Four Attacks and a Proof for Telegram

Martin R. Albrecht*, Lenka Mareková*, Kenneth G. Paterson† and Igors Stepanovs†
*Information Security Group, Royal Holloway, University of London, {martin.albrecht,lenka.marekova.2018}@rhul.ac.uk
†Applied Cryptography Group, ETH Zurich, {kenny.paterson,istepanovs}@inf.ethz.ch

***Abstract*—We study the use of symmetric cryptography in the MTProto 2.0 protocol, Telegram's equivalent of the TLS record protocol. We give positive and negative results. On the one hand, we formally and in detail model a slight variant of Telegram's "record protocol" and prove that it achieves security in a suitable bidirectional secure channel model, albeit under unstudied assumptions; this model itself advances the state-of-the-art for secure channels. On the other hand, we first motivate our modelling deviation from MTProto as deployed by giving two attacks – one of practical, one of theoretical interest – against MTProto without our modifications. We then also give a third attack exploiting timing side channels, of varying strength, in three official Telegram clients. On its own this attack is thwarted by the secrecy of salt and id fields that are established by Telegram's key exchange protocol. To recover these, we chain the third attack with a fourth one against the implementation of the key exchange protocol on Telegram's servers. In totality, our results provide the first comprehensive study of MTProto's use of symmetric cryptography.**

I. Introduction

Telegram is a chat platform that in January 2021 reportedly had 500M monthly users [1]. It provides a host of multimedia and chat features, such as one-on-one chats, public and private group chats for up to 200,000 users as well as public channels with an unlimited number of subscribers. Prior works establish the popularity of Telegram with higher-risk users such as activists [2] and participants of protests [3]. In particular, it is reported in [2], [3] that these groups of users shun Signal in favour of Telegram, partly due to the absence of some key features, but mostly due to Signal's reliance on phone numbers as contact handles.

This heavy usage contrasts with the scant attention paid to Telegram's bespoke cryptographic design – MTProto – by the cryptographic community. To date, only four works treat Telegram. In [4] an attack against the IND-CCA security of MTProto 1.0 was reported, in response to which the protocol was updated. In [5] a replay attack based on improper validation in the Android client was reported. Similarly, [6] reports input validation bugs in Telegram's Windows Phone client. Recently, in [7] MTProto 2.0 (the current version) was proven secure in a symbolic model, but assuming ideal building blocks and abstracting away all implementation/primitive details. In short, the security that Telegram offers is not well understood.

Telegram uses its MTProto "record layer" – offering protection based on symmetric cryptographic techniques – for two different types of chats. By default, messages are encrypted and authenticated between a client and a server, but not end-to-end encrypted: such chats are referred to as *cloud chats*. Here Telegram's MTProto protocol plays the same role that TLS plays in e.g. Facebook Messenger. In addition, Telegram offers optional end-to-end encryption for one-on-one chats which are referred to as *secret chats* (these are tunnelled over cloud chats). So far, the focus in the cryptographic literature has been on secret chats [4], [6] as opposed to cloud chats. In contrast, in [3] it is established that the one-on-one chats played only a minor role for the protest participants interviewed in the study; significant activity was reportedly coordinated using group chats secured by the MTProto protocol between Telegram clients and the Telegram servers. For this reason, we focus here on cloud chats. Given the similarities between the cryptography used in secret and cloud chats, our positive results can be modified to apply to the case of secret chats (but we omit any detailed analysis).

A. Contributions

We provide an in-depth study of how Telegram uses symmetric cryptography inside MTProto for cloud chats. We give four distinctive contributions: our security model for secure channels, the formal model of our variant of MTProto, our attacks on the original protocol and our security proofs for the formal model of MTProto.

Security model: Starting from the observation that MTProto entangles the keys of the two channel directions, in Section III we develop a bidirectional security model for two-party secure channels that allows an adversary full control over generating and delivering ciphertexts from/to either party (client or server). The model assumes that the two parties start with a shared key and use stateful algorithms. Our security definitions come in two flavours, one capturing confidentiality, the other integrity. Our formalisation is broad enough to consider a variety of different styles of secure channels – for example, allowing channels where messages can be delivered out-of-order within some bounds, or where messages can be dropped (neither of which we consider appropriate for secure messaging). This caters for situations where the secure channel operates over an unreliable transport protocol, but where the channel is designed to recover from accidental errors in message delivery as well as from certain permitted adversarial behaviours.

This is done technically by introducing the concept of *support functions*, inspired by the support predicates recently introduced by [8] but extending them to cater for a wider range of situations. Here the core idea is that a support function operates on the transcript of messages and ciphertexts sent and received (in both directions) and its output is used to decide whether an adversarial behaviour – say, dropping or reordering messages – counts as a "win" in the security games. It is also used to define a suitable correctness notion with respect to expected behaviours of the channel.

As a final feature, our secure channel definitions allow the adversary complete control over all randomness used by the two parties, since we can achieve security against such a strong adversary in the stateful setting. This decision reflects a concern about Telegram clients expressed by Telegram developers [9].

Formal model of MTProto: In Section IV, we provide a detailed formal model of Telegram's symmetric encryption. Our model is computational and does not abstract away the building blocks used in Telegram. This in itself is a non-trivial task as no formal specification exists and behaviour can only be derived from official (but incomplete) documentation and from observation; moreover different clients do not have the same behaviour.

Formally, we define an MTProto-based bidirectional channel MTP-CH as a composition of multiple cryptographic primitives. This allows us to recover a variant of the real-world MTProto protocol by instantiating the primitives with specific constructions, and to study whether each of them satisfies the security notions that are required in order to achieve the desired security of MTP-CH. This allows us to work at two different levels of abstraction, and significantly simplifies the analysis. However, we emphasise that our goal is to be descriptive, not prescriptive, i.e. we do not suggest alternative instantiations of MTP-CH.

To arrive at our model, we had to make several decisions on what behaviour to model and where to draw the line of abstraction. Notably, there are various behaviours exhibited by (official) Telegram implementations that lead to attacks.

In particular, we verified in practice that current implementations allow an attacker on the network to reorder messages from a client to the server, with the transcript on the client being updated later to reflect the attacker-altered server's view. We stress, though, that this trivial yet practical attack is not inherent in MTProto and can be avoided by updating the processing of message metadata in Telegram's servers.

Further, if a message is not acknowledged within a certain time in MTProto, it is resent using the same metadata and with fresh random padding. While this appears to be a useful feature and a mitigation against message deletion, it would actually enable an attack in our formal model if such retransmissions were included. In particular, an adversary who also has control over the randomness can break stateful IND-CPA security with 2 encryption queries, while an attacker without that control could do so with about 2^{64} encryption queries. We use these more theoretical attacks to motivate our decision not to allow re-encryption with fixed metadata in our formal model of MTProto, i.e. we insist that the state is evolving.

Proof of security: We then claim in Section V that our slight variant of MTProto achieves channel confidentiality and integrity in our model, under certain assumptions on the components used in its construction. As described in Section I-B, Telegram has implemented our proposed alterations so that there can be some assurances about MTProto as currently deployed.[1]

We use code-based game hopping proofs in which the analysis is modularised into a sequence of small steps that can be individually verified. As well as providing all details of the proofs (in the full version), we also give high-level intuitions. Significant complexity arises in the proofs from two sources: the entanglement of keys used in the two channel directions, and the detailed nature of the model of MTProto that we use (so that our proof rules out as many attacks as possible).

We eschew an asymptotic approach in favour of concrete security analysis. This results in security theorems that quantitatively relate the confidentiality and integrity of MTProto as a secure channel to the security of its underlying cryptographic components. Our main security results, Theorems 1 and 2 and Corollaries 1 and 2, provide confidentiality and integrity bounds containing terms equivalent to $\approx q/2^{64}$ where q is the number of queries an attacker makes. We discuss this further in Section V.

However, our security proofs rely on several assumptions about cryptographic primitives that, while plausible, have not been considered in the literature. In more detail, due to the way Telegram makes use of SHA-256 as a MAC algorithm and as a KDF, we have to rely on the novel assumption that the block cipher SHACAL-2 underlying the SHA-256 compression function is a leakage-resilient PRF under related-key attacks, where "leakage-resilient" means that the adversary can choose a part of the key. Our proofs rely on two distinct variants of such an assumption. These assumptions hold in the ideal cipher model, but further cryptanalysis is needed to validate them for SHACAL-2. For similar reasons, we also require a dual-PRF assumption of SHACAL-2. We stress that such assumptions are likely necessary for our or any other computational security proofs for MTProto. This is due to the specifics of how MTProto uses SHA-256 and how it constructs keys and tags from public inputs and overlapping key bits of a master secret. Given the importance of Telegram, these assumptions provide new, significant cryptanalysis targets as well as motivate further research on related-key attacks. Our proofs side-step concerns about length-extension attacks by relying on the MTProto plaintext encoding format which mandates the presence of certain metadata in the first block of the encrypted payload.

Attacks: We present further implementation attacks against Telegram in Section VI and Appendix A. These attacks highlight the limits of our formal modelling and the fragility of MTProto implementations. The first of these, a timing attack against Telegram's use of IGE mode encryption, can be avoided by careful implementation, but we found multiple vulnerable clients.[2] The attack takes inspiration from an attack on SSH [12]. It exploits that Telegram encrypts a length field and checks integrity of plaintexts rather than ciphertexts. If this process is not implemented whilst taking care to avoid a timing side channel, it can be turned into an attack recovering up to 32

[1]Clients still differ in their implementation of the protocol and in particular in payload validation, which our model does not capture.

[2]We note that Telegram's TDLib [10] library manages to avoid this leak [11].

bits of plaintext. We give an example from the official Desktop Telegram client in Section VI and treat the Android and iOS clients in the full version of this work. However, we stress that the conditions of this attack are difficult to meet in practice. In particular, to recover bits from a plaintext message block m_i we assume knowledge of message block m_{i-1} (we consider this a relatively mild assumption) and, critically, message block m_1 which contains two 64-bit random values negotiated between client and server. Thus, confidentiality hinges on the secrecy of two random strings – a salt and an id. Notably, these fields were not designated for this purpose in the Telegram documentation.

In order to recover m_1 and thereby enable our plaintext-recovery attack, in Appendix A we chain it with another attack on the server-side implementation of Telegram's key exchange protocol. This attack exploits how Telegram servers process RSA ciphertexts. While the exploited behaviour was confirmed by the Telegram developers, we did not verify it with an experiment.[3] This attack actually breaks server authentication – allowing a MiTM attack – assuming the attack can be completed before a session times out. But, more germanely, it also allows us to recover the id field. This essentially reduces the overall security of Telegram to guessing the 64-bit salt field. We give a sketch in Appendix A and details in the full version. We stress, though, that even if all assumptions we make are met, our exploit chain – while being considerably cheaper than breaking the underlying `AES-256` encryption – is far from practical. Yet, it demonstrates the fragility of MTProto, which could be avoided – along with unstudied assumptions – by relying on standard authenticated encryption or, indeed, just using TLS.

We conclude with a broader discussion of Telegram security and with our recommendations in Section VII.

B. Disclosure

We notified Telegram's developers about the vulnerabilities we found in MTProto on 16 April 2021. They acknowledged receipt soon after and the behaviours we describe on 8 June 2021. They awarded a bug bounty for the timing side channel and for the overall analysis. We were informed by the Telegram developers that they do not do security or bugfix releases except for immediate post-release crash fixes. The development team also informed us that they did not wish to issue security advisories at the time of patching nor commit to release dates for specific fixes. Therefore, the fixes were rolled out as part of regular Telegram updates. The Telegram developers informed us that as of version 7.8.1 for Android, 7.8.3 for iOS and 2.8.8 for Telegram Desktop all vulnerabilities reported here were addressed. When we write "the current version of MTProto" or "current implementations", we refer to the versions prior to those version numbers, i.e. the versions we analysed.

II. Preliminaries

A. Notational conventions

1) Basic notation: Let $\mathbb{N} = \{1, 2, \ldots\}$. For $i \in \mathbb{N}$ let $[i]$ be the set $\{1, \ldots, i\}$. We denote the empty string by ε, the empty set by $\emptyset$, and the empty tuple by $()$. We let $x_1 \leftarrow x_2 \leftarrow v$ denote assigning the value v to both x_1 and x_2. Let $x \in \{0,1\}^*$ be any string; then $|x|$ denotes its bit-length, $x[i]$ denotes its i-th bit for $0 \leq i < |x|$, and $x[a : b] = x[a] \ldots x[b-1]$ for $0 \leq a < b \leq |x|$. For any $x \in \{0,1\}^*$ and $\ell \in \mathbb{N}$ such that $|x| \leq \ell$, we write $\langle x \rangle_\ell$ to denote the bit-string of length ℓ that is built by padding x with leading zeros. For any two strings $x, y \in \{0,1\}^*$, $x \parallel y$ denotes their concatenation. If X is a finite set, we let $x \leftarrow\$\, X$ denote picking an element of X uniformly at random and assigning it to x. If T is a table, $\mathsf{T}[i]$ denotes the element of the table that is indexed by i. We use `int64` as a shorthand for a 64-bit integer data type. We use `0x` to prefix a hexadecimal string in big-endian order. All variables are represented in big-endian unless specified otherwise. The symbol $\perp \notin \{0,1\}^*$ denotes an empty table position or an error code that indicates rejection, such as invalid input to an algorithm. Uninitialised integers are assumed to be initialised to 0, Booleans to false, strings to ε, sets to $\emptyset$, tuples to $()$, and tables are initially empty.

2) Algorithms and adversaries: Algorithms may be randomised unless otherwise indicated. Running time is worst case. If A is an algorithm, $y \leftarrow A(x_1, \ldots ; r)$ denotes running A with random coins r on inputs $x_1, \ldots$ and assigning the output to y. If any of inputs taken by A is $\perp$, then all of its outputs are $\perp$. We let $y \leftarrow\$\, A(x_1, \ldots)$ be the result of picking r at random and letting $y \leftarrow A(x_1, \ldots ; r)$. We let $[A(x_1, \ldots)]$ denote the set of all possible outputs of A when invoked with inputs $x_1, \ldots$. Adversaries are algorithms. We require that adversaries never pass $\perp$ as input to their oracles.

3) Security games and reductions: We use the code-based game-playing framework of [13]. $\Pr[\mathrm{G}]$ denotes the probability that game G returns true. Variables in each game are shared with its oracles. In the security reductions, we omit specifying the running times of the constructed adversaries when they are roughly the same as the running time of the initial adversary.

B. Standard definitions

1) Collision-resistant functions: Let $f\colon \mathcal{D}_f \to \mathcal{R}_f$ be a function. Consider game G^{cr} of Fig. 1, defined for f and an adversary $\mathcal{F}$. The advantage of $\mathcal{F}$ in breaking the CR-security of f is defined as $\mathsf{Adv}^{\mathsf{cr}}_f(\mathcal{F}) = \Pr[\mathrm{G}^{\mathsf{cr}}_{f,\mathcal{F}}]$. To win the game, adversary $\mathcal{F}$ has to find two distinct inputs $x_0, x_1 \in \mathcal{D}_f$ such that $f(x_0) = f(x_1)$. Note that f is *unkeyed*, so there exists a trivial adversary $\mathcal{F}$ with $\mathsf{Adv}^{\mathsf{cr}}_f(\mathcal{F}) = 1$ whenever f is not injective. We will use this notion in a constructive way, to build a specific collision-resistance adversary $\mathcal{F}$ (for $f = \mathsf{SHA\text{-}256}$ with a truncated output) in a security reduction.

```
Game $\mathrm{G}^{\mathsf{cr}}_{f,\mathcal{F}}$
$(x_0, x_1) \leftarrow\$\, \mathcal{F}$ ; Return $(x_0 \neq x_1) \wedge (f(x_0) = f(x_1))$
```

Figure 1: Collision-resistance of function f.

2) Function families: A family of functions F specifies a deterministic algorithm $\mathsf{F.Ev}$, a key set $\mathsf{F.Keys}$, an input set $\mathsf{F.In}$ and an output length $\mathsf{F.ol} \in \mathbb{N}$. $\mathsf{F.Ev}$ takes a function

[3]Verification would require sending a significant number of requests to the Telegram servers from a geographically close host.

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

3) 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 $\mathsf{E.Ev}$) an inverse algorithm $\mathsf{E.Inv}\colon \{0,1\}^{\mathsf{E.ol}} \to \mathsf{E.In}$ such that $\mathsf{E.Inv}(ek, \mathsf{E.Ev}(ek, x)) = x$ for all $ek \in \mathsf{E.Keys}$ and all $x \in \mathsf{E.In}$. We refer to $\mathsf{E.ol}$ as the block length of E. Our pictures and attacks use E_K and E_K^{-1} as a shorthand for $\mathsf{E.Ev}(ek, \cdot)$ and $\mathsf{E.Inv}(ek, \cdot)$ respectively.

