12 \leq 4\,. \tag{16}$$

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

$$l^- + 1008 - \mathsf{msg\_length}(m^\star) > 1024 \tag{17}$$

$$\text{and hence} \quad 4 < l^- - \mathsf{msg\_length}(m^\star) - 16 < 12\,. \tag{18}$$

In other words: 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 that of the 29 most significant bits (MSBs) of $\mathsf{msg\_length}(m^\star)$ and finally to the corresponding bits of m_i.

The number of queries needed for the above attack is the sum of queries in the linear phase and in the binary search phase. It amounts to approximately

$$\frac{\text{msg\_length}(m^\star)}{1008} + \log_2(63) \approx \frac{\text{msg\_length}(m^\star)}{1008} + 6. \qquad (19)$$

The experimentally measured number of queries for varying values of msg_length($m^\star$) matches the expected behaviour: for msg_length($m^\star$) $\leq 2^{10}$ the number of queries is dominated by the binary search. For larger msg_length($m^\star$), the number of queries grows linearly because it is dominated by the linear search. Note that the number of queries needed only depends on the value of msg_length($m^\star$). If we limit $m^\star$ to have msg_length $\leq L$ for an arbitrary $L < 2^{32}$, then the average value of $m^\star$ is $\frac{L}{2}$. The attacker thus needs $\approx L \cdot 2^{-11}$ queries on average in the clean oracle setting.

5.2.4 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 was 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 msg_length($m^\star$) is too big, then the attack does not finish in reasonable time. Furthermore, the attacker needs to send a message with length on the order of msg_length($m^\star$) bytes which may trigger an OS exception due to a large amount of memory being allocated, leading to a crash of the client. For our experimental setup, we had to require msg_length $\leq 2^{22}$. To summarize, the attack is successful in the following cases:

(1) msg_id($m^\star$) is smaller than the maximum limit of about 2^{63}.
(2) msg_id($m^\star$) has odd parity.
(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. So the length of a message is no longer obfuscated.

If $i \neq 2$, the success probability is reduced to $\approx 2^{-1} \cdot 2^{-1} \cdot 2^{-10} \cdot 2^{-2} = 2^{-14}$ since the four conditions listed above must all hold (here we rely for the probability analysis that msg_length($m^\star$) is effectively randomised since it arises as a 32-bit subfield of $m_i \oplus c_{i-1} \oplus c_1$ and we can treat ciphertext blocks c_{i-1}, c_1 as being random 128-bit strings). Here, as above, we assume 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 expect to recover 29 bits of plaintext from one in every 2^{14} blocks.

5.2.5 Attack with noisy oracles. In a real-world setting, we have to take into account the fact that the correctness of the oracle responses is probabilistic, with sources of noise coming from the presence of other processes running on the client and from network jitter arising between the client and the attacker.

However, the attacker can repeatedly send the same message, observe the client processing times, and average over multiple measurements to improve the timing accuracy. Here we are assisted by the fact that the MadelineProto client does not force a re-establishment of MTProto sessions on decryption failures. This makes it possible to build an (effectively) clean oracle from the actual noisy one that is available. Assuming an independent and identically distributed Gaussian distribution of the response time for multiple queries, the variance of the average scales down by the square root of the number of trials. Using standard statistical techniques involving the different means and variances for the 3 different timing distributions (cf. Fig. 4), it is possible to compute how many trials are required to obtain an (effectively) clean oracle as a function of the noise distribution. In short, it is sufficient to make each pair of peaks of average timings for the 3 different distributions lie a few standard deviations apart to obtain a reliable oracle. As Fig. 4 and Table 4 show, there is a gap on the order of 16 µs between the different failure types. This should already be large enough to carry out the attack with the client and attacker located in the same LAN. As a point of comparison [42] showed this to be the case with timing differences on the order of only 1 µs.

However, we again stress that the required knowledge of the values of the server_salt and the session_id makes the attack mostly of theoretical interest. For this reason we have not implemented the full attack against the actual client with a remote attacker, but were content to describe the attack and experiment with it in a simulated environment. This said, the values server_salt and session_id are not specified to be secret [12] so the two values may be revealed in a future implementation.

