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. We chain the third attack with a fourth one against the implementation of the key exchange protocol on Telegram's servers. This fourth attack breaks the authentication properties of Telegram's key exchange, allowing a MitM attack. More mundanely, it also recovers the id field, reducing the cost of the plaintext recovery attack to guessing the 64-bit salt field. In totality, our results provide the first comprehensive study of MTProto's use of symmetric cryptography, as well as highlight weaknesses in its key exchange.**

This is the full version of a work to appear at IEEE S&P 2022, preparation date: 16 July 2021.

I. Introduction

Telegram is a chat platform that, as of January 2021, reportedly has 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 these works 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.

Security model: Starting from the observation that MTProto entangles the keys of the two channel directions, we develop in Section III 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. We also consider a combined security notion and its relationship to the individual notions. 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 altogether (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 how messages are delivered, 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: We then provide a formal and detailed model for Telegram's symmetric encryption in Section IV. 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, and different clients do not implement the same behaviour.

Formally, we define an MTProto-based bidirectional channel $\mathsf{MTP\text{-}CH}$ as a composition of multiple cryptographic primitives. This allows us to then recover a variant of the real-world MTProto protocol by instantiating these primitives with specific constructions, and separately study whether each of them satisfies the security notions that are required in order to achieve the desired security of $\mathsf{MTP\text{-}CH}$. 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 $\mathsf{MTP\text{-}CH}$.

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 to reflect the attacker-altered server's view later. 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. The consequences of such an attack can be quite severe, as we discuss further in Appendix C.

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 prove in Section V that our slight variant of MTProto can achieve channel confidentiality and integrity in our model. While our proof does not carry over to MTProto as currently deployed by Telegram (as explained above), it shows that strong notions of channel security are achievable with only minor alterations.

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, 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 obtaining a concrete security analysis. This results in security theorems that tightly 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, show that MTProto achieves security of $q/2^{64}$ 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 SHA-256 compression function – based on SHACAL-2 – 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 motivating further research on related-key attacks. On the other hand, we note that our proofs side-step concerns about length-extension attacks by relying on the underlying payload format.

Attacks: We present further implementation attacks against Telegram in Section VI and Appendix F. 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.[1] 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 bits of plaintext. We give examples from the official Desktop, Android and iOS Telegram clients, each exhibiting a different timing side channel. 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 enable our plaintext-recovery attack, i.e. to recover m_1, in Appendix F we chain it with another attack on the implementation of Telegram's server-side key exchange protocol. This attack exploits how Telegram servers process RSA ciphertexts. We note that while the exploited behaviour was confirmed by the Telegram developers we did not verify it with an experiment.[2] It uses a combination of lattice reduction and Bleichenbacher-like techniques [13]. 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 reduces the overall security of Telegram, essentially, to guessing the 64-bit salt field. Details can be found in Appendix F. We stress, though, that even if all assumptions we make in Appendix F are met, our exploit chain (Section VI, Appendix F) – 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.

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

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

B. Disclosure

We notified Telegram's developers about the vulnerabilities that 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. As a consequence the fixes were being 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.

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 $x, y \in \{0,1\}^*$, $x \,\|\, y$ 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 $\bot \notin \{0,1\}^*$ denotes an empty table position or an error code that indicates rejection, such as invalid input to an algorithm. We may use subscripts to indicate that $\bot_0, \bot_1, \ldots$ denote distinct error codes.

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 $\bot$, then all of its outputs are $\bot$. 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 $x_1, \ldots$. The instruction **abort**$(x_1, \ldots)$ is used to immediately halt the algorithm with output $(x_1, \ldots)$. Adversaries are algorithms. We require that adversaries never pass $\bot$ as input to their oracles.

3) Security games and reductions: We use the code-based game-playing framework of [14]. (See Fig. 2 for an example.) $\Pr[\,\mathrm{G}\,]$ 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. Let $\mathrm{G}_{\mathcal{D}}$ be any security game defining a decision-based problem that requires an adversary $\mathcal{D}$ to guess a challenge bit d; let d' denote the output of $\mathcal{D}$, and let game $\mathrm{G}_{\mathcal{D}}$ return true iff $d' = d$. Depending on the context, we interchangeably use the two equivalent advantage definitions for such games: $\mathsf{Adv}(\mathcal{D}) = 2 \cdot \Pr[\,\mathrm{G}_{\mathcal{D}}\,] - 1$, and $\mathsf{Adv}(\mathcal{D}) = \Pr[\,d' = 1 \mid d = 1\,] - \Pr[\,d' = 1 \mid d = 0\,]$.

4) Implicit initialisation values: In algorithms and games, 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.

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 $\mathsf{Adv}^{\mathsf{cr}}_f(\mathcal{F}) = \Pr\left[\,\mathrm{G}^{\mathsf{cr}}_{f,\mathcal{F}}\,\right]$. 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 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 pseudorandomness 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 $\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{F}}(\mathcal{D}) = 2 \cdot \Pr\left[\,\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}\,\right] - 1$. The game samples a uniformly random challenge bit b and runs adversary $\mathcal{D}$, providing it with access to oracle $\textsc{RoR}$. The oracle takes $x \in \mathsf{F.In}$ as input, and the adversary is allowed to query the oracle arbitrarily many times. Each time $\textsc{RoR}$ 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 $G^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$
    $b \leftarrow\$ \{0,1\}$ ; $b' \leftarrow\$ \mathcal{D}^{\text{RoR}}$
    Return $b' = b$
RoR($x$)
    $fk \leftarrow\$ \mathsf{F.Keys}$ ; $y_1 \leftarrow \mathsf{F.Ev}(fk, x)$
    $y_0 \leftarrow\$ \{0,1\}^{\mathsf{F.ol}}$ ; Return $y_b$
```

Figure 2: One-time pseudorandomness of function family F.

5) Symmetric encryption schemes: A symmetric encryption scheme SE specifies algorithms SE.Enc and SE.Dec, where 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 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 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 SE.Enc is deterministic.

6) One-time indistinguishability of SE: Consider game $G^{\mathsf{otind\$}}$ 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 $\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}) = 2 \cdot \Pr\left[G^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}} \right] - 1$. The game proceeds as the OTPRF game.

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

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

7) CBC block cipher mode of operation: Let E be a block cipher. Define the Cipher Block Chaining (CBC) mode of operation as a symmetric encryption scheme CBC[E] as shown in Fig. 4, where key length is $\mathsf{SE.kl} = \mathsf{E.kl} + \mathsf{E.ol}$, 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 SE.cl is the identity function. Note that Fig. 4 gives a somewhat non-standard definition for CBC, as it includes the IV (c_0) as part of the key material. However, in this work, we are only interested in one-time security of SE, so keys and IVs are generated together and the IV is not included as part of the ciphertext.

8) IGE block cipher mode of operation: Let E be a block cipher. Define the Infinite Garble Extension (IGE) mode of operation as IGE[E] as in Fig. 4, with parameters as in the CBC mode except for key length $\mathsf{SE.kl} = \mathsf{E.kl} + 2 \cdot \mathsf{E.ol}$ (since IGE has two IV blocks which we again include as part of the key). We depict IGE decryption in Fig. 5 as we rely on this in Section VI. IGE was first defined in [15], which claims it has infinite error propagation and thus can provide integrity. This claim was disproved in an attack on Free-MAC [16], which has the same specification as IGE. [16] shows that given a plaintext-ciphertext pair it is possible to construct another ciphertext that will correctly decrypt to a plaintext such that only two of its blocks differ from the original plaintext, i.e. the "errors" introduced in the ciphertext do not propagate forever. IGE also appears as a special case of the Accumulated Block Chaining (ABC) mode [17]. A chosen-plaintext attack on ABC that relied on IV reuse between encryptions was described in [18].

```
CBC[E].Enc($k,m$)
    $K \leftarrow k[0 : \mathsf{E.kl}]$
    $c_0 \leftarrow k[\mathsf{E.kl} : \mathsf{SE.kl}]$
    For $i = 1, \ldots, t$ do
        $c_i \leftarrow \mathsf{E.Ev}(K, m_i \oplus c_{i-1})$
    Return $c_1 \| \ldots \| c_t$
CBC[E].Dec($k,c$)
    $K \leftarrow k[0 : \mathsf{E.kl}]$
    $c_0 \leftarrow k[\mathsf{E.kl} : \mathsf{SE.kl}]$
    For $i = 1, \ldots, t$ do
        $m_i \leftarrow \mathsf{E.Inv}(K, c_i) \oplus c_{i-1}$
    Return $m_1 \| \ldots \| m_t$

IGE[E].Enc($k,m$)
    $K \leftarrow k[0 : \mathsf{E.kl}]$
    $c_0 \leftarrow k[\mathsf{E.kl} : \mathsf{E.kl} + \mathsf{E.ol}]$
    $m_0 \leftarrow k[\mathsf{E.kl} + \mathsf{E.ol} : \mathsf{SE.kl}]$
    For $i = 1, \ldots, t$ do
        $c_i \leftarrow \mathsf{E.Ev}(K, m_i \oplus c_{i-1}) \oplus m_{i-1}$
    Return $c_1 \| \ldots \| c_t$
IGE[E].Dec($k,c$)
    $K \leftarrow k[0 : \mathsf{E.kl}]$
    $c_0 \leftarrow k[\mathsf{E.kl} : \mathsf{E.kl} + \mathsf{E.ol}]$
    $m_0 \leftarrow k[\mathsf{E.kl} + \mathsf{E.ol} : \mathsf{SE.kl}]$
    For $i = 1, \ldots, t$ do
        $m_i \leftarrow \mathsf{E.Inv}(K, c_i \oplus m_{i-1}) \oplus c_{i-1}$
    Return $m_1 \| \ldots \| m_t$
```

Figure 4: Constructions of deterministic symmetric encryption schemes CBC[E] and IGE[E] from block cipher E. Consider t as the number of blocks of m (or c), i.e. $m = m_1 \| \ldots \| m_t$.

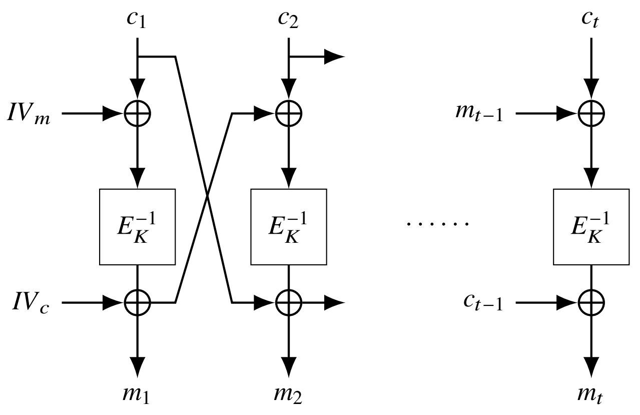


Figure 5: IGE mode decryption, where $c_0 = IV_c$ and $m_0 = IV_m$ are the initial values so decryption can be expressed as $m_i = E_K^{-1}(c_i \oplus m_{i-1}) \oplus c_{i-1}$.

9) MD transform: Fig. 6 defines the Merkle-Damgård transform as a function family MD[h] for a given compression function $h\colon \{0,1\}^{\ell} \times \{0,1\}^{\ell'} \to \{0,1\}^{\ell}$, with $\mathsf{MD.In} = \bigcup_{t \in \mathbb{N}} \{0,1\}^{\ell' \cdot t}$, $\mathsf{MD.Keys} = \{0,1\}^{\ell}$ and $\mathsf{MD.ol} = \ell$.[3]

10) SHA-1 and SHA-256: 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 as defined in [19]. We will refer to their compression functions as $h_{160} : \{0,1\}^{160} \times \{0,1\}^{512} \to$

[3]Traditionally, MD[h] is unkeyed, but it is convenient at points in our analysis to think of it as being keyed. When creating a hash function like SHA-1 or SHA-256 from MD[h], the key is fixed to a specific IV value.

```
MD.Ev(k, $x_1 \| \dots \| x_t$)
    $H_0 \leftarrow$ k
    For $i = 1, \dots, t$ do $H_i \leftarrow h(H_{i-1}, x_i)$
    Return $H_t$
```

```
SHA-pad($x$)    // $|x| < 2^{64}$
    $L \leftarrow (447 - |x|) \bmod 512$
    $x' \leftarrow x \| 1 \| 0^L \| \langle |x| \rangle_{64}$
    Return $x'$
```

Figure 6: Left pane: Construction of MD-transform $\mathsf{MD} = \mathsf{MD}[h]$ from compression function h. Right pane: SHA-pad pads SHA-1 or SHA-256 input x to a length that is a multiple of 512 bits.

$\{0,1\}^{160}$ and $h_{256} : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$, and to their initial states as IV_{160} and IV_{256}. We can write $\mathsf{SHA\text{-}1}(x) = \mathsf{MD}[h_{160}].\mathsf{Ev}(\mathsf{IV}_{160}, \mathsf{SHA\text{-}pad}(x))$ and $\mathsf{SHA\text{-}256}(x) = \mathsf{MD}[h_{256}].\mathsf{Ev}(\mathsf{IV}_{256}, \mathsf{SHA\text{-}pad}(x))$ where SHA-pad is defined in Fig. 6.

11) SHACAL-1 and SHACAL-2: Let $\hat{+}$ be an addition operator over 32-bit words, meaning for any $x, y \in \bigcup_{t \in \mathbb{N}} \{0,1\}^{32 \cdot t}$ with $|x| = |y|$ the instruction $z \leftarrow x \hat{+} y$ splits x and y into 32-bit words and independently adds together words at the same positions, each modulo 2^{32}; it then computes z by concatenating together the resulting 32-bit words. Let SHACAL-1 [20] be the block cipher defined with $\mathsf{SHACAL\text{-}1.kl} = 512$, $\mathsf{SHACAL\text{-}1.ol} = 160$ such that $h_{160}(k, x) = k \hat{+} \mathsf{SHACAL\text{-}1.Ev}(x, k)$. Similarly, let SHACAL-2 be the block cipher defined with $\mathsf{SHACAL\text{-}2.kl} = 512$, $\mathsf{SHACAL\text{-}2.ol} = 256$ such that $h_{256}(k, x) = k \hat{+} \mathsf{SHACAL\text{-}2.Ev}(x, k)$.

III. Bidirectional channels

A. Prior work

There is a significant body of prior work on modelling and constructing secure channels. Relevant here are the early work of [21] which introduced stateful security notions for symmetric encryption and used them to analyse SSH; a follow-up [22] which provided formal definitions for channels permitting message replay, reordering and deletion; follow-up works in this direction [23] and [8] (the latter introducing notions of robustness via support predicates, which we extend); recent work in the context of messaging protocols, e.g. [24], [25]; and work treating the case of causality in bidirectional channels [26]. We draw on all of this work in this section to develop functional and security definitions for bidirectional secure channels.

B. Definitions

A *channel* provides a method for two users to exchange messages. We refer to the two users of a channel as the initiator $\mathcal{I}$ and the receiver $\mathcal{R}$. These will map to client and server in the setting of MTProto. We use u as a variable to represent an arbitrary user and $\overline{u}$ to represent the other user. We use st_u to represent the channel state of user u. We associate abstract auxiliary information aux to each sent/received message. This should not be thought of as additional data in an AEAD scheme but rather a way to express e.g. time when message processing may depend on it.

Definition 1. *A channel* CH *specifies algorithms* CH.Init, CH.Send *and* CH.Recv, *where* CH.Recv *is deterministic. Associated to* CH *is a message space* CH.MS *and a randomness space* CH.SendRS *of* CH.Send. *The initialisation algorithm* CH.Init *returns $\mathcal{I}$'s and $\mathcal{R}$'s initial states $st_{\mathcal{I}}$ and $st_{\mathcal{R}}$. The sending algorithm* CH.Send *takes st_u for some $u \in \{\mathcal{I}, \mathcal{R}\}$, a plaintext $m \in$* CH.MS*, and auxiliary information aux to return the updated state st_u and a ciphertext c, where $c = \perp$ may be used to indicates a failure to send. We may surface random coins $r \in$* CH.SendRS *as an additional input to* CH.Send. *The receiving algorithm takes st_u, c, and auxiliary information aux' to return the updated state st_u and a plaintext $m \in$* CH.MS$\cup\{\perp\}$*, where $\perp$ indicates a failure to recover a message. The syntax used for the algorithms of* CH *is given in Fig. 7.*

```
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{CH.Init}()$
$(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m, aux; r)$
$(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux')$
```

Figure 7: Syntax of the constituent algorithms of channel CH.

We use transcripts to represent a record of all messages sent and received on the channel, indexed by an abstract label that could be the ciphertext or a unique encoding of each message.[4] Transcripts can include entries where the message m equals $\perp$ to capture that a received ciphertext 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). A label can also be equal to $\perp$, e.g. to indicate that a message 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* op = sent *indicates that user u attempted to send message m with auxiliary information aux, encoded into* label*. An entry with* op = recv *indicates that user u received* label *with auxiliary information aux, and decoded it into message m.*

We expand the definition of a support predicate from [8] to a support *function*, so that instead of representing merely the decision to accept/reject a given ciphertext, it either returns the message corresponding to a given ciphertext (signifying acceptance) or returns $\perp$. This will simplify our security definitions. To work in the bidirectional setting, the support function takes transcripts of both users as input. Our transcripts use abstract labels instead of ciphertexts, so we define a support function to take a label as input. We also let the support function take the auxiliary information 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 decode* label, aux *into message m^*.*

In our games, a call to $\mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$ is used to determine whether user u should accept an incoming message

[4] In the main channel security notions, this will be the ciphertext, but for notions that only reason about the plaintext it will be a message encoding.

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

$\textsc{Send}(u, m, aux, r)$
$(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m, aux; r)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$ ; Return $c$

$\textsc{Recv}(u, c, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
If $m^* = \perp$ then return $\perp$
$(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
If $m^* \neq m$ then $\mathsf{win} \leftarrow \mathsf{true}$
Return $m$
```

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

$\textsc{Send}(u, m, aux, r)$
$(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m, aux; r)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$ ; Return $c$

$\textsc{Recv}(u, c, aux)$
$(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
If $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
Return $m$
```

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

$\textsc{Ch}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m_b, aux; r)$
Return $c$

$\textsc{Recv}(u, c, aux)$
$(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
Return $\perp$
```

Figure 8: Correctness of channel CH; integrity of channel CH; indistinguishability of channel CH.

from $\overline{u}$ that is associated to label. We define two correctness properties of a support function. First, supp always returns a message that was honestly sent and delivered, i.e. it supports in-order delivery as in [8].[5] Second, supp always outputs $\perp$ if the queried label does not appear in $\mathsf{tr}_{\overline{u}}$. Formal definitions of both properties are in Fig. 35 and Fig. 36 in Appendix A. We do not constrain the support function further so that a range of channel behaviours such as strict in-order delivery or out-of-order delivery within a given time window can be captured.

C. Correctness and security of channels

For the following properties, consider the games in Fig. 8. We allow the adversary to control the randomness used by $\mathsf{CH.Send}$ since stateful encryption can achieve strong notions of security even in this setting.

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

[5] [8] defines this notion as part of the channel correctness game, but we choose to surface it as a separate property since for instance non-robust channels which output $\perp$ once a number of errors occurs cannot meet it.

2) Integrity: Consider the adversary $\mathcal{F}$ in the $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ game 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 $\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}\right]$. The adversary gets access to a $\textsc{Send}$ and a $\textsc{Recv}$ oracle (but not to the users' states). Both calls proceed as in the correctness game except that $\textsc{Recv}$ now does not limit $\mathcal{F}$ to only honestly produced ciphertexts, to capture the intuition that the adversary can manipulate ciphertexts on the network in an attempt to create a forgery. For example, let CH be a channel that produces unique ciphertexts. Take $\mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$ that returns m^* iff $(\mathsf{sent}, m^*, c, aux) \in \mathsf{tr}_{\overline{u}}$, and returns $\perp$ otherwise. Then integrity of CH with respect supp implies the standard notions of ciphertext integrity and plaintext integrity.