4) One-time PRF-security of function family: Consider game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$ of Fig. 2, defined for a function family F and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the OTPRF-security of F is defined as $\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{F}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}] - 1$. The game samples a uniformly random challenge bit b and runs adversary $\mathcal{D}$, providing it with access to oracle RoR. The oracle takes $x \in \mathsf{F.In}$ as input, and the adversary is allowed to query the oracle arbitrarily many times. Each time RoR is queried on any x, it samples a uniformly random key fk from $\mathsf{F.Keys}$ and returns either $\mathsf{F.Ev}(fk, x)$ (if $b = 1$) or a uniformly random element from $\{0,1\}^{\mathsf{F.ol}}$ (if $b = 0$). $\mathcal{D}$ wins if it returns a bit b' that is equal to the challenge bit.

```
Game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$                         $\underline{\mathrm{RoR}(x)}$   // $x \in \mathsf{F.In}$
$b \leftarrow_\$ \{0,1\}$ ; $b' \leftarrow_\$ \mathcal{D}^{\mathrm{RoR}}$    $fk \leftarrow_\$ \mathsf{F.Keys}$ ; $y_1 \leftarrow \mathsf{F.Ev}(fk, x)$
Return $b' = b$                               $y_0 \leftarrow_\$ \{0,1\}^{\mathsf{F.ol}}$ ; Return $y_b$
```

Figure 2: One-time PRF-security of function family F.

5) Symmetric encryption schemes: A symmetric encryption scheme SE specifies algorithms $\mathsf{SE.Enc}$ and $\mathsf{SE.Dec}$, where $\mathsf{SE.Dec}$ is deterministic. Associated to SE is a key length $\mathsf{SE.kl} \in \mathbb{N}$, a message space $\mathsf{SE.MS} \subseteq \{0,1\}^* \setminus \{\varepsilon\}$, and a ciphertext length function $\mathsf{SE.cl}\colon \mathbb{N} \to \mathbb{N}$. The encryption algorithm $\mathsf{SE.Enc}$ takes a key $k \in \{0,1\}^{\mathsf{SE.kl}}$ and a message $m \in \mathsf{SE.MS}$ to return a ciphertext $c \in \{0,1\}^{\mathsf{SE.cl}(|m|)}$. We write $c \leftarrow_\$ \mathsf{SE.Enc}(k, m)$. The decryption algorithm $\mathsf{SE.Dec}$ takes k, c to return message $m \in \mathsf{SE.MS} \cup \{\bot\}$, where $\bot$ denotes incorrect decryption. We write $m \leftarrow \mathsf{SE.Dec}(k, c)$. Decryption correctness requires that $\mathsf{SE.Dec}(k, c) = m$ for all $k \in \{0,1\}^{\mathsf{SE.kl}}$, all $m \in \mathsf{SE.MS}$, and all $c \in [\mathsf{SE.Enc}(k, m)]$. We say that SE is deterministic if $\mathsf{SE.Enc}$ is deterministic.

6) One-time indistinguishability of SE: Consider game $\mathrm{G}^{\mathsf{otind\$}}$ of Fig. 3, defined for a deterministic symmetric encryption scheme SE and an adversary $\mathcal{D}$. We define the advantage of $\mathcal{D}$ in breaking the OTIND\$-security of SE as $\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}}] - 1$. The game proceeds as the OTPRF game.

7) IGE block cipher mode of operation: Let E be a block cipher. Define the Infinite Garble Extension (IGE) mode of operation as $\mathsf{SE} = \mathsf{IGE}[\mathsf{E}]$ as in Fig. 4, where key length is $\mathsf{SE.kl} = \mathsf{E.kl} + 2 \cdot \mathsf{E.ol}$, the message space $\mathsf{SE.MS} = \bigcup_{t \in \mathbb{N}} \{0,1\}^{\mathsf{E.ol} \cdot t}$ consists of messages whose lengths are multiples of the block length, and the ciphertext length function $\mathsf{SE.cl}$ is the identity function. IGE was first defined

```
Game $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}}$                        $\underline{\mathrm{RoR}(m)}$   // $m \in \mathsf{SE.MS}$
$b \leftarrow_\$ \{0,1\}$ ; $b' \leftarrow_\$ \mathcal{D}^{\mathrm{RoR}}$   $k \leftarrow_\$ \{0,1\}^{\mathsf{SE.kl}}$ ; $c_1 \leftarrow \mathsf{SE.Enc}(k, m)$
Return $b' = b$                              $c_0 \leftarrow_\$ \{0,1\}^{\mathsf{SE.cl}(|m|)}$ ; Return $c_b$
```

Figure 3: One-time real-or-random indistinguishability of deterministic symmetric encryption scheme SE.

in [14] without proof of security. Attacks relying on key/IV reuse were described in [15], [16]. Fig. 4 is somewhat nonstandard, as it includes the IV (c_0, m_0) as part of the key material. However, in this work, we only require one-time security of SE, so keys and IVs are generated together and the IV is not included as part of the ciphertext.

```
$\underline{\mathsf{IGE[E].Enc}(k, m)}$                       $\underline{\mathsf{IGE[E].Dec}(k, c)}$
For $i = 1, \ldots, t$ do                       For $i = 1, \ldots, t$ do
    $c_i \leftarrow \mathsf{E.Ev}(K, m_i \oplus c_{i-1})$          $m_i \leftarrow \mathsf{E.Inv}(K, c_i \oplus m_{i-1})$
        $\oplus m_{i-1}$                                 $\oplus c_{i-1}$
Return $c_1 \| \ldots \| c_t$                       Return $m_1 \| \ldots \| m_t$
```

Figure 4: Construction of $\mathsf{IGE[E]}$ as SE from block cipher E. Let t be the number of blocks of m (or c), i.e. $m = m_1 \| \ldots \| m_t$. Parse $K \| c_0 \| m_0 \leftarrow k$ where $|K| = \mathsf{E.kl}$, $|c_0| = |m_0| = \mathsf{E.ol}$.

8) SHA hash functions: Let $\mathsf{SHA\text{-}1} : \{0,1\}^* \to \{0,1\}^{160}$ and $\mathsf{SHA\text{-}256} : \{0,1\}^* \to \{0,1\}^{256}$ be the hash functions of [17] and let $h_{160} : \{0,1\}^{160} \times \{0,1\}^{512} \to \{0,1\}^{160}$ and $h_{256} : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$ be their compression functions. Let $\mathsf{SHACAL\text{-}1}$ [18] be the block cipher defined by $\mathsf{SHACAL\text{-}1.kl} = 512$, $\mathsf{SHACAL\text{-}1.ol} = 160$ and $h_{160}(k, x) = k \mathbin{\hat{+}} \mathsf{SHACAL\text{-}1.Ev}(x, k)$, where $\hat{+}$ is a modular addition over 32-bit words. Let $\mathsf{SHACAL\text{-}2}$ be the block cipher defined by $\mathsf{SHACAL\text{-}2.kl} = 512$, $\mathsf{SHACAL\text{-}2.ol} = 256$ and $h_{256}(k, x) = k \mathbin{\hat{+}} \mathsf{SHACAL\text{-}2.Ev}(x, k)$.

III. Bidirectional channels

A. Our formal model in context of prior work

We model Telegram's MTProto protocol as a bidirectional cryptographic channel. A *channel* provides a method for two users to exchange messages, and it is *bidirectional* [19] when both users can send and receive messages. There is a significant body of prior work on primitives that can be thought of as special cases of a bidirectional channel, building on the early work of [20] which introduced stateful security notions for symmetric encryption and used them to analyse SSH. MTProto uses distinct but related secret keys to send messages in the opposite directions on the channel, so the simpler primitives are not sufficient for our analysis.

MTProto cryptographically enforces a complex set of rules regarding the order in which messages can be decrypted, allowing out-of-order delivery. Channels are normally required to satisfy the strongest possible integrity notion, ensuring strict in-order delivery. But some prior work considers relaxed integrity requirements, defining security notions that permit message replay, reordering, or omission [20], [21], [22]. Fine-grained message delivery rules are captured in [23]. A more powerful framework for *robust channels* is defined in [8]. None of this work targets bidirectional channels.

We extend the framework of [8], lifting it to the bidirectional setting. Most notably, our framework uses more information to make the support decisions. These decisions are based on per-user communication transcripts. For each sent or received ciphertext, a user's transcript includes a plaintext-ciphertext pair, where one of them can be $\perp$ to denote a failure. Keeping track of failures allows us to capture fine-grained notions of robustness; keeping track of plaintexts allows to define simpler security definitions. In the full version of this work we provide a detailed comparison between our framework and that of [8].

B. Syntax of channels

We refer to the two users of a channel as $\mathcal{I}$ and $\mathcal{R}$. These will map to client and server in the setting of MTProto. We use $u \in \{\mathcal{I}, \mathcal{R}\}$ as a variable to represent an arbitrary user and $\overline{u}$ to represent the other user, meaning $\overline{u}$ denotes the sole element of $\{\mathcal{I}, \mathcal{R}\} \setminus \{u\}$. We use st_u to represent the internal state of user u.

Definition 1. *A channel* CH *specifies algorithms* $\mathsf{CH.Init}$, $\mathsf{CH.Send}$ *and* $\mathsf{CH.Recv}$*, where* $\mathsf{CH.Recv}$ *is deterministic. Associated to* CH *is a plaintext space* $\mathsf{CH.MS}$ *and a randomness space* $\mathsf{CH.SendRS}$ *of* $\mathsf{CH.Send}$*. The initialisation algorithm* $\mathsf{CH.Init}$ *returns* $\mathcal{I}$*'s and* $\mathcal{R}$*'s initial states* $st_{\mathcal{I}}$ *and* $st_{\mathcal{R}}$*. The sending algorithm* $\mathsf{CH.Send}$ *takes* st_u *for some* $u \in \{\mathcal{I}, \mathcal{R}\}$*, a plaintext* $m \in \mathsf{CH.MS}$*, and auxiliary information* aux *to return the updated state* st_u *and a ciphertext* c*, where* $c = \perp$ *may be used to indicate a failure to send. We may surface random coins* $r \in \mathsf{CH.SendRS}$ *as an additional input to* $\mathsf{CH.Send}$*. The receiving algorithm* $\mathsf{CH.Recv}$ *takes* st_u*,* c *and auxiliary information* aux *to return the updated state* st_u *and a plaintext* $m \in \mathsf{CH.MS} \cup \{\perp\}$*, where* $\perp$ *indicates a failure to recover a plaintext. The syntax used for the algorithms of* CH *is given in Fig. 5.*

$$\boxed{\begin{array}{l}(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{CH.Init}() \\ (st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m, aux; r) \\ (st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)\end{array}}$$

Figure 5: Syntax of the constituent algorithms of channel CH.

The abstract auxiliary information field aux will be used to associate timestamps to each sent and received message. It should not be thought of as an associated data that needs to be authenticated; we do not model associated data.

We define a support transcript to represent a record of all messages sent and received by a single user on a channel. Each transcript entry includes a plaintext m and a label (denoted by label); we use labels to distinguish between exchanged user messages that encrypt or encode different plaintexts. Depending on the level of abstraction, for any m we will use the corresponding ciphertext or message encoding as its label.[4] But we will make use only of the equality patterns that arise between labels, not of the exact values. Transcripts can include entries with plaintexts $m = \perp$ to capture that a received message was rejected. This allows us to model a range of channel behaviours in the event of an error (from terminating after the first error to full recovery). Transcript entries can also include $\mathsf{label} = \perp$, e.g. to indicate that a plaintext could not be sent over a terminated channel.

Definition 2. *A support transcript* tr_u *for user* $u \in \{\mathcal{I}, \mathcal{R}\}$ *is a list of entries of the form* $(\mathsf{op}, m, \mathsf{label}, aux)$*, where* $\mathsf{op} \in \{\mathsf{sent}, \mathsf{recv}\}$*. An entry with* $\mathsf{op} = \mathsf{sent}$ *indicates that user* u *attempted to send a message that encrypts or encodes plaintext* m *with auxiliary information* aux*, associated to* label*. An entry with* $\mathsf{op} = \mathsf{recv}$ *indicates that user* u *received a message associated to* label *with auxiliary information* aux*, and used it to recovered plaintext* m*.*

We define a support function supp that uses user support transcripts to determine whether a user $u \in \{\mathcal{I}, \mathcal{R}\}$ should accept an incoming message from $\overline{u}$ that is associated to label. If the message should be accepted, then supp must return a plaintext m^* to indicate that u is expected to recover m^* from the incoming message; otherwise supp must return $\perp$ to indicate that the message is expected to be rejected. We also let supp take the auxiliary information aux as input so that timestamps can be captured in our definitions.

Definition 3. *A support function* supp *is a function with syntax* $\mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux) \to m^*$ *where* $u \in \{\mathcal{I}, \mathcal{R}\}$*, and* tr_u*,* $\mathsf{tr}_{\overline{u}}$ *are support transcripts for users* u *and* $\overline{u}$*. It indicates that, according to the transcripts, user* u *is expected to recover plaintext* m^* *from the incoming message that is associated to* label *with auxiliary information* aux*.*

A support function does not take a channel's state information as input, so it can only rely on equality patterns between labels across the transcripts of both users. This is sufficient to specify message delivery rules that can capture attempted forgeries, replays, reordering and omissions.[5] Thus we will use support functions to specify the permissible adversarial behaviour on the network that should be supported by a channel.

In the full version of this work we formalise two correctness properties of a support function supp, but we do not mandate that they must always be met. Both properties were also considered in [8]. The *order correctness* requires that in-order delivery is supported in either direction if each message is assigned a distinct label.[6] The *integrity* of supp requires that it always returns $\perp$ if the queried label does not appear in $\mathsf{tr}_{\overline{u}}$.

C. Correctness and security of channels

For the following properties, consider the games in Fig. 6. We allow the adversary to control the randomness used by $\mathsf{CH.Send}$. We show our games to be equivalent to an authenticated encryption style security notion for channels in the full version of this work.

[4]This will be a ciphertext c when channel security notions are considered. This will be a message encoding p when properties of the message encoding schemes (defined in Section III-D) are considered.

[5]For example, the supp. function in Fig. 23 mandates strict in-order delivery.

[6][8] defines this notion as a part of the channel correctness game. We note that this notion cannot be met by some non-robust channels, e.g. those that close the connection once a number of errors occur.

```
Game $G^{corr}_{CH,supp,\mathcal{F}}$
  $win \leftarrow false$ ; $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ CH.Init()$
  $\mathcal{F}^{Send,Recv}(st_{\mathcal{I}}, st_{\mathcal{R}})$ ; Return $win$
Send$(u, m, aux, r)$
  $(st_u, c) \leftarrow CH.Send(st_u, m, aux; r)$
  $tr_u \leftarrow tr_u \,\|\, (sent, m, c, aux)$ ; Return $c$
Recv$(u, c, aux)$
  $m^* \leftarrow supp(u, tr_u, tr_{\overline{u}}, c, aux)$
  If $m^* = \bot$ then return $\bot$
  $(st_u, m) \leftarrow CH.Recv(st_u, c, aux)$
  $tr_u \leftarrow tr_u \,\|\, (recv, m, c, aux)$
  If $m^* \neq m$ then $win \leftarrow true$
  Return $m$
```

```
Game $G^{int}_{CH,supp,\mathcal{F}}$
  $win \leftarrow false$ ; $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ CH.Init()$
  $\mathcal{F}^{Send,Recv}$ ; Return $win$
Send$(u, m, aux, r)$
  $(st_u, c) \leftarrow CH.Send(st_u, m, aux; r)$
  $tr_u \leftarrow tr_u \,\|\, (sent, m, c, aux)$ ; Return $c$
Recv$(u, c, aux)$
  $(st_u, m) \leftarrow CH.Recv(st_u, c, aux)$
  $m^* \leftarrow supp(u, tr_u, tr_{\overline{u}}, c, aux)$
  $tr_u \leftarrow tr_u \,\|\, (recv, m, c, aux)$
  If $m \neq m^*$ then $win \leftarrow true$
  Return $m$
```

```
Game $G^{ind}_{CH,\mathcal{D}}$
  $b \leftarrow\$ \{0,1\}$ ; $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ CH.Init()$
  $b' \leftarrow\$ \mathcal{D}^{Ch,Recv}$ ; Return $b' = b$
Ch$(u, m_0, m_1, aux, r)$
  If $|m_0| \neq |m_1|$ then return $\bot$
  $(st_u, c) \leftarrow CH.Send(st_u, m_b, aux; r)$
  Return $c$
Recv$(u, c, aux)$
  $(st_u, m) \leftarrow CH.Recv(st_u, c, aux)$
  Return $\bot$
```

Figure 6: Correctness of channel CH; integrity of channel CH; indistinguishability of channel CH.

1) Correctness: Consider adversary $\mathcal{F}$ in game $G^{corr}_{CH,supp,\mathcal{F}}$ associated to a channel CH and a support function supp. The advantage of $\mathcal{F}$ in breaking the correctness of CH with respect to supp is defined as $Adv^{corr}_{CH,supp}(\mathcal{F}) = \Pr[G^{corr}_{CH,supp,\mathcal{F}}]$. The game initialises users $\mathcal{I}$ and $\mathcal{R}$. The adversary is given their initial states and gets access to a sending oracle Send and to a receiving oracle Recv. Calling Send(u, m, aux, r) encrypts the plaintext m with auxiliary data aux and randomness r from user u to the other user $\overline{u}$; the resulting tuple $(sent, m, c, aux)$ is added to the sender's transcript tr_u. Recv can only be called on honestly produced ciphertexts, meaning it exits when supp returns $m^* \neq \bot$. Calling Recv(u, c, aux) thus recovers the plaintext m^* from the support function, decrypts the corresponding ciphertext c and adds $(recv, m, c, aux)$ to the receiver's transcript tr_u; the game verifies that the recovered plaintext m is equal to the originally encrypted plaintext m^*. If the adversary can cause the channel to output a different m, then the adversary wins. This game captures the *minimal* requirement one would expect from a communication channel: honestly sent ciphertexts should decrypt to the correct plaintexts. It is similar in spirit to the correctness game of [8].