6 SECURITY IN A PROLIFERATING ECOSYSTEM

The presence of the replay vulnerability in three different Telegram libraries indicates the difficulty that developers face in implementing the required checks correctly. Two implementations (Listings 2 and 3) explicitly mention the missing implementation within a comment. Additionally, during the disclosure, one developer asked us for help to implement the fixes correctly. This indicates a more fundamental problem: the replay protection as specified in [11] and described in Section 3.2 is too large a burden for developers. It is neither straight forward to implement them, nor does it seem that enough help is provided. The specification requires a relatively complex text and no implementation examples are provided. To illustrate the complexity of the specifications, we quote the security guidelines for client developers for the checks on the msg_id [11]:

> The client must check that msg_id has even parity for messages from client to server, and odd parity for messages from server to client.
> In addition, the identifiers (msg_id) of the last N messages received from the other side must be stored, and if a message comes in with an msg_id lower than all or equal to any of the stored values, that message is to be ignored. Otherwise, the new message msg_id is added to the set, and, if the number of stored msg_id values is greater than N, the oldest (i. e. the lowest) is discarded.
> In addition, msg_id values that belong over 30 seconds in the future or over 300 seconds in the past are to be ignored (recall that msg_id approximately equals unixtime * 2^{32}). This is especially important for the server. The client would also find this useful (to protect from a replay attack), but only

if it is certain of its time (for example, if its time has been synchronized with that of the server).
Certain client-to-server service messages containing data sent by the client to the server (for example, msg_id of a recent client query) may, nonetheless, be processed on the client even if the time appears to be "incorrect". This is especially true of messages to change server_salt and notifications about invalid time on the client.

The specification is not only quite long and contains checks that are not relevant for a client developer, it is also not very clear and proposes to accept certain messages even if they do not strictly match the requirements. It is worth mentioning that even the official implementation in TDLib deviates from the specification by keeping at most $2 \cdot N$ message IDs and removing the oldest N message IDs in one go [43].

Since vulnerabilities in third-party clients affect not only Telegram in terms of reputational damage, but also directly on the application level (e.g., by the use of a vulnerable client to broadcast messages [38]), we argue that Telegram should address the underlying issue of hard-to-implement specifications directly. The handling of a sliding window for incoming messages required in MTProto 2.0 is significantly more complex compared to other protocols such as DTLS [44, Section 4.5] and IPsec [45, Section B.2]. It would be straightforward to require and check that the msg_id increases by exactly 1 for every sent message, as it is implicitly done in the TLS record protocol [46, Section 5.3]. This is easy to implement and would directly rule out any reordering attack. Since Telegram de facto relies on TCP and thus on reliable transport, it is unclear why the current complexity is needed. Our proposed change can thus be implemented without further consequences.

Furthermore, the vulnerabilities that we found in third-party clients and libraries together with the ones discussed in [6] suggest a much wider question: how can security be guaranteed in an environment consisting of a variety of independent implementations?

The origin of the problems seems to be three-fold. Firstly, Telegram is developer-friendly and encourages developers to implement their own clients and bots [7]. This attracts developers without a cryptographic background. Secondly, as shown above, the custom protocol MTProto 2.0 does not make it easy to build a secure implementation. Thirdly, when security issues in the Telegram protocol and official clients were discovered [6], Telegram's official clients were patched without any vulnerability announcement being made by Telegram. There is no communication channel between Telegram and third-party developers to publish vulnerabilities and give security advices. The lack of transparency could hint at a strategy that favours adoption over security. We consider this as being a missed opportunity for Telegram to draw the attention of developers in their broader ecosystem to the security issues.

The first problem is partially addressed by the introduction of the cryptographic library TDLib in 2018 [47]. We propose that Telegram formally verifies TDLib and makes a strong recommendation to use the library. A downside of adopting this recommendation would be the introduction of a single point of failure on the implementation side. But, as our and prior work on other ecosystems suggests, secure implementations and fixes against known attacks are hard to implement in a broad ecosystem, cf. [13–18]. While the question about the optimal trade-off is open for future work, we consequently advocate for security over having an open and distributed development. To reduce the potentially resulting centralization of clients, Telegram could further separate security critical parts from the rest of the client code. This way, openness on the application level could be guaranteed while improving security.

However, not all developers will use TDLib. Although the library 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 issue. Otherwise, the Telegram specifications and security guidelines need to be more precise and easily understandable for non-cryptographers. Improvements in this direction include publishing pseudocode of the correct implementation of MTProto 2.0, providing test vectors, and applying formal methods.