3) Privacy: Consider the adversary $\mathcal{D}$ in the $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ game associated to a channel CH. The advantage of $\mathcal{D}$ in breaking the IND-CPA security of CH is defined as $\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}\right] - 1$. The adversary can query the challenge oracle $\textsc{Ch}(u, m_0, m_1, aux, r)$ as an encryption oracle for user u with two messages 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 $\textsc{Recv}$ oracle. This is needed to model the feature that each user's state st_u may be updated every time a ciphertext is processed, potentially influencing subsequent encryption operations. However, the $\textsc{Recv}$ oracle does not return any information to $\mathcal{D}$ directly.

4) Authenticated encryption: Following the all-in-one definitional style of [27], Appendix B defines a single authenticated encryption game to capture both integrity and privacy, and shows equivalence with the combination of $\mathrm{G}^{\mathsf{ind}}$ and $\mathrm{G}^{\mathsf{int}}$ games.

D. Message encoding

To help with separation of functions within the channel, we define a primitive for message encoding such that $\mathsf{CH.Send}$ and $\mathsf{CH.Recv}$ could call it as a subroutine.

Definition 4. *A message encoding scheme* ME *specifies algorithms* $\mathsf{ME.Init}$, $\mathsf{ME.Encode}$ *and* $\mathsf{ME.Decode}$, *where*

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

Figure 9: Syntax of message encoding scheme ME.

ME.Decode is deterministic. Associated to ME is a message set $\mathsf{ME.MS} \subseteq \{0,1\}^*$*, a payload set* ME.Out*, a randomness space* ME.EncRS *of* ME.Encode*, a payload length function* $\mathsf{ME.pl}\colon (\mathbb{N} \cup \{0\}) \times \mathsf{ME.EncRS} \to \mathbb{N}$*, and the maximum number of messages* $\mathsf{ME.T} \in \mathbb{N}$ *the scheme can encode. The initialisation algorithm* ME.Init *returns* $\mathcal{I}$*'s and* $\mathcal{R}$*'s initial states* $st_{\mathcal{I}}$ *and* $st_{\mathcal{R}}$*. The encoding algorithm* 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* $p \in \mathsf{ME.Out}$*. We may surface random coins* $\nu \in \mathsf{ME.EncRS}$ *as an additional input to* ME.Encode*; then a message* m *should be encoded into a payload of length* $|p| = \mathsf{ME.pl}(|m|, \nu)$*. The decoding algorithm* 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. 9.*

This primitive allows us to reason more modularly about security properties of the channel using an encoding integrity notion defined in Fig. 10. The advantage of $\mathcal{F}$ in breaking the EINT-security of ME with respect to supp is defined as $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME,supp}}(\mathcal{F}) = \Pr[\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME,supp},\mathcal{F}}]$. Both ME and supp are concerned with whether or not a given payload (i.e. message encoding) should be accepted, and satisfying this notion ensures that the behaviour of ME matches the constraints specified by supp. Since the notion only concerns honestly generated messages, the support function can use plaintext payloads as labels instead of ciphertexts.

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

$\mathrm{SEND}(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, p, aux)$ ; Return $p$

$\mathrm{RECV}(u, p, aux)$
If $\nexists m', aux' \colon (\mathsf{sent}, m', p, aux') \in \mathsf{tr}_{\overline{u}}$ then return $\bot$
$(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, p, aux)$ ; If $m \neq m^*$ then win $\leftarrow$ true
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, p, aux)$ ; Return $m$
```

Figure 10: 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 [28] and as implemented in the official desktop[6] and Android clients.[7] We focus on *cloud chats*, i.e. chats that are only encrypted at the transport layer between the clients and Telegram servers. The end-to-end encrypted *secret chats* are implemented on top of this transport layer and only available for one-on-one chats. Figures 11 and 12 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 Appendix F. 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 [29].

2) The API requests and responses are serialised according to the type language (TL) [30] 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-IGE. The AES-256-IGE ciphertext c is a part of an MTProto ciphertext $\mathsf{auth\_key\_id} \,\|\, \mathsf{msg\_key} \,\|\, c$, where (recalling that $z[a:b]$ denotes bits a to $b-1$, inclusive, of string z):

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

Here, the first two fields form an *external header*. The AES-256-IGE 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 [31] of IGE, which has 2-block IVs.

4) MTProto ciphertexts are encapsulated in a "transport protocol". The MTProto documentation defines multiple such protocols [32], but the default appears to be 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

[6]https://github.com/telegramdesktop/tdesktop/, versions 2.3.2 to 2.7.1
[7]https://github.com/DrKLO/Telegram/, versions 6.1.1 to 7.6.0

Table I: MTProto payload format.

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.

- 0x7f || length || c_{MTP} where length is encoded in 3 bytes.

5) All the resulting packets are obfuscated by default using AES-128-CTR encryption. The key and IV are transmitted at the beginning of the stream, so the obfuscation provides no cryptographic protection and we ignore it henceforth.[8]

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, in which the MAC tag msg_key is computed using SHA-256 with a prepended key derived from (certain bits of) auth_key, and in which the key and IV for IGE mode are derived using a KDF based on SHA-256 on a per-message basis using msg_key as a diversifier (also using certain bits of auth_key as the key-deriving key). Note also that the bits from auth_key used by client and server to derive keys in both the "Encrypt" and "MAC" operations overlap with one another. Any formal security analysis needs to take this into account.

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: In what follows, we consider a network attacker that sits between the client and the Telegram servers, attempting to manipulate the conversation transcript. We distinguish between two cases: when the client is the sender of a message and when it is the receiver. 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 messaging protocol. The MTProto documentation mentions reordering attacks as something to protect against in secret chats but does not discuss it for cloud chats [33]. 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. On mobile clients messages are also delivered via push notification systems which are typically secured with TLS. Note that service messages of MTProto typically do not have such a timestamp so reordering is theoretically possible, but it is unclear whether it would affect the client's state since such messages tend to be responses to particular requests or notices of errors, which are not expected to arrive in a given order.
- 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 [34] 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 modified order that will match that of Bob.

A stronger form of reordering resistance can also be required from a protocol if one considers the order in the transcript as a whole, so that the order of sent messages with respect to received messages has to be preserved. We discuss this further and compare Telegram's behaviour to other messenger systems and protocols in Appendix C.

[8]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.

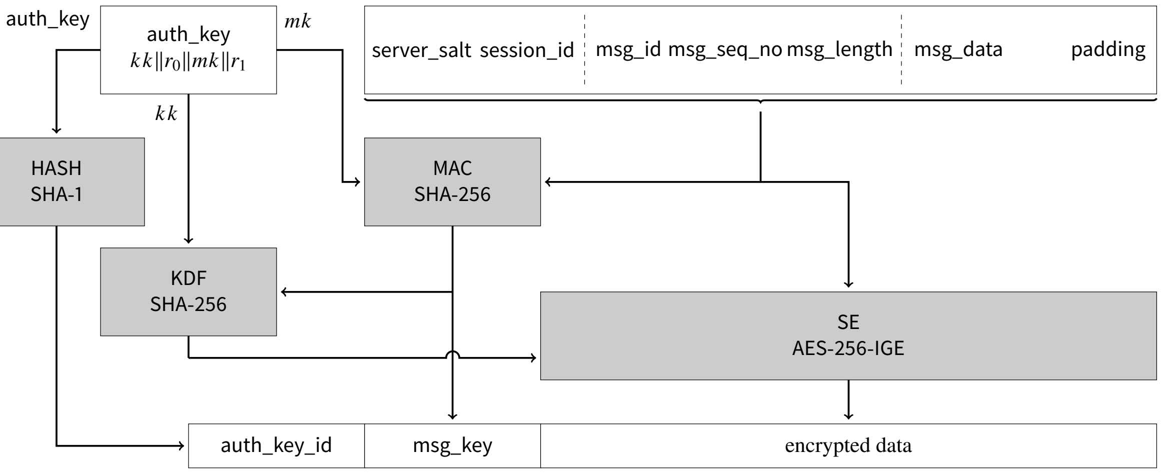


Figure 11: Overview of message processing in MTProto 2.0.

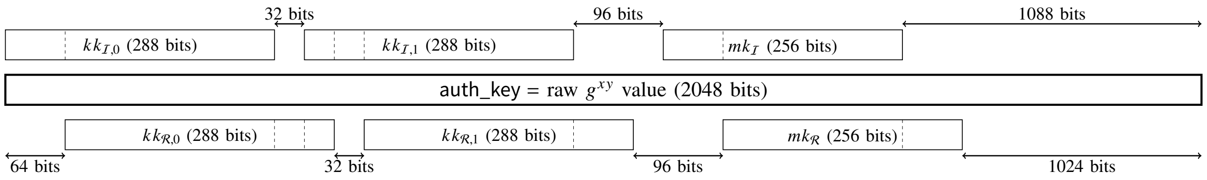

32 bits
96 bits
1088 bits
$kk_{\mathcal{I},0}$ (288 bits)
$kk_{\mathcal{I},1}$ (288 bits)
$mk_{\mathcal{I}}$ (256 bits)
auth_key = raw g^{xy} value (2048 bits)
$kk_{\mathcal{R},0}$ (288 bits)
$kk_{\mathcal{R},1}$ (288 bits)
$mk_{\mathcal{R}}$ (256 bits)
64 bits
32 bits
96 bits
1024 bits

Figure 12: 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) Deletion: MTProto makes it possible to silently drop a message both when the client is the sender[9] and when it is the receiver, but it is difficult to exploit in practice. Clients and the server attempt to resend messages they did not get acknowledgement for. Such messages have the same 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 [35]. 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 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, we will use "✓" below to denote this, 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 $c_{\mathcal{R}}$.[10]

```
Adversary $\mathcal{D}_{\mathrm{IND},q}^{\mathrm{CH,RECV}}$
Let $aux = \varepsilon$. Choose any $m_0, m_1 \in \mathsf{CH.MS} \setminus \{✓\}$.
Require $\forall i \in \mathbb{N}$: $r_{\mathcal{I},i}, r_{\mathcal{R},i} \in \mathsf{CH.SendRS}$.
For $i = 1, \ldots, q$ do
    $c_{\mathcal{I},i} \leftarrow \mathrm{CH}(\mathcal{I}, m_0, m_0, aux, r_{\mathcal{I},i})$
    $c_{\mathcal{R},i} \leftarrow \mathrm{CH}(\mathcal{R}, ✓, m_1, aux, r_{\mathcal{R},i})$ ; $\mathrm{RECV}(\mathcal{I}, c_{\mathcal{R},i}, aux)$
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 13: Adversary against the IND-security of MTProto (modelled as channel CH) when permitting re-encryption under reused msg_id and 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 msg_key for $c_{\mathcal{I},i}$ and let $c^{(i)}$ be the i-th block of ciphertext c.

[9]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).

[10]Note that here we are breaking the confidentiality of the ciphertext carrying "✓". 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.

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.[11] To sketch a model for acknowledgement messages for the purposes of explaining this attack and as mentioned above, we define a special plaintext symbol ✓ that, when received, indicates acknowledgement for the last sent message. As in Telegram, ✓ messages are encrypted. Further, we model re-encryptions by insisting that if the CH.Send algorithm is queried again on an unacknowledged message m then CH.Send will produce another ciphertext c' for m but using the same headers, including msg_id and msg_seq_no, as previously used. Critically, this means the same state in the form of msg_id and msg_seq_no is used for two different encryptions.

We use this behaviour to break the indistinguishability of an encrypted ✓. Consider the adversary given in Fig. 13. In that figure, if $c_{\mathcal{R},i}$ encrypts an ✓ (i.e. case $b = 0$) then $c_{\mathcal{I},i+1}$ will not be a re-encryption of m_0 under the same msg_id and msg_seq_no that were used for $c_{\mathcal{I},i}$. In contrast, if $b = 1$, then we have $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$, where $c^{(i)}$ denotes the i-th block of c, with probability 1 whenever $\mathsf{msg\_key}_k = \mathsf{msg\_key}_j$. 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 msg_id and msg_seq_no in the second block, encrypted under the same key. In the setting where the adversary controls the randomness of the encryption, 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 (of the padding) then we use the fact that msg_key is computed via SHA-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. We note that the adversary can check when a collision is found. On the other hand, 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.

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.[12] 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" [36]. In practice the clients receive salts that change every hour and which overlap with each other. For client differences, consider padding generation: on desktop [35], a given message length will always result in the same padding length, whereas on Android [37], 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 server and the client 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.[13] 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 [39] 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

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

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

[13]The Android client accepts any value in the place of server_salt, and the desktop client [38] 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.

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-derivation 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 outputs 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 current implementation of MTProto avoids in some cases only due to application level processing.[14]

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. 14. The users $\mathcal{I}$ and $\mathcal{R}$ represent the client and the server. We abstract the individual keyed primitives into function families.[15]

CH.Init generates the keys for both users and initialises the message encoding scheme. Note that auth_key as described in Section IV-A does not appear in the code in Fig. 14, 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.[16] 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. 14, 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. 19. It is a simplified MTProto message encoding scheme for strict in-order delivery without replays (see Appendix D 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 its internal counters wrap around when seq_no would "overflow".

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. 19, with* $\mathsf{ME.MS} = \bigcup_{i=1}^{2^{24}} \{0,1\}^{8 \cdot i}$, $\mathsf{ME.Out} = \bigcup_{i \in \mathbb{N}} \{0,1\}^{\mathsf{bl} \cdot i}$, $\mathsf{ME.T} = 2^{96} - 1$ *and* $\mathsf{ME.pl}(\ell, \nu) = 256 + \ell + |\mathsf{GenPadding}(\ell; \nu)|$.[17]

The following SHA-1 and SHA-256 based function families capture the MTProto primitives that are used to derive

[14]Secret chats implement more elaborate measures against replay/reordering [33], however this complexity is not required when in-order delivery is required for each direction separately.

[15]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.

[16]The comments in Fig. 15 show how the exact 2048-bit value of auth_key can be reconstructed by combining bits of hk, kk, mk. Note that the key hk used for HASH is deliberately chosen to contain all bits of auth_key that are *not* used for KDF and MAC keys kk, mk.

[17]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. 19 and instead handle this implicitly.

```
$\underline{\mathsf{CH.Init}()}$
$hk \leftarrow_\$ \{0,1\}^{\mathsf{HASH.kl}}$
$kk \leftarrow_\$ \{0,1\}^{672}$ ; $mk \leftarrow_\$ \{0,1\}^{320}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, kk \,\|\, mk)$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$\mathsf{key}_{\mathcal{I}} \leftarrow (kk_{\mathcal{I}}, mk_{\mathcal{I}})$
$\mathsf{key}_{\mathcal{R}} \leftarrow (kk_{\mathcal{R}}, mk_{\mathcal{R}})$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow_\$ \mathsf{ME.Init}()$
$st_{\mathcal{I}} \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_{\mathcal{I}}, \mathsf{key}_{\mathcal{R}}, st_{\mathsf{ME},\mathcal{I}})$
$st_{\mathcal{R}} \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_{\mathcal{R}}, \mathsf{key}_{\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
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 14: The 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 derivation functions ϕ_{MAC} and ϕ_{KDF}, and from deterministic symmetric encryption scheme SE.

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* MTP-HASH.Ev *given in Fig. 15.*

```
$\underline{\mathsf{MTP\text{-}HASH.Ev}(hk, x)}$   // $|hk| = 1056$, $|x| = 992$
$kk \leftarrow x[0:672]$                  // auth_key[0 : 672]
$r_0 \leftarrow hk[0:32]$                 // auth_key[672 : 704]
$mk \leftarrow x[672:992]$                // auth_key[704 : 1024]
$r_1 \leftarrow hk[32:1056]$              // auth_key[1024 : 2048]
$\mathsf{auth\_key} \leftarrow kk \,\|\, r_0 \,\|\, mk \,\|\, r_1$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{SHA\text{-}1}(\mathsf{auth\_key})[96:160]$
Return auth_key_id
```

Figure 15: Construction of MTP-HASH.

Definition 8. MTP-MAC *is the function family with* $\mathsf{MAC.Keys} = \{0,1\}^{256}$, $\mathsf{MAC.In} = \{0,1\}^*$, $\mathsf{MAC.ol} = 128$ *and* MTP-MAC.Ev *given in Fig. 16.*

```
$\underline{\mathsf{MTP\text{-}MAC.Ev}(mk_u, p)}$   // $|mk_u| = 256$, $p \in \{0,1\}^*$
$\mathsf{msg\_key} \leftarrow \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$
Return msg_key
```

Figure 16: Construction of MTP-MAC.

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. 17.*

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

Figure 17: Construction of MTP-KDF.

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 [40], but define our related-key derivation function ϕ_{KDF} (resp. ϕ_{MAC}) to output both keys at once, as a function of kk (resp. mk). See Fig. 18 for precise details of ϕ_{KDF} and ϕ_{MAC}.

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

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

Figure 18: Related-key derivation functions $\phi_{\mathsf{KDF}}\colon \{0,1\}^{672} \rightarrow \mathsf{KDF.Keys} \times \mathsf{KDF.Keys}$ and $\phi_{\mathsf{MAC}}\colon \{0,1\}^{320} \rightarrow \mathsf{MAC.Keys} \times \mathsf{MAC.Keys}$.

Finally, we define the deterministic symmetric encryption scheme.

Definition 10. *Let* AES-256 *be the standard AES block cipher with* $\mathsf{AES\text{-}256.kl} = 256$ *and* $\mathsf{AES\text{-}256.ol} = 128$*, 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 prove that MTP-CH satisfies correctness, indistinguishability and integrity. Our proofs use games and hops between them. In our games, we annotate some lines with comments of the form "G_i–G_j" to indicate that these lines belong only to games G_i through G_j (inclusive). The lines not annotated with such comments are shared by all of the games that are shown in the particular figure.

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. 20.

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

GenPadding($\ell$)
$\ell' \leftarrow \text{bl} - \ell \bmod \text{bl}$
$bn \leftarrow_\$ \{1, \cdots, \text{pb}\}$
$\text{padding} \leftarrow_\$ \{0,1\}^{\ell' + bn * \text{bl}}$
Return padding

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

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

Figure 19: The construction of a simplified message encoding scheme for strict in-order delivery $\text{ME} = \text{MTP-ME}[\text{session\_id}, \text{pb}, \text{bl}]$ for session identifier session_id, maximum padding length (in full blocks) pb, and output block length bl.

The security game $G^{\text{otwind}}_{\text{HASH},\mathcal{D}}$ in Fig. 20 evaluates 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 the function family; 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 $\text{Adv}^{\text{otwind}}_{\text{HASH}}(\mathcal{D}) = 2 \cdot \Pr\left[G^{\text{otwind}}_{\text{HASH},\mathcal{D}} \right] - 1$. Appendix E1 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 $G^{\text{otwind}}_{\text{HASH},\mathcal{D}}$
$b \leftarrow_\$ \{0,1\}$ ; $hk \leftarrow_\$ \{0,1\}^{\text{HASH.kl}}$ ; $x_0 \leftarrow_\$ \text{HASH.In}$
$x_1 \leftarrow_\$ \text{HASH.In}$ ; $\text{auth\_key\_id} \leftarrow \text{HASH.Ev}(hk, x_b)$
$b' \leftarrow_\$ \mathcal{D}(x_0, x_1, \text{auth\_key\_id})$ ; Return $b' = b$
```

Figure 20: 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. 21. The security game $G^{\text{rkprf}}_{\text{KDF},\phi_{\text{KDF}},\mathcal{D}}$ in Fig. 21 defines a variant of the standard PRF notion, except it allows adversary $\mathcal{D}$ to use its RoR oracle to evaluate function family KDF on either of the two secret, related function keys $kk_\mathcal{I}, kk_\mathcal{R}$ (both computed using key-derivation function ϕ_{KDF}). The advantage of $\mathcal{D}$ in breaking the RKPRF-security of KDF with respect to ϕ_{KDF} is defined as $\text{Adv}^{\text{rkprf}}_{\text{KDF},\phi_{\text{KDF}}}(\mathcal{D}) = 2 \cdot \Pr\left[G^{\text{rkprf}}_{\text{KDF},\phi_{\text{KDF}},\mathcal{D}} \right] - 1$.

Appendix E2 provides a formal reduction from the RKPRF-security of MTP-KDF to a novel security notion for SHACAL-2 that roughly requires it to be a leakage-resilient PRF under related-key attacks. 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 key-derivation functions ϕ_{KDF} and $\phi_{\text{SHACAL-2}}$ (as defined in Appendix E2).