2) Integrity: Consider adversary $\mathcal{F}$ in game $G^{int}_{CH,supp,\mathcal{F}}$ associated to a channel CH and a support function supp. The advantage of $\mathcal{F}$ in breaking the integrity of CH with respect to supp is defined as $Adv^{int}_{CH,supp}(\mathcal{F}) = \Pr[G^{int}_{CH,supp,\mathcal{F}}]$. The adversary gets access to oracles Send and Recv (but not to the users' states). Both calls proceed as in the correctness game except that now Recv does not limit $\mathcal{F}$ to querying only honestly produced ciphertexts. This captures the intuition that the adversary can manipulate ciphertexts on the network in an attempt to create a forgery. Take $supp(u, tr_u, tr_{\overline{u}}, c, aux)$ that returns m^* iff $(sent, m^*, c, aux) \in tr_{\overline{u}}$, and returns $\bot$ otherwise. Then integrity with respect to supp mandates that a conventional ciphertext forgery is impossible, but all ciphertext replays, reordering, and omissions are permitted by the channel.

3) Confidentiality: Consider adversary $\mathcal{D}$ in game $G^{ind}_{CH,\mathcal{D}}$ associated to a channel CH. The advantage of $\mathcal{D}$ in breaking the IND-security of CH is defined as $Adv^{ind}_{CH}(\mathcal{D}) = 2 \cdot \Pr[G^{ind}_{CH,\mathcal{D}}] - 1$. The adversary can query the challenge oracle Ch(u, m_0, m_1, aux, r) as an encryption oracle for user u with two plaintexts m_0, m_1 of the same size, auxiliary information aux and randomness r, to obtain the ciphertext c that encrypts m_b. The adversary wins if it can guess the challenge bit b. The game also contains a Recv oracle. It is needed to model that each user's state st_u may be updated every time a ciphertext is processed, potentially influencing subsequent encryption operations. However, the Recv oracle does not return any information directly to $\mathcal{D}$.

D. Message encoding schemes

At its core, a channel can be expected to have a mechanism that handles encoding of plaintexts into payloads, and decoding of payloads back into plaintexts. Such a mechanism does not need to provide any security assurances, and can be intended for use over a communication channel that already guarantees integrity and confidentiality. We formalise it as a separate primitive called a *message encoding scheme*. It can then be composed with appropriate cryptographic primitives to build a cryptographic channel.

A modular approach leads to defining a syntax for message encoding that is similar to that of cryptographic channels. A message encoding scheme needs to have stateful encoding and decoding algorithms. Auxiliary information can be used to relay and verify information such as timestamps. One could expect all algorithms of a message encoding scheme to be deterministic; our definition uses randomness purely because it is necessary when modelling Telegram.

```
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ ME.Init()$
$(st_u, p) \leftarrow ME.Encode(st_u, m, aux; \nu)$
$(st_u, m) \leftarrow ME.Decode(st_u, p, aux)$
```

Figure 7: Syntax of message encoding scheme ME.

Definition 4. *A message encoding scheme* ME *specifies algorithms* ME.Init, ME.Encode *and* ME.Decode, *where* ME.Decode *is deterministic. Associated to* ME *is a plaintext space* $ME.MS \subseteq \{0,1\}^*$, *a payload space* ME.Out, *a randomness space* ME.EncRS *of* ME.Encode, *and a payload length function*

$\mathsf{ME.pl}\colon (\mathbb{N}\cup\{0\})\times\mathsf{ME.EncRS}\to\mathbb{N}$. *The initialisation algorithm* $\mathsf{ME.Init}$ *returns* $\mathcal{I}$*'s and* $\mathcal{R}$*'s initial states* $st_{\mathcal{I}}$ *and* $st_{\mathcal{R}}$*. The encoding algorithm* $\mathsf{ME.Encode}$ *takes* st_u *for* $u\in\{\mathcal{I},\mathcal{R}\}$*, a message* $m\in\mathsf{ME.MS}$*, and auxiliary information* aux *to return the updated state* st_u *and a payload* $p\in\mathsf{ME.Out}$.[7] *We may surface random coins* $\nu\in\mathsf{ME.EncRS}$ *as an additional input to* $\mathsf{ME.Encode}$*; then a message* m *should be encoded into a payload of length* $|p|=\mathsf{ME.pl}(|m|,\nu)$*. The decoding algorithm* $\mathsf{ME.Decode}$ *takes* st_u, p*, and auxiliary information* aux *to return the updated state* st_u *and a message* $m\in\mathsf{ME.MS}\cup\{\bot\}$*. The syntax used for the algorithms of* ME *is given in Fig. 7.*

A message encoding scheme needs to provide correctness-style properties and some form of non-cryptographic integrity. We expect it to arbitrate whether payloads that are sent over the channel can be silently replayed, reordered or omitted. In contrast, the cryptographic (non-encoding) parts of the channel can be expected to enforce that all received payloads were at some point honestly produced by the opposite user.

We define integrity of a message encoding scheme ME based on the security game in Fig. 8. The advantage of adversary $\mathcal{F}$ in breaking the EINT-security of ME with respect to supp is defined as $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{F})=\Pr[\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}]$. The two core differences from the corresponding channel notion in Section III-C are as follows. First, the message encoding scheme is meant to be run within an authenticated communication channel, so the $\textsc{Recv}$ oracle now starts by checking that the queried payload p was returned by a prior call to the opposite user's $\textsc{Send}$ oracle in response to some message m and auxiliary information aux. Second, the message encoding is not meant to serve any cryptographic purpose, meaning the initial states $st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}$ should not contain any secret information and are both given as inputs to adversary $\mathcal{F}$.

```
Game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$
$\mathsf{win}\leftarrow\mathsf{false}$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$\ \mathsf{ME.Init}()$
$\mathcal{F}^{\textsc{Send},\textsc{Recv}}(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$ ; Return $\mathsf{win}$

$\textsc{Send}(u,m,aux,r)$   // $u\in\{\mathcal{I},\mathcal{R}\}$, $m\in\mathsf{ME.MS}$, $r\in\mathsf{ME.EncRS}$
$(st_{\mathsf{ME},u}, p)\leftarrow\mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{tr}_u\leftarrow\mathsf{tr}_u \,\|\, (\mathsf{sent}, m, p, aux)$ ; Return $p$

$\textsc{Recv}(u,p,aux)$   // $u\in\{\mathcal{I},\mathcal{R}\}$, $p\in\mathsf{ME.Out}$
If $\nexists m', aux' : (\mathsf{sent}, m', p, aux')\in\mathsf{tr}_{\overline{u}}$ then return $\bot$
$(st_{\mathsf{ME},u}, m)\leftarrow\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
$m^*\leftarrow\mathsf{supp}(u,\mathsf{tr}_u,\mathsf{tr}_{\overline{u}}, p, aux)$
$\mathsf{tr}_u\leftarrow\mathsf{tr}_u \,\|\, (\mathsf{recv}, m, p, aux)$ ; If $m\neq m^*$ then $\mathsf{win}\leftarrow\mathsf{true}$
Return $m$
```

Figure 8: Integrity of message encoding scheme ME with respect to support function supp.

IV. Modelling MTProto 2.0

In this section, we describe our modelling of the MTProto 2.0 record protocol as a bidirectional channel. First, in Section IV-A we give an informal description of MTProto based on Telegram documentation and client implementations. Next, in Section IV-B we outline attacks that motivate protocol changes required to achieve security. We list further modelling issues and points where we depart from Telegram documentation in Section IV-C. We conclude with Section IV-D where we give our formal model for a fixed version of the protocol.

A. Telegram description

We studied MTProto 2.0 as described in the online documentation [24] and as implemented in the official desktop[8] and Android[9] clients. We focus on *cloud chats*. Figures 9 and 10 give a visual summary of the following description.

Key exchange: A Telegram client must first establish a symmetric 2048-bit auth_key with the server via a version of the Diffie-Hellman key exchange. We defer the details of the key exchange to the full version of this work. In practice, this key exchange first results in a permanent auth_key for each of the Telegram data centres the client connects to. Thereafter, the client runs a new key exchange on a daily basis to establish a temporary auth_key that is used instead of the permanent one.

"Record protocol": Messages are protected as follows.

1) API calls are expressed as functions in the TL schema [25].
2) The API requests and responses are serialised according to the type language (TL) [26] and embedded in the msg_data field of a payload p, shown in Table I. The first two 128-bit blocks of p have a fixed structure and contain various metadata. The maximum length of msg_data is 2^{24} bytes.
3) The payload is encrypted using AES-256 in IGE mode. The ciphertext c is a part of an MTProto ciphertext $\mathsf{auth\_key\_id}\,\|\,\mathsf{msg\_key}\,\|\,c$, where:

$$\begin{aligned}\mathsf{auth\_key\_id} &:= \mathsf{SHA\text{-}1}(\mathsf{auth\_key})[96:160]\\ \mathsf{msg\_key} &:= \mathsf{SHA\text{-}256}(\mathsf{auth\_key}[704+x:960+x]\,\|\,p)[64:192]\\ c &:= \mathsf{IGE[AES\text{-}256].Enc}(\mathsf{key}\,\|\,\mathsf{iv}, p)\end{aligned}$$

The IGE[AES-256] keys and IVs are computed via:

$$\begin{aligned}A &:= \mathsf{SHA\text{-}256}(\mathsf{msg\_key}\,\|\,\mathsf{auth\_key}[x:288+x])\\ B &:= \mathsf{SHA\text{-}256}(\mathsf{auth\_key}[320+x:608+x]\,\|\,\mathsf{msg\_key})\\ \mathsf{key} &:= A[0:64]\,\|\,B[64:192]\,\|\,A[192:256]\\ \mathsf{iv} &:= B[0:64]\,\|\,A[64:192]\,\|\,B[192:256]\end{aligned}$$

In the above steps, $x=0$ for messages from the client and $x=64$ from the server. Telegram clients use the BoringSSL implementation [27] of IGE, which has 2-block IVs.

4) MTProto ciphertexts are encapsulated in a "transport protocol". The MTProto documentation defines multiple such protocols [28], but the default is the *abridged* format that begins the stream with a fixed value of `0xefefefef` and then wraps each MTProto ciphertext c_{MTP} in a transport packet as:

- $\mathsf{length}\,\|\,c_{\mathsf{MTP}}$ where 1-byte length contains the c_{MTP} length divided by 4, if the resulting packet length is <127, or
- `0x7f` $\|\,\mathsf{length}\,\|\,c_{\mathsf{MTP}}$ where length is encoded in 3 bytes.

5) All the resulting packets are obfuscated by default using AES-128 in CTR mode. The key and IV are transmitted at

[7]For full generality, the algorithm ME.Encode could also be allowed to return $p=\bot$. However, the message encoding schemes we define in this work can never return $\bot$, so for simplicity we do not allow such output.

[8]https://github.com/telegramdesktop/tdesktop/, versions 2.3.2 to 2.7.1

[9]https://github.com/DrKLO/Telegram/, versions 6.1.1 to 7.6.0

the beginning of the stream, so the obfuscation provides no cryptographic protection and we ignore it henceforth.[10]
6) Communication is over TCP (port 443) or HTTP. Clients attempt to choose the best available connection. There is support for TLS in the client code, but it does not seem to be used.

In combination, these operations mean that MTProto 2.0 at its core uses a "stateful Encrypt & MAC" construction. Here the MAC tag msg_key is computed using SHA-256 with a prepended key derived from (certain bits of) auth_key. The key and IV for IGE mode are derived on a per-message basis using a KDF based on SHA-256, using certain bits of auth_key as the key-deriving key and the msg_key as a diversifier. Note that the bit ranges of auth_key used by the client and the server to derive keys in both operations overlap with one another. Any formal security analysis needs to take this into account.

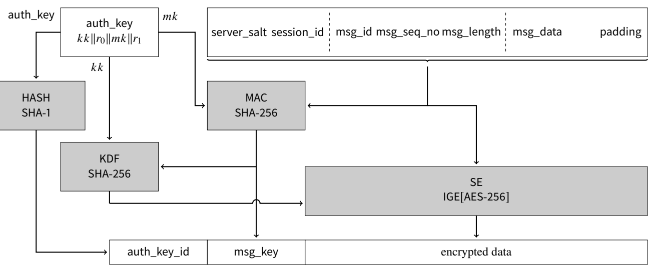


Figure 9: Overview of message processing in MTProto 2.0.

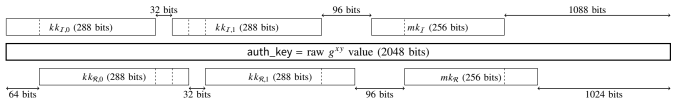

32 bits
96 bits
1088 bits
$kk_{\mathcal{I},0}$ (288 bits)
$kk_{\mathcal{I},1}$ (288 bits)
$mk_{\mathcal{I}}$ (256 bits)
auth_key = raw g^{xy} value (2048 bits)
$kk_{\mathcal{R},0}$ (288 bits)
$kk_{\mathcal{R},1}$ (288 bits)
$mk_{\mathcal{R}}$ (256 bits)
64 bits
32 bits
96 bits
1024 bits

Figure 10: Parsing auth_key in MTProto 2.0. User $u \in \{\mathcal{I}, \mathcal{R}\}$ derives a KDF key $kk_u = (kk_{u,0}, kk_{u,1})$ and a MAC key mk_u.

B. Attacks against MTProto metadata validation

We describe adversarial behaviours that are permitted in current Telegram implementations and that mostly depend on how clients and servers validate metadata information in the payload (especially the second 128-bit block containing msg_id, msg_seq_no and msg_length).

1) Reordering and deletion: We consider a network attacker that sits between the client and the Telegram servers, attempting to manipulate the conversation transcript. By *message* we mean any msg_data exchanged via MTProto, but we pay particular attention to when it contains a chat message.

a) Reordering: By reordering we mean that an adversary can swap messages sent by one party so that they are processed in the wrong order by the receiving party. Preventing such attacks is a basic property that one would expect in a secure

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.
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.

Table I: MTProto payload format.

messaging protocol. The MTProto documentation mentions reordering attacks as something to protect against in secret chats but does not discuss it for cloud chats [29]. The implementation of cloud chats provides some protection, but not fully:

- When the client is the receiver, the order of displayed chat messages is determined by the date and time values within the TL message object (which are set by the server), so adversarial reordering of packets has no effect on the order of chat messages as seen by the client. Service messages of MTProto typically do not have such a timestamp so reordering is theoretically possible, though with unclear impact.
- When the client is the sender, the order of chat messages can be manipulated because the server sets the date and time value for the Telegram user to whom the message was addressed based on when the server itself receives the message, and because the server will accept a message with a lower msg_id than that of a previous message as long as its msg_seq_no is also lower than that of a previous message. The server does not take the timestamp implicit within msg_id into account except to check whether it is at most 300s in the past or 30s in the future, so within this time interval reordering is possible. A message outside of this time interval is not ignored, but a request for time synchronisation is triggered, after receipt of which the client sends the message again with a fresh msg_id. So an attacker can also simply delay a chosen message to cause messages to be accepted out of order. In Telegram, the rotation of the server_salt every 30 to 60 minutes may be an obstacle to carrying out this attack in longer time intervals.

We have verified that reordering between a sending client and a receiving server is possible in practice using unmodified Android clients (v6.2.0) and a malicious WiFi access point running a TCP proxy [30] with custom rules to suppress and later release certain packets. Suppose an attacker sits between Alice and a server, and Alice is in a chat with Bob. The attacker can reorder messages that Alice is sending, so the server receives them in the wrong order and forwards them in the wrong order to Bob. While Alice's client will initially display her sent messages in the order she sent them, once it fetches history from the server it will update to display the

[10]This feature is meant to prevent ISP blocking. In addition to this, clients can route their connections through a Telegram proxy. The obfuscation key is then derived from a shared secret (e.g. from proxy password) between the client and the proxy.

modified order that will match that of Bob.

b) Deletion: MTProto makes it possible to silently drop a message both when the client is the sender[11] and when it is the receiver, but it is difficult to exploit in practice. Clients and the server attempt to resend messages for which did not get acknowledgements. Such messages have the same $\mathsf{msg\_ids}$ but are enclosed in a fresh ciphertext with random padding so the attacker must be able to distinguish the repeated encryptions to continue dropping the same payload. This is possible e.g. with the desktop client as sender, since padding length is predictable based on the message length [31]. When the client is a receiver, other message delivery mechanisms such as batching of messages inside a container or API calls like `messages.getHistory` make it hard for an attacker to identify repeated encryptions. So although MTProto does not prevent deletion in the latter case, there is likely no practical attack.

2) Re-encryption: If a message is not acknowledged within a certain time in MTProto, it is re-encrypted using the same $\mathsf{msg\_id}$ and with fresh random padding. While this appears to be a useful feature and a mitigation against message deletion, it enables attacks in the IND-CPA setting, as we explain next.