The second problem is not addressed yet. To the contrary, design choices such as the relatively complex checks on the message ID could be simplified without loss of security. Similarly, the design choice to use encrypt-and-MAC opens the door for bad implementations of the decryption process and the introduction of potential timing side-channels. This was also observed in [6] but is endemic to encrypt-and-MAC, see for example the analysis of encrypt-and-MAC in SSH in [48] and the generic treatment in [49]. The use of encrypt-then-MAC in place of encrypt-and-MAC would significantly lower the potential for timing side-channel vulnerabilities in implementations because the MAC would be verified on the ciphertext, removing the temptation to perform decryption at all if the verification fails. Even better, Telegram could switch to using an AEAD scheme (as TLS has done exclusively in TLS 1.3). To summarize, the number of Telegram clients having critical security vulnerabilities shows that the security checks should be as simple as possible so that they will be correctly implemented.

More fundamentally, the justification to use a custom protocol in Telegram is questionable. Telegram mentions reliability for weak mobile connections and speed for cryptographic processing of large files as the reason for introducing MTProto 2.0 [50]. 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 the literature and many state-of-the-art libraries for various languages exist.[4]

However, one argument in favour of MTProto 2.0 lies in the root of trust. By designing and deploying their own protocol, Telegram can carefully choose the root of trust for server authentication and need not rely on trust in dozens to hundreds of root certficate authorities (CAs) as TLS 1.3 does. But this argument is weakened by the reliance on secure transport of the client software itself to the user: most likely this will be secured by TLS. Telegram could moreover use TLS but hard-code the trusted root CAs.

Finally, the solution to the third problem is simple: Telegram should communicate in an open and transparent way with their developer community (and with their users) when security vulnerabilities are disclosed to them. Indeed, prior work has shown, that

[4]We note that 0-RTT messages are not replayable following the TLS 1.3 specification.

full public disclosure does not reduce security [51]. In a competitive ecosystem with multiple providers of software with the same functionality, security may even be increased by this practice [52]. Moreover, a recent study suggests that well-documented security changes with minimal migration effort have a high chance to be quickly adopted by open source developers [53]. Lastly, a policy of transparent disclosure would align with Telegram's will to attract developers and let Telegram assume their responsibility: while Telegram provides a wide range of functions and flexibility, they should also allow developers to learn from previous mistakes.

7 CONCLUSION AND FUTURE WORK

We have shown replay and reordering attacks against the Pyrogram, Telethon, and GramJS Telegram clients. The attacks are practical and can be exploited by, e.g., running a malicious Wi-Fi access point. The attacks are powerful in that they can significantly alter the view of a conversation for any participant using a vulnerable client.

We have also presented a timing side-channel attack against MadelineProto that lets an attacker learn the true length of a message as well as 29 bits of an arbitrary plaintext block with a probability of 2^{-14}. We have shown how to implement the attack. This attack is mostly of theoretical interest due to the hard-to-achieve requirements of knowing the server_salt and the session_id.

Most important, we have explained why our attacks should not be viewed as isolated vulnerabilities, but how they highlight the need for action on a deeper level to improve the security of the Telegram ecosystem, to the eventual benefit of its users' privacy. The fact that developers systematically fail to implement MTProto 2.0 correctly as well as the severe consequences a vulnerability in one client may imply for the others motivates the production of a precise, complete specification of the MTProto 2.0 protocol, akin to the TLS 1.3 specification [46]. This would facilitate secure implementation of MTProto 2.0, as well as promote interoperability and further security analysis.

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, a lot of interesting work remains. Future work includes research on secret chats, bots, and control messages.

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

ACKNOWLEDGEMENTS

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. The authors received a bug bounty from Telegram after disclosing the research. We thank the anonymous reviewers of AsiaCCS for their valuable comments.