We stress that we have to assume a property of SHACAL-2 that has not been studied in the literature. Related-key attacks on reduced-round SHACAL-2 have been considered [41], [42], but they ordinarily work with a *known difference* relation between unknown keys. In MTProto, the keys produced by ϕ_{KDF} and $\phi_{\text{SHACAL-2}}$ differ by random, unknown parts. However, 224 out of 512 bits of each key produced by $\phi_{\text{SHACAL-2}}$ are known to the adversary, out of which 128 bits, corresponding to msg_key, can be directly influenced by the adversary. It is straightforward to show that the LRKPRF-security of SHACAL-2 holds 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.

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

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

Figure 21: Related-key PRF-security of function family KDF with respect to key-derivation function ϕ_{KDF}.

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. 22. The security game $G^{\text{rkcr}}_{\text{MAC},\phi_{\text{MAC}},\mathcal{F}}$ in Fig. 22 gives adversary $\mathcal{F}$ two related function keys $mk_\mathcal{I}, mk_\mathcal{R}$ (created by key-derivation 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 $\text{msg\_key}_0, \text{msg\_key}_1$ of function family MAC. The advantage of $\mathcal{F}$ in breaking the RKCR-security of MAC with respect to ϕ_{MAC} is defined as $\text{Adv}^{\text{rkcr}}_{\text{MAC},\phi_{\text{MAC}}}(\mathcal{F}) = \Pr\left[G^{\text{rkcr}}_{\text{MAC},\phi_{\text{MAC}},\mathcal{F}} \right]$. It is clear by inspection that the RKCR-security of $\text{MTP-MAC.Ev}(mk_u, p) = \text{SHA-256}(mk_u \,\|\, p)[64:192]$ (with respect to ϕ_{MAC} from Fig. 18) reduces to the collision resistance of truncated SHA-256.

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

Figure 22: Related-key collision resistance of function family MAC with respect to key-derivation function ϕ_{MAC}.

4) MTP-MAC **is a PRF under RKA for inputs with unique prefixes:** 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. 23. The security game $\mathsf{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 23 extends the standard PRF notion to use two related ϕ_{MAC}-derived function keys $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ for function family MAC (similar to the RKPRF-security notion we defined above); but it also enforces that 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 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 on 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 $\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}\right] - 1$.

Appendix E3 shows that the UPRKPRF-security of MTP-MAC reduces to a novel assumption on SHACAL-2 as a leakage-resilient PRF under related-key attacks (defined as HRKPRF in Fig. 50), and to the one-time PRF-security of the SHA-256 compression function h_{256}. Analogously to RKPRF-security of MTP-KDF we emphasise that, while this assumption on SHACAL-2 holds in the ideal cipher model, it is unstudied in the literature.

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

$\mathrm{RoR}(u, p)$
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 23: Related-key PRF-security of function family MAC for inputs with unique 256-bit prefixes, with respect to key derivation function ϕ_{MAC}.

5) MTP-SE **is a one-time indistinguishable symmetric encryption scheme:** For any block cipher E, Appendix E4 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 on CBC in our analysis of MTProto.

B. Security requirements on message encoding

1) Prefix uniqueness of MTP-ME**:** We require that payloads produced by MTP-ME have distinct prefixes of size 256 bits, as defined in Fig. 24. The advantage of $\mathcal{F}$ in breaking the UPREF-security of ME is defined as $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{F}) = \Pr\left[\mathsf{G}^{\mathsf{upref}}_{\mathsf{ME},\mathcal{F}}\right]$. Given the fixed prefix size, this notion cannot be satisfied against unbounded adversaries. Our MTP-ME scheme ensures unique prefixes using seq_no which is of size 96 bits, so we have $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{MTP\text{-}ME}}(\mathcal{F}) = 0$ only for $\mathcal{F}$ making less than 2^{96} queries, and otherwise $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{MTP\text{-}ME}}(\mathcal{F}) = 1$. Note that MTP-ME always has payloads larger than 256 bits. The current MTProto implementation of message encoding is not UPREF-secure as it allows repeated msg_id (cf. Section IV-C).

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

$\mathrm{SEND}(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
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.

2) MTP-ME **ensures in-order delivery:** We require that MTP-ME is EINT-secure (Fig. 10) with respect to the support function SUPP defined in Fig. 25. SUPP enforces in-order delivery for each user's sent messages, thus preventing unidirectional reordering attacks, replays and message deletion. It is formalised using a function find(op, tr, label) which searches a given transcript's sent or received entries for the message corresponding to label and also counts the number of valid entries up to a successful find. Correctness is ensured by the search of entries sent by the other user $\overline{u}$ so that valid messages are returned, which holds as long as label serves as a unique label. This is the case for SUPP and MTP-ME for less than 2^{96} queries.[18] 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.[19] For reasons outlined in Section IV-B, the current MTProto construction of ME (cf. Appendix D) is not EINT-secure with respect to SUPP. Appendix E5 shows that $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{MTP\text{-}ME},\mathsf{SUPP}}(\mathcal{F}) = 0$ for $\mathcal{F}$ making less than 2^{96} queries to SEND.

3) Encoding robustness of MTP-ME**:** We require that decoding in MTP-ME should not affect the state in such a way that would be visible in future encoded outputs, as defined in Fig. 26. The advantage of $\mathcal{D}$ in breaking the ENCROB-security of ME is defined as $\mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}}\right] - 1$. This advantage is trivially zero both for MTP-ME and the

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

[19]Note that aux is not used in SUPP or MTP-ME. It would be possible to add time synchronisation using aux captured in a 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 25: Support function SUPP for strict in-order delivery.

original MTProto message encoding scheme (cf. Appendix D). 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 26: Encoding robustness of message encoding scheme ME.

4) Combined security of MTP-SE and MTP-ME: We require that decryption in MTP-SE with random keys has unpredictable outputs with respect to MTP-ME, as defined in Fig. 27. The advantage of $\mathcal{F}$ in breaking the UNPRED-security of SE with respect to ME is defined as $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}\right]$. $\mathcal{F}$ is given access to two oracles. For a given user u and msg_key, $\textsc{Ch}$ decrypts a given ciphertext c_{se} under a random key and then decodes it using the given message encoding state st_{ME}, returning no output. $\textsc{Expose}$ lets $\mathcal{F}$ learn the key corresponding to the given u and msg_key, which disallows the adversary from querying $\textsc{Ch}$ with this u and msg_key. $\mathcal{F}$ 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. Appendix E6 shows that $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{MTP\text{-}SE},\mathsf{MTP\text{-}ME}}(\mathcal{F}) \leq q_{\textsc{Ch}}/2^{64}$ for $\mathcal{F}$ making $q_{\textsc{Ch}}$ queries.

C. Correctness of MTP-CH

Consider the correctness game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ (Fig. 8) for channel CH = MTP-CH (Fig. 14) and support function supp = SUPP (Fig. 25). We only consider $\textsc{Recv}$ queries for c produced by an honest $\textsc{Send}$ query, since supp always outputs $\perp$ otherwise (Fig. 36). Informally, we claim that $\mathcal{F}$ cannot win because the primitives of MTP-CH satisfy perfect correctness and because MTP-ME "matches" SUPP for less than 2^{96} queries (cf. Appendix E5).[20] The latter is easy to see when comparing

[20]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.

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

$\textsc{Expose}(u, \mathsf{msg\_key})$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow \mathsf{true}$ ; Return $\mathsf{T}[u, \mathsf{msg\_key}]$

$\textsc{Ch}(u, \mathsf{msg\_key}, c_{se}, st_{\mathsf{ME}}, aux)$
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 27: Unpredictability of deterministic symmetric encryption scheme SE with respect to message encoding scheme ME.

the correctness game with the EINT-security game (Fig. 10). Given an adversary $\mathcal{F}$ for the former, we can build an adversary $\mathcal{F}_{\mathrm{EINT}}$ for the latter: generate the initial states for $\mathcal{F}$ as MTP-CH does and simulate its oracles until the ME.Encode and ME.Decode calls, which are replaced with the oracles of $\mathcal{F}_{\mathrm{EINT}}$.

D. IND-security of MTP-CH

We begin our IND-security reduction by considering an arbitrary adversary $\mathcal{D}_{\mathrm{IND}}$ playing in the IND-security game against channel CH = MTP-CH (i.e. $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\mathrm{IND}}}$ in Fig. 8), and we gradually change this game until we can show that $\mathcal{D}_{\mathrm{IND}}$ 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 receiver's channel state by calling CH.Recv. We assume that calls to CH.Recv never affect the ciphertexts that are returned by future calls to 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 CH.Recv. (2) We use the UPRKPRF-security of MAC to show that the ciphertexts returned by oracle $\textsc{Ch}$ contain 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 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 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* MTP-CH. *Let* CH = MTP-CH[ME, HASH, MAC, KDF, ϕ_{MAC}, ϕ_{KDF}, SE]. *Let* $\mathcal{D}_{\mathrm{IND}}$ *be any adversary against the* IND*-security of* CH*, making* q_{CH} *queries to its* CH *oracle. Then there exist adversaries* $\mathcal{D}_{\mathrm{OTWIND}}$, $\mathcal{D}_{\mathrm{RKPRF}}$, $\mathcal{D}_{\mathrm{ENCROB}}$, $\mathcal{F}_{\mathrm{UPREF}}$, $\mathcal{D}_{\mathrm{UPRKPRF}}$, $\mathcal{D}_{\mathrm{OTIND\$}}$ *such that*

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

Proof. This proof uses games G_0–G_8 in Fig. 28. The adversaries for transitions between games are provided in Fig. 29.

G_0: Game G_0 is equivalent to game $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\mathrm{IND}}}$. It expands the code of algorithms CH.Init, CH.Send and CH.Send; the expanded instructions are highlighted in gray. It follows that

$$\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}_{\mathrm{IND}}) = 2 \cdot \Pr[\mathrm{G}_0] - 1.$$

$\mathrm{G}_0 \to \mathrm{G}_1$: Note that adversary $\mathcal{D}_{\mathrm{IND}}$ can learn the value of auth_key_id from any ciphertext returned by oracle CH which depends on the KDF and MAC keys. To invoke PRF-style security notions for either primitive in later steps, we appeal to the OTWIND-security of HASH (Fig. 20), which essentially guarantees that auth_key_id leaks no information about KDF and MAC keys. Game G_1 is the same as game G_0, except auth_key_id ← HASH.Ev(hk, ·) is evaluated on a uniformly random string x rather than on $kk \,\|\, mk$. We claim that $\mathcal{D}_{\mathrm{IND}}$ cannot distinguish between these two games. More formally, given $\mathcal{D}_{\mathrm{IND}}$, in Fig. 29a we define an adversary $\mathcal{D}_{\mathrm{OTWIND}}$ attacking the OTWIND-security of HASH as follows. According to the definition of game $\mathrm{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}_{\mathrm{OTWIND}}}$, adversary $\mathcal{D}_{\mathrm{OTWIND}}$ takes $(x_0, x_1, \mathsf{auth\_key\_id})$ as input. We define adversary $\mathcal{D}_{\mathrm{OTWIND}}$ to sample a challenge bit b, to parse $kk \,\|\, mk \leftarrow x_1$, and to subsequently use the obtained values of $b, kk, mk, \mathsf{auth\_key\_id}$ in order to simulate either of the games G_0, G_1 for adversary $\mathcal{D}_{\mathrm{IND}}$ (both games are equivalent from the moment these 4 values are chosen). If $\mathcal{D}_{\mathrm{IND}}$ guesses the challenge bit b then we let adversary $\mathcal{D}_{\mathrm{OTWIND}}$ return 1; otherwise we let it return 0. Now let d be the challenge bit in game $\mathrm{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}_{\mathrm{OTWIND}}}$, and let d' be the value returned by $\mathcal{D}_{\mathrm{OTWIND}}$. If $d = 1$ then $\mathcal{D}_{\mathrm{OTWIND}}$ simulates game G_0 for $\mathcal{D}_{\mathrm{IND}}$, and otherwise it simulates game G_1. It follows that $\Pr[\mathrm{G}_0] = \Pr[\, d' = 1 \mid d = 1 \,]$ and $\Pr[\mathrm{G}_1] = \Pr[\, d' = 1 \mid d = 0 \,]$, and hence

$$\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] = \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\mathrm{OTWIND}}).$$

$\mathrm{G}_1 \to \mathrm{G}_2$: In the transition between games G_1 and G_2, we use the RKPRF-security of KDF (with respect to ϕ_{KDF}, Fig. 21) in order to replace KDF.Ev(kk_u, msg_key) with a uniformly random value from $\{0,1\}^{\mathsf{KDF.ol}}$ (and for consistency store the latter in T[u, msg_key]). Similarly to the above, in Fig. 29b we build an adversary $\mathcal{D}_{\mathrm{RKPRF}}$ attacking the RKPRF-security of KDF that simulates G_1 or G_2 for adversary $\mathcal{D}_{\mathrm{IND}}$, depending on the challenge bit in game $\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\mathrm{RKPRF}}}$. We have

$$\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] = \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}).$$

$\mathrm{G}_2 \to \mathrm{G}_3$: We invoke the ENCROB property of ME (Fig. 26) to transition from G_2 to G_3. This property states that calls to ME.Decode do not change ME's state in a way that affects the payloads returned by any future calls to ME.Encode, allowing us to remove the ME.Decode call from inside the oracle RECV in game G_3. In Fig. 29c we build an adversary $\mathcal{D}_{\mathrm{ENCROB}}$ against ENCROB of ME that simulates either G_2 or G_3 for $\mathcal{D}_{\mathrm{IND}}$, depending on the challenge bit in game $\mathrm{G}^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}_{\mathrm{ENCROB}}}$, such that

$$\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] = \mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}_{\mathrm{ENCROB}}).$$

$\mathrm{G}_3 \to \mathrm{G}_4$: Game G_4 differs from game G_3 in the following ways. (1) The KDF keys kk, $kk_{\mathcal{I}}$, $kk_{\mathcal{R}}$ are no longer used in our reduction games starting from G_3, so they are not included in game G_4 and onwards. (2) The calls to oracle RECV in game G_3 no longer change the receiver's channel state, so game G_4 immediately returns $\perp$ on every call to RECV. (3) Game G_4 rewrites, in a functionally equivalent way, the initialisation and usage of values from the PRF-table T inside oracle CH. (4) Game G_4 adds a set X_u, for each $u \in \{\mathcal{I}, \mathcal{R}\}$, that stores fixed-size prefixes of payloads that were produced by calling the specific user's CH oracle. Every time a new payload p is generated, the new code inside oracle CH checks whether X_u contains a prefix ω of a previously generated payload such that it is the same as $p[0:256]$, the prefix of p. Then the new prefix is added to X_u. We note that the output of oracle CH in game G_4 does not change depending on whether this condition passes or fails. (5) Game G_4 adds Boolean flags bad_0 and bad_1 that are set to true when the corresponding conditions inside oracle CH are satisfied. These flags do not affect the functionality of the games, and will only be used for the formal analysis that we provide below. Both games are functionally equivalent, so

$$\Pr[\mathrm{G}_4] = \Pr[\mathrm{G}_3].$$

$\mathrm{G}_4 \to \mathrm{G}_5$: The transition from game G_4 to G_5 replaces the value assigned to msg_key when the newly added unique-prefixes condition (Fig. 24) is satisfied; the value of msg_key changes from MAC.Ev(mk_u, p) to a uniformly random string from $\{0,1\}^{\mathsf{MAC.ol}}$. Games G_4 and G_5 are identical until bad_0 is set. According to the Fundamental Lemma of Game Playing [14] we have

$$\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] \le \Pr[\mathsf{bad}_0^{\mathrm{G}_4}],$$

where $\Pr[\mathsf{bad}^{\mathrm{Q}}]$ denotes the probability of setting flag bad in game Q. The UPREF property of ME states that it is hard to find two payloads returned by ME.Encode such that their 256-bit prefixes are the same; we use this property to upper-bound the probability of setting bad_0 in game G_4. In Fig. 29d we build an adversary $\mathcal{F}_{\mathrm{UPREF}}$ attacking the UPREF of ME that simulates game G_4 for adversary $\mathcal{D}_{\mathrm{IND}}$. Every time bad_0 is

```
Games $G_0$–$G_3$
$b \leftarrow_\$ \{0,1\}$ ; $hk \leftarrow_\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $kk \leftarrow_\$ \{0,1\}^{672}$ ; $mk \leftarrow_\$ \{0,1\}^{320}$
$x \leftarrow kk \,\|\, mk$                                  // $G_0$
$x \leftarrow_\$ \{0,1\}^{992}$                               // $G_1$–$G_3$ (OTWIND of HASH)
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$ ; $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow_\$ \mathsf{ME.Init}()$
$b' \leftarrow_\$ \mathcal{D}_{\mathrm{IND}}^{\mathrm{CH},\mathrm{RECV}}$ ; Return $b' = b$

$\mathrm{CH}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathrm{T}[u, \mathsf{msg\_key}] = \perp$ then $\mathrm{T}[u, \mathsf{msg\_key}] \leftarrow_\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$           // $G_0$–$G_1$
$k \leftarrow \mathrm{T}[u, \mathsf{msg\_key}]$                    // $G_2$–$G_3$ (RKPRF of KDF)
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$ ; $c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; Return $c$

$\mathrm{RECV}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathrm{T}[\overline{u}, \mathsf{msg\_key}] = \perp$ then $\mathrm{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow_\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$           // $G_0$–$G_1$
$k \leftarrow \mathrm{T}[\overline{u}, \mathsf{msg\_key}]$                    // $G_2$–$G_3$ (RKPRF of KDF)
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ ; $\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
If $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$ then
    $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$    // $G_0$–$G_2$ (ENCROB of ME)
Return $\perp$
```

```
Games $G_4$–$G_8$
$b \leftarrow_\$ \{0,1\}$ ; $hk \leftarrow_\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow_\$ \{0,1\}^{320}$
$x \leftarrow_\$ \{0,1\}^{992}$ ; $\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow_\$ \mathsf{ME.Init}()$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$ ; $b' \leftarrow_\$ \mathcal{D}_{\mathrm{IND}}^{\mathrm{CH},\mathrm{RECV}}$ ; Return $b' = b$

$\mathrm{CH}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
If $\exists \omega \in X_u : \omega = p[0:256]$ then
    $\mathsf{bad}_0 \leftarrow \mathsf{true}$
    $\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$   // $G_4$
    $\mathsf{msg\_key} \leftarrow_\$ \{0,1\}^{\mathsf{MAC.ol}}$       // $G_5$–$G_8$ (UPREF of ME)
Else
    $\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$   // $G_4$–$G_5$
    $\mathsf{msg\_key} \leftarrow_\$ \{0,1\}^{\mathsf{MAC.ol}}$    // $G_6$–$G_8$ (UPRKPRF of MAC)
$X_u \leftarrow X_u \cup \{p[0:256]\}$ ; $k \leftarrow_\$ \{0,1\}^{\mathsf{KDF.ol}}$
If $\mathrm{T}[u, \mathsf{msg\_key}] \neq \perp$ then
    $\mathsf{bad}_1 \leftarrow \mathsf{true}$
    $k \leftarrow \mathrm{T}[u, \mathsf{msg\_key}]$              // $G_4$–$G_6$ (Birthday bound)
$\mathrm{T}[u, \mathsf{msg\_key}] \leftarrow k$
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$                  // $G_4$–$G_7$
$c_{se} \leftarrow_\$ \{0,1\}^{\mathsf{SE.cl}(\mathsf{ME.pl}(|m_b|, r))}$     // $G_8$ (OTIND\$ of SE)
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; Return $c$

$\mathrm{RECV}(u, c, aux)$: Return $\perp$
```

Figure 28: Games G_0–G_8 for proof of Theorem 1. Left pane: The code added by expanding the algorithms of CH in game $G^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\mathrm{IND}}}$ is highlighted in gray. Right pane: The code highlighted in gray was rewritten in a way that is functionally equivalent to the corresponding code in G_3. Both panes: The code added for the transitions between games is highlighted in green.

set in game G_4, this corresponds to adversary $\mathcal{F}_{\mathrm{UPREF}}$ setting flag win to true in its own game $G^{\mathsf{upref}}_{\mathsf{ME},\mathcal{F}_{\mathrm{UPREF}}}$. It follows that

$$\Pr[\mathsf{bad}_0^{G_4}] \leq \mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{F}_{\mathrm{UPREF}}).$$