As a motivation, consider a local passive adversary that tries to establish whether $\mathcal{R}$ responded to $\mathcal{I}$ when looking at a transcript of three ciphertexts $(c_{\mathcal{I},0}, c_{\mathcal{R}}, c_{\mathcal{I},1})$, where c_u represents a ciphertext sent from u. In particular, it aims to establish whether $c_{\mathcal{R}}$ encrypts an automatically generated acknowledgement, denoted by "$\checkmark$", or a new message from $\mathcal{R}$. If $c_{\mathcal{I},1}$ is a re-encryption of the same message as $c_{\mathcal{I},0}$, re-using the state, this leaks that bit of information about $c_{\mathcal{R}}$.[12]

Suppose we have a channel CH that models the MTProto protocol as described in Section IV-A and uses the payload format given in Table I.[13] To sketch a model for acknowledgement messages for the purpose of explaining this attack, we define a special plaintext symbol $\checkmark$ that, when received, indicates acknowledgement for the last sent message. As in Telegram, $\checkmark$ messages are encrypted. Further, we model re-encryptions by insisting that if the $\mathsf{CH.Send}$ algorithm is queried again on an unacknowledged message m then $\mathsf{CH.Send}$ will produce another ciphertext c' for m using the same headers, including $\mathsf{msg\_id}$ and $\mathsf{msg\_seq\_no}$, as previously used. Critically, this means the same state in the form of $\mathsf{msg\_id}$ and $\mathsf{msg\_seq\_no}$ is used for two different encryptions.

[11]There are scenarios where deletion can be impactful. Telegram offers its users the ability to delete chat history for the other party (or all members of a group) – if such a request is dropped, severing the connection, the chat history will appear to be cleared in the user's app even though the request never made it to the Telegram servers (cf. [3] for the significance of history deletion in some settings).

[12]Note that here we are breaking the confidentiality of the ciphertext carrying "$\checkmark$". In addition to these encrypted acknowledgement messages, the underlying transport layer, e.g. TCP, may also issue unencrypted ACK messages or may resend ciphertexts as is. The difference between these two cases is that in the former case the acknowledgement message is encrypted, in the latter it is not. For completeness, note that Telegram clients do not resend cached ciphertext blobs when unacknowledged, but re-encrypt the underlying message under the same state but with fresh random padding.

[13]We give a formal definition of the channel in Section IV-D, but it is not necessary to outline the attack.

We use this behaviour to break the indistinguishability of an encrypted $\checkmark$. Consider the adversary given in Fig. 11. If $b = 0$, $c_{\mathcal{R},i}$ encrypts an $\checkmark$ and so $c_{\mathcal{I},i+1}$ will not be a re-encryption of m_0 under the same $\mathsf{msg\_id}$ and $\mathsf{msg\_seq\_no}$ that were used for $c_{\mathcal{I},i}$. In contrast, if $b = 1$, then we have $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ for some j, k, where $c^{(i)}$ denotes the i-th block of c, with probability 1 whenever $\mathsf{msg\_key}_j = \mathsf{msg\_key}_k$. This is true because the payloads of $c_{\mathcal{I},j}$ and $c_{\mathcal{I},k}$ share the same header fields, in particular including the $\mathsf{msg\_id}$ and $\mathsf{msg\_seq\_no}$ in the second block, encrypted under the same key. In the setting where the adversary controls the randomness of the padding, the condition $\mathsf{msg\_key}_j = \mathsf{msg\_key}_k$ can be made to always hold and thus $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ holds with probability 1. As a consequence two queries to the oracle suffice. When the adversary does not control the randomness, we can use the fact that $\mathsf{msg\_key}$ is computed via $\mathsf{SHA\text{-}256}$ truncated to 128 bits and the birthday bound applies for finding collisions. Thus after 2^{64} queries we expect a collision with constant probability (note that the adversary can check when a collision is found). Finally, in either setting, when $b = 0$ we have $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ with probability 0 since the underlying payloads differ, the key is the same and AES is a permutation for a fixed key.

```
Adversary $\mathcal{D}^{\text{CH},\text{RECV}}_{\text{IND},q}$
Let $aux = \varepsilon$. Choose any $m_0, m_1 \in \mathsf{CH.MS} \setminus \{\checkmark\}$.
Require $\forall i \in \mathbb{N}$: $r_{\mathcal{I},i}, r_{\mathcal{R},i} \in \mathsf{CH.SendRS}$.
For $i = 1, \ldots, q$ do
    $c_{\mathcal{I},i} \leftarrow \text{CH}(\mathcal{I}, m_0, m_0, aux, r_{\mathcal{I},i})$
    $c_{\mathcal{R},i} \leftarrow \text{CH}(\mathcal{R}, \checkmark, m_1, aux, r_{\mathcal{R},i})$ ; $\text{RECV}(\mathcal{I}, c_{\mathcal{R},i}, aux)$
If $\exists j \neq k$: $\mathsf{msg\_key}_j = \mathsf{msg\_key}_k$ then
    If $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ then return 1 else return 0
Else return $\perp$
```

Figure 11: Adversary against the IND-security of MTProto (modelled as channel CH) when permitting re-encryption under reused $\mathsf{msg\_id}$ and $\mathsf{msg\_seq\_no}$. If the adversary controls the randomness, then set $q = 2$ and choose $r_{\mathcal{I},0} = r_{\mathcal{I},1}$. Otherwise (i.e. all $r_{\mathcal{I},i}, r_{\mathcal{R},i}$ values are uniformly random) set $q = 2^{64}$. In this figure, let $\mathsf{msg\_key}_i$ be the $\mathsf{msg\_key}$ for $c_{\mathcal{I},i}$ and let $c^{(i)}$ be the i-th block of ciphertext c.

C. Modelling differences

In general, we would like our formal model of MTProto 2.0 to stay as close as possible to the real protocol, so that when we prove statements about the model, we obtain meaningful assurances about the security of the real protocol. However, as the previous section demonstrates, the current protocol has flaws. These prevent meaningful security analysis and can be removed by making small changes to the protocol's handling of metadata. Further, the protocol has certain features that make it less amenable to formal analysis. Here we describe the modelling decisions we have taken that depart from the current version of MTProto 2.0 and justify each change.

1) Inconsistency: There is no authoritative specification of the protocol. The Telegram documentation often differs from the implementations and the clients are not consistent with each other.[14] Where possible, we chose a sensible "default" choice from the observed set of possibilities, but we stress that it is in general impossible to create a formal specification of MTProto that would be valid for all current implementations. For instance, the documentation defines server_salt as "A (random) 64-bit number periodically (say, every 24 hours) changed (separately for each session) at the request of the server" [32]. In practice the clients receive salts that change every hour and which overlap with each other.[15] For client differences, consider padding generation: on desktop [31], a given message length will always result in the same padding length, whereas on Android [33], the padding length is randomised.

2) Application layer: Similarly, there is no clear separation between the cryptographic protocol of MTProto and the application data processing (expressed using the TL schema). However, to reason succinctly about the protocol we require a certain level of abstraction. In concrete terms, this means that we consider the msg_data field as "the message", without interpreting its contents and in particular without modelling TL constructors. However, this separation does not exist in implementations of MTProto – for instance, message encoding behaves differently for some constructors (e.g. container messages) – and so our model does not capture these details.

3) Client/server roles: The client and the server are not considered equal in MTProto. For instance, the server is trusted to timestamp TL messages for history, while the clients are not, which is why our reordering attacks only work in the client to server direction. The client chooses the session_id, the server generates the server_salt. The server accepts any session_id given in the first message and then expects that value, while the client checks the session_id but may accept any server_salt given.[16] Clients do not check the msg_seq_no field. The protocol implements elaborate measures to synchronise "bad" client time with server time, which includes: checks on the timestamp within msg_id as well as the salt, special service messages [35] and the resending of messages with regenerated headers. Since much of this behaviour is not critical for security, we model both parties of the protocol as equals. Expanding our model with this behaviour should be possible without affecting most of the proofs.

4) Key exchange: We are concerned with the symmetric part of the protocol, and thus assume that the shared auth_key is a uniformly random string rather than of the form $g^{ab} \bmod p$ resulting from the actual key exchange.

5) Bit mixing: MTProto uses specific bit ranges of auth_key as KDF and MAC inputs. These ranges do not overlap for different primitives (i.e. the KDF key inputs are wholly distinct from the MAC key inputs), and we model auth_key as a random value, so without loss of generality our model generates the KDF and MAC key inputs as separate random values. The key input ranges for the client and the server do overlap for KDF and MAC separately, however, so we model this in the form of related-key-deriving functions.

Further, the KDF intermixes specific bit ranges of the outputs of two SHA-256 calls to derive the encryption keys and IVs. We argue that this is unnecessary – the intermixed KDF output is indistinguishable from random (the usual security requirement of a key derivation function) if and only if the concatenation of the two SHA-256 outputs is indistinguishable from random. Hence in our model the KDF just returns the concatenation.

6) Order: Given that MTProto operates over reliable transport channels, it is not necessary to allow messages arriving out of order. Our model imposes stricter validation on metadata upon decryption via a single sequence number that is checked by both sides and only the next expected value is accepted. Enforcing strict ordering also automatically rules out replay and deletion attacks, which the implementation of MTProto as studied avoided in some cases only due to application-level processing.[17]

7) Re-encryption: Because of the attacks in Section IV-B2, we insist in our formalisation that all sent messages include a fresh value in the header. This is achieved via a stateful secure channel definition in which either a client or server sequence number is incremented on each call to the CH.Send oracle.

8) Message encoding: Some of the previous points outline changes to message encoding. We simplify the scheme, keeping to the format of Table I but not modelling diverging behaviours upon decoding. The implemented MTProto message encoding scheme behaves differently depending on whether the user is a client or a server, but each of them checks a 64-bit value in the first plaintext block, session_id and server_salt respectively. To prove security of the channel, it is enough that there is a single such value that both parties check, and it does not need to be randomised, so we model a constant session_id and we leave the salt as an empty field. We also merge the msg_id and msg_seq_no fields into a single sequence number field of corresponding size, reflecting that a simple counter suffices in place of the original fields. Note that though we only prove security with respect to this particular message encoding scheme, our modelling approach is flexible and can accommodate more complex message encoding schemes.

D. MTProto-based channel

Our model of the MTProto channel is given in Definition 5 and Fig. 12. We abstract the individual keyed primitives into function families.[18]

CH.Init generates the keys for both users and initialises the message encoding scheme. Note that auth_key as described

[14]Since the server code was not available, we inferred its behaviour from observing the communication.

[15]The documentation was updated in response to our paper.

[16]The Android client accepts any value in the place of server_salt, and the desktop client [34] compares it with a previously saved value and resends the message if they do not match and if the timestamp within msg_id differs from the acceptable time window.

[17]Secret chats implement more elaborate measures against replay/reordering [29], however this complexity is not required when in-order delivery is required for each direction separately.

[18]While the definition itself could admit many different implementations of the primitives, we are interested in modelling MTProto and thus do not define our channel in a fully general way, e.g. we fix some key sizes.

```
$\underline{\mathsf{CH.Init}()}$
$hk \leftarrow_\$ \{0,1\}^{\mathsf{HASH.kl}}$
$kk \leftarrow_\$ \{0,1\}^{672}$ ; $mk \leftarrow_\$ \{0,1\}^{320}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, kk \,\|\, mk)$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$\mathsf{key}_{\mathcal{I}} \leftarrow (kk_{\mathcal{I}}, mk_{\mathcal{I}})$
$\mathsf{key}_{\mathcal{R}} \leftarrow (kk_{\mathcal{R}}, mk_{\mathcal{R}})$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow_\$ \mathsf{ME.Init}()$
$st_{\mathcal{I}} \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_{\mathcal{I}}, \mathsf{key}_{\mathcal{R}}, st_{\mathsf{ME},\mathcal{I}})$
$st_{\mathcal{R}} \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_{\mathcal{R}}, \mathsf{key}_{\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
Return $(st_{\mathcal{I}}, st_{\mathcal{R}})$
```

```
$\underline{\mathsf{CH.Send}(st_u, m, aux; r)}$
$(\mathsf{auth\_key\_id}, \mathsf{key}_u, \mathsf{key}_{\overline{u}}, st_{\mathsf{ME}}) \leftarrow st_u$
$(kk_u, mk_u) \leftarrow \mathsf{key}_u$
$(st_{\mathsf{ME}}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME}}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
$k \leftarrow \mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
$st_u \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_u, \mathsf{key}_{\overline{u}}, st_{\mathsf{ME}})$
Return $(st_u, c)$
```

```
$\underline{\mathsf{CH.Recv}(st_u, c, aux)}$
$(\mathsf{auth\_key\_id}, \mathsf{key}_u, \mathsf{key}_{\overline{u}}, st_{\mathsf{ME}}) \leftarrow st_u$
$(kk_{\overline{u}}, mk_{\overline{u}}) \leftarrow \mathsf{key}_{\overline{u}}$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{auth\_key\_id} \neq \mathsf{auth\_key\_id}'$ then
    Return $(st_u, \perp)$
$k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
$\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
If $\mathsf{msg\_key}' \neq \mathsf{msg\_key}$ then return $(st_u, \perp)$
$(st_{\mathsf{ME}}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME}}, p, aux)$
$st_u \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_u, \mathsf{key}_{\overline{u}}, st_{\mathsf{ME}})$
Return $(st_u, m)$
```

Figure 12: Construction of MTProto-based channel $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$ from message encoding scheme ME, function families HASH, MAC and KDF, related-key-deriving functions ϕ_{MAC} and ϕ_{KDF}, and from deterministic symmetric encryption scheme SE.

in Section IV-A does not appear in the code in Fig. 12, since each part of auth_key that is used for keying the primitives can be generated independently. These parts are denoted by hk, kk and mk. The function ϕ_{KDF} (resp. ϕ_{MAC}) is then used to derive the (related) keys for each user from kk (resp. mk).

CH.Send proceeds by first using ME to encode a message m into a payload p. The MAC is computed on this payload to produce a msg_key, and the KDF is called on the msg_key to compute the key and IV for symmetric encryption SE, here abstracted as k. The payload is encrypted with SE using this key material, and the resulting ciphertext is called c_{se}. The CH ciphertext c consists of auth_key_id, msg_key and the symmetric ciphertext c_{se}.

CH.Recv reverses the steps by first computing k from the msg_key parsed from c, then decrypting c_{se} to the payload p, and recomputing the MAC of p to check whether it equals msg_key. If not, it returns $\perp$ (without changing the state) to signify failure. If the check passes, it uses ME to decode the payload into a message m. It is important the MAC check is performed before ME.Decode is called, otherwise this opens the channel to attacks – as we show later in Section VI.

Definition 5. *Let* ME *be a message encoding scheme. Let* HASH *be a function family such that* $\{0,1\}^{992} \subseteq \mathsf{HASH.In}$. *Let* MAC *be a function family such that* $\mathsf{ME.Out} \subseteq \mathsf{MAC.In}$. *Let* KDF *be a function family such that* $\{0,1\}^{\mathsf{MAC.ol}} \subseteq \mathsf{KDF.In}$. *Let* $\phi_{\mathsf{MAC}}\colon \{0,1\}^{320} \to \mathsf{MAC.Keys} \times \mathsf{MAC.Keys}$ *and* $\phi_{\mathsf{KDF}}\colon \{0,1\}^{672} \to \mathsf{KDF.Keys} \times \mathsf{KDF.Keys}$. *Let* SE *be a deterministic symmetric encryption scheme with* $\mathsf{SE.kl} = \mathsf{KDF.ol}$ *and* $\mathsf{SE.MS} = \mathsf{ME.Out}$. *Then* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$ *is the channel as defined in Fig. 12, with* $\mathsf{CH.MS} = \mathsf{ME.MS}$ *and* $\mathsf{CH.SendRS} = \mathsf{ME.EncRS}$.

The message encoding scheme MTP-ME is specified in Definition 6 and Fig. 13. It is a simplified scheme for strict in-order delivery without replays (see the full version of this work for the actual MTProto scheme that permits reordering). As justified in Section IV-C, MTP-ME follows the header format of Table I, but it does not use the server_salt field (we define salt as filled with zeros to preserve the field order) and we merge the 64-bit msg_id and 32-bit msg_seq_no fields into a single 96-bit seq_no field. Note that the internal counters of MTP-ME wrap around when seq_no "overflows" modulo 2^{96}, so MTP-ME can only provide encoding integrity against adversaries that make at most 2^{96} oracle SEND queries.

Definition 6. *Let* $\mathsf{session\_id} \in \{0,1\}^{64}$ *and* $\mathsf{pb}, \mathsf{bl} \in \mathbb{N}$. *Then* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathsf{session\_id}, \mathsf{pb}, \mathsf{bl}]$ *is the message-encoding scheme given in Fig. 13, with* $\mathsf{ME.MS} = \bigcup_{i=1}^{2^{24}} \{0,1\}^{8 \cdot i}$, $\mathsf{ME.Out} = \bigcup_{i \in \mathbb{N}} \{0,1\}^{\mathsf{bl} \cdot i}$ *and* $\mathsf{ME.pl}(\ell, \nu) = 256 + \ell + |\mathsf{GenPadding}(\ell; \nu)|$.[19]

The following SHA-1 and SHA-256-based function families capture the MTProto primitives that are used to derive auth_key_id, the message key msg_key and the symmetric encryption key k.