REFERENCES

[1] P. Durov, "500 million users," https://t.me/durov/147, 2021.
[2] Statista, "Most popular global mobile messenger apps as of October 2021, based on number of monthly active users," https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/, 2021.
[3] M. R. Albrecht, J. Blasco, R. B. Jensen, and L. Mareková, "Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong," in *USENIX Security 2021*.
[4] N. Kobeissi, "Formal Verification for Real-World Cryptographic Protocols and Implementations," Ph.D. dissertation, 2018.
[5] T. Sušánka and J. Kokeš, "Security Analysis of the Telegram IM," in *ACM ROOTS*, 2017.
[6] M. R. Albrecht, L. Mareková, K. G. Paterson, and I. Stepanovs, "Four attacks and a proof for Telegram," in *IEEE S&P 2022*.
[7] Telegram, "API," https://web.archive.org/web/20211127010953/https://core.telegram.org/api, 2021.
[8] ——, "MTProto mobile protocol," https://web.archive.org/web/20211213201047/https://core.telegram.org/mtproto, 2021.
[9] ——, "Telegram schema," https://telegram.org/schema, 2022.
[10] ——, "Telegram applications," https://web.archive.org/web/20211201125716/https://telegram.org/apps, 2021.
[11] ——, "Security guidelines for client developers," https://web.archive.org/web/20211028151304/https://core.telegram.org/mtproto/security_guidelines, 2021.
[12] ——, "Mobile Protocol: Detailed Description," https://web.archive.org/web/20211016013637/https://core.telegram.org/mtproto/description/, 2021.
[13] B. Reaves, J. Bowers, N. Scaife, A. Bates, A. Bhartiya, P. Traynor, and K. R. B. Butler, "Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications," *ACM Transactions on Privacy and Security*, 2017.
[14] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *IEEE S&P 2015*.
[15] E. Ronen, K. G. Paterson, and A. Shamir, "Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure," in *ACM CCS 2018*.
[16] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes, and S. Fahl, "Why Eve and Mallory still love Android: Revisiting TLS (in)security in android applications," in *USENIX Security 2021*.
[17] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *ACM CCS 2012*.
[18] M. R. Albrecht, J. Massimo, K. G. Paterson, and J. Somorovsky, "Prime and prejudice: Primality testing under adversarial conditions," in *ACM SIGSAC CCS 2018*.
[19] "Security Vulnerability: Replay Attack Against Telethon · Issue #3753," 2022. [Online]. Available: https://github.com/LonamiWebs/Telethon/issues/3753
[20] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced Encryption Standard (AES)," 2001.
[21] N. Sombatruang, M. A. Sasse, and M. Baddeley, "Why do people use unsecure public Wi-Fi? An investigation of behaviour and factors driving decisions," in *ACM STAST 2016*, 2016.
[22] P. Durov, "Why Isn't Telegram End-to-End Encrypted by Default?" Telegraph. [Online]. Available: https://web.archive.org/web/20220310005110/https://telegra.ph/Why-Isnt-Telegram-End-to-End-Encrypted-by-Default-08-14
[23] Telegram Support Force, "End-to-End Encryption FAQ," 2022. [Online]. Available: https://web.archive.org/web/20220315180848/https://tsf.telegram.org/manuals/e2ee-simple
[24] 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, 2019.
[25] ——, "TDLib, mtproto_received_ids_manager.h," https://github.com/telegramdesktop/tdesktop/blob/dev/Telegram/SourceFiles/mtproto/details/mtproto_received_ids_manager.h#L15, 2021.
[26] J. Kohout and T. Pevný, "Network Traffic Fingerprinting Based on Approximated Kernel Two-Sample Test," *IEEE Transactions on Information Forensics and Security*, 2018.
[27] Dan "delivrance", "Pyrogram. Telegram MTProto API framework for Python," https://github.com/pyrogram/pyrogram, 2017–.
[28] "LonamiWebs", "Telethon," https://github.com/LonamiWebs/Telethon, 2016–.
[29] GramJS, "Gramjs. NodeJS/Browser MTProto API Telegram client library," https://github.com/gram-js/gramjs, 2019–.
[30] Dan "delivrance", "mtproto.py," https://github.com/pyrogram/pyrogram/blob/34b6002c689273d7233ca1a0976da009a3aafe09/pyrogram/crypto/mtproto.py#L52, 2021.
[31] "LonamiWebs", "mtprotostate.py," https://github.com/LonamiWebs/Telethon/blob/f9643bf7376a5953da2050a5361c9b465f7ee7d9/telethon/network/mtprotostate.py#L133, 2021.
[32] GramJS, "MTProtoState.ts," https://github.com/gram-js/gramjs/blob/7474e57e1f5e392ce9750871db1ca78bf3fcc453/gramjs/network/MTProtoState.ts#L190, 2021.
[33] A. Zinchuk, "Telegram Web Z," https://github.com/Ajaxy/telegram-tt, 2021.
[34] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A free and open source interactive HTTPS proxy," 2010–. [Online]. Available: https://mitmproxy.org/