$G_5 \rightarrow G_6$: We use the UPRKPRF-security of MAC (with respect to ϕ_{MAC}, Fig. 23) in order to replace the value of msg_key from $\mathsf{MAC.Ev}(mk_u, p)$ to a uniformly random value from $\{0,1\}^{\mathsf{MAC.ol}}$ in the transition from G_5 to G_6. Note that the notion of UPRKPRF-security only guarantees the indistinguishability from random when MAC is evaluated on inputs with unique prefixes, whereas games G_5, G_6 ensure that this prerequisite is satisfied by only evaluating MAC if $p[0:256] \notin X_u$ has payloads with unique prefixes. In Fig. 29e we build an adversary $\mathcal{D}_{\mathrm{UPRKPRF}}$ attacking the UPRKPRF-security of MAC that simulates G_5 or G_6 for adversary $\mathcal{D}_{\mathrm{IND}}$, depending on the challenge bit in game $G^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$. It follows that

$$\Pr[G_5] - \Pr[G_6] = \mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}).$$

$G_6 \rightarrow G_7$: Games G_6 and G_7 are identical until bad_1 is set; so, as above, we have

$$\Pr[G_6] - \Pr[G_7] \leq \Pr[\mathsf{bad}_1^{G_6}].$$

The values of $\mathsf{msg\_key} \in \{0,1\}^{\mathsf{MAC.ol}}$ in game G_6 are sampled uniformly at random and independently across the q_{CH} different calls to oracle SEND, so we can apply the birthday bound to claim the following:

$$\Pr[\mathsf{bad}_1^{G_6}] \leq \frac{q_{\mathrm{CH}} \cdot (q_{\mathrm{CH}} - 1)}{2 \cdot 2^{\mathsf{MAC.ol}}}.$$

$G_7 \rightarrow G_8$: In the transition from G_7 to G_8, we replace the value of ciphertext c_{se} from $\mathsf{SE.Enc}(k, p)$ to a uniformly random value from $\{0,1\}^{\mathsf{SE.cl}(\mathsf{ME.pl}(|m_b|, r))}$ by appealing to the OTIND\$-security of SE (Fig. 3). Recall that $\mathsf{ME.pl}(|m_b|, r)$ is the length of the payload p that is produced by calling ME.Encode on any message of length $|m_b|$ and on random coins r, whereas $\mathsf{SE.cl}(\cdot)$ maps the former to the resulting ciphertext length of SE. In Fig. 29f we build an adversary $\mathcal{D}_{\mathrm{OTIND\$}}$ attacking the OTIND\$-security of SE that simulates G_7 or G_8 for adversary $\mathcal{D}_{\mathrm{IND}}$, depending on the challenge bit in game $G^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}_{\mathrm{OTIND\$}}}$. It follows that

$$\Pr[G_7] - \Pr[G_8] = \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}_{\mathrm{OTIND\$}}).$$

Finally, the output of oracle CH in game G_8 no longer depends on the challenge bit b, so we have

$$\Pr[G_8] = \frac{1}{2}.$$

The theorem statement follows. □

E. INT-security of MTP-CH

Our integrity proof begins by showing 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

```
Adversary $\mathcal{D}_{\text{OTWIND}}(x_0, x_1, \mathsf{auth\_key\_id})$
$kk \,\|\, mk \leftarrow x_1$   // Such that $|kk| = 672$ and $|mk| = 320$.
$b \leftarrow\!\$ \{0,1\}$ ; $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$ \mathsf{ME.Init}()$
$b' \leftarrow\!\$ \mathcal{D}_{\text{IND}}^{\textsc{ChSim},\textsc{RecvSim}}$ ; If $b' = b$ then return 1 else return 0
$\textsc{ChSim}(u, m_0, m_1, aux, r)$ and $\textsc{RecvSim}(u, c, aux)$
// Identical to oracles $\textsc{Ch}$ and $\textsc{Recv}$ in games $\mathrm{G}_0$, $\mathrm{G}_1$ of Fig. 28.
```

(a) Adversary $\mathcal{D}_{\text{OTWIND}}$ against the OTWIND-security of HASH for transition between games G_0–G_1.

```
Adversary $\mathcal{D}_{\text{RKPRF}}^{\textsc{RoR}}$
$b \leftarrow\!\$ \{0,1\}$ ; $hk \leftarrow\!\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\!\$ \{0,1\}^{320}$ ; $x \leftarrow\!\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$ \mathsf{ME.Init}()$
$b' \leftarrow\!\$ \mathcal{D}_{\text{IND}}^{\textsc{ChSim},\textsc{RecvSim}}$ ; If $b' = b$ then return 1 else return 0
$\textsc{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$ ; $k \leftarrow \textsc{RoR}(u, \mathsf{msg\_key})$
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$ ; $c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; Return $c$
$\textsc{RecvSim}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$ ; $k \leftarrow \textsc{RoR}(\overline{u}, \mathsf{msg\_key})$
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ ; $\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
If $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$ then
    $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
Return $\perp$
```

(b) Adversary $\mathcal{D}_{\text{RKPRF}}$ against the RKPRF-security of KDF for transition between games G_1–G_2.

```
Adversary $\mathcal{D}_{\text{ENCROB}}^{\textsc{Send},\textsc{Recv}}$
$b \leftarrow\!\$ \{0,1\}$ ; $hk \leftarrow\!\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\!\$ \{0,1\}^{320}$ ; $x \leftarrow\!\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$b' \leftarrow\!\$ \mathcal{D}_{\text{IND}}^{\textsc{ChSim},\textsc{RecvSim}}$ ; If $b' = b$ then return 1 else return 0
$\textsc{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$p \leftarrow \textsc{Send}(u, m_b, aux, r)$ ; $\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; $c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; Return $c$
$\textsc{RecvSim}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
$\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
If $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$ then
    $\textsc{Recv}(u, p, aux)$
Return $\perp$
```

(c) Adversary $\mathcal{D}_{\text{ENCROB}}$ against the ENCROB-security of ME for transition between games G_2–G_3.

```
Adversary $\mathcal{F}_{\text{UPREF}}^{\textsc{Send}}$
$b \leftarrow\!\$ \{0,1\}$ ; $hk \leftarrow\!\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\!\$ \{0,1\}^{320}$
$x \leftarrow\!\$ \{0,1\}^{992}$ ; $\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $b' \leftarrow\!\$ \mathcal{D}_{\text{IND}}^{\textsc{ChSim},\textsc{RecvSim}}$
$\textsc{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$p \leftarrow \textsc{Send}(u, m_b, aux, r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$ ; $k \leftarrow\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
If $\mathsf{T}[u, \mathsf{msg\_key}] \neq \perp$ then $k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$
$\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow k$ ; $c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; Return $c$
$\textsc{RecvSim}(u, c, aux)$:  Return $\perp$
```

(d) Adversary $\mathcal{F}_{\text{UPREF}}$ against the UPREF-security of ME for transition between games G_4–G_5.

```
Adversary $\mathcal{D}_{\text{UPRKPRF}}^{\textsc{RoR}}$
$b \leftarrow\!\$ \{0,1\}$ ; $hk \leftarrow\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
$x \leftarrow\!\$ \{0,1\}^{992}$ ; $\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$ \mathsf{ME.Init}()$ ; $b' \leftarrow\!\$ \mathcal{D}_{\text{IND}}^{\textsc{ChSim},\textsc{RecvSim}}$
If $b' = b$ then return 1 else return 0
$\textsc{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$\mathsf{msg\_key} \leftarrow \textsc{RoR}(u, p)$
If $\mathsf{msg\_key} = \perp$ then $\mathsf{msg\_key} \leftarrow\!\$ \{0,1\}^{\mathsf{MAC.ol}}$
$k \leftarrow\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
If $\mathsf{T}[u, \mathsf{msg\_key}] \neq \perp$ then $k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$
$\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow k$ ; $c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; Return $c$
$\textsc{RecvSim}(u, c, aux)$:  Return $\perp$
```

(e) Adversary $\mathcal{D}_{\text{UPRKPRF}}$ against the UPRKPRF-security of MAC for transition between games G_5–G_6.

```
Adversary $\mathcal{D}_{\text{OTIND\$}}^{\textsc{RoR}}$
$b \leftarrow\!\$ \{0,1\}$ ; $hk \leftarrow\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
$x \leftarrow\!\$ \{0,1\}^{992}$ ; $\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$ \mathsf{ME.Init}()$ ; $b' \leftarrow\!\$ \mathcal{D}_{\text{IND}}^{\textsc{ChSim},\textsc{RecvSim}}$
If $b' = b$ then return 1 else return 0
$\textsc{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$\mathsf{msg\_key} \leftarrow\!\$ \{0,1\}^{\mathsf{MAC.ol}}$ ; $c_{se} \leftarrow \textsc{RoR}(p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; Return $c$
$\textsc{RecvSim}(u, c, aux)$:  Return $\perp$
```

(f) Adversary $\mathcal{D}_{\text{OTIND\$}}$ against the OTIND\$-security of SE for transition between games G_7–G_8.

Figure 29: Adversaries for proof of Theorem 1. The highlighted instructions mark the changes in the code of the simulated games.

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, mirror, 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* $\mathsf{session\_id} \in \{0,1\}^{64}$ *and* $\mathsf{pb}, \mathsf{bl} \in \mathbb{N}$. *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 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. 25. Let* $\mathcal{F}_{\mathrm{INT}}$ *be any adversary against the* INT-*security of* CH *with respect to* supp. *Then there exist adversaries* $\mathcal{D}_{\mathrm{OTWIND}}$, $\mathcal{D}_{\mathrm{RKPRF}}$, $\mathcal{F}_{\mathrm{UNPRED}}$, $\mathcal{F}_{\mathrm{RKCR}}$, $\mathcal{F}_{\mathrm{EINT}}$ *such that*

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

Before providing the detailed proof, we provide some discussion of our approach and a high-level overview of the different parts of the proof.

1) Invisible terms based on correctness of ME, SE, supp: We state and prove our INT-security claim for channel $\mathsf{MTP\text{-}CH}$ with respect to fixed choices of MTProto-based constructions $\mathsf{ME} = \mathsf{MTP\text{-}ME}$ (Definition 6) and $\mathsf{SE} = \mathsf{MTP\text{-}SE}$ (Definition 10), and with respect to the support function $\mathsf{supp} = \mathsf{SUPP}$ that is defined in Fig. 25. Our security reduction relies on six correctness-style properties of these primitives (one for ME, two for SE, three for supp). Each of them can be observed to be always true for the corresponding scheme, and hence does not contribute an additional term to the advantage statement in Theorem 2. These notions are also simple enough that we choose not to define them in a game-based style. Our security reduction nonetheless introduces and justifies a game hop for each of the correctness notions. This necessitates the use of 14 security games to prove Theorem 2, including some that are meant to be equivalent by observation (i.e. the corresponding game transitions do not rely on any correctness or security property). However, some of these reduction steps require a detailed analysis.

Theorem 2 could be stated in a more general way, fully formalising the aforementioned correctness notions and stating our claims with respect to any SE, ME, supp. We lose this generality by instantiating these primitives. Our motivation is twofold. On the one hand, we state our claims in a way that highlights the parts of MTProto (as captured by our model) that are critical for its security analysis, and omit spending too much attention on parts of the reduction that can be "taken for granted". On the other hand, our work studies MTProto and the abstractions that we use are meant to simplify and aid this analysis. We discourage the reader from treating $\mathsf{MTP\text{-}CH}$ in a prescriptive way, e.g. from trying to instantiate it with different primitives to build a secure channel since standard, well-studied cryptographic protocols such as TLS already exist.

2) Proof phase I. Forging a ciphertext is hard: Let $\mathcal{F}_{\mathrm{INT}}$ be an adversary playing in the INT-security game against channel $\mathsf{MTP\text{-}CH}$. Consider an arbitrary call made by $\mathcal{F}_{\mathrm{INT}}$ to its oracle $\textsc{Recv}$ on inputs u, c, aux such that $c = (\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se})$. The oracle evaluates $\mathsf{MTP\text{-}CH.Recv}(st_u, c, aux)$. Recall that $\mathsf{MTP\text{-}CH.Recv}$ attempts to validate $\mathsf{msg\_key}$ by checking whether $\mathsf{msg\_key} = \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ for an appropriately recovered payload p (i.e. $k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$ and $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$). If this $\mathsf{msg\_key}$ verification passes (and if $\mathsf{auth\_key\_id}' = \mathsf{auth\_key\_id}$), then $\mathsf{MTP\text{-}CH.Recv}$ attempts to decode the payload by computing $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$.

We consider two cases, and claim the following. (A) If $\mathsf{msg\_key}$ was not previously returned by oracle $\textsc{Send}$ as a part of any ciphertext sent by user $\overline{u}$, then with high probability an evaluation of $\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ would return $m = \perp$ *regardless of whether the* $\mathsf{msg\_key}$ *verification passed or failed*; so in this case we are not concerned with assessing the likelihood that the $\mathsf{msg\_key}$ verification passes. (B) If $\mathsf{msg\_key}$ was previously returned by oracle $\textsc{Send}$ as a part of some ciphertext $c' = (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c'_{se})$ sent by user $\overline{u}$, and if $\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}'$, then with high probability $c_{se} = c'_{se}$ (and hence $c = c'$) whenever the $\mathsf{msg\_key}$ verification passes. We now justify both claims.

Case A. Assume $\mathsf{msg\_key}$ is fresh: Our analysis of this case will rely on a property of the symmetric encryption scheme SE, and will require that its key k is chosen uniformly at random. Thus we begin by invoking the OTWIND-security of HASH and the RKPRF-security of KDF in order to claim that the output of $\mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$ is indistinguishable from random; this mirrors the first two steps of the IND-security reduction of $\mathsf{MTP\text{-}CH}$. We formalise this by requiring that $\mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$ is indistinguishable from a uniformly random value stored in the PRF table's entry $\mathsf{T}[\overline{u}, \mathsf{msg\_key}]$.

Our analysis of Case A now reduces roughly to the following: we need to show that it is hard to find any SE ciphertext c_{se} such that its decryption p under a uniformly random key k has a non-negligible chance of being successfully decoded by $\mathsf{ME.Decode}$ (i.e. returning $m \neq \perp$). As a part of this experiment, the adversary is allowed to query many different values of $\mathsf{msg\_key}$ and c_{se} (recall that an $\mathsf{MTP\text{-}CH}$ ciphertext contains both). At this point the $\mathsf{msg\_key}$ is only used to select a uniformly random SE key k from $\mathsf{T}[\overline{u}, \mathsf{msg\_key}]$, but the adversary can reuse the same key k in combination with many different choices of c_{se}. The Case A assumption that $\mathsf{msg\_key}$ is "fresh" means that the $\mathsf{msg\_key}$ was not seen during previous calls to the $\textsc{Send}$ oracle, so the adversary has no additional leakage on key k. All of the above is formalised by the UNPRED-security notion of SE with respect to ME.

The above security notion can be trivially broken if $\mathsf{ME.Decode}$ is defined in a way that it successfully decodes every possible payload $p \in \mathsf{ME.Out}$. It can also be trivially broken for contrived examples of SE like the one defining $\forall k \in \{0,1\}^{\mathsf{SE.kl}}, \forall x \in \mathsf{SE.MS}\colon \mathsf{SE.Enc}(k,x) = x \wedge \mathsf{SE.Dec}(k,x) = x$, assuming that $\mathsf{ME.Decode}$ can successfully decode even a single payload p from $\mathsf{SE.MS}$. But the more structure $\mathsf{ME.Decode}$ requires from its input p, and the more "unpredictable" is the function $\mathsf{SE.Dec}(k,\cdot)$ for a uniformly random k, the harder it is to break the UNPRED-security of SE, ME. We note that $\mathsf{MTP\text{-}ME}$ requires every p to contain a constant $\mathsf{session\_id} \in \{0,1\}^{64}$ in the second half of its 128-bit block, whereas $\mathsf{MTP\text{-}SE}$ implements the IGE block cipher mode of operation. In Appendix E6 we show that the output p of $\mathsf{MTP\text{-}SE.Dec}$ is highly unlikely to contain $\mathsf{session\_id}$ at the necessary position, i.e. if $\mathcal{F}_{\mathrm{INT}}$ makes $q_{\textsc{Send}}$ queries to its $\textsc{Send}$ oracle then it can find such p with probability at most $q_{\textsc{Send}}/2^{64}$. In Appendix E6 we also discuss the possibility of improving this bound.

Case B. Assume $\mathsf{msg\_key}$ is reused: In this case, we know that adversary $\mathcal{F}_{\mathrm{INT}}$ previously called its $\textsc{Send}$ oracle on inputs $\overline{u}, m', aux', r'$ for some m', aux', r', and received back a ciphertext $c' = (\mathsf{auth\_key\_id}, \mathsf{msg\_key}', c'_{se})$ such that $\mathsf{msg\_key}' = \mathsf{msg\_key}$. Let p' be the payload that was built and used inside this oracle call. Recall that we are currently considering $\mathcal{F}_{\mathrm{INT}}$'s ongoing call to its oracle $\textsc{Recv}$ on inputs u, c, aux such that $c = (\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se})$; we are only interested in the event that the $\mathsf{msg\_key}$ verification passed (and that $\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}'$), meaning that $\mathsf{msg\_key} = \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ holds for an appropriately recovered p.

It follows that $\mathsf{MAC.Ev}(mk_{\overline{u}}, p') = \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$. If $p' \neq p$ then this breaks the RKCR-security of MAC. Recall that MTProto instantiates MAC with $\mathsf{MTP\text{-}MAC}$ where $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$. So this attack against MAC reduces to breaking some variant of $\mathsf{SHA\text{-}256}$'s collision-resistance that restricts the set of allowed inputs but only requires to find a collision in a 128-bit fragment of the output.

Based on the the above, we obtain $(\mathsf{msg\_key}', p') = (\mathsf{msg\_key}, p)$. Let $k = \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$. Note that $c'_{se} \leftarrow \mathsf{SE.Enc}(k, p')$ was computed during the $\textsc{Send}$ call, and $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ was computed during the ongoing $\textsc{Recv}$ call. The equality $p' = p$ implies $c'_{se} = c_{se}$ if SE guarantees that for any key k, the algorithms of SE match every plaintext message $p \in \mathsf{SE.MS}$ with a unique ciphertext c_{se}. When this condition holds, we say that SE has *unique ciphertexts*. We note that $\mathsf{MTP\text{-}SE}$ satisfies this property; it follows that $c'_{se} = c_{se}$ and therefore the $\mathsf{MTP\text{-}CH}$ ciphertext c that was queried to $\textsc{Recv}$ (for user u) is equal to the ciphertext c' that was previously returned by $\textsc{Send}$ (by user $\overline{u}$). Implicit in this argument is an assumption that SE has the decryption correctness property; $\mathsf{MTP\text{-}ME}$ satisfies this property as well.

3) Proof phase II: $\mathsf{MTP\text{-}CH}$ acts as an authenticated channel: We can rewrite the claims we stated and justified in the first phase of the proof as follows. When adversary $\mathcal{F}_{\mathrm{INT}}$ queries its oracle $\textsc{Recv}$ on inputs u, c, aux, it gets back $m = \perp$ with high probability, unless c was honestly returned in response to $\mathcal{F}_{\mathrm{INT}}$'s prior call to $\textsc{Send}(\overline{u}, \ldots)$, meaning $\exists m', aux'\colon (\mathsf{sent}, m', c, aux') \in \mathsf{tr}_{\overline{u}}$. Furthermore, we claim that the channel state for user u does not change when $\mathcal{F}_{\mathrm{INT}}$'s queries to oracle $\textsc{Recv}$ result in $m = \perp$. This could only happen in Case A above, assuming that the $\mathsf{msg\_key}$ verification succeeds but then the $\mathsf{ME.Decode}$ call returns $m = \perp$ and changes user u's message encoder state $st_{\mathsf{ME},u}$. We note that $\mathsf{MTP\text{-}ME}$ never updates $st_{\mathsf{ME},u}$ when decoding fails, and hence it satisfies this requirement.