Definition 7. MTP-HASH *is the function family with* $\mathsf{MTP\text{-}HASH.Keys} = \{0,1\}^{1056}$, $\mathsf{MTP\text{-}HASH.In} = \{0,1\}^{992}$, $\mathsf{MTP\text{-}HASH.ol} = 128$ *and* $\mathsf{MTP\text{-}HASH.Ev}(hk, x) = \mathsf{SHA\text{-}1}(x[0:672] \,\|\, hk[0:32] \,\|\, x[672:992] \,\|\, hk[32:1056])[96:160]$.

Definition 8. MTP-MAC *is the function family with* $\mathsf{MTP\text{-}MAC.Keys} = \{0,1\}^{256}$, $\mathsf{MTP\text{-}MAC.In} = \{0,1\}^{*}$, $\mathsf{MTP\text{-}MAC.ol} = 128$ *and* $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$. *We refer to its output as* msg_key.

Definition 9. MTP-KDF *is the function family with* $\mathsf{MTP\text{-}KDF.Keys} = \{0,1\}^{288} \times \{0,1\}^{288}$, $\mathsf{MTP\text{-}KDF.In} = \{0,1\}^{128}$, $\mathsf{MTP\text{-}KDF.ol} = 2 \cdot \mathsf{SHA\text{-}256.ol}$ *and* MTP-KDF.Ev *given in Fig. 14.*

Since the keys for KDF and MAC in MTProto are not independent for the two users, we have to work in a related-key setting. We are inspired by the RKA framework of [36], but define our related-key-deriving function ϕ_{KDF} (resp. ϕ_{MAC}) to output both keys at once, as a function of kk (resp. mk). See Fig. 15 for precise details of ϕ_{KDF} and ϕ_{MAC}.

We now define the symmetric encryption scheme.

[19]The definition of ME.pl assumes that GenPadding is invoked with the random coins of the corresponding ME.Encode call. For simplicity, we chose to not surface these coins in Fig. 13 and instead handle this implicitly.

```
ME.Init()
  $N_{\mathsf{sent}} \leftarrow 0$ ; $N_{\mathsf{recv}} \leftarrow 0$
  $st_{\mathsf{ME},\mathcal{I}} \leftarrow (\mathsf{session\_id}, N_{\mathsf{sent}}, N_{\mathsf{recv}})$
  $st_{\mathsf{ME},\mathcal{R}} \leftarrow (\mathsf{session\_id}, N_{\mathsf{sent}}, N_{\mathsf{recv}})$
  Return $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$

GenPadding($\ell$)   // $\ell \in \bigcup_{i=1}^{2^{24}} \{0,1\}^{8 \cdot i}$
  $\ell' \leftarrow \mathsf{bl} - \ell \bmod \mathsf{bl}$
  $bn \leftarrow_{\$} \{1, \cdots, \mathsf{pb}\}$
  $\mathsf{padding} \leftarrow_{\$} \{0,1\}^{\ell' + bn * \mathsf{bl}}$
  Return padding
```

```
ME.Encode($st_{\mathsf{ME},u}, m, aux$)
  $(\mathsf{session\_id}, N_{\mathsf{sent}}, N_{\mathsf{recv}}) \leftarrow st_{\mathsf{ME},u}$
  $\mathsf{salt} \leftarrow \langle 0 \rangle_{64}$ ; $\mathsf{seq\_no} \leftarrow \langle N_{\mathsf{sent}} \rangle_{96}$
  $\mathsf{length} \leftarrow \langle |m|/8 \rangle_{32}$
  $\mathsf{padding} \leftarrow_{\$} \mathsf{GenPadding}(|m|)$
  $p_0 \leftarrow \mathsf{salt} \,\|\, \mathsf{session\_id}$
  $p_1 \leftarrow \mathsf{seq\_no} \,\|\, \mathsf{length}$
  $p_2 \leftarrow m \,\|\, \mathsf{padding}$ ; $p \leftarrow p_0 \,\|\, p_1 \,\|\, p_2$
  $N_{\mathsf{sent}} \leftarrow (N_{\mathsf{sent}} + 1) \bmod 2^{96}$
  $st_{\mathsf{ME},u} \leftarrow (\mathsf{session\_id}, N_{\mathsf{sent}}, N_{\mathsf{recv}})$
  Return $(st_{\mathsf{ME},u}, p)$
```

```
ME.Decode($st_{\mathsf{ME},u}, p, aux$)
  If $|p| < 256$ then return $(st_{\mathsf{ME},u}, \perp)$
  $(\mathsf{session\_id}, N_{\mathsf{sent}}, N_{\mathsf{recv}}) \leftarrow st_{\mathsf{ME},u}$ ; $\ell \leftarrow |p| - 256$
  $\mathsf{salt} \leftarrow p[0:64]$ ; $\mathsf{session\_id}' \leftarrow p[64:128]$
  $\mathsf{seq\_no} \leftarrow p[128:224]$ ; $\mathsf{length} \leftarrow p[224:256]$
  If $(\mathsf{session\_id}' \neq \mathsf{session\_id}) \vee$
      $(\mathsf{seq\_no} \neq N_{\mathsf{recv}}) \vee$
      $\neg(0 < \mathsf{length} \leq |\ell|/8)$ then return $(st_{\mathsf{ME},u}, \perp)$
  $m \leftarrow p[256 : 256 + \mathsf{length} \cdot 8]$
  $N_{\mathsf{recv}} \leftarrow (N_{\mathsf{recv}} + 1) \bmod 2^{96}$
  $st_{\mathsf{ME},u} \leftarrow (\mathsf{session\_id}, N_{\mathsf{sent}}, N_{\mathsf{recv}})$ ; Return $(st_{\mathsf{ME},u}, m)$
```

Figure 13: Construction of a simplified message encoding scheme for strict in-order delivery $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathsf{session\_id}, \mathsf{pb}, \mathsf{bl}]$ for session identifier session_id, maximum padding length (in full blocks) pb, and output block length bl.

```
MTP-KDF.Ev($kk_u$, msg_key)   // |msg_key| = 128
  $(kk_0, kk_1) \leftarrow kk_u$ ; $k_0 \leftarrow \mathsf{SHA\text{-}256}(\mathsf{msg\_key} \,\|\, kk_0)$
  $k_1 \leftarrow \mathsf{SHA\text{-}256}(kk_1 \,\|\, \mathsf{msg\_key})$ ; $k \leftarrow k_0 \,\|\, k_1$ ; Return $k$
```

Figure 14: Construction of function family MTP-KDF.

```
$\phi_{\mathsf{KDF}}(kk)$   // $|kk| = 672$
  $kk_{\mathcal{I},0} \leftarrow kk[0:288]$
  $kk_{\mathcal{R},0} \leftarrow kk[64:352]$
  $kk_{\mathcal{I},1} \leftarrow kk[320:608]$
  $kk_{\mathcal{R},1} \leftarrow kk[384:672]$
  $kk_{\mathcal{I}} \leftarrow (kk_{\mathcal{I},0}, kk_{\mathcal{I},1})$
  $kk_{\mathcal{R}} \leftarrow (kk_{\mathcal{R},0}, kk_{\mathcal{R},1})$
  Return $(kk_{\mathcal{I}}, kk_{\mathcal{R}})$
```

```
$\phi_{\mathsf{MAC}}(mk)$   // $|mk| = 320$
  $mk_{\mathcal{I}} \leftarrow mk[0:256]$
  $mk_{\mathcal{R}} \leftarrow mk[64:320]$
  Return $(mk_{\mathcal{I}}, mk_{\mathcal{R}})$
```

Figure 15: Related-key-deriving functions $\phi_{\mathsf{KDF}} \colon \{0,1\}^{672} \rightarrow \mathsf{MTP\text{-}KDF.Keys} \times \mathsf{MTP\text{-}KDF.Keys}$ and $\phi_{\mathsf{MAC}} \colon \{0,1\}^{320} \rightarrow \mathsf{MTP\text{-}MAC.Keys} \times \mathsf{MTP\text{-}MAC.Keys}$.

Definition 10. *Let* AES-256 *be the standard AES block cipher with* AES-256.kl = 256 *and* AES-256.ol = 128*, and let* IGE *be the block cipher mode in Fig. 4. Let* MTP-SE = IGE[AES-256].

V. Formal security analysis

We first define the central security notions required from each of the primitives used in MTP-CH. Then, we state that MTP-CH satisfies correctness, indistinguishability and integrity.

A. Security requirements on standard primitives

1) MTP-HASH is a one-time indistinguishable function family: We require that MTP-HASH meets the one-time weak indistinguishability notion (OTWIND) defined in Fig. 16. The security game $\mathsf{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}$ in Fig. 16 evaluates the function family HASH on a challenge input x_b using a secret uniformly random function key hk. Adversary $\mathcal{D}$ is given x_0, x_1 and the output of HASH; it is required to guess the challenge bit $b \in \{0,1\}$. The game samples inputs x_0, x_1 uniformly at random rather than allowing $\mathcal{D}$ to choose them, so this security notion requires HASH to provide only a *weak* form of one-time indistinguishability. The advantage of $\mathcal{D}$ in breaking the OTWIND-security of HASH is defined as $\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}) = 2 \cdot \Pr[\mathsf{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}] - 1$. The full version of this work provides a formal reduction from the OTWIND-security of MTP-HASH to the one-time PRF-security of SHACAL-1 (as defined in Section II-B).

```
Game $\mathsf{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}$
  $b \leftarrow_{\$} \{0,1\}$ ; $hk \leftarrow_{\$} \{0,1\}^{\mathsf{HASH.kl}}$ ; $x_0 \leftarrow_{\$} \mathsf{HASH.In}$
  $x_1 \leftarrow_{\$} \mathsf{HASH.In}$ ; $\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x_b)$
  $b' \leftarrow_{\$} \mathcal{D}(x_0, x_1, \mathsf{auth\_key\_id})$ ; Return $b' = b$
```

Figure 16: One-time weak indistinguishability of function family HASH.

2) MTP-KDF is a PRF under related-key attacks: We require that MTP-KDF behaves like a pseudorandom function in the RKA setting (RKPRF) as defined in Fig. 17. The security game $\mathsf{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}$ in Fig. 17 defines a variant of the standard PRF notion allowing the adversary $\mathcal{D}$ to use its RoR oracle to evaluate the function family KDF on either of the two secret, related function keys $kk_{\mathcal{I}}, kk_{\mathcal{R}}$ (both computed using related-key-deriving function ϕ_{KDF}). The advantage of $\mathcal{D}$ in breaking the RKPRF-security of KDF with respect to ϕ_{KDF} is defined as $\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}) = 2 \cdot \Pr[\mathsf{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}] - 1$.

```
Game $\mathsf{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}$
  $b \leftarrow_{\$} \{0,1\}$ ; $kk \leftarrow_{\$} \{0,1\}^{672}$
  $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
  $b' \leftarrow_{\$} \mathcal{D}^{\mathrm{RoR}}$ ; Return $b' = b$
```

```
$\mathrm{RoR}(u, \mathsf{msg\_key})$
  $k_1 \leftarrow \mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$
  If $\mathsf{T}[u, \mathsf{msg\_key}] = \perp$ then
      $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow_{\$} \{0,1\}^{\mathsf{KDF.ol}}$
  $k_0 \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; Return $k_b$
```

Figure 17: Related-key PRF-security of function family KDF with respect to related-key-deriving function ϕ_{KDF}.

In Section V-B1 we define a novel security notion for SHACAL-2 that roughly requires it to be a leakage-resilient PRF under related-key attacks; in the full version of this work we provide a formal reduction from the RKPRF-security of MTP-KDF to the new security notion. In this context, "leakage resilience" means that the adversary can adaptively choose a part of the SHACAL-2 key. However, we limit the adversary to being able to evaluate SHACAL-2 only on a single known, constant input (which is IV_{256}, the initial state of SHA-256). The new security notion is formalised as the LRKPRF-security of SHACAL-2 with respect to a pair of related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ (the latter is defined in Section V-B1).

3) MTP-MAC **is collision-resistant under RKA:** We require that collisions in the outputs of MTP-MAC under related keys are hard to find (RKCR), as defined in Fig. 18. The security game $G^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{F}}$ in Fig. 18 gives the adversary $\mathcal{F}$ two related function keys $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ (created by the related-key-deriving function ϕ_{MAC}), and requires it to produce two payloads p_0, p_1 (for either user u) such that there is a collision in the corresponding outputs $\mathsf{msg\_key}_0, \mathsf{msg\_key}_1$ of the function family MAC. The advantage of $\mathcal{F}$ in breaking the RKCR-security of MAC with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{F}) = \Pr[G^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{F}}]$. It is clear by inspection that the RKCR-security of $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$ (with respect to ϕ_{MAC} from Fig. 15) reduces to the collision resistance of truncated-output SHA-256.

```
Game $G^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{F}}$
$mk \leftarrow\!\$\ \{0,1\}^{320}$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(u, p_0, p_1) \leftarrow\!\$\ \mathcal{F}(mk_{\mathcal{I}}, mk_{\mathcal{R}})$ ; $\mathsf{msg\_key}_0 \leftarrow \mathsf{MAC.Ev}(mk_u, p_0)$
$\mathsf{msg\_key}_1 \leftarrow \mathsf{MAC.Ev}(mk_u, p_1)$ ; $\mathsf{dist\_inp} \leftarrow (p_0 \neq p_1)$
$\mathsf{eq\_out} \leftarrow (\mathsf{msg\_key}_0 = \mathsf{msg\_key}_1)$ ; Return $\mathsf{dist\_inp} \wedge \mathsf{eq\_out}$
```

Figure 18: Related-key collision resistance of function family MAC with respect to related-key-deriving function ϕ_{MAC}.

4) MTP-MAC **is a PRF under RKA for unique-prefix inputs:** We require that MTP-MAC behaves like a pseudorandom function in the RKA setting when it is evaluated on a set of inputs that have unique 256-bit prefixes (UPRKPRF), as defined in Fig. 19. The security game $G^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 19 extends the standard PRF notion to use two related ϕ_{MAC}-derived function keys $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ for the function family MAC (similar to the RKPRF-security notion we defined above); but it also enforces that the adversary $\mathcal{D}$ cannot query its oracle RoR on two inputs (u, p_0) and (u, p_1) for any $u \in \{\mathcal{I}, \mathcal{R}\}$ such that p_0, p_1 share the same 256-bit prefix. The unique-prefix condition means that the game does not need to maintain a PRF table to achieve output consistency. Note that this security game only allows to call the oracle RoR with inputs of length $|p| \geq 256$; this is sufficient for our purposes, because in MTP-CH the function family MTP-MAC is only used with payloads that are longer than 256 bits. The advantage of $\mathcal{D}$ in breaking the UPRKPRF-security of MAC with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr[G^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}] - 1$.

In Section V-B2 we define a novel security notion that requires SHACAL-2 to be a leakage-resilient, related-key PRF when evaluated on a fixed input; in the full version of this work we show that the UPRKPRF-security of MTP-MAC reduces to this security notion and to the one-time PRF-security (OTPRF) of the SHA-256 compression function h_{256}. The new security notion is similar to the notion discussed in Section V-A2 and defined in Section V-B1, in that it only allows the adversary to evaluate SHACAL-2 on the fixed input IV_{256}. However, the underlying security game derives the related SHACAL-2 keys

```
Game $G^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}$
$b \leftarrow\!\$\ \{0,1\}$
$mk \leftarrow\!\$\ \{0,1\}^{320}$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$
$b' \leftarrow\!\$\ \mathcal{D}^{\mathrm{RoR}}$
Return $b' = b$

$\mathrm{RoR}(u, p)$   // $p \in \{0,1\}^*$
If $|p| < 256$ then return $\perp$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then return $\perp$
$X_u \leftarrow X_u \cup \{p_0\}$
$\mathsf{msg\_key}_1 \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
$\mathsf{msg\_key}_0 \leftarrow\!\$\ \{0,1\}^{\mathsf{MAC.ol}}$
Return $\mathsf{msg\_key}_b$
```

Figure 19: Related-key PRF-security of function family MAC for inputs with unique 256-bit prefixes, with respect to key derivation function ϕ_{MAC}.

differently, partially based on the function ϕ_{MAC} defined in Fig. 15 (as opposed to ϕ_{KDF}). The new notion is formalised as the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC}.

5) MTP-SE **is a one-time indistinguishable SE scheme:** For any block cipher E, the full version of this work shows that IGE[E] as used in MTProto is OTIND\$-secure (defined in Fig. 3) if CBC[E] is OTIND\$-secure. This enables us to use standard results [37], [38] on CBC in our analysis of MTProto.