[35] M. Jafari, "telminal. A terminal in Telegram!" https://github.com/fristhon/telminal, 2021.
[36] S. Affan and R. Pathiyil, "WhatsGram. Yet another userbot for Whatsapp," https://github.com/WhatsGram/WhatsGram, 2021.
[37] S. Almeroth, "telegram-signals," https://github.com/stav/telegram-signals, 2021.
[38] F. Noushad, N. Eashy, and "MrBotDeveloper", "BroadcastBot," https://github.com/nacbots/BroadcastBot, 2021.
[39] D. Gentili, "MadelineProto – Readme.md," https://github.com/danog/MadelineProto/blob/5969ebe783692c8c7aa1b38d380489954a540f66/README.md, 2021.
[40] ——, "MadelineProto – MsgIdHandler64.php," https://github.com/danog/MadelineProto/blob/1389b24751fa3f06ba783888c4eee7b1c42dea84/src/danog/MadelineProto/MTProtoSession/MsgIdHandler/MsgIdHandler64.php#L50, 2021.
[41] ——, "MadelineProto – ReadLoop.php," https://github.com/danog/MadelineProto/blob/1389b24751fa3f06ba783888c4eee7b1c42dea84/src/danog/MadelineProto/Loop/Connection/ReadLoop.php#L106, 2020.
[42] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *IEEE S&P 2013*.
[43] Telegram, "TDLib, AuthData.cpp," 2022. [Online]. Available: https://github.com/tdlib/td/blob/fa8feefed70d64271945e9d5fd010b957d93c8cd/td/mtproto/AuthData.cpp
[44] E. Rescorla, H. Tschofenig, and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3," IETF, RFC 9147, 2022.
[45] S. Kent, "IP Authentication Header," IETF, RFC 4302, 2020.
[46] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," IETF, RFC 8446, 2018.
[47] Telegram, "TDLib -- Build your own Telegram," https://telegram.org/blog/tdlib, 2018.
[48] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *IEEE S&P 2009*.
[49] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *ASIACRYPT 2000*.
[50] Telegram, "FAQ for the technically inclined," https://web.archive.org/web/20211115225615/https://core.telegram.org/techfaq, 2021.
[51] D. Nizovtsev and M. Thursby, "To disclose or not? An analysis of software user behavior," *Information Economics and Policy*, 2007.
[52] A. Arora, R. Krishnan, R. Telang, and Y. Yang, "An Empirical Analysis of Software Vendors ' Patch Release Behavior: Impact of Vulnerability Disclosure," *Information Systems Research*, 2010.
[53] N. Imtiaz, A. Khanom, and L. Williams, "Open or Sneaky? Fast or Slow? Light or Heavy?: Investigating Security Releases of Open Source Packages," *arXiv:2112.06804*, 2021.

A IMPLEMENTATION OF REPLAY AND REORDERING ATTACKS

A.1 Client implementations

The Listings 5 to 7 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.

A.1.1 Pyrogram.

Listing 5: `pyrogram_client.py`: a simple Pyrogram receiver. The use of the proxy must be specified in the `config.ini` file.

```python
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()
```

A.1.2 Telethon.

Listing 6: `telethon_client.py`: a simple Telethon receiver.

```python
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()
```

A.1.3 GramJS.

Listing 7: `gramJS_client.js`: a simple GramJS receiver.

```javascript
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();
```

A.2 Mitmproxy add-ons

Listing 8 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.

A.2.1 Replay attack.

Listing 8: `replay_addon.py`

```python
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()
]
```

A.2.2 *Reordering attack.*

Listing 9: `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))

        if 700 < message_len < 1000: # Only deal with
             text messages
            if self.basic_message == None:
                ctx.log.info("SAVE basic message")
                self.basic_message = message.content
                return

            # Store the first four packets. Replace them
                 with the basic_message
            if 0 <= len(self.packets) < 4:
                ctx.log.info("SAVE packet")
                self.packets += [message.content]
                message.content = self.basic_message

            # Only reorder the first 4 packets after
                 basic_message
            elif self.next < 12:
                ctx.log.info("LOAD packet")
                message.content = self.packets[self.next
                     % len(self.packets)]
                self.next += 3




addons = [
    Reorder()
]
```