We now know that oracle $\textsc{Recv}$ accepts only honestly forwarded ciphertexts from the opposite user, and that it never changes the channel's state otherwise. This allows us to rewrite the INT-security game to ignore all cryptographic algorithms in the $\textsc{Recv}$ oracle. More specifically, oracle $\textsc{Send}$ can use the opposite user's transcript to check which ciphertexts were produced honestly, and simply reject the ones that are not on this transcript. For each ciphertext c that is on the transcript, the game can maintain a table that maps it to the payload p that was used to generate it; oracle $\textsc{Recv}$ can take this payload and immediately call $\mathsf{ME.Decode}$ to decode it.

4) Proof phase III: Interaction between ME and supp: By now, we have transformed our INT-security game to an extent that it roughly captures the requirement that the behaviour of ME should match that of supp (i.e. adversary $\mathcal{F}_{\mathrm{INT}}$ wins the game iff the message m produced by $\mathsf{ME.Decode}$ inside oracle $\textsc{Recv}$ is not equal to the corresponding output m^* of supp). However, the support function supp uses the $\mathsf{MTP\text{-}CH}$ encryption c of payload p as its label, and it is not necessarily clear what information about c can or should be used to define the behaviour of supp. In order the simplify the security game we have arrived to, we will rely on three correctness-style notions as follows. (1) Integrity of a support function requires that the support function returns $m^* = \perp$ when it is called on a ciphertext that cannot be found in the opposite user's transcript $\mathsf{tr}_{\overline{u}}$. (2) Robustness of a support function requires that adding failed decryption events (i.e. $m = \perp$) to a transcript does not affect the future outputs supp on any inputs. (3) We also rely on a property requiring that a support function uses no information about its labels beyond their equality pattern, separately for either direction of communication ($u \rightarrow \overline{u}$ and $\overline{u} \rightarrow u$). For the last property, we observe that in our game $p_0 = p_1$ iff the corresponding $\mathsf{MTP\text{-}CH}$ ciphertexts are also equal. This allows us to switch from using ciphertexts to using payloads as the labels for the supp, and simultaneously change the transcripts to also store payloads instead of ciphertexts. Our theorem is stated with respect to $\mathsf{supp} = \mathsf{SUPP}$ that satisfies all three of the above properties.

The introduced properties of a support function allow us to further simplify the INT-security game. This helps us to remove the corner case that deals with $\textsc{Recv}$ being queried on an invalid ciphertext (i.e. one that was not honestly forwarded). And finally this lets us reduce our latest version of the INT-security game for $\mathsf{MTP\text{-}CH}$ to the EINT property of $\mathsf{ME}, \mathsf{supp}$

(see Fig. 10) that is defined to match ME against supp in the presence of adversarial behaviour on an authenticated channel that exchanges ME payloads between two users. In Appendix E5 we show that this notion holds for MTP-ME with respect to SUPP.

5) Case A does not have to rely on ME.Decode: In the earlier analysis of Case A, we relied on a certain property of the message encoding scheme ME. Roughly speaking, we reasoned that the algorithm ME.Decode should not be able to successfully decode random-looking strings, meaning it should require that decodable payloads are structured in a certain way. We now briefly outline a proof strategy that might be applicable if one could not rely on such properties of ME.

In Case A adversary $\mathcal{F}_{\text{INT}}$ calls its oracle $\text{Recv}(u, c, aux)$ on $c = (\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se})$ with a msg_key value that was never previously returned by oracle Send as a part of a ciphertext produced by user $\overline{u}$. This, in particular, means that $k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$ can be thought of as a uniformly random key (due to the assumed OTWIND-security of HASH, and RKPRF-security of KDF) that was never previously used inside oracle Send but could have been used in previous queries to oracle Recv. Let us modify our initial goal for Case A (which required to show that ME.Decode will likely fail) as follows: we want to show that evaluating $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ and $\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ is very unlikely to result in $\mathsf{msg\_key}' = \mathsf{msg\_key}$.

A straightforward approach now is to assume that the function family MAC satisfies some form of either PRF-security or preimage-resistance. Then we would be able to argue that it is hard to find any $\mathsf{MAC.Ev}(mk_{\overline{u}}, \cdot)$ input p that maps to a fixed target output msg_key that was never seen before. The challenge here is that neither of the two security assumptions holds for MAC = MTP-MAC, which defines $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$, because it is based on SHA-256 and hence permits length extension attacks. The length extension can be applied to known input-output pairs of $\mathsf{MAC.Ev}(mk_{\overline{u}}, \cdot)$ in order to derive new valid input-output pairs, even without knowing the key $mk_{\overline{u}}$. Furthermore, in this case the length extension attack cannot be ruled out by assuming that $\mathsf{MAC.Ev}(mk_{\overline{u}}, \cdot)$ will be evaluated on a prefix-free set of inputs, because one could query Recv on c_{se} and c^*_{se} (with respect to the same key k) such that c_{se} is a prefix of c^*_{se}. Since SE = MTP-SE is based on a block cipher mode of operation, then $p = \mathsf{SE.Dec}(k, c_{se})$ will likewise be a prefix of $p^* = \mathsf{SE.Dec}(k, c^*_{se})$. However, as long as the SE key k can be shown to be uniformly random and unknown to the adversary, it should be hard to find the specific prefix value x such that $p \,\|\, x = p^*$; this non-standard condition might help to rule out the length extension attacks. One also has to take care of the possibility that a future call to oracle Send might hit the currently targeted challenge value of msg_key, especially if this proof step relies on the hardness of a decision problem (e.g. on a variant of PRF-security of MAC). Overall, this seems to be a viable proof strategy, but it would be much more involved than our approach that relies on the properties of ME.

Proof of Theorem 2. This proof uses games G_0–G_2 in Fig. 30a, and games G_3–G_{14} in Fig. 31. The adversaries for transitions between games are provided in Fig. 30, Fig. 32, and Fig. 33.

Games G_0–G_2 and the transitions between them ($G_0 \rightarrow G_1$ based on the OTWIND-security of HASH, and $G_1 \rightarrow G_2$ based on the RKPRF-security of KDF) are very similar to the corresponding games and transitions in our IND-security reduction. We refer to the proof of Theorem 1 for a detailed explanation of both transitions.

G_0: Game G_0 is equivalent to game $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\text{INT}}}$. It expands the code of algorithms CH.Init, CH.Send and CH.Send. The expanded instructions are highlighted in gray. It follows that

$$\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}_{\text{INT}}) = \Pr[G_0].$$

$G_0 \rightarrow G_1$: The value of auth_key_id in game G_0 depends on the initial KDF key kk. In contrast, game G_1 computes auth_key_id by evaluating HASH on uniformly random inputs that are independent of kk. We invoke the OTWIND-security of HASH (Fig. 20) in order to claim that adversary $\mathcal{F}_{\text{INT}}$ cannot distinguish between playing in G_0 and G_1. In Fig. 30b we build an adversary $\mathcal{D}_{\text{OTWIND}}$ against the OTWIND-security of HASH. When adversary $\mathcal{D}_{\text{OTWIND}}$ plays in game $G^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}_{\text{OTWIND}}}$ with challenge bit $d \in \{0,1\}$, it simulates game G_0 (when $d = 1$) or game G_1 (when $d = 0$) for adversary $\mathcal{F}_{\text{INT}}$. Adversary $\mathcal{D}_{\text{OTWIND}}$ returns $d' = 1$ iff $\mathcal{F}_{\text{INT}}$ sets win, so we have

$$\Pr[G_0] - \Pr[G_1] = \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}_{\text{OTWIND}}).$$

$G_1 \rightarrow G_2$: Going from G_1 to G_2, we switch the outputs of KDF.Ev to uniformly random values. Since the adversary can call $k \leftarrow \mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$ on the same inputs multiple times, we use the PRF table T to enforce the consistency between calls; the output of $\mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$ in G_1 corresponds to a uniformly random value that is sampled and stored in the table entry $\mathsf{T}[u, \mathsf{msg\_key}]$. In Fig. 30c we build an adersary $\mathcal{D}_{\text{RKPRF}}$ against the RKPRF-security of KDF (with respect to ϕ_{KDF}, Fig. 21) with respect to ϕ_{KDF}. When adversary $\mathcal{D}_{\text{RKPRF}}$ plays in game $G^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\text{RKPRF}}}$ with challenge bit $d \in \{0,1\}$, it simulates game G_1 (when $d = 1$) or game G_2 (when $d = 0$) for adversary $\mathcal{F}_{\text{INT}}$. Adversary $\mathcal{D}_{\text{RKPRF}}$ returns $d' = 1$ iff $\mathcal{F}_{\text{INT}}$ sets win, so we have

$$\Pr[G_1] - \Pr[G_2] = \mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\text{RKPRF}}).$$

$G_2 \rightarrow G_3$: Game G_3 differs from game G_2 in the following ways. (1) The KDF keys kk, $kk_{\mathcal{I}}$, $kk_{\mathcal{R}}$ are no longer used in our reduction games starting from G_2, so they are not included in game G_3 and onwards. (2) Game G_3 adds a table S that is updated during each call to oracle Send. We set $\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$ to remember that user u produced msg_key when sending (to user $\overline{u}$) an SE ciphertext c_{se}, that encrypts payload p. (3) Oracle Recv in game G_3, prior to calling ME.Decode, now saves a backup copy of $st_{\mathsf{ME},u}$ in variable $st^*_{\mathsf{ME},u}$. It then adds four new conditional statements that do not serve any purpose in game G_3. Four of the future game transitions in our security reduction ($G_3 \rightarrow G_4$, $G_4 \rightarrow G_5$, $G_5 \rightarrow G_6$, $G_7 \rightarrow G_8$) will do the following. Each of them

```
Games $G_0$–$G_2$
$\mathsf{win} \leftarrow \mathsf{false}$ ; $hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
$kk \leftarrow\!\!\$ \{0,1\}^{672}$ ; $mk \leftarrow\!\!\$ \{0,1\}^{320}$
$x \leftarrow kk \,\|\, mk$                                   // $G_0$
$x \leftarrow\!\!\$ \{0,1\}^{992}$                            // $G_1$–$G_2$ (OTWIND of HASH)
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$ ; $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$ \mathsf{ME.Init}()$
$\mathcal{F}_{\mathrm{INT}}^{\mathrm{SEND},\mathrm{RECV}}$ ; Return $\mathsf{win}$

SEND$(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathrm{T}[u, \mathsf{msg\_key}] = \bot$ then $\mathrm{T}[u, \mathsf{msg\_key}] \leftarrow\!\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$     // $G_0$–$G_1$
$k \leftarrow \mathrm{T}[u, \mathsf{msg\_key}]$              // $G_2$ (RKPRF of KDF)
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$ ; $c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$ ; Return $c$

RECV$(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathrm{T}[\overline{u}, \mathsf{msg\_key}] = \bot$ then $\mathrm{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow\!\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$     // $G_0$–$G_1$
$k \leftarrow \mathrm{T}[\overline{u}, \mathsf{msg\_key}]$              // $G_2$ (RKPRF of KDF)
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ ; $\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ ; $m \leftarrow \bot$
If $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$ then
        $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$ ; If $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$ ; Return $m$
```

(a) Games G_0–G_2. The code added by expanding the algorithms of CH in game $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\mathrm{INT}}}$ is highlighted in gray. The code added for the transitions between games is highlighted in green.

```
Adversary $\mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, \mathsf{auth\_key\_id})$
$kk \,\|\, mk \leftarrow x_1$     // Such that $|kk| = 672$ and $|mk| = 320$.
$\mathsf{win} \leftarrow \mathsf{false}$ ; $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$ \mathsf{ME.Init}()$
$\mathcal{F}_{\mathrm{INT}}^{\mathrm{SENDSIM},\mathrm{RECVSIM}}$ ; If win then return 1 else return 0
SENDSIM$(u, m, aux, r)$ and RECVSIM$(u, c, aux)$
// Identical to oracles SEND and RECV in games $G_0$, $G_1$.
```

(b) Adversary $\mathcal{D}_{\mathrm{OTWIND}}$ against the OTWIND-security of HASH for transition between games G_0–G_1.

```
Adversary $\mathcal{D}_{\mathrm{RKPRF}}^{\mathrm{ROR}}$
$\mathsf{win} \leftarrow \mathsf{false}$ ; $hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\!\!\$ \{0,1\}^{320}$
$x \leftarrow\!\!\$ \{0,1\}^{992}$ ; $\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$ \mathsf{ME.Init}()$
$\mathcal{F}_{\mathrm{INT}}^{\mathrm{SENDSIM},\mathrm{RECVSIM}}$ ; If win then return 1 else return 0

SENDSIM$(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$ ; $k \leftarrow \mathrm{ROR}(u, \mathsf{msg\_key})$
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$ ; $c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$ ; Return $c$

RECVSIM$(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$ ; $k \leftarrow \mathrm{ROR}(\overline{u}, \mathsf{msg\_key})$
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ ; $\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ ; $m \leftarrow \bot$
If $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$ then
    $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$ ; If $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$ ; Return $m$
```

(c) Adversary $\mathcal{D}_{\mathrm{RKPRF}}$ against the RKPRF-security of KDF for transition between games G_1–G_2.

Figure 30: Games G_0–G_2 and the corresponding adversaries for proof of Theorem 2. The instructions that are highlighted inside adversaries mark the changes in the code of the simulated security reduction games.

will add an instruction, inside the corresponding conditional statement, that reverts the pair of variables $(st_{\mathsf{ME},u}, m)$ to their initial values $(st^*_{\mathsf{ME},u}, \bot)$ that they had at the beginning of the ongoing RECV oracle call. Each of the new conditional statements also contains its own bad flag; these flags are only used for the formal analysis that we provide below. (4) Similar to above, game G_3 adds two conditional statements to the SEND oracle, and both serve no purpose in game G_3. In future games they will be used to roll back the message encoder's state $st_{\mathsf{ME},u}$ to its initial value that it had at the beginning of the ongoing SEND oracle call, followed by exiting this oracle call with $\bot$ as output. Games G_3 and G_2 are functionally equivalent, so

$$\Pr[G_3] = \Pr[G_2].$$

$G_3 \rightarrow G_4$*:* Games G_3 and G_4 are identical until bad_0 is set. According to the Fundamental Lemma of Game Playing [14],

$$\Pr[G_3] - \Pr[G_4] \leq \Pr[\mathsf{bad}_0^{G_3}],$$

where $\Pr[\mathsf{bad}^{\mathrm{Q}}]$ denotes the probability of setting flag bad in game Q. The bad_0 flag can be set in G_3 only when the instruction $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ simultaneously changes the value of $st_{\mathsf{ME},u}$ and returns $m = \bot$. Recall that the statement of Theorem 2 restricts ME to an instantiation of MTP-ME. But the latter never modifies its state $st_{\mathsf{ME},u}$ when the decoding fails (i.e. $m = \bot$), so

$$\Pr[\mathsf{bad}_0^{G_3}] = 0.$$

$G_4 \rightarrow G_5$*:* Games G_4 and G_5 are identical until bad_1 is set. According to the Fundamental Lemma of Game Playing [14],

$$\Pr[G_4] - \Pr[G_5] \leq \Pr[\mathsf{bad}_1^{G_5}].$$

When the bad_1 flag is set in G_5, we know that the SE key $k = \mathrm{T}[\overline{u}, \mathsf{msg\_key}]$ was sampled uniformly at random and never used inside the SEND oracle before (because $\mathrm{S}[\overline{u}, \mathsf{msg\_key}] = \bot$). Yet the adversary $\mathcal{F}_{\mathrm{INT}}$ found an SE ciphertext c_{se} such that the payload $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ was successfully decoded by $\mathsf{ME.Decode}$ (i.e. $m \neq \bot$). We note that $\mathcal{F}_{\mathrm{INT}}$ is allowed to query its RECV oracle on arbitrarily many ciphertexts c_{se} with respect to the same SE key k, by repeatedly using the same pair of values for $(\overline{u}, \mathsf{msg\_key})$. But it might nonetheless be hard for $\mathcal{F}_{\mathrm{INT}}$ to obtain a decodable payload p if (1) the outputs of function $\mathsf{SE.Dec}(k, \cdot)$ are sufficiently "unpredictable" for an unknown uniformly random k, and (2) the $\mathsf{ME.Decode}$ algorithm is sufficiently "restrictive" (e.g. designed to run some sanity checks on its payloads, hence rejecting a

```
Games $G_3$–$G_8$
win ← false ; $hk \leftarrow_{\$} \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow_{\$} \{0,1\}^{320}$
$x \leftarrow_{\$} \{0,1\}^{992}$ ; auth_key_id ← HASH.Ev($hk, x$)
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow_{\$}$ ME.Init()
$\mathcal{F}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$ ; Return win

$\textsc{Send}(u, m, aux, r)$
$st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$ ; $(st_{\mathsf{ME},u}, p) \leftarrow$ ME.Encode($st_{\mathsf{ME},u}, m, aux; r$)
msg_key ← MAC.Ev($mk_u, p$)
If T[$u$, msg_key] $=\perp$ then T[$u$, msg_key] $\leftarrow_{\$} \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow$ T[$u$, msg_key] ; $c_{se} \leftarrow$ SE.Enc($k, p$)
If S[$u$, msg_key] $\neq\perp$ then
    $(p', c'_{se}) \leftarrow$ S[$u$, msg_key]
    If $p \neq p'$ then
        $\mathsf{bad}_2 \leftarrow$ true
        $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\perp$    // $G_6$–$G_8$ (RKCR of MAC)
If SE.Dec($k, c_{se}$) $\neq p$ then
    $\mathsf{bad}_3 \leftarrow$ true
    $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\perp$    // $G_7$–$G_8$ (SE = MTP-SE)
S[$u$, msg_key] ← $(p, c_{se})$ ; $c \leftarrow$ (auth_key_id, msg_key, $c_{se}$)
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\,$ (sent, $m, c, aux$) ; Return $c$

$\textsc{Recv}(u, c, aux)$
(auth_key_id′, msg_key, $c_{se}$) ← $c$
If T[$\overline{u}$, msg_key] $=\perp$ then T[$\overline{u}$, msg_key] $\leftarrow_{\$} \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow$ T[$\overline{u}$, msg_key] ; $p \leftarrow$ SE.Dec($k, c_{se}$)
msg_key′ ← MAC.Ev($mk_{\overline{u}}, p$) ; $m \leftarrow \perp$
If (msg_key′ = msg_key) ∧ (auth_key_id = auth_key_id′) then
    $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$ ; $(st_{\mathsf{ME},u}, m) \leftarrow$ ME.Decode($st_{\mathsf{ME},u}, p, aux$)
    If S[$\overline{u}$, msg_key] $=\perp$ then
        If $(m = \perp) \wedge (st_{\mathsf{ME},u} \neq st^*_{\mathsf{ME},u})$ then
            $\mathsf{bad}_0 \leftarrow$ true
            $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$    // $G_4$–$G_8$ (ME = MTP-ME)
        If $m \neq \perp$ then
            $\mathsf{bad}_1 \leftarrow$ true
            $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \perp)$    // $G_5$–$G_8$ (UNPRED of SE, ME)
    If S[$\overline{u}$, msg_key] $\neq\perp$ then
        $(p', c'_{se}) \leftarrow$ S[$\overline{u}$, msg_key]
        If $p \neq p'$ then
            $\mathsf{bad}_2 \leftarrow$ true
            $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \perp)$    // $G_6$–$G_8$ (RKCR of MAC)
        Else if $c_{se} \neq c'_{se}$ then
            $\mathsf{bad}_4 \leftarrow$ true
            $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \perp)$    // $G_8$ (SE = MTP-SE)
$m^* \leftarrow$ supp($u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux$) ; If $m \neq m^*$ then win ← true
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\,$ (recv, $m, c, aux$) ; Return $m$
```

```
Games $G_9$–$G_{13}$
win ← false ; $hk \leftarrow_{\$} \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow_{\$} \{0,1\}^{320}$
$x \leftarrow_{\$} \{0,1\}^{992}$ ; auth_key_id ← HASH.Ev($hk, x$)
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow_{\$}$ ME.Init()
$\mathcal{F}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$ ; Return win

$\textsc{Send}(u, m, aux, r)$
$st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$ ; $(st_{\mathsf{ME},u}, p) \leftarrow$ ME.Encode($st_{\mathsf{ME},u}, m, aux; r$)
msg_key ← MAC.Ev($mk_u, p$)
If T[$u$, msg_key] $=\perp$ then T[$u$, msg_key] $\leftarrow_{\$} \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow$ T[$u$, msg_key] ; $c_{se} \leftarrow$ SE.Enc($k, p$)
If (S[$u$, msg_key] $\neq\perp$) ∧ (S[$u$, msg_key] $\neq (p, c_{se})$) then
    $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\perp$
If SE.Dec($k, c_{se}$) $\neq p$ then
    $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\perp$
S[$u$, msg_key] ← $(p, c_{se})$ ; $c \leftarrow$ (auth_key_id, msg_key, $c_{se}$)
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\,$ (sent, $m, c, aux$)    // $G_9$–$G_{11}$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\,$ (sent, $m, p, aux$)    // $G_{12}$–$G_{13}$ (supp = SUPP)
P[$u, c$] ← $p$ ; Return $c$

$\textsc{Recv}(u, c, aux)$
If P[$\overline{u}, c$] $\neq\perp$ then    // $\exists m', aux'$ : (sent, $m', c, aux'$) $\in \mathsf{tr}_{\overline{u}}$
    $p \leftarrow$ P[$\overline{u}, c$] ; $(st_{\mathsf{ME},u}, m) \leftarrow$ ME.Decode($st_{\mathsf{ME},u}, p, aux$)
    $m^* \leftarrow$ supp($u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux$)  }
    $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\,$ (recv, $m, c, aux$)  }    // $G_9$–$G_{11}$
    $m^* \leftarrow$ supp($u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, p, aux$)  }
    $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\,$ (recv, $m, p, aux$)  }    // $G_{12}$–$G_{13}$ (supp = SUPP)
Else
    $m \leftarrow \perp$ ; $m^* \leftarrow$ supp($u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux$)
    If $m^* \neq \perp$ then
        $\mathsf{bad}_5 \leftarrow$ true
        $m^* \leftarrow \perp$    // $G_{10}$–$G_{13}$ (supp = SUPP)
    $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\,$ (recv, $m, c, aux$)    // $G_9$–$G_{10}$ (supp = SUPP)
If $m \neq m^*$ then
    $\mathsf{bad}_6 \leftarrow$ true
    win ← true    // $G_9$–$G_{12}$ (EINT of ME, supp)
Return $m$
```