B. Novel assumptions about SHACAL-2

In this section we define two novel assumptions about SHACAL-2. Both assumptions require SHACAL-2 to be a related-key PRF when evaluated on the fixed input IV_{256} (i.e. on the initial state of SHA-256), meaning that the adversary can obtain the values of $\mathsf{SHACAL\text{-}2.Ev}(\cdot, \mathsf{IV}_{256})$ for a number of different but related keys. We formalise the two assumptions as security notions, called LRKPRF and HRKPRF, each defined with respect to different related-key-deriving functions; this reflects the fact that these security notions allow the adversary to choose the keys in substantially different ways. The notion of LRKPRF-security derives the SHACAL-2 keys partially based on the function ϕ_{KDF}, whereas the notion of HRKPRF-security derives SHACAL-2 keys partially based on the function ϕ_{MAC} (both functions are defined in Fig. 15). Both security notions also have different flavours of leakage resilience: (1) the security game defining LRKPRF allows the adversary to directly choose 128 bits of the 512-bit long SHACAL-2 key, with another 96 bits of this key fixed and known (due to being chosen by the SHA padding function SHA-pad), and (2) the security game defining HRKPRF allows the adversary to directly choose 256 bits of the 512-bit long SHACAL-2 key.

We use the notion of LRKPRF-security to justify the RKPRF-security of MTP-KDF with respect to ϕ_{KDF} (as explained in Section V-A2), which is needed in both the IND-security and the INT-security proofs of MTP-CH. We use the notion of HRKPRF-security to justify the UPRKPRF-security of MTP-MAC with respect to ϕ_{MAC} (as explained in Section V-A4), which is needed in the IND-security proof of MTP-CH.

We stress that we have to assume properties of SHACAL-2 that have not been studied in the literature. Related-key attacks on reduced-round SHACAL-2 have been considered [39], [40], but they ordinarily work with a *known difference* relation between unknown keys. In contrast, our LRKPRF-security notion uses keys that differ by random, unknown parts. Both of our security notions consider keys that are partially chosen or known by the adversary. It is straightforward to show that both the LRKPRF-security and the HRKPRF-security of SHACAL-2 hold in the ideal cipher model (i.e. when SHACAL-2 is modelled as the ideal cipher). However, we cannot rule out the possibility of attacks on SHACAL-2 due to its internal structure in the setting of related-key attacks combined with key leakage. We leave this as an open question.

1) SHACAL-2 is a PRF with ϕ_{KDF}-based related keys: Our LRKPRF-security notion for SHACAL-2 is defined with respect to related-key-deriving functions ϕ_{KDF} (from Fig. 15) and $\phi_{\mathsf{SHACAL\text{-}2}}$ from Fig. 20. The latter mirrors the design of MTP-KDF that (in Definition 9) is defined to return SHA-256(msg_key $\|$ kk_0) $\|$ SHA-256(kk_1 $\|$ msg_key) for the target key $kk_u = (kk_0, kk_1)$, except $\phi_{\mathsf{SHACAL\text{-}2}}$ only needs to produce the corresponding SHA-padded inputs.

```
$\phi_{\mathsf{SHACAL\text{-}2}}(kk_u, \mathsf{msg\_key})$   // $|\mathsf{msg\_key}| = 128$
$(kk_0, kk_1) \leftarrow kk_u$ ;  $sk_0 \leftarrow \mathsf{SHA\text{-}pad}(\mathsf{msg\_key} \,\|\, kk_0)$
$sk_1 \leftarrow \mathsf{SHA\text{-}pad}(kk_1 \,\|\, \mathsf{msg\_key})$ ;  Return $(sk_0, sk_1)$
```

Figure 20: Related-key-deriving function $\phi_{\mathsf{SHACAL\text{-}2}}$: $(\mathsf{MTP\text{-}KDF.Keys} \times \mathsf{MTP\text{-}KDF.Keys}) \times \{0,1\}^{128} \rightarrow \{0,1\}^{512}$.

Consider the game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ in Fig. 21. Adversary $\mathcal{D}$ is given access to the RoR oracle that takes $u, i, \mathsf{msg\_key}$ as input; all inputs to the oracle serve as parameters for the SHACAL-2 key derivation, used to determine the challenge key sk_i. The adversary gets back either the output of $\mathsf{SHACAL\text{-}2.Ev}(sk_i, \mathsf{IV}_{256})$ (if $b = 1$), or a uniformly random value (if $b = 0$), and is required to guess the challenge bit. The PRF table T is used to ensure consistency, so that a single random value is sampled and remembered for each set of used key derivation parameters $u, i, \mathsf{msg\_key}$. The advantage of $\mathcal{D}$ in breaking the LRKPRF-security of SHACAL-2 with respect to ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ is defined as $\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}] - 1$.

```
Game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$
$b \leftarrow_{\$} \{0,1\}$ ;  $kk \leftarrow_{\$} \{0,1\}^{672}$ ;  $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$b' \leftarrow_{\$} \mathcal{D}^{\mathrm{RoR}}$ ;  Return $b' = b$
$\mathrm{RoR}(u, i, \mathsf{msg\_key})$   // $u \in \{\mathcal{I}, \mathcal{R}\}$, $i \in \{0,1\}$, $|\mathsf{msg\_key}| = 128$
$(sk_0, sk_1) \leftarrow \phi_{\mathsf{SHACAL\text{-}2}}(kk_u, \mathsf{msg\_key})$
$y_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(sk_i, \mathsf{IV}_{256})$
If $\mathsf{T}[u, i, \mathsf{msg\_key}] = \perp$ then
    $\mathsf{T}[u, i, \mathsf{msg\_key}] \leftarrow_{\$} \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
$y_0 \leftarrow \mathsf{T}[u, i, \mathsf{msg\_key}]$ ;  Return $y_b$
```

Figure 21: Leakage-resilient, related-key PRF-security of function family SHACAL-2 on fixed input IV_{256} with respect to related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$.

2) SHACAL-2 is a PRF with ϕ_{MAC}-based related keys: Consider the game $\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 22. Adversary $\mathcal{D}$ is given access to RoR oracle, and is required to choose the 256-bit suffix p of each challenge key used for evaluating $\mathsf{SHACAL\text{-}2.Ev}(\cdot, \mathsf{IV}_{256})$. The value of mk_u is then used to set the 256-bit prefix of the challenge key, where u is also chosen by the adversary, but the $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ values themselves are related secrets that are not known to $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}] - 1$.

```
Game $\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$
$b \leftarrow_{\$} \{0,1\}$
$mk \leftarrow_{\$} \{0,1\}^{320}$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$b' \leftarrow_{\$} \mathcal{D}^{\mathrm{RoR}}$
Return $b' = b$

$\mathrm{RoR}(u, p)$   // $u \in \{\mathcal{I}, \mathcal{R}\}$, $|p| = 256$
$y_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(mk_u \,\|\, p, \mathsf{IV}_{256})$
If $\mathsf{T}[u, p] = \perp$ then
    $\mathsf{T}[u, p] \leftarrow_{\$} \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
$y_0 \leftarrow \mathsf{T}[u, p]$
Return $y_b$
```

Figure 22: Leakage-resilient, related-key PRF-security of function family SHACAL-2 on fixed input IV_{256} with respect to related-key-deriving function ϕ_{MAC}.

C. Security requirements on message encoding

1) MTP-ME ensures in-order delivery: We require that MTP-ME is EINT-secure (Fig. 8) with respect to the support function SUPP defined in Fig. 23. SUPP enforces in-order delivery for each user's sent messages, thus preventing uni-directional reordering attacks, replays and message deletion. It is formalised using a function find(op, tr, label) that searches a given transcript for a sent or recv entry that corresponds to label, and also counts the number of valid entries encountered prior to finding the target. For any label that corresponds to the N_{sent}-th valid sent-type entry in $\mathsf{tr}_{\overline{u}}$, the support function SUPP checks that tr_u contains $N_{\mathsf{recv}} = N_{\mathsf{sent}} - 1$ valid recv-type entries, and that none of them contains the label itself. Here we rely on each label being unique, which is true for MTP-ME as long as it encodes at most 2^{96} messages.[20] Replays are prevented by the search of entries received by u. The count from both searches is used to ensure that there are no gaps between the number of sent and received ciphertexts, thus preventing deletion and reordering.[21] As outlined in Section IV-B1, the MTProto implementation of ME we studied allowed reordering so it was not EINT-secure with respect to SUPP. The full version of this work shows that $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{MTP\text{-}ME},\mathsf{SUPP}}(\mathcal{F}) = 0$ for any $\mathcal{F}$ making at most 2^{96} queries to SEND.

2) Prefix uniqueness of MTP-ME: We require that payloads produced by MTP-ME have distinct prefixes of size 256 bits (independently for each user $u \in \{\mathcal{I}, \mathcal{R}\}$), as defined by the security game in Fig. 24. The advantage of an adversary $\mathcal{F}$ in breaking the UPREF-security of a message encoding scheme ME is defined as $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{F}) = \Pr[\mathrm{G}^{\mathsf{upref}}_{\mathsf{ME},\mathcal{F}}]$. Given the fixed prefix size, this notion cannot be satisfied against unbounded adversaries. Our MTP-ME scheme ensures unique prefixes using the 96-bit counter seq_no that contains the number of messages

[20] A limitation on number of queries is inherent as long as fixed-length sequence numbers are used.

[21] Note that aux is not used in SUPP or MTP-ME. It would be possible to add time synchronisation using the timestamp captured in the msg_id field just as the current MTProto ME implementation does.

```
$\mathsf{SUPP}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$
$(N_{\mathsf{recv}}, m_{\mathsf{recv}}) \leftarrow$
    $\mathsf{find}(\mathsf{recv}, \mathsf{tr}_u, \mathsf{label})$
If $m_{\mathsf{recv}} \neq \perp$ then return $\perp$
$(N_{\mathsf{sent}}, m_{\mathsf{sent}}) \leftarrow$
    $\mathsf{find}(\mathsf{sent}, \mathsf{tr}_{\overline{u}}, \mathsf{label})$
If $N_{\mathsf{sent}} \neq N_{\mathsf{recv}} + 1$ then
    Return $\perp$
Return $m_{\mathsf{sent}}$

$\mathsf{find}(\mathsf{op}, \mathsf{tr}, \mathsf{label})$
$N_{\mathsf{op}} \leftarrow 0$
For $(\mathsf{op}, m, \mathsf{label}', aux) \in \mathsf{tr}$ do
    If $(\mathsf{op} = \mathsf{recv} \wedge m \neq \perp) \vee$
        $(\mathsf{op} = \mathsf{sent} \wedge \mathsf{label}' \neq \perp)$ then
            $N_{\mathsf{op}} \leftarrow N_{\mathsf{op}} + 1$
            If $\mathsf{label}' = \mathsf{label}$ then
                Return $(N_{\mathsf{op}}, m)$
Return $(N_{\mathsf{op}}, \perp)$
```

Figure 23: Support function SUPP for strict in-order delivery.

sent by user u, so we have $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{MTP\text{-}ME}}(\mathcal{F}) = 0$ for any $\mathcal{F}$ making at most 2^{96} queries, and otherwise there exists an adversary $\mathcal{F}$ such that $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{MTP\text{-}ME}}(\mathcal{F}) = 1$. Note that MTP-ME always has payloads larger than 256 bits. The MTProto implementation of message encoding we analysed was not UPREF-secure as it allowed repeated msg_id (cf. Section IV-B2).

```
Game $\mathrm{G}^{\mathsf{upref}}_{\mathsf{ME},\mathcal{F}}$
win $\leftarrow$ false
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
    $\leftarrow\!\$\ \mathsf{ME.Init}()$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$
$\mathcal{F}^{\textsc{Send}}$; Return win

$\textsc{Send}(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
If $|p| < 256$ then return $\perp$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then win $\leftarrow$ true
$X_u \leftarrow X_u \cup \{p_0\}$ ; Return $p$
```

Figure 24: Prefix uniqueness of message encoding scheme ME.

3) Encoding robustness of MTP-ME: We require that decoding in MTP-ME should not affect its state in such a way that would be visible in future encoded payloads, as defined by the security game in Fig. 25. The advantage of an adversary $\mathcal{D}$ in breaking the ENCROB-security of a message encoding scheme ME is defined as $\mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}}] - 1$. This advantage is trivially zero both for MTP-ME and the original MTProto message encoding scheme (modelled in the full version). Note, however, that this property is incompatible with stronger notions of resistance against reordering attacks such as causality preservation.

```
Game $\mathrm{G}^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}}$
$b \leftarrow\!\$\ \{0,1\}$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$\ \mathsf{ME.Init}()$
$b' \leftarrow\!\$\ \mathcal{D}^{\textsc{Send},\textsc{Recv}}$ ; Return $b' = b$

$\textsc{Send}(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$ ; Return $p$

$\textsc{Recv}(u, p, aux)$
If $b = 1$ then $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
Return $\perp$
```

Figure 25: Encoding robustness of message encoding scheme ME.

4) Combined security of MTP-SE and MTP-ME: We require that decryption in MTP-SE with uniformly random keys has unpredictable outputs with respect to MTP-ME, as defined in Fig. 26. The security game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}$ in Fig. 26 gives adversary $\mathcal{F}$ access to two oracles. For any user $u \in \{\mathcal{I}, \mathcal{R}\}$ and message key msg_key, oracle $\textsc{Ch}$ decrypts a given ciphertext c_{se} of deterministic symmetric encryption scheme SE under a uniformly random key $k \in \{0,1\}^{\mathsf{SE.kl}}$, and then decodes it using the given message encoding state st_{ME} of message encoding scheme ME, returning no output. The adversary is allowed to choose arbitrary values of c_{se} and st_{ME}; it is allowed to repeatedly query oracle $\textsc{Ch}$ on inputs that contain the same values for u, msg_key in order to reuse a fixed, secret SE key k with different choices of c_{se}. Oracle $\textsc{Expose}$ lets $\mathcal{F}$ learn the SE key corresponding to the given u and msg_key; the table S is then used to disallow the adversary from querying $\textsc{Ch}$ with this pair of u and msg_key values again. $\mathcal{F}$ wins if it can cause ME.Decode to output a valid $m \neq \perp$. Note that msg_key in this game merely serves as a label for the tables, so we allow it to be an arbitrary string $\mathsf{msg\_key} \in \{0,1\}^*$. The advantage of $\mathcal{F}$ in breaking the UNPRED-security of SE with respect to ME is defined as $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}) = \Pr[\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}]$. In the full version of this work we show that $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{MTP\text{-}SE},\mathsf{MTP\text{-}ME}}(\mathcal{F}) \leq q_{\textsc{Ch}}/2^{64}$ for any $\mathcal{F}$ making $q_{\textsc{Ch}}$ queries.

```
Game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}$
win $\leftarrow$ false ; $\mathcal{F}^{\textsc{Expose},\textsc{Ch}}$ ; Return win

$\textsc{Expose}(u, \mathsf{msg\_key})$    // $\mathsf{msg\_key} \in \{0,1\}^*$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow$ true ; Return $\mathsf{T}[u, \mathsf{msg\_key}]$

$\textsc{Ch}(u, \mathsf{msg\_key}, c_{se}, st_{\mathsf{ME}}, aux)$    // $\mathsf{msg\_key} \in \{0,1\}^*$
If $\neg\mathsf{S}[u, \mathsf{msg\_key}]$ then
    If $\mathsf{T}[u, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow\!\$\ \{0,1\}^{\mathsf{SE.kl}}$
    $k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
    $(st_{\mathsf{ME}}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME}}, p, aux)$
    If $m \neq \perp$ then win $\leftarrow$ true
Return $\perp$
```

Figure 26: Unpredictability of deterministic symmetric encryption scheme SE with respect to message encoding scheme ME.