Figure 31: Games G_3–G_{13} for proof of Theorem 2. Right pane: The code highlighted in gray is functionally equivalent to the corresponding code in G_8. Both panes: The code added for the transitions between games is highlighted in green.

fraction of them). We use the unpredictability notion of SE with respect to ME, which captures this intuition. In Fig. 32a we build an adversary $\mathcal{F}_{\mathrm{UNPRED}}$ against the UNPRED-security of SE, ME (Fig. 27) as follows. When adversary $\mathcal{F}_{\mathrm{UNPRED}}$ plays in game $G^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}_{\mathrm{UNPRED}}}$, it simulates game G_5 for adversary $\mathcal{F}_{\mathrm{INT}}$. Adversary $\mathcal{F}_{\mathrm{UNPRED}}$ wins in its own game whenever $\mathcal{F}_{\mathrm{INT}}$ sets bad_1, so we have

$$\Pr[\mathsf{bad}_1^{G_5}] \leq \mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}_{\mathrm{UNPRED}}).$$

First of all, adversary $\mathcal{F}_{\mathrm{UNPRED}}$ does not maintain its own transcripts $\mathsf{tr}_u, \mathsf{tr}_{\overline{u}}$, and hence does not evaluate the support function supp at the end of the simulated $\textsc{Recv}$ oracle. This is because supp's outputs do not affect the input-output behaviour of the simulated oracles $\textsc{Send}$ and $\textsc{Recv}$, and because this reduction step does not rely on whether adversary $\mathcal{F}_{\mathrm{INT}}$ succeeds to win in the simulated game (but rather only whether it sets bad_1). Some of the adversaries we construct for the next reduction steps will likewise not maintain the

```
Adversary $\mathcal{F}_{\mathrm{UNPRED}}^{\textsc{Expose},\textsc{Ch}}$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\$ \{0,1\}^{320}$ ; $x \leftarrow\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$ ; $\mathcal{F}_{\mathrm{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$
$\textsc{SendSim}(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathsf{S}[u, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow \textsc{Expose}(u, \mathsf{msg\_key})$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; $c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$ ; $c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
Return $c$
$\textsc{RecvSim}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] \neq \perp$ then
    If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
    $k \leftarrow \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
    $\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ ; $m \leftarrow \perp$
    If $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$ then
        $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \perp$ then
    $m \leftarrow \perp$ ; $\textsc{Ch}(\overline{u}, \mathsf{msg\_key}, c_{se}, st_{\mathsf{ME},u}, aux)$
Return $m$
```

(a) Adversary $\mathcal{F}_{\mathrm{UNPRED}}$ against the UNPRED-security of SE, ME for transition between games G_4–G_5.

```
Adversary $\mathcal{F}_{\mathrm{RKCR}}(mk_{\mathcal{I}}, mk_{\mathcal{R}})$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $x \leftarrow\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$ ; $\mathcal{F}_{\mathrm{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$ ; Return out
$\textsc{SendSim}(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; $c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
If $\mathsf{S}[u, \mathsf{msg\_key}] \neq \perp$ then
    $(p', c_{se}') \leftarrow \mathsf{S}[u, \mathsf{msg\_key}]$ ; If $p \neq p'$ then $\mathrm{out} \leftarrow (u, p, p')$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$ ; $c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
Return $c$
$\textsc{RecvSim}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \perp$ then $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
$\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ ; $m \leftarrow \perp$
If $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$ then
    $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$ ; $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
    If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \perp$ then $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \perp)$
    If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] \neq \perp$ then
        $(p', c_{se}') \leftarrow \mathsf{S}[\overline{u}, \mathsf{msg\_key}]$ ; If $p \neq p'$ then $\mathrm{out} \leftarrow (\overline{u}, p, p')$
Return $m$
```

(b) Adversary $\mathcal{F}_{\mathrm{RKCR}}$ against the RKCR-security of MAC for transition between games G_5–G_6.

Figure 32: Adversaries for transitions between games G_4–G_6 in proof of Theorem 2. The highlighted instructions mark the changes in the code of the simulated security reduction games.

transcripts.

Adversary $\mathcal{F}_{\mathrm{UNPRED}}$ splits the simulation of game G_5's $\textsc{Recv}$ oracle into two cases. (1) If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] \neq \perp$, then $\mathcal{F}_{\mathrm{UNPRED}}$ honestly simulates all instructions that would have been evaluated by $\textsc{Recv}$. (2) If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \perp$, then $\mathcal{F}_{\mathrm{UNPRED}}$ does not modify $st_{\mathsf{ME},u}$ and always returns $m = \perp$; this is consistent with the behaviour of oracle $\textsc{Recv}$ in game G_5. In addition to the latter, adversary $\mathcal{F}_{\mathrm{UNPRED}}$ also makes a call to its oracle $\textsc{Ch}$. This oracle simulates all instructions that would have been evaluated by $\textsc{Recv}$ when $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \perp$, except it omits the condition checking $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$. This condition is a prerequisite to setting flag bad_1 in game G_5; the change is fine because adversary $\mathcal{F}_{\mathrm{INT}}$ will set this flag in the simulated game at least as often as in the real game. Finally, adversary $\mathcal{F}_{\mathrm{UNPRED}}$ uses its $\textsc{Expose}$ oracle to learn the values from the PRF table that is maintained by the UNPRED-security game, and synchronizes them with its own PRF table T inside the simulated oracle $\textsc{Send}$ (intuitively, this appears unnecessary, but it helps us avoid further analysis to show that $\mathcal{F}_{\mathrm{UNPRED}}$ perfectly simulates game G_5).

$\mathrm{G}_5 \rightarrow \mathrm{G}_6$: Games G_5 and G_6 are identical until bad_2 is set. According to the Fundamental Lemma of Game Playing [14],

$$\Pr[\mathrm{G}_5] - \Pr[\mathrm{G}_6] \leq \Pr[\mathsf{bad}_2^{\mathrm{G}_5}].$$

Game G_5 sets the bad_2 flag in two different places: one inside oracle $\textsc{Send}$, and one inside oracle $\textsc{Recv}$. In either case, this happens when the table entry $\mathsf{S}[w, \mathsf{msg\_key}] = (p', c_{se}')$, for some $w \in \{\mathcal{I}, \mathcal{R}\}$, indicates that a prior call to oracle $\textsc{Send}$ obtained $\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_w, p')$, and now we found p such that $p \neq p'$ and $\mathsf{msg\_key} = \mathsf{MAC.Ev}(mk_w, p)$. This results in a collision for MAC under related keys, and hence breaks its RKCR-security (with respect to ϕ_{MAC}, Fig. 22). In Fig. 32b we build an adversary $\mathcal{F}_{\mathrm{RKCR}}$ against the RKCR-security of MAC with respect to ϕ_{MAC} as follows. When adversary $\mathcal{F}_{\mathrm{RKCR}}$ plays in game $\mathrm{G}^{\mathrm{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{F}_{\mathrm{RKCR}}}$, it simulates game G_5 for adversary $\mathcal{F}_{\mathrm{INT}}$. Adversary $\mathcal{F}_{\mathrm{RKCR}}$ wins in its own game whenever $\mathcal{F}_{\mathrm{INT}}$ sets bad_2, so we have

$$\Pr[\mathsf{bad}_2^{\mathrm{G}_5}] \leq \mathsf{Adv}^{\mathrm{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{F}_{\mathrm{RKCR}}).$$

$\mathrm{G}_6 \rightarrow \mathrm{G}_7$: Games G_6 and G_7 are identical until bad_3 is set. According to the Fundamental Lemma of Game Playing [14],

$$\Pr[\mathrm{G}_6] - \Pr[\mathrm{G}_7] \leq \Pr[\mathsf{bad}_3^{\mathrm{G}_6}].$$

If bad_3 is set in G_6, it means that adversary $\mathcal{F}_{\mathrm{INT}}$ found a payload p and an SE key $k \in \{0,1\}^{\mathsf{SE.kl}}$ such that $\mathsf{SE.Dec}(k, \mathsf{SE.Enc}(k, p)) \neq p$. This violates the *decryption correctness* of SE. Recall that the statement of Theorem 2 considers $\mathsf{SE} = \mathsf{MTP\text{-}SE}$. The MTP-SE scheme satisfies decryption correctness, so

$$\Pr[\mathsf{bad}_3^{\mathrm{G}_6}] = 0.$$

$\mathbf{G}_7 \to \mathbf{G}_8$*:* Games G_7 and G_8 are identical until bad_4 is set. According to the Fundamental Lemma of Game Playing [14],

$$\Pr[\mathrm{G}_7] - \Pr[\mathrm{G}_8] \le \Pr[\mathsf{bad}_4^{\mathrm{G}_7}].$$

Whenever bad_4 is set in game G_7, we know that (1) $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ was computed during the ongoing RECV call, and (2) $c'_{se} \leftarrow \mathsf{SE.Enc}(k,p)$ was computed during an earlier call to SEND, which also verified that $\mathsf{SE.Dec}(k, c'_{se}) = p$. Importantly, we also know that $c_{se} \neq c'_{se}$. The statement of Theorem 2 considers $\mathsf{SE} = \mathsf{MTP\text{-}SE}$. The latter is a deterministic symmetric encryption scheme that is based on the IGE block cipher mode of operation. For each key $k \in \{0,1\}^{\mathsf{SE.kl}}$ and each length $\ell \in \mathbb{N}$ such that $\{0,1\}^\ell \subseteq \mathsf{SE.MS}$, this scheme specifies a permutation between all plaintexts from $\{0,1\}^\ell$ and all ciphertexts from $\{0,1\}^\ell$. In particular, this means that $\mathsf{MTP\text{-}SE}$ has *unique ciphertexts*, meaning it is impossible to find $c_{se} \neq c'_{se}$ that, under any fixed choice of key k, decrypt to the same payload p. It follows that bad_4 can never be set when $\mathsf{SE} = \mathsf{MTP\text{-}SE}$, so we have

$$\Pr[\mathsf{bad}_4^{\mathrm{G}_7}] = 0.$$

$\mathbf{G}_8 \to \mathbf{G}_9$*:* We say that a ciphertext c belongs to (or appears in) a support transcript tr iff $\exists m', aux' \colon (\mathsf{sent}, m', c, aux') \in \mathsf{tr}$.

Let us start by showing that oracle RECV in game G_8 evaluates $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ *and* does not subsequently roll back the variables $st_{\mathsf{ME},u}, m$ to the initial values they had at the beginning of the ongoing oracle call iff c belongs to $\mathsf{tr}_{\overline{u}}$. (1) If oracle RECV evaluates $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ and does not restore the values of $st_{\mathsf{ME},u}, m$, then $\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}'$ and $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = (p, c_{se})$ (the latter implies $\mathsf{msg\_key}' = \mathsf{msg\_key}$). According to the construction of oracle SEND, this means that the ciphertext $c' = (\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se})$ appears in transcript $\mathsf{tr}_{\overline{u}}$. (2) Let $c = (\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se})$ be any $\mathsf{MTP\text{-}CH}$ ciphertext, and let $\overline{u} \in \{\mathcal{I}, \mathcal{R}\}$. If c belongs to $\mathsf{tr}_{\overline{u}}$, then by construction of oracle SEND we know that $\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}'$ and $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = (p, c_{se})$ for the payload p such that $k = \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$, and $c_{se} = \mathsf{SE.Enc}(k, p)$, and $p = \mathsf{SE.Dec}(k, c_{se})$. The latter equality is guaranteed by the decryption correctness of $\mathsf{SE} = \mathsf{MTP\text{-}SE}$ that we used for transition $\mathrm{G}_6 \to \mathrm{G}_7$. The RKCR-security of MAC guarantees that once $\mathsf{S}[\overline{u}, \mathsf{msg\_key}]$ is populated, a future call to oracle SEND cannot overwrite $\mathsf{S}[\overline{u}, \mathsf{msg\_key}]$ with a different pair of values. All of the above implies that if c belongs to $\mathsf{tr}_{\overline{u}}$ at the beginning of a call to oracle RECV, then this oracle will successfully verify that $\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}'$ and $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = (p, c_{se})$ for $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ (whereas $\mathsf{msg\_key}' = \mathsf{msg\_key}$ follows from $\mathsf{S}[\overline{u}, \mathsf{msg\_key}]$ containing the payload p). It means that the instruction $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ will be evaluated, and the variables $st_{\mathsf{ME},u}, m$ will not be subsequently rolled back to their initial values.

Game G_9 differs from game G_8 in the following ways. (1) Game G_9 adds a payload table P that is updated during each call to oracle SEND. We set $\mathsf{P}[u, c] \leftarrow p$ to indicate that the $\mathsf{MTP\text{-}CH}$ ciphertext c, which was sent from user u to user $\overline{u}$, encrypts the payload p. Observe that any pair (u, c) with $c = (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ corresponds to a unique payload that can be recovered as $p \leftarrow \mathsf{SE.Dec}(\mathsf{T}[u, \mathsf{msg\_key}], c_{se})$. This relies on decryption correctness of SE, which is guaranteed to hold for ciphertexts inside table P due to the changes that we introduced in the transition between games $\mathrm{G}_6 \to \mathrm{G}_7$. (2) Game G_9 rewrites the code of game G_8's oracle RECV to run $\mathsf{ME.Decode}$ iff the ciphertext c belongs to the transcript $\mathsf{tr}_{\overline{u}}$; otherwise, the RECV oracle does not change $st_{\mathsf{ME},u}$ and simply sets $m \leftarrow \perp$. This follows from the analysis of G_8 that we provided above. We note that checking whether c belongs to $\mathsf{tr}_{\overline{u}}$ is equivalent to checking $\mathsf{P}[\overline{u}, c] \neq \perp$. For simplicity, we do the latter; and if the condition is satisfied, then we set $p \leftarrow \mathsf{P}[\overline{u}, c]$ and run $\mathsf{ME.Decode}$ with this payload as input. As discussed above, the $\mathsf{MTP\text{-}CH}$ ciphertext c that is issued by user $\overline{u}$ always encrypts a unique payload p, and hence we can rely that the table entry $\mathsf{P}[\overline{u}, c]$ stores this unique payload value. (3) Game G_9 also rewrites one condition inside oracle SEND, in a more compact but equivalent way. It also adds one new conditional statement to oracle RECV (checking $m^* \neq \perp$), but it serves no purpose in G_9. Games G_9 and G_8 are functionally equivalent, so

$$\Pr[\mathrm{G}_9] = \Pr[\mathrm{G}_8].$$

$\mathbf{G}_9 \to \mathbf{G}_{10}$*:* Game G_{10} enforces that $m^* = \perp$ whenever its oracle RECV is called on a ciphertext that cannot be found in the appropriate user's transcript. Games G_9 and G_{10} are identical until bad_5 is set. According to the Fundamental Lemma of Game Playing [14],

$$\Pr[\mathrm{G}_9] - \Pr[\mathrm{G}_{10}] \le \Pr[\mathsf{bad}_5^{\mathrm{G}_9}].$$

If bad_5 is set in game G_9 then the support function supp returned $m^* \neq \perp$ in response to an $\mathsf{MTP\text{-}CH}$ ciphertext c that does not belong to the opposite user's transcript $\mathsf{tr}_{\overline{u}}$. The statement of Theorem 2 considers $\mathsf{supp} = \mathsf{SUPP}$. The latter is defined to always return $m^* = \perp$ when its input label does not appear in $\mathsf{tr}_{\overline{u}}$, so

$$\Pr[\mathsf{bad}_5^{\mathrm{G}_9}] = 0.$$

In Section III we refer to this property as the *integrity* of supp, and we also formally define it in Fig. 36.

$\mathbf{G}_{10} \to \mathbf{G}_{11}$*:* Game G_{11} stops adding all entries of the form $(\mathsf{recv}, \perp, c, aux)$ to the transcripts of both users. Once this is done, it becomes pointless for adversary $\mathcal{F}_{\mathrm{INT}}$ to call its RECV oracle on any ciphertext that does not appear in the appropriate user's transcript. This is because such a call will never set the win flag (due to the change introduced in transition $\mathrm{G}_9 \to \mathrm{G}_{10}$) and will never affect the transcript of either user (due to the change introduced in this transition). The statement of Theorem 2 considers $\mathsf{supp} = \mathsf{SUPP}$. The latter is defined to ignore all transcript entries of the form $(\mathsf{recv}, \perp, c, aux)$, so removing the instruction $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$ for $m = \perp$ will not affect the outputs of any future calls to this support function. We have

$$\Pr[\mathrm{G}_{11}] = \Pr[\mathrm{G}_{10}].$$

In Section III we refer to this property as the *robustness* of supp.

$\mathbf{G}_{11} \rightarrow \mathbf{G}_{12}$: When discussing the differences between games G_9 and G_8, we showed that for each pair of sender $u \in \{\mathcal{I}, \mathcal{R}\}$ and MTP-CH ciphertext c, the encrypted payload p is unique. It is also true that for each pair of $u \in \{\mathcal{I}, \mathcal{R}\}$ and payload p, there is a unique MTP-CH ciphertext c that encrypts p in the direction from u to $\overline{u}$. It follows that in games G_{11} and G_{12} for any fixed user $u \in \{\mathcal{I}, \mathcal{R}\}$ there is a 1-to-1 correspondence between payloads and MTP-CH ciphertexts that could be successfully sent from u to $\overline{u}$ (i.e. this property does not hold if SE does not have decryption correctness, but the code added for the transition $\mathrm{G}_6 \rightarrow \mathrm{G}_7$ already identifies and discards the corresponding ciphertexts). The statement of Theorem 2 considers $\mathsf{supp} = \mathsf{SUPP}$. Observe that for any label z sent from u to $\overline{u}$, the support function SUPP cheks only its equality with every z^* such that $(\mathsf{sent}, m, z^*, aux) \in \mathsf{tr}_u$ or $(\mathsf{recv}, m, z^*, aux) \in \mathsf{tr}_{\overline{u}}$ for any values of m, aux. In other words, this support function only looks at the *equality pattern* of the labels, and it does this independently in each of the two directions between the users. The 1-to-1 correspondence between c and p, with respect to any fixed user u, means we can replace the labels used in transcripts from c to p, and replace the label inputs to the support function SUPP in the same way; the future outputs of the support function will remain the same. We have

$$\Pr[\mathrm{G}_{12}] = \Pr[\mathrm{G}_{11}].$$

```
Adversary $\mathcal{F}_{\mathrm{EINT}}^{\mathrm{SEND},\mathrm{RECV}}(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
$hk \leftarrow_{\$} \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow_{\$} \{0,1\}^{320}$ ; $x \leftarrow_{\$} \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$ ; $\mathcal{F}_{\mathrm{INT}}^{\mathrm{SENDSIM},\mathrm{RECVSIM}}$

$\mathrm{SENDSIM}(u, m, aux, r)$
$st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$ ; $(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \bot$ then $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow_{\$} \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; $c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
If $(\mathsf{S}[u, \mathsf{msg\_key}] \neq \bot) \wedge (\mathsf{S}[u, \mathsf{msg\_key}] \neq (p, c_{se}))$ then
    $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\bot$
If $\mathsf{SE.Dec}(k, c_{se}) \neq p$ then
    $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\bot$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$ ; $c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
$\mathrm{SEND}(u, m, aux, r)$ ; $\mathsf{P}[u, c] \leftarrow p$ ; Return $c$

$\mathrm{RECVSIM}(u, c, aux)$
If $\mathsf{P}[\overline{u}, c] \neq \bot$ then
    $p \leftarrow \mathsf{P}[\overline{u}, c]$ ; $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
    $m \leftarrow \mathrm{RECV}(u, p, aux)$ ; Return $m$
Else return $\bot$
```

Figure 33: Adversary $\mathcal{F}_{\mathrm{EINT}}$ against the EINT-security of $\mathsf{ME}, \mathsf{supp}$ for transition between games G_{12}–G_{13} in proof of Theorem 2. The highlighted instructions mark the locations in the pseudocode of the simulated game G_{13} where adversary $\mathcal{F}_{\mathrm{EINT}}$ uses its own oracles.

$\mathbf{G}_{12} \rightarrow \mathbf{G}_{13}$: Games G_{12} and G_{13} are identical until bad_6 is set. According to the Fundamental Lemma of Game Playing [14],

$$\Pr[\mathrm{G}_{12}] - \Pr[\mathrm{G}_{13}] \leq \Pr[\mathsf{bad}_6^{\mathrm{G}_{13}}].$$

Games G_{12} and G_{13} can be thought of as simulating a bidirectional authenticated channel that allows the two users to exchange ME payloads. The adversary $\mathcal{F}_{\mathrm{INT}}$ is allowed to forward, mirror, drop, and replay the payloads; but it is not allowed to modify or forge them. This description roughly corresponds to the definition of EINT-security of ME with respect to supp (Fig. 10). In games G_{12}–G_{13} the oracle SEND still runs cryptographic algorithms in order to generate and output MTP-CH ciphertexts, but we will build an EINT-security adversary that simulates these additional instructions for $\mathcal{F}_{\mathrm{INT}}$. In Fig. 33 we build an adversary $\mathcal{F}_{\mathrm{EINT}}$ against the EINT-security of $\mathsf{ME}, \mathsf{supp}$ as follows. When adversary $\mathcal{F}_{\mathrm{EINT}}$ plays in game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}_{\mathrm{EINT}}}$, it simulates game G_{13} for adversary $\mathcal{F}_{\mathrm{INT}}$. Adversary $\mathcal{F}_{\mathrm{EINT}}$ wins in its own game whenever $\mathcal{F}_{\mathrm{INT}}$ sets bad_6, so we have

$$\Pr[\mathsf{bad}_6^{\mathrm{G}_{13}}] \leq \mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{F}_{\mathrm{EINT}}).$$

Observe that $\mathcal{F}_{\mathrm{EINT}}$ takes $\mathcal{I}$'s and $\mathcal{R}$'s initial ME states as input, and repeatedly calls the ME algorithms to manually update these states (as opposed to relying on its SEND and RECV oracles). This allows $\mathcal{F}_{\mathrm{EINT}}$ to correctly identify the two conditional statements inside the simulated oracle SENDSIM that require to roll back the most recent update to $st_{\mathsf{ME},u}$ and to exit the oracle with $\bot$ as output.

Adversary $\mathcal{F}_{\mathrm{INT}}$ can no longer win in game G_{13}, because the only instruction that sets the win flag in games G_0–G_{12} was removed in transition to game G_{13}. It follows that

$$\Pr[\mathrm{G}_{13}] = 0.$$

The theorem statement follows. □

F. 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-B and the proofs in Appendix E. 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* MTP-ME, MTP-HASH, MTP-MAC, MTP-KDF, ϕ_{MAC}, ϕ_{KDF}, MTP-SE *be the primitives of MTProto defined in Section IV-D. Let* $\phi_{\mathsf{SHACAL\text{-}2}}$ *be the key-derivation function defined in Appendix E2. Let* h_{256} *be the* SHA-256 *compression function, and let* H *be the corresponding function family with* $\mathsf{H.Ev} = h_{256}$, $\mathsf{H.kl} = \mathsf{H.ol} = 256$, $\mathsf{H.In} = \{0,1\}^{512}$. *Let* CH = MTP-CH[MTP-ME, MTP-HASH, MTP-MAC, MTP-KDF, $\phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}$, MTP-SE]. *Let* $\ell \in \mathbb{N}$. *Let* $\mathcal{D}_{\mathrm{IND}}$ *be any adversary against the* IND*-security of* CH, *making* $q_{\mathrm{CH}} < 2^{96}$ *queries to its* CH *oracle, each query made for messages of length* *at most $\ell \leq 2^{27}$ bits.*[21] *Then there exist adversaries $\mathcal{D}_{\mathrm{OTPRF}}^{\mathsf{SHACAL\text{-}1}}$, $\mathcal{D}_{\mathrm{LRKPRF}}$, $\mathcal{D}_{\mathrm{HRKPRF}}$, $\mathcal{D}_{\mathrm{OTPRF}}^{\mathsf{H}}$, $\mathcal{D}_{\mathrm{OTIND\$}}$ such that*

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

Corollary 1 follows from Theorem 1 together with Proposition 3, Proposition 4, Proposition 5 with Lemma 1 and Proposition 6. The two terms in Theorem 1 related to ME are zero for ME = MTP-ME when an adversary is restricted to making $q_{\mathrm{CH}} < 2^{96}$ queries. Qualitatively, Corollary 1 shows that the privacy 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) 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 $q_{\mathrm{CH}} < 2^{64}$, 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* MTP-ME, MTP-HASH, MTP-MAC, MTP-KDF, ϕ_{MAC}, ϕ_{KDF}, MTP-SE *be the primitives of MTProto defined in Section IV-D. Let $\phi_{\mathsf{SHACAL\text{-}2}}$ be the key-derivation function defined in Appendix E2. Let* SHA-256′ *be* SHA-256 *with its output truncated to the middle 128 bits. Let* CH = MTP-CH[MTP-ME, MTP-HASH, MTP-MAC, MTP-KDF, ϕ_{MAC}, ϕ_{KDF}, MTP-SE]. *Let* supp *be the support function as defined in Fig. 25. Let $\mathcal{F}_{\mathrm{INT}}$ be any adversary against the* INT*-security of* CH *with respect to* supp, *making $q_{\mathrm{SEND}} < 2^{96}$ queries to its* SEND *oracle. Then there exist adversaries $\mathcal{D}_{\mathrm{OTPRF}}$, $\mathcal{D}_{\mathrm{LRKPRF}}$, $\mathcal{F}_{\mathrm{CR}}$ such that*

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

Corollary 2 follows from Theorem 2 together with Proposition 3, Proposition 4 and Proposition 8. The term $\mathsf{Adv}_{\mathsf{MTP\text{-}ME},\mathsf{SUPP}}^{\mathsf{eint}}(\mathcal{F}_{\mathrm{EINT}})$ from Theorem 2 resolves to 0 for adversaries making $q_{\mathrm{SEND}} < 2^{96}$ queries. 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 the MAC is constructed, the result also depends on the collision resistance of truncated SHA-256. Quantitatively, the advantage is again bounded by $q_{\mathrm{SEND}} < 2^{64}$. 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.

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-A. We describe the side channels found in the implementations of several Telegram clients in Section VI-B and experimentally demonstrate the existence of a timing side channel in the desktop client in Section VI-G.

A. 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.[22] 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 F 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 (cf. [12]).

[21] The maximum message length in MTProto is 2^{27} bits.

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

B. Leaky length field

The preceding attack assumes we have a side channel that enables us to learn part of the second plaintext block. 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, we now proceed to a case-by-case analysis, showing relevant code excerpts in each case.

C. Android

The field msg_length is referred to as messageLength in this implementation. The check is performed in decryptServerResponse of Datacenter.cpp [43], which compares messageLength with another length field (see code below). If the messageLength check fails, the MAC check is still performed. The timing difference thus consists only of two conditional jumps, which would be small in practice. The length field is taken from the first four bytes of the transport protocol format and is not checked against the actual packet size, so an attacker can substitute arbitrary values. Using multiple queries with different length values could thus enable extraction of up to 32 bits of plaintext from the messageLength field.

```
if (messageLength > length - 32) {
        error = true;
} else if (paddingLength < 12 || paddingLength > 1024) {
        error = true;
}
messageLength += 32;
if (messageLength > length) {
        messageLength = length;
}
// compute messageKey [redacted due to space]
return memcmp(messageKey + 8, key, 16) == 0 && !error;
```

D. Desktop

Here the length check is performed in the method handleReceived of session_private.cpp [44], 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}, allowing those bits to be recovered with certainty after about 2^8 attempts on average.[23]

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

```
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
```

E. iOS

The field msg_length is referred to as messageDataLength here. The method _decryptIncomingTransportData of MTProto.m [45] compares the messageDataLength field with the length of the decrypted data first in a padding length check and then directly, see code below. If either check fails, it hashes the complete decrypted payload. A timing side channel arises because sometimes this countermeasure hashes fewer bytes than a genuine MAC check (the latter also hashes 32 bytes of auth_key, here effectiveAuthKey.authKey; hence one more 512-bit block will be hashed unless the length of the decrypted payload in bits modulo 512 is 184 or less[24], this condition being due to padding). If an attacker can change the value of decryptedData.length directly or by attaching additional ciphertext blocks, this could leak up to 32 bits of plaintext as in the Android client.

```
int32_t paddingLength =
        ((int32_t)decryptedData.length)
        - messageDataLength;
if (paddingLength < 12 || paddingLength > 1024) {
        __unused NSData *result = MTSha256(decryptedData);
        return nil;
}

if (messageDataLength < 0 ||
    messageDataLength > (int32_t)decryptedData.length) {
        __unused NSData *result = MTSha256(decryptedData);
        return nil;
}

int xValue = 8;
NSMutableData *msgKeyLargeData =
        [[NSMutableData alloc] init];
[msgKeyLargeData appendBytes:effectiveAuthKey.authKey.bytes
        + 88 + xValue length:32];
[msgKeyLargeData appendData:decryptedData];

NSData *msgKeyLarge = MTSha256(msgKeyLargeData);
NSData *messageKey =
        [msgKeyLarge subdataWithRange:NSMakeRange(8, 16)];

if (![messageKey isEqualToData:embeddedMessageKey])
        return nil;
```

F. Discussion

Note that all three of the above implementations are in violation of Telegram's own security guidelines [46] 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." Interestingly, TDLib [11],

[24]This condition holds for payloads of length 191 bits or less modulo 512, but interface to hash functions in OpenSSL and derived libraries only accepts inputs in multiples of bytes not bits.

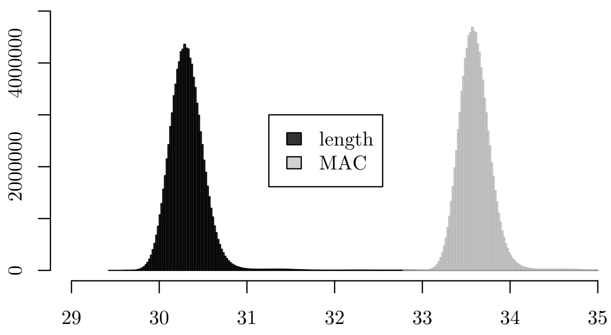
the cross-platform library for building Telegram clients, does avoid timing leaks by running the MAC check first.

Remark 1. *Recall that in Section IV-D, we define a simplified message encoding scheme which uses a constant in place of* session_id *and* server_salt*. This change would make the above attack more practical. However, the attack is enabled by a misplaced* msg_key *check and the mitigation offered by those values being secret in the implementations is accidental. Put differently, the attacks described in this section do not justify their secrecy; our proofs of security do not rely on them being secret.*

G. 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. The code can be found in Appendix H. 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. 34. This should be sufficiently large for a remote attacker to detect, even with network and other noise sources (cf. [47], where sub-microsecond timing differences were successfully resolved over a LAN).

Figure 34: Processing time of `SessionPrivate::handleReceived` in microseconds.



error type	# trials	mean	st. dev.	median
msg_length	97820883	30.330652	0.267439	30.308
MAC	96908852	33.603296	0.190341	33.589

VII. Discussion

The central result of this work is a proof that the use of symmetric encryption in Telegram's MTProto 2.0 can achieve the security of a robust bidirectional channel if small modifications are made. Thus, when those changes are made 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 are possible against current implementations, 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. We recommend that Telegram adopts these changes.

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 enables the potential for a leak that we then exploit. This "brittleness" of MTProto is of particular relevance due to the surfeit of implementations of the protocol, and security advice may not be heeded by all authors.[25]

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 swaps out the low-level cryptographic processing in all official clients by 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 $q/2^{64}$ 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 allow to tighten these bounds.

Future work: Our attack in Appendix F 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 required to rely on unstudied assumptions about the underlying primitives used in MTProto. In particular, we have to make related-key

[25]Indeed, the Telegram developers rule out length-extension attacks in [48] 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 is violated by several official Telegram clients.

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 the Telegram adopts well-studied primitives for future versions of MTProto to ease analysis and thus to increase confidence in their design; or, indeed, adopt TLS.

Telegram: While we prove security 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, which emphasises the need to study these technologies and how they serve those who rely on them.

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," 2020, https://arxiv.org/abs/2012.03141.
[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] 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.
[14] 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.
[15] C. Campbell, "Design and specification of cryptographic capabilities," *IEEE Communications Society Magazine*, vol. 16, no. 6, pp. 15–19, 1978.
[16] C. Jutla, "Attack on free-mac, sci.crypt," https://groups.google.com/forum/#!topic/sci.crypt/4bkzm_n7UGA, Sep 2000.
[17] L. R. Knudsen, "Block chaining modes of operation," 2000.
[18] 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.
[19] NIST, "FIPS 180-4: Secure Hash Standard," 2015, http://dx.doi.org/10.6028/NIST.FIPS.180-4.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] J. Jaeger and I. Stepanovs, "Optimal channel security against fine-grained state compromise: The safety of messaging," in *CRYPTO 2018, Part I*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, Heidelberg, Aug. 2018, pp. 33–62.
[25] B. Poettering and P. Rösler, "Towards bidirectional ratcheted key exchange," in *CRYPTO 2018, Part I*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, Heidelberg, Aug. 2018, pp. 3–32.
[26] P. Eugster, G. A. Marson, and B. Poettering, "A cryptographic look at multi-party channels," in *CSF 2018Computer Security Foundations Symposium*, S. Chong and S. Delaune, Eds. IEEE Computer Society Press, 2018, pp. 31–45.
[27] T. Shrimpton, "A characterization of authenticated-encryption as a form of chosen-ciphertext security," Cryptology ePrint Archive, Report 2004/272, 2004, http://eprint.iacr.org/2004/272.
[28] Telegram, "Mobile protocol: Detailed description," http://web.archive.org/web/20210126200309/https://core.telegram.org/mtproto/description, Jan 2021.
[29] ——, "Schema," https://core.telegram.org/schema, Sep 2020.
[30] ——, "TL language," https://core.telegram.org/mtproto/TL, Sep 2020.
[31] Google, "BoringSSL AES IGE implementation," https://github.com/DrKLO/Telegram/blob/d073b80063c568f31d81cc88c927b47c01a1dbf4/TMessagesProj/jni/boringssl/crypto/fipsmodule/aes/aes_ige.c, Jul 2018.
[32] Telegram, "MTProto transports," http://web.archive.org/web/20200527124125/https://core.telegram.org/mtproto/mtproto-transports, May 2020.
[33] ——, "Sequence numbers in secret chats," http://web.archive.org/web/20201031115541/https://core.telegram.org/api/end-to-end/seq_no, Jan 2021.
[34] K. Ludwig, "Trudy - Transparent TCP proxy," 2017, https://github.com/praetorian-inc/trudy.
[35] 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.
[36] ——, "Mobile protocol: Detailed description – server salt," http://web.archive.org/web/20210221134408/https://core.telegram.org/mtproto/description#server-salt, Feb 2021.

[37] ——, “Telegram Android – Datacenter.cpp,” https://github.com/DrKLO/Telegram/blob/release-7.4.0_2223/TMessagesProj/jni/tgnet/Datacenter.cpp#L1171, Feb 2021.
[38] ——, “Telegram Desktop – session_private.cpp,” https://github.com/telegramdesktop/tdesktop/blob/v2.6.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1338, Mar 2021.
[39] ——, “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.
[40] 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.
[41] 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.
[42] 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.
[43] Telegram, “Telegram Android – Datacenter.cpp,” https://github.com/DrKLO/Telegram/blob/release-7.6.0_2264/TMessagesProj/jni/tgnet/Datacenter.cpp#L1250, Apr 2021.
[44] ——, “Telegram Desktop – session_private.cpp,” https://github.com/telegramdesktop/tdesktop/blob/v2.7.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1258, Apr 2021.
[45] ——, “Telegram iOS – MTProto.m,” https://github.com/TelegramMessenger/Telegram-iOS/blob/release-7.6.2/submodules/MtProtoKit/Sources/MTProto.m#L2144, Apr 2021.
[46] ——, “Security guidelines for client developers,” http://web.archive.org/web/20210203134436/https://core.telegram.org/mtproto/security_guidelines#mtproto-encrypted-messages, Feb 2021.
[47] 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.
[48] 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.
[49] E. Rescorla, H. Tschofenig, and N. Modadugu, “The Datagram Transport Layer Security (DTLS) protocol version 1.3,” https://datatracker.ietf.org/doc/draft-ietf-tls-dtls13/, Feb. 2021, draft Version 41.
[50] J. Iyengar and M. Thomson, “QUIC: A UDP-based multiplexed and secure transport,” https://datatracker.ietf.org/doc/draft-ietf-quic-transport/, Mar. 2021, draft Version 34.
[51] M. Bellare, R. Canetti, and H. Krawczyk, “Pseudorandom functions revisited: The cascade construction and its concrete security,” in *37th FOCS*. IEEE Computer Society Press, Oct. 1996, pp. 514–523.
[52] M. Bellare, J. Kilian, and P. Rogaway, “The security of the cipher block chaining message authentication code,” *Journal of Computer and System Sciences*, vol. 61, no. 3, pp. 362–399, 2000.
[53] M. Bellare, K. Pietrzak, and P. Rogaway, “Improved security analyses for CBC MACs,” in *CRYPTO 2005*, ser. LNCS, V. Shoup, Ed., vol. 3621. Springer, Heidelberg, Aug. 2005, pp. 527–545.
[54] 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.
[55] 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.
[56] Telegram, “Telegram MTProto – creating an authorization key,” http://web.archive.org/web/20210112084225/https://core.telegram.org/mtproto/auth_key, Jan 2021.
[57] R. Merget, M. Brinkmann, N. Aviram, J. Somorovsky, J. Mittmann, and J. Schwenk, “Raccoon Attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E),” https://raccoon-attack.com/RacoonAttack.pdf, Sep. 2020, accessed 11 September 2020.
[58] H. Shacham and A. Boldyreva, Eds., *CRYPTO 2018, Part I*, ser. LNCS, vol. 10991. Springer, Heidelberg, Aug. 2018.

Appendix

A. Correctness of the support function

We present two correctness properties of a support function.