D. Correctness of MTP-CH

We claim that our MTProto-based channel satisfies our correctness definition. Consider any adversary $\mathcal{F}$ playing in the correctness game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ (Fig. 6) for channel CH = MTP-CH (Fig. 12) and support function supp = SUPP (Fig. 23). Due to the definition of SUPP, the $\textsc{Recv}$ oracle in game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{MTP\text{-}CH},\mathsf{SUPP},\mathcal{F}}$ rejects all CH ciphertexts that were not previously returned by $\textsc{Send}$. The encryption and decryption algorithms of channel MTP-CH rely in a modular way on the message encoding scheme MTP-ME, deterministic function families MTP-KDF, MTP-MAC, and deterministic symmetric encryption scheme MTP-SE; the latter provides decryption correctness, so any valid ciphertext processed by oracle $\textsc{Recv}$ correctly recovers the originally encrypted payload p. Thus we need to show that MTP-ME always recovers the expected plaintext m from payload p, meaning m matches the corresponding output of SUPP. This is implied by the EINT-security of MTP-ME with respect to SUPP; we prove the latter in the full version of this work for adversaries that make at most 2^{96} queries.[22]

E. IND-security of MTP-CH

Due to lack of space, here we provide only a very high-level overview of how we prove IND-security of MTP-CH and a theorem statement. We begin our IND-security reduction by considering an arbitrary adversary $\mathcal{D}_{\text{IND}}$ playing in the IND-security game against channel $\mathsf{CH} = \mathsf{MTP\text{-}CH}$ (i.e. $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\text{IND}}}$ in Fig. 6), and we gradually change this game until we can show that $\mathcal{D}_{\text{IND}}$ can no longer win. To this end, we make three key observations. (1) Recall that oracle $\textsc{Recv}$ always returns $\perp$, and the only functionality of this oracle is to update the state of receiver's channel by calling $\mathsf{CH.Recv}$. We assume that calls to $\mathsf{CH.Recv}$ never affect the ciphertexts that are returned by future calls to $\mathsf{CH.Send}$ (more precisely, we use the ENCROB property of ME that reasons about payloads rather than ciphertexts). This allows us to completely disregard the $\textsc{Recv}$ oracle, making it immediately return $\perp$ without calling $\mathsf{CH.Recv}$. (2) We use the UPRKPRF-security of MAC to show that the ciphertexts returned by oracle $\textsc{Ch}$ contain $\mathsf{msg\_key}$ values that look uniformly random and are independent of each other. Roughly, this security notion requires that MAC can only be evaluated on a set of inputs with unique prefixes. To ensure this, we assume that the payloads produced by ME meet this requirement (as formalised by the UPREF property of ME). (3) In order to prove that oracle $\textsc{Ch}$ does not leak the challenge bit, it remains to show that ciphertexts returned by $\textsc{Ch}$ contain c_{se} values that look uniformly random and independent of each other. This follows from the OTIND\$-security of SE. We invoke the OTWIND-security of HASH to show that $\mathsf{auth\_key\_id}$ does not leak any information about the KDF keys; we then use the RKPRF-security of KDF to show that the keys used for SE are uniformly random. Finally, we use the birthday bound to argue that the uniformly random values of $\mathsf{msg\_key}$ are unlikely to collide, and hence the keys used for SE are also one-time. Formally, we have:

Theorem 1. *Let* ME, HASH, MAC, KDF, ϕ_{MAC}, ϕ_{KDF}, SE *be any primitives that meet the requirements stated in Definition 5 of channel* $\mathsf{MTP\text{-}CH}$. *Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$. *Let* $\mathcal{D}_{\text{IND}}$ *be any adversary against the* IND-*security of* CH, *making* $q_{\textsc{Ch}}$ *queries to its* $\textsc{Ch}$ *oracle. Then there exist adversaries* $\mathcal{D}_{\text{OTWIND}}$, $\mathcal{D}_{\text{RKPRF}}$, $\mathcal{D}_{\text{ENCROB}}$, $\mathcal{F}_{\text{UPREF}}$, $\mathcal{D}_{\text{UPRKPRF}}$, $\mathcal{D}_{\text{OTIND\$}}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}_{\text{IND}}) \le 2 \cdot \Big(&\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\text{OTWIND}}) + \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\text{RKPRF}}) \\ &+ \mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}_{\text{ENCROB}}) + \mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{F}_{\text{UPREF}}) \\ &+ \mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\text{UPRKPRF}}) + \frac{q_{\textsc{Ch}} \cdot (q_{\textsc{Ch}} - 1)}{2 \cdot 2^{\mathsf{MAC.ol}}} \\ &+ \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}_{\text{OTIND\$}})\Big).\end{aligned}$$

The proof can be found in the full version of this work.

[22]There are other ways to handle counters which could imply correctness for unbounded adversaries – MTP-ME wraps its counters to stay close to the actual MTProto implementations.

F. INT-security of MTP-CH

Due to lack of space, here we provide only a very high-level overview of how we prove integrity of $\mathsf{MTP\text{-}CH}$ and a theorem statement. Details are in the full version. The first half of our integrity proof shows that it is hard to forge ciphertexts; in order to justify this, we rely on security properties of the cryptographic primitives that are used to build the channel $\mathsf{MTP\text{-}CH}$ (i.e. HASH, KDF, SE, and MAC). Once ciphertext forgery is ruled out, we are guaranteed that $\mathsf{MTP\text{-}CH}$ broadly matches an intuition of an *authenticated channel*: it prevents an attacker from modifying or creating its own ciphertexts but still allows it to intercept and subsequently drop, reorder or replay honestly produced ciphertexts. So it remains to show that the message encoding scheme ME appropriately resolves all of the possible adversarial interaction with an authenticated channel; formally, we require that it behaves according to the requirements that are specified by some support function supp. Our main result is then:

Theorem 2. *Let* $\mathsf{session\_id} \in \{0,1\}^{64}$, $\mathsf{pb} \in \mathbb{N}$, *and* $\mathsf{bl} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathsf{session\_id}, \mathsf{pb}, \mathsf{bl}]$ *be the message encoding scheme as defined in Definition 6. Let* $\mathsf{SE} = \mathsf{MTP\text{-}SE}$ *be the deterministic symmetric encryption scheme as defined in Definition 10. Let* HASH, MAC, KDF, ϕ_{MAC}, ϕ_{KDF} *be any primitives that, together with* ME *and* SE, *meet the requirements stated in Definition 5 of channel* $\mathsf{MTP\text{-}CH}$. *Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$. *Let* $\mathsf{supp} = \mathsf{SUPP}$ *be the support function as defined in Fig. 23. Let* $\mathcal{F}_{\text{INT}}$ *be any adversary against the* INT-*security of* CH *with respect to* supp. *Then there exist adversaries* $\mathcal{D}_{\text{OTWIND}}$, $\mathcal{D}_{\text{RKPRF}}$, $\mathcal{F}_{\text{UNPRED}}$, $\mathcal{F}_{\text{RKCR}}$, $\mathcal{F}_{\text{EINT}}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}_{\text{INT}}) \le\ &\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\text{OTWIND}}) + \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\text{RKPRF}}) \\ &+ \mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}_{\text{UNPRED}}) + \mathsf{Adv}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{F}_{\text{RKCR}}) \\ &+ \mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{F}_{\text{EINT}}).\end{aligned}$$

The proof can be found in the full version of this work.

G. Instantiation and Interpretation

We are now ready to combine the theorems from the previous two sections with the notions defined in Section V-A and Section V-C and the proofs in the full version of this work. This is meant to allow interpretation of our main results: qualitatively (what security assumptions are made) and quantitatively (what security level is achieved). Note that in both of the following corollaries, the adversary is limited to making 2^{96} queries. This is due to the wrapping of counters in MTP-ME, since beyond this limit the advantage in breaking UPREF-security and EINT-security of MTP-ME becomes 1.

Corollary 1. *Let* $\mathsf{session\_id} \in \{0,1\}^{64}$, $\mathsf{pb} \in \mathbb{N}$ *and* $\mathsf{bl} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathsf{session\_id}, \mathsf{pb}, \mathsf{bl}]$, $\mathsf{MTP\text{-}HASH}$, $\mathsf{MTP\text{-}MAC}$, $\mathsf{MTP\text{-}KDF}$, ϕ_{MAC}, ϕ_{KDF}, $\mathsf{MTP\text{-}SE}$ *be the primitives of MTProto defined in Section IV-D. Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{MTP\text{-}HASH}, \mathsf{MTP\text{-}MAC}, \mathsf{MTP\text{-}KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{MTP\text{-}SE}]$. *Let* $\phi_{\mathsf{SHACAL\text{-}2}}$ *be the related-key-deriving function defined in Fig. 20. Let* h_{256} *be the* $\mathsf{SHA\text{-}256}$ *compression function, and let* H *be the corresponding function family with* $\mathsf{H.Ev} = h_{256}$, $\mathsf{H.kl} = \mathsf{H.ol} = 256$ *and* $\mathsf{H.In} = \{0,1\}^{512}$. *Let* $\ell \in \mathbb{N}$. *Let* $\mathcal{D}_{\text{IND}}$ *be any adversary against the* IND-*security of* CH, *making* $q_{\textsc{Ch}} \le 2^{96}$ *queries to its* $\textsc{Ch}$ *oracle, each query made for a message of length at most*

$\ell \leq 2^{27}$ *bits.*[23] *Then there exist adversaries* $\mathcal{D}_{\text{OTPRF}}^{\mathsf{shacal}}$, $\mathcal{D}_{\text{LRKPRF}}$, $\mathcal{D}_{\text{HRKPRF}}$, $\mathcal{D}_{\text{OTPRF}}^{\mathsf{compr}}$, $\mathcal{D}_{\text{OTIND\$}}$ *such that*

$$\begin{aligned}\mathsf{Adv}_{\mathsf{CH}}^{\mathsf{ind}}(\mathcal{D}_{\text{IND}}) \leq 4 \cdot \Big(&\mathsf{Adv}_{\mathsf{SHACAL\text{-}1}}^{\mathsf{otprf}}(\mathcal{D}_{\text{OTPRF}}^{\mathsf{shacal}})\\ &+ \mathsf{Adv}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}^{\mathsf{lrkprf}}(\mathcal{D}_{\text{LRKPRF}})\\ &+ \mathsf{Adv}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}^{\mathsf{hrkprf}}(\mathcal{D}_{\text{HRKPRF}})\\ &+ \left\lfloor \frac{\ell + 256}{512} + \frac{\mathsf{pb}+1}{4} \right\rfloor \cdot \mathsf{Adv}_{\mathsf{H}}^{\mathsf{otprf}}(\mathcal{D}_{\text{OTPRF}}^{\mathsf{compr}})\Big)\\ &+ \frac{q_{\text{CH}} \cdot (q_{\text{CH}} - 1)}{2^{128}}\\ &+ 2 \cdot \mathsf{Adv}_{\mathsf{CBC[AES\text{-}256]}}^{\mathsf{otind\$}}(\mathcal{D}_{\text{OTIND\$}}).\end{aligned}$$

Qualitatively, Corollary 1 shows that the confidentiality of the MTProto-based channel depends on whether SHACAL-1 and SHACAL-2 can be considered as pseudorandom functions in a variety of modes: with keys used only once, related keys, partially chosen-keys when evaluated on fixed inputs and when the key and input switch positions. Especially the related-key assumptions (LRKPRF and HRKPRF given in Section V-B) are highly unusual; both assumptions hold in the ideal cipher model, but require further study in the standard model. Quantitatively, a limiting term in the advantage, which implies security only if $q_{\text{CH}} < 2^{64}$, is a result of the birthday bound on the MAC output, though we note that we do not have a corresponding attack in this setting and thus the bound may not be tight.

Corollary 2. *Let* $\mathsf{session\_id} \in \{0,1\}^{64}$, $\mathsf{pb} \in \mathbb{N}$ *and* $\mathsf{bl} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathsf{session\_id}, \mathsf{pb}, \mathsf{bl}]$, $\mathsf{MTP\text{-}HASH}$, $\mathsf{MTP\text{-}MAC}$, $\mathsf{MTP\text{-}KDF}$, ϕ_{MAC}, ϕ_{KDF}, $\mathsf{MTP\text{-}SE}$ *be the primitives of MTProto defined in Section IV-D. Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{MTP\text{-}HASH}, \mathsf{MTP\text{-}MAC}, \mathsf{MTP\text{-}KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{MTP\text{-}SE}]$. *Let* $\phi_{\mathsf{SHACAL\text{-}2}}$ *be the related-key-deriving function defined in Fig. 20. Let* $\mathsf{SHA\text{-}256}'$ *be* $\mathsf{SHA\text{-}256}$ *with its output truncated to the middle 128 bits. Let* $\mathsf{supp} = \mathsf{SUPP}$ *be the support function as defined in Fig. 23. Let* $\mathcal{F}_{\text{INT}}$ *be any adversary against the* INT-*security of* CH *with respect to* supp, *making* $q_{\text{SEND}} \leq 2^{96}$ *queries to its* SEND *oracle. Then there exist adversaries* $\mathcal{D}_{\text{OTPRF}}$, $\mathcal{D}_{\text{LRKPRF}}$, $\mathcal{F}_{\text{CR}}$ *such that*

$$\begin{aligned}\mathsf{Adv}_{\mathsf{CH},\mathsf{supp}}^{\mathsf{int}}(\mathcal{F}_{\text{INT}}) \leq 2 \cdot \Big(&\mathsf{Adv}_{\mathsf{SHACAL\text{-}1}}^{\mathsf{otprf}}(\mathcal{D}_{\text{OTPRF}})\\ &+ \mathsf{Adv}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}^{\mathsf{lrkprf}}(\mathcal{D}_{\text{LRKPRF}})\Big)\\ &+ \frac{q_{\text{SEND}}}{2^{64}} + \mathsf{Adv}_{\mathsf{SHA\text{-}256}'}^{\mathsf{cr}}(\mathcal{F}_{\text{CR}}).\end{aligned}$$

Qualitatively, Corollary 2 shows that also the integrity of the MTProto-based channel depends on SHACAL-1 and SHACAL-2 behaving as PRFs. Due to the way MTP-MAC is constructed, the result also depends on the collision resistance of truncated-output SHA-256 (as discussed in Section V-A3). Quantitatively, the advantage is again bounded by $q_{\text{SEND}} < 2^{64}$. This bound follows from the fact that the first block of payload contains a 64-bit constant session_id which has to match upon decoding. If the MTProto message encoding scheme consistently checked more fields during decoding (especially in the first block), the bound could be improved.

[23]The length of plaintext m in MTProto is $\ell := |m| \leq 2^{27}$ bits. To build a payload p, algorithm ME.Encode prepends a 256-bit header, and appends at most $\mathsf{bl} \cdot (\mathsf{pb}+1)$-bit padding. Further evaluation of MAC on p might append at most 512 additional bits of SHA padding.

VI. Timing side-channel attack

We present a timing side-channel attack against implementations of MTProto. The attack arises from MTProto's reliance on an Encrypt & MAC construction, the malleability of IGE mode, and specific weaknesses in implementations. The attack proceeds in the spirit of [12]: move a target ciphertext block to a position where the underlying plaintext will be interpreted as a length field and use the resulting behaviour to learn some information. The attack is complicated by Telegram using IGE mode instead of CBC mode analysed in [12]. We begin by describing a generic way to overcome this obstacle in Section VI-1. We describe a side channel found in the Telegram desktop client in Section VI-2 (we treat the iOS and Android clients in the full version) and experimentally demonstrate the existence of a timing side channel in that client in Section VI-4.

1) Manipulating IGE: Suppose we intercept an IGE ciphertext c consisting of t blocks (for any block cipher E): $c_1 \mid c_2 \mid \ldots \mid c_t$ where $\mid$ denotes a block boundary. Further, suppose we have a side channel that enables us to learn some bits of m_2, the second plaintext block.[24] In IGE mode, we have $c_i = E_K(m_i \oplus c_{i-1}) \oplus m_{i-1}$ for $i = 1, 2, \ldots, t$ (see Section II). Fix a target block number i for which we are interested in learning a portion of m_i that is encrypted in c_i. Assume we know the plaintext blocks m_1 and m_{i-1}.

We construct a ciphertext $c_1 \mid c^\star$ where $c^\star := c_i \oplus m_{i-1} \oplus m_1$. This is decrypted in IGE mode as follows:

$$\begin{aligned}m_1 &= E_K^{-1}(c_1 \oplus IV_m) \oplus IV_c\\ m^\star &= E_K^{-1}(c^\star \oplus m_1) \oplus c_1 = E_K^{-1}(c_i \oplus m_{i-1}) \oplus c_1\\ &= m_i \oplus c_{i-1} \oplus c_1\end{aligned}$$

Since we know c_1 and c_{i-1}, we can recover some bits of m_i if we can obtain the corresponding bits of $m^\star$ (e.g. through a side channel leak).

To motivate our known plaintext assumption, consider a message where m_{i-1} = "Today's password" and m_i = "is SECRET". Here m_{i-1} is known, while learning bytes of m_i is valuable. On another hand, the requirement of knowing m_1 may not be easy to fulfil in MTProto. The first plaintext block of an MTProto payload always contains server_salt || session_id, both of which are random values. It is unclear whether they were intended to be secret, but in effect they are, limiting the applicability of this attack. Appendix A gives an attack to recover these values. Note that these values are the same for all ciphertexts within a single session, so if they were recovered, then we could carry out the attack on each of the ciphertexts in turn. This allows the basic attack above to be iterated when the target m_i is fixed across all the ciphertexts, e.g. in order to amplify the total information learned about m_i when a single ciphertext allows to infer only a partial or noisy information about it (cf. [12]).

[24]The attack is easy to adapt to a different block.

2) Leaky length field: The preceding attack assumes we have a side channel that enables us to learn part of m_2. We now show how such side channels arise in implementations.

The msg_length field occupies the last four bytes of the second block of every MTProto cloud message plaintext (see Section IV-A). After decryption, the field is checked for validity in Telegram clients. Crucially, in several implementations this check is performed *before* the MAC check, i.e. before msg_key is recomputed from the decrypted plaintext. If either of those checks fails, the client closes the connection without outputting a specific error message. However, if an implementation is not constant time, an attacker who submits modified ciphertexts of the form described above may be able to distinguish between an error arising from validity checking of msg_length and a MAC error, and thus learn something about the bits of plaintext in the position of the msg_length field.

Since different Telegram clients implement different checks on the msg_length field, the full version proceeds to a case-by-case analysis for the Android, Desktop and iOS clients. Due to space restrictions we only treat the Desktop client here.

Here the length check is performed in the method `handleReceived` of `session_private.cpp` [41], which compares the `messageLength` field with a fixed value of `kMaxMessageLength` = 2^{24}. When this check fails, the connection is closed and no MAC check is performed, providing a potentially large timing difference. Because of the fixed value 2^{24}, this check would leak the 8 most significant bits of the target block m_i with probability 2^{-8}, i.e. the eight most significant bits of the 32-bit length field, allowing those bits to be recovered after about 2^8 attempts on average.[25]

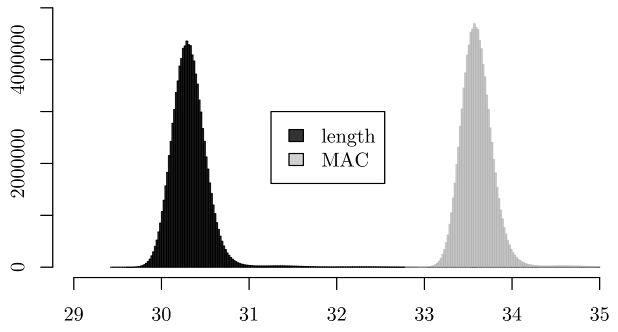
```
if (messageLength > kMaxMessageLength) {
        LOG(("TCP Error: bad messageLength %1").arg(
                messageLength));
        TCP_LOG(("TCP Error: bad message %1").arg(
                Logs::mb(ints,
                         intsCount * kIntSize).str()));

        return restart();
}
// ...
// MAC computation and check follow
```

3) Discussion: Note that all three of the Desktop, Android and iOS clients were in violation of Telegram's own security guidelines [42] which state: "If an error is encountered before this check could be performed, the client must perform the msg_key check anyway before returning any result. Note that the response to any error encountered before the msg_key check must be the same as the response to a failed msg_key check." In contrast, `TDLib` [11], the cross-platform library for building Telegram clients, does avoid timing leaks by running the MAC check first.

4) Practical experiments: We ran experiments to verify whether the side channel present in the desktop client code is exploitable. We measured the time difference between processing a message with a wrong msg_length and processing a message with a correct msg_length but a wrong MAC. This was done using the Linux desktop client, modified to process messages generated on the client side without engaging the network. We collected data for 10^8 trials for each case under ideal conditions, i.e. with hyper-threading, Turbo Boost etc. disabled. After removing outliers, the difference in means was about 3 microseconds, see Fig. 27. This should be sufficiently large for a remote attacker to detect, even with network and other noise sources (cf. [43], where sub-microsecond timing differences were successfully resolved over a LAN).

Figure 27: Processing time of `SessionPrivate::handleReceived` in microseconds.



VII. Discussion

The central result of this work is that the use of symmetric encryption in Telegram's MTProto 2.0 can provide the basic security expected from a bidirectional channel if small modifications are made. The Telegram developers have indicated that they implemented most of these changes. Thus, our work can give some assurance to those reliant on Telegram providing confidential and integrity-protected cloud chats – at a comparable level to chat protocols that run over TLS's record protocol. However, our work comes with a host of caveats.

Attacks: Our work also presents attacks against the symmetric encryption in Telegram. These highlight the gap between the variant of MTProto 2.0 that we model and Telegram's implementations. While the reordering attack in Section IV-B1 and the attack on IND-CPA security in Section IV-B2 were possible against implementations that we studied, they can easily be avoided without making changes to the on-the-wire format of MTProto, i.e. by only changing processing in clients and servers. After disclosing our findings, Telegram informed us that they have changed this processing accordingly.

Our attacks in Section VI are attacks on the implementation. As such, they can be considered outside the model: our model only shows that there *can* be secure instantiations of MTProto but does not cover the actual implementations; in particular, we do not model timing differences. That said, protocol design has a significant impact on the ease with which secure implementations can be achieved. Here, the decision in MTProto to adopt Encrypt & MAC results in the potential for a leak that we can exploit in specific implementations. This "brittleness" of MTProto is of particular relevance due

[25]Note that this beats random guessing as the correct value can be recognised.

to the surfeit of implementations of the protocol, and the fact that security advice may not be heeded by all authors.[26] Here Telegram's apparent ambition to provide TDLib as a one-stop solution for clients across platforms will allow security researchers to focus their efforts. We thus recommend that Telegram replaces the low-level cryptographic processing in all official clients with a carefully vetted library.

Tightness: On the other hand, our proofs are not necessarily tight. That is, our theorem statements contain terms bounding the advantage by $\approx q/2^{64}$ where q is the number of queries sent by the adversary. Yet, we have no attacks matching these bounds (our attacks with complexity 2^{64} are outside the model). Thus, it is possible that a refined analysis would yield tighter bounds.

Future work: Our attack in Appendix A is against the implementation of Telegram's key exchange and is thus outside of our model for two reasons: as before, we do not consider timing side channels in our model and, critically, we only model the symmetric part of MTProto. This highlights a second significant caveat for our results that large parts of Telegram's design remain unstudied: multi-user security, the key exchange, the higher-level message processing, secret chats, forward secrecy, control messages, bot APIs, CDNs, cloud storage, the Passport feature, to name but a few. These are pressing topics for future work.

Assumptions: In our proofs we are forced to rely on unstudied assumptions about the underlying primitives used in MTProto. In particular, we have to make related-key assumptions about the compression function of SHA-256 which could be easily avoided by tweaking the use of these primitives in MTProto. In the meantime, these assumptions represent interesting targets for symmetric cryptography research. Similarly, the complexity of our proofs and assumptions largely derives from MTProto deploying hash functions in place of (domain-separated) PRFs such as HMAC. We recommend that Telegram either adopts well-studied primitives for future versions of MTProto to ease analysis and thus to increase confidence in the design, or adopts TLS.

Telegram: While we prove security of the symmetric part of MTProto at a protocol level, we recall that by default communication via Telegram must trust the Telegram servers, i.e. end-to-end encryption is optional and not available for group chats. We thus, on the one hand, (a) recommend that Telegram open-sources the cryptographic processing on their servers and (b) recommend to avoid referencing Telegram as an "encrypted messenger" which – post-Snowden – has come to mean end-to-end encryption. On the other hand, discussions about end-to-end encryption aside, echoing [2], [3] we note that many higher-risk users *do* rely on MTProto and Telegram and shun Signal. This emphasises the need to study these technologies and how they serve those who rely on them.

[26]Indeed, the Telegram developers rule out length-extension attacks in [44] because the MAC is computed on the plaintext and because any change in the MAC affects the decryption key and thus the decrypted plaintext, which makes it unlikely that the integrity check passes. This is largely correct but only under the assumption that msg_id is actually unique and re-encryption of messages with the same msg_id is not allowed. That is, the condition given by the developers in the FAQ was violated by several official Telegram clients.

Acknowledgements

We thank Mihir Bellare for discussions and insights. The research of Mareková was supported by the EPSRC and the UK Government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1). The research of Paterson was supported in part by a gift from VMware.

References

[1] Telegram, "500 million users," https://t.me/durov/147, Feb 2021.

[2] K. Ermoshina, H. Halpin, and F. Musiani, "Can Johnny build a protocol? co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols," in *European Workshop on Usable Security*, 2017.

[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," to appear at USENIX'21, pre-print at https://arxiv.org/abs/2105.14869, 2021.

[4] J. Jakobsen and C. Orlandi, "On the CCA (in)security of MTProto," *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices - SPSM'16*, 2016. [Online]. Available: http://dx.doi.org/10.1145/2994459.2994468

[5] T. Sušánka and J. Kokeš, "Security analysis of the Telegram IM," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, 2017, pp. 1–8.

[6] N. Kobeissi, "Formal Verification for Real-World Cryptographic Protocols and Implementations," Theses, INRIA Paris ; Ecole Normale Supérieure de Paris - ENS Paris, Dec. 2018, https://hal.inria.fr/tel-01950884.

[7] M. Miculan and N. Vitacolonna, "Automated symbolic verification of Telegram's MTProto 2.0," in *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021*, S. De Capitani di Vimercati and P. Samarati, Eds. SciTePress, 2021, pp. 185–197.

[8] M. Fischlin, F. Günther, and C. Janson, "Robust channels: Handling unreliable networks in the record layers of QUIC and DTLS 1.3," Cryptology ePrint Archive, Report 2020/718, 2020, https://eprint.iacr.org/2020/718.

[9] Telegram, "End-to-end encryption, secret chats – sending a request," http://web.archive.org/web/20210126013030/https://core.telegram.org/api/end-to-end#sending-a-request, Feb 2021.

[10] ——, "tdlib," https://github.com/tdlib/td, Sep 2020.

[11] ——, "tdlib – Transport.cpp," https://github.com/tdlib/td/blob/v1.7.0/td/mtproto/Transport.cpp#L272, Apr 2021.

[12] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *2009 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2009, pp. 16–26.

[13] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *EUROCRYPT 2006*, ser. LNCS, S. Vaudenay, Ed., vol. 4004. Springer, Heidelberg, May / Jun. 2006, pp. 409–426.

[14] C. Campbell, "Design and specification of cryptographic capabilities," *IEEE Communications Society Magazine*, vol. 16, no. 6, pp. 15–19, 1978.

[15] C. Jutla, "Attack on free-mac, sci.crypt," https://groups.google.com/forum/#!topic/sci.crypt/4bkzm_n7UGA, Sep 2000.

[16] M. Bellare, A. Boldyreva, L. R. Knudsen, and C. Namprempre, "On-line ciphers and the hash-CBC constructions," *Journal of Cryptology*, vol. 25, no. 4, pp. 640–679, Oct. 2012.

[17] NIST, "FIPS 180-4: Secure Hash Standard," 2015, http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[18] H. Handschuh and D. Naccache, "SHACAL (-submission to NESSIE-)," *Proceedings of First Open NESSIE Workshop*, 2000, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.4066&rep=rep1&type=pdf.

[19] G. A. Marson and B. Poettering, "Security notions for bidirectional channels," *IACR Trans. Symm. Cryptol.*, vol. 2017, no. 1, pp. 405–426, 2017.

[20] M. Bellare, T. Kohno, and C. Namprempre, "Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol," in *ACM CCS 2002*, V. Atluri, Ed. ACM Press, Nov. 2002, pp. 1–11.

[21] T. Kohno, A. Palacio, and J. Black, "Building secure cryptographic transforms, or how to encrypt and MAC," Cryptology ePrint Archive, Report 2003/177, 2003, http://eprint.iacr.org/2003/177.

[22] C. Boyd, B. Hale, S. F. Mjølsnes, and D. Stebila, “From stateless to stateful: Generic authentication and authenticated encryption constructions with application to TLS,” in *CT-RSA 2016*, ser. LNCS, K. Sako, Ed., vol. 9610. Springer, Heidelberg, Feb. / Mar. 2016, pp. 55–71.
[23] P. Rogaway and Y. Zhang, “Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE,” in *CRYPTO 2018, Part II*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10992. Springer, Heidelberg, Aug. 2018, pp. 3–32.
[24] Telegram, “Mobile protocol: Detailed description,” http://web.archive.org/web/20210126200309/https://core.telegram.org/mtproto/description, Jan 2021.
[25] ——, “Schema,” https://core.telegram.org/schema, Sep 2020.
[26] ——, “TL language,” https://core.telegram.org/mtproto/TL, Sep 2020.
[27] Google, “BoringSSL AES IGE implementation,” https://github.com/DrKLO/Telegram/blob/d073b80063c568f31d81cc88c927b47c01a1dbf4/TMessagesProj/jni/boringssl/crypto/fipsmodule/aes/aes_ige.c, Jul 2018.
[28] Telegram, “MTProto transports,” http://web.archive.org/web/20200527124125/https://core.telegram.org/mtproto/mtproto-transports, May 2020.
[29] ——, “Sequence numbers in secret chats,” http://web.archive.org/web/20201031115541/https://core.telegram.org/api/end-to-end/seq_no, Jan 2021.
[30] K. Ludwig, “Trudy - Transparent TCP proxy,” 2017, https://github.com/praetorian-inc/trudy.
[31] Telegram, “Telegram Desktop – mtproto_serialized_request.cpp,” https://github.com/telegramdesktop/tdesktop/blob/v2.5.8/Telegram/SourceFiles/mtproto/details/mtproto_serialized_request.cpp#L15, Feb 2021.
[32] ——, “Mobile protocol: Detailed description – server salt,” http://web.archive.org/web/20210221134408/https://core.telegram.org/mtproto/description#server-salt, Feb 2021.
[33] ——, “Telegram Android – Datacenter.cpp,” https://github.com/DrKLO/Telegram/blob/release-7.4.0_2223/TMessagesProj/jni/tgnet/Datacenter.cpp#L1171, Feb 2021.
[34] ——, “Telegram Desktop – session_private.cpp,” https://github.com/telegramdesktop/tdesktop/blob/v2.6.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1338, Mar 2021.
[35] ——, “Notice of ignored error message,” http://web.archive.org/web/20200527121939/https://core.telegram.org/mtproto/service_messages_about_messages#notice-of-ignored-error-message, May 2020.
[36] M. Bellare and T. Kohno, “A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications,” in *EUROCRYPT 2003*, ser. LNCS, E. Biham, Ed., vol. 2656. Springer, Heidelberg, May 2003, pp. 491–506.
[37] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, “A concrete security treatment of symmetric encryption,” in *38th FOCS*. IEEE Computer Society Press, Oct. 1997, pp. 394–403.
[38] P. Rogaway, “Nonce-based symmetric encryption,” in *FSE 2004*, ser. LNCS, B. K. Roy and W. Meier, Eds., vol. 3017. Springer, Heidelberg, Feb. 2004, pp. 348–359.
[39] J. Kim, G. Kim, S. Lee, J. Lim, and J. H. Song, “Related-key attacks on reduced rounds of SHACAL-2,” in *INDOCRYPT 2004*, ser. LNCS, A. Canteaut and K. Viswanathan, Eds., vol. 3348. Springer, Heidelberg, Dec. 2004, pp. 175–190.
[40] J. Lu, J. Kim, N. Keller, and O. Dunkelman, “Related-key rectangle attack on 42-round SHACAL-2,” in *ISC 2006*, ser. LNCS, S. K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, Eds., vol. 4176. Springer, Heidelberg, Aug. / Sep. 2006, pp. 85–100.
[41] Telegram, “Telegram Desktop – session_private.cpp,” https://github.com/telegramdesktop/tdesktop/blob/v2.7.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1258, Apr 2021.
[42] ——, “Security guidelines for client developers,” http://web.archive.org/web/20210203134436/https://core.telegram.org/mtproto/security_guidelines#mtproto-encrypted-messages, Feb 2021.
[43] N. J. AlFardan and K. G. Paterson, “Lucky thirteen: Breaking the TLS and DTLS record protocols,” in *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 526–540.
[44] Telegram, “FAQ for the Technically Inclined – length extension attacks,” http://web.archive.org/web/20210203134422/https://core.telegram.org/techfaq#length-extension-attacks, Feb 2021.
[45] D. Bleichenbacher, “Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1,” in *CRYPTO’98*, ser. LNCS, H. Krawczyk, Ed., vol. 1462. Springer, Heidelberg, Aug. 1998, pp. 1–12.
[46] G. D. Micheli and N. Heninger, “Recovering cryptographic keys from partial information, by example,” Cryptology ePrint Archive, Report 2020/1506, 2020, https://eprint.iacr.org/2020/1506.
[47] M. R. Albrecht and N. Heninger, “On Bounded Distance Decoding with predicate: Breaking the "lattice barrier" for the Hidden Number Problem,” Cryptology ePrint Archive, Report 2020/1540, 2020, https://eprint.iacr.org/2020/1540.

Appendix

A. Attacking the key exchange

During the key exchange, a client sends an RSA-encrypted message $m := (h_r, \gamma, n', p_r)$ to the server with $h_r := \mathsf{SHA\text{-}1}(\gamma \parallel n')$, γ a known constant, $n' \in \{0,1\}^{256}$ and p_r some unknown padding. The tag h_r is meant to provide integrity. Our target secret is n'. Note that the $\mathsf{SHA\text{-}1}(\cdot)$ does not include the padding but that γ has a variable bit-length (known to the attacker). Thus, the payload must be parsed after decryption *before* verifying its integrity, which enables a potential timing side channel. While we were unable to establish the parsing order of the Telegram servers or if they defend against such leaks, the Telegram developers confirmed to us the existence of vulnerable behaviour on the server during the disclosure process.

If we assume that server-side parsing proceeds analogously to similar parsing in TDlib then a 32-bit header value ζ (part of γ) is checked first and the parsing function terminates early when it does not match. Assuming further that this event is detectable through a time difference, this instantiates an oracle leaking when certain 32 bits match a known value. Our attack then proceeds in the style of Bleichenbacher's attack [45]. Writing $c := m^e \bmod N'$ for the ciphertext observed by the attacker – where e, N' is the server's public RSA key – we submit $s_i^e \cdot c$ – for carefully sampled s_i – to our oracle to learn whether $s_i \cdot m$ is such that the target 32 bits match the expected header value. Collecting several such answers we can then recover m and thus n'.

A complication is that Bleichenbacher's adaptive recovery method – iteratively restricting the interval – is not available to us since we learn the value of some middle bits rather than the most significant bits. Writing y for the bit position of ζ, we observe that $(s_i \cdot m \bmod N') - 2^y \cdot \zeta \bmod 2^{y+32} \ll 2^{y+32}$, i.e. that the correct value m produces an unusually short value modulo 2^{y+32}. This enables us to use lattice reduction to find m using known techniques [46], [47].

Knowing n' implies knowing $\mathsf{server\_salt}$. To recover $\mathsf{session\_id}$ we can then run a guess and verify attack using the techniques from Section VI. Alternatively, we observe that n' is later used in the key exchange to protect the integrity of Diffie-Hellman shares g^a and g^b. Thus, our attack would also enable an attacker-in-the-middle (MitM) attack on the key exchange. The full version contains the details and a proof of concept implementation of the lattice reduction part.