1) Order correctness of the support function: The game in Fig. 35 captures the fact that the support function should return m if m and label appear together in the transcript of the sender and m was delivered in-order. To express this, we define a function getPairs and we use $\preccurlyeq$ to denote when a list is a prefix of another list. The advantage of $\mathcal{F}$ in breaking the ORD-security of supp is defined as $\mathsf{Adv}^{\mathsf{ord}}_{\mathsf{supp}}(\mathcal{F}) = \Pr\left[\, \mathrm{G}^{\mathsf{ord}}_{\mathsf{supp},\mathcal{F}} \,\right]$. We have $\mathsf{Adv}^{\mathsf{ord}}_{\mathsf{SUPP}}(\mathcal{F}) = 0$ for SUPP in Fig. 25.

Note that the game requires the submitted labels to be unique. For instance, if that was not so, $\mathcal{F}$ could win trivially against a supp which rejects replays by querying SEND with the same label twice. Thus when supp is used in conjunction with some labelling scheme, it is only "well-behaved" as long as the labels are unique. (This becomes apparent in Section V-B2, where we match a supp to a message encoding scheme which can only produce a fixed number of unique payloads.)

```
Game $\mathrm{G}^{\mathsf{ord}}_{\mathsf{supp},\mathcal{F}}$
$\mathsf{win} \leftarrow \mathsf{false}$ ; $X \leftarrow \emptyset$ ; $\mathcal{F}^{\textsc{Send},\textsc{Recv}}$ ; Return win
$\textsc{Send}(u, m, \mathsf{label}, aux)$
If $\mathsf{label} \in X$ then return $\bot$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, \mathsf{label}, aux)$ ; $X \leftarrow X \cup \{\mathsf{label}\}$ ; Return $\bot$
$\textsc{Recv}(u, m, \mathsf{label}, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, \mathsf{label}, aux)$
$\mathsf{inOrder} \leftarrow \mathsf{getPairs}(\mathsf{recv}, \mathsf{tr}_u) \preccurlyeq \mathsf{getPairs}(\mathsf{sent}, \mathsf{tr}_{\overline{u}})$
If $\mathsf{inOrder} \wedge m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
Return $\bot$
$\mathsf{getPairs}(\mathsf{op}, \mathsf{tr}_u)$
$\mathsf{pairs} \leftarrow [\,]$
For $(\mathsf{op}, m, \mathsf{label}, aux) \in \mathsf{tr}_u$ do $\mathsf{pairs} \leftarrow \mathsf{pairs} \,\|\, (m, \mathsf{label})$
Return pairs
```

Figure 35: Game defining the order correctness of supp.

2) Integrity of the support function: The game in Fig. 36 captures the fact that the support function should return $\bot$ if the given label does not appear in the transcript of the sender. The advantage of $\mathcal{F}$ in breaking the SINT-security of supp is defined as $\mathsf{Adv}^{\mathsf{sint}}_{\mathsf{supp}}(\mathcal{F}) = \Pr\left[\, \mathrm{G}^{\mathsf{sint}}_{\mathsf{supp},\mathcal{F}} \,\right]$. We have $\mathsf{Adv}^{\mathsf{sint}}_{\mathsf{SUPP}}(\mathcal{F}) = 0$ for SUPP in Fig. 25.

```
Game $\mathrm{G}^{\mathsf{sint}}_{\mathsf{supp},\mathcal{F}}$
$(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux) \leftarrow\!\!\$\ \mathcal{F}$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$
$\mathsf{forge} \leftarrow (\nexists m', aux' \colon (\mathsf{sent}, m', \mathsf{label}, aux') \in \mathsf{tr}_{\overline{u}})$
Return $\mathsf{forge} \wedge (m^* \neq \bot)$
```

Figure 36: Game defining the integrity of supp.

B. Combined security for bidirectional channels

Consider the authenticated encryption game for combining privacy and integrity in Fig. 37. The advantage of $\mathcal{A}$ in breaking the AE-security of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) = 2 \cdot \Pr\left[\, \mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}} \,\right] - 1$. The CH oracle copies the SEND oracle of $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ (Fig. 8), but amends it for the left-or-right setting. If RECV is queried on an honestly produced and forwarded ciphertext which encrypts a challenge message (i.e. for a CH call with $m_0 \neq m_1$), then the adversary $\mathcal{A}$ is not allowed to learn its decryption, and otherwise (i.e. if CH was called with $m_0 = m_1$) the adversary knows the encrypted message without the help of a decryption oracle, so RECV returns $\bot_0$. If $\mathcal{A}$ calls RECV on a forged ciphertext that decrypts correctly, then its output depends on the challenge bit b: RECV returns the decryption of the forged ciphertext if $b = 1$, and it returns $\bot_1$ otherwise. This ensures that breaking the integrity of CH allows $\mathcal{A}$ to learn the challenge bit in $\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$. In the following two propositions, we show that this combined notion is equivalent to the individual games together.

```
Game $\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$
$b \leftarrow\!\!\$\ \{0,1\}$ ; $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{CH.Init}()$
$b' \leftarrow\!\!\$\ \mathcal{D}^{\textsc{Ch},\textsc{Recv}}$ ; Return $b' = b$
$\textsc{Ch}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$(st_u, c) \leftarrow \mathsf{CH.Send}(st_u, m_b, aux; r)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m_b, c, aux)$ ; Return $c$
$\textsc{Recv}(u, c, aux)$
$(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
If $m \neq m^*$ then
    If $b = 1$ then return $m$ else return $\bot_1$
Return $\bot_0$
```

Figure 37: Game defining authenticated encryption security of channel CH.

Proposition 1. *Let* CH *be a channel. Let* supp *be a support function. Let $\mathcal{A}$ be an adversary against the* AE*-security of* CH *with respect to* supp*. Then we can build an adversary $\mathcal{F}$ against the* INT*-security of* CH *with respect to* supp*, and an adversary $\mathcal{D}$ against the* IND*-security of* CH *such that*

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}) + \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}).$$

Proof. We rewrite the $\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathcal{A},\mathsf{supp}}$ game as game G_0 in Fig. 38, so $\Pr[\, \mathrm{G}_0 \,] = \frac{1}{2}\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) - \frac{1}{2}$ by definition. We modify this game to obtain G_1 by removing the one before last line, and denote this part of the code by setting $\mathsf{bad} \leftarrow \mathsf{true}$.

Construct the adversaries $\mathcal{F}$ for $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ and $\mathcal{D}$ for $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ (both games in Fig. 8) as shown in Fig. 39. Consider $\mathcal{D}$ first. By inspection, it simulates the oracles of G_1 perfectly for $\mathcal{A}$, so we can write $\Pr[\, \mathrm{G}_1 \,] = \Pr\left[\, \mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}} \,\right] = \frac{1}{2}\mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) - \frac{1}{2}$.

```
Games G_0–G_1
$b \leftarrow_{\$} \{0,1\}$ ; $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow_{\$}$ CH.Init() ; $b' \leftarrow_{\$} \mathcal{A}^{\textsc{Ch},\textsc{Recv}}$
Return $b' = b$

Ch($u, m_0, m_1, aux, r$)
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_u, c) \leftarrow$ CH.Send($st_u, m_b, aux; r$)
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m_b, c, aux)$
Return $c$

Recv($u, c, aux$)
$(st_u, m) \leftarrow$ CH.Recv($st_u, c, aux$)
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
If $m \neq m^*$ then
    bad $\leftarrow$ true
    If $b = 1$ then return $m$ else return $\perp_1$        // G_0
Return $\perp_0$
```

Figure 38: Games G_0–G_1 for proof of Proposition 1.

Second, according to the Fundamental Lemma of Game Playing [14] we have

$$\Pr[\,G_0\,] - \Pr[\,G_1\,] \leq \Pr[\,\mathsf{bad}\,],$$

where $\Pr[\,\mathsf{bad}\,]$ denotes the probability of setting the bad flag in games G_0–G_1. Finally, consider $\mathcal{F}$, which simulates G_0 for $\mathcal{A}$. If bad = true, then the $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ game sets win = true, hence $\Pr[\,\mathsf{bad}\,] \leq \Pr\left[\,G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}\,\right] = \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F})$. Taken together, we can write

$$\frac{1}{2}\left(\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) - \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D})\right) \leq \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}),$$

which concludes the proof.

```
Adversary $\mathcal{F}^{\textsc{Send},\textsc{Recv}}$
$b \leftarrow_{\$} \{0,1\}$
$b' \leftarrow_{\$} \mathcal{A}^{\textsc{ChSim},\textsc{RecvSim}}$

ChSim($u, m_0, m_1, aux, r$)
If $|m_0| \neq |m_1|$ then return $\perp$
$c \leftarrow$ Send($u, m_b, aux, r$)
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m_b, c, aux)$
Return $c$

RecvSim($u, c, aux$)
$m \leftarrow$ Recv($u, c, aux$)
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
If $m \neq m^*$ then
    If $b = 1$ then return $m$
    Else return $\perp_1$
Return $\perp_0$
```

```
Adversary $\mathcal{D}^{\textsc{Ch},\textsc{Recv}}$
$b' \leftarrow_{\$} \mathcal{A}^{\textsc{ChSim},\textsc{RecvSim}}$
Return $b'$

ChSim($u, m_0, m_1, aux, r$)
$c \leftarrow$ Ch($u, m_0, m_1, aux, r$)
Return $c$

RecvSim($u, c, aux$)
err $\leftarrow$ Recv($u, c, aux$)
Return $\perp_0$
```

Figure 39: Adversaries $\mathcal{F}$, $\mathcal{D}$ for proof of Proposition 1.

□

Proposition 2. *Let* CH *be a channel. Let* supp *be a support function. Let $\mathcal{F}$ be an adversary against the* INT-*security of* CH *with respect to* supp, *and let $\mathcal{D}$ be an adversary against the* IND-*security of* CH. *Then we can build adversaries $\mathcal{A}_{\mathrm{INT}}$ and $\mathcal{A}_{\mathrm{IND}}$ against the* AE-*security of* CH *with respect to* supp *such that*

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) \geq \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}) \text{ and}$$
$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{IND}}) \geq \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}).$$

Proof. Let $\mathcal{F}$ be the adversary in $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ (Fig. 8). Build the adversary $\mathcal{A}_{\mathrm{INT}}$ as shown in Fig. 40. Note that it makes use of the **abort**(x) instruction, which allows it to end the simulation for $\mathcal{F}$ and return x to its own game. The reason for using this instruction is that $\mathcal{A}_{\mathrm{INT}}$ could only simulate Recv perfectly if its challenge bit $b = 1$, because then it can return the m value that $\mathcal{F}$ is expecting. If $b = 0$, $\mathcal{A}_{\mathrm{INT}}$ does not get this value from Recv, but it can win its game so the simulation can end.

We let $\mathcal{A}_{\mathrm{INT}}$ return 0 by default if one of the abort conditions is not triggered during the run of $\mathcal{F}$. By this construction, if $b = 0$ then $\mathcal{A}_{\mathrm{INT}}$ never returns 1. If $b = 1$, then $\mathcal{A}_{\mathrm{INT}}$ wins if $\mathcal{F}$ sets win = true during its run, because $m \neq m^*$ corresponds to RecvSim returning err $\neq \perp_0$. Then we can write

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{INT}}) = \Pr[\,b' = 1 \mid b = 1\,] - \Pr[\,b' = 1 \mid b = 0\,]$$
$$\geq \Pr\left[\,G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}\,\right] - 0 = \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}).$$

Let $\mathcal{D}$ be the adversary in $G^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ (Fig. 8). Build the adversary $\mathcal{A}_{\mathrm{IND}}$ as shown in Fig. 40. Both simulated oracles run the same code that $\mathcal{D}$ would be expecting, and the additional processing with transcripts and the support function does not affect the state of the channel or what is returned. $\mathcal{A}_{\mathrm{IND}}$ could parse the err in RecvSim, but it is not necessary. If $\mathcal{D}$ returns the correct challenge bit, then $\mathcal{A}_{\mathrm{IND}}$ does, so we can write

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{IND}}) = \Pr\left[\,G^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{IND}}}\,\right]$$
$$\geq \Pr\left[\,G^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}\,\right] = \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}).$$

```
Adversary $\mathcal{A}_{\mathrm{INT}}^{\textsc{Ch},\textsc{Recv}}$
$\mathcal{F}^{\textsc{SendSim},\textsc{RecvSim}}$
Return 0

SendSim($u, m, aux, r$)
$c \leftarrow$ Ch($u, m, m, aux, r$)
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$
Return $c$

RecvSim($u, c, aux$)
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
err $\leftarrow$ Recv($u, c, aux$)
If err $\neq \perp_0$ then
    If err $= \perp_1$ then abort(0)
    Else abort(1)
$m \leftarrow m^*$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
Return $m$
```

```
Adversary $\mathcal{A}_{\mathrm{IND}}^{\textsc{Ch},\textsc{Recv}}$
$b' \leftarrow_{\$} \mathcal{D}^{\textsc{ChSim},\textsc{RecvSim}}$
Return $b'$

ChSim($u, m_0, m_1, aux, r$)
$c \leftarrow$ Ch($u, m, aux, r$)
Return $c$

RecvSim($u, c, aux$)
err $\leftarrow$ Recv($u, c, aux$)
Return $\perp$
```

Figure 40: Adversaries $\mathcal{A}_{\mathrm{INT}}$, $\mathcal{A}_{\mathrm{IND}}$ for proof of Proposition 2.

□

```
ME.Init()
$N_{\mathsf{sent}} \leftarrow 0$ ; session_id $\leftarrow 0$ ; last_sent_msg_id $\leftarrow 0$
S $\leftarrow_{\$}$ GenerateSalts() ; M $\leftarrow \emptyset$
For $u \in \{\mathcal{I}, \mathcal{R}\}$ do
    $st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}$, session_id, last_sent_msg_id, S, M$)$
Return $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
```

```
ME.Encode($st_{\mathsf{ME},u}, m, aux$)
$(N_{\mathsf{sent}}$, session_id, last_sent_msg_id, S, M$) \leftarrow st_{\mathsf{ME},u}$
If $u = \mathcal{I}$ and $N_{\mathsf{sent}} = 0$ then session_id $\leftarrow_{\$} \{0,1\}^{64}$
server_salt $\leftarrow$ GetSalt(S, $aux$)
msg_id $\leftarrow$ GetMsgId($u, aux$, last_sent_msg_id)
msg_seq_no $\leftarrow \langle 2 \cdot N_{\mathsf{sent}} + 1 \rangle_{32}$
msg_length $\leftarrow \langle |m|/8 \rangle_{32}$
padding $\leftarrow_{\$}$ GenPadding($|m|$)
$p_0 \leftarrow$ server_salt || session_id
$p_1 \leftarrow$ msg_id || msg_seq_no || msg_length
$p_2 \leftarrow m$ || padding
$p \leftarrow p_0 \| p_1 \| p_2$
$N_{\mathsf{sent}} \leftarrow (N_{\mathsf{sent}} + 1) \mod 2^{32}$
last_sent_msg_id $\leftarrow$ msg_id
$st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}$, session_id, last_sent_msg_id, S, M$)$
Return $(st_{\mathsf{ME},u}, p)$

GetMsgId($u, aux$, last_sent_msg_id)
msg_id $\leftarrow aux \ll 32$
If msg_id $\leq$ last_sent_msg_id then
    msg_id $\leftarrow$ last_sent_msg_id + 1
$i_{\mathcal{I}} \leftarrow 0$ ; $i_{\mathcal{R}} \leftarrow 1$ ; $t \leftarrow (i_u -$ msg_id$) \bmod 4$
Return $\langle$msg_id $+ t\rangle_{64}$

GenPadding($\ell$)
$\ell' \leftarrow 128 - \ell \bmod 128$ ; $bn \leftarrow_{\$} \{2, 3, \cdots, 63\}$
padding $\leftarrow_{\$} \{0,1\}^{\ell' + bn * 128}$
Return padding
```

```
ME.Decode($st_{\mathsf{ME},u}, p, aux'$)
$(N_{\mathsf{sent}}$, session_id, last_sent_msg_id, S, M$) \leftarrow st_{\mathsf{ME},u}$
server_salt $\leftarrow p[0:64]$ ; session_id' $\leftarrow p[64:128]$
msg_id $\leftarrow p[128:192]$ ; msg_seq_no $\leftarrow p[192:224]$
msg_length $\leftarrow p[224:256]$ ; $\ell \leftarrow |p| - 256$
If $u = \mathcal{R} \wedge$ server_salt $\notin$ ValidSalts(S, $aux'$) then
    Return $(st_{\mathsf{ME},u}, \perp)$
If $u = \mathcal{R} \wedge N_{\mathsf{recv}} = 0$ then session_id $\leftarrow$ session_id'
Else if session_id' $\neq$ session_id then return $(st_{\mathsf{ME},u}, \perp)$
If $\neg(aux' - t_p \leq$ (msg_id $\gg 32$) $\leq aux' + t_f) \vee$
    msg_id $\in$ M.IDs $\vee$ msg_id $<$ min(M.IDs) then
        Return $(st_{\mathsf{ME},u}, \perp)$
If $u = \mathcal{R} \wedge \exists (i, s) \in$ M :
    (msg_seq_no $\leq s \wedge$ msg_id $> i) \vee$
    (msg_seq_no $\geq s \wedge$ msg_id $< i$) then
        Return $(st_{\mathsf{ME},u}, \perp)$
If ($u = \mathcal{I} \wedge$ msg_id mod $4 \neq 1) \vee$
    ($u = \mathcal{R} \wedge$ msg_id mod $4 \neq 0$) then
        Return $(st_{\mathsf{ME},u}, \perp)$
padding_length $\leftarrow \ell/8 -$ msg_length
If $\neg(0 <$ msg_length $\leq \ell/8) \vee$
    $\neg(12 \leq$ padding_length $\leq 1024)$ then
        Return $(st_{\mathsf{ME},u}, \perp)$
$m \leftarrow p[256 : 256 +$ msg_length $\cdot 8]$
M $\leftarrow$ M.add(msg_id, msg_seq_no)
$st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}$, session_id, last_sent_msg_id, S, M$)$
Return $(st_{\mathsf{ME},u}, m)$
```

Figure 41: Construction of MTProto's message encoding scheme ME where aux, aux' are 32-bit timestamps. Table S contains 64-bit server_salt values, each associated to some time period; algorithm GenerateSalts generates this table; algorithms GetSalt and ValidSalts are used to choose and validate salt values depending on the current timestamp. M is a fixed-size set that stores (msg_id, msg_seq_no) for each of recently received messages; when M reaches its maximum size, the entries with the smallest msg_id are removed first. M.IDs is the set of msg_ids in M. Time constants t_p and t_f determine the range of timestamps (from the past or future) that should be accepted; these constants are in the same encoding as aux, aux'. We assume all strings are byte-aligned.

C. Causality preservation

Recall that in Telegram as currently implemented, an adversary on the network can reorder messages, e.g. changing the role of pizza and crime in the sequence of messages transmitted from a single client ("I say yes to", "all the pizza", "I say no to", "all the crimes"). This sort of reordering may be particularly devastating when a protocol is used to transport control messages – as is the case for MTProto which carries control messages both for Telegram directly and to third-party bots – but we know of no such exploitable example.

Such reordering attacks are not possible against e.g. Signal or MTProto's closest "competitor" TLS. TLS-like protocols over UDP such as DTLS [49] or QUIC [50] either leave it to the application to handle packet reordering (DTLS, i.e. they are possible against DTLS itself) or have built-in mechanisms to handle these (QUIC, i.e. they are not possible against QUIC itself). As discussed in the main text, in the case of Telegram higher levels of the application do not prevent packet reordering.

The prevention of reordering attacks in one direction can be strengthened to also cover the order of packets flowing in both directions. This is sometimes referred to as *causality preservation* in the literature [26], and is generally considered to be more complex to achieve. In particular, the following is possible in both Telegram and e.g. Signal. Alice sends a message "Let's commit all the crimes". Then, simultaneously both Alice and Bob send a message. Alice: "Just kidding"; Bob: "Okay". Depending on the order in which these messages arrive, the transcript on either side might be (Alice: "Let's commit all the crimes", Alice:"Just kidding", Bob: "Okay") or (Alice: "Let's commit all the crimes", Bob: "Okay", Alice:"Just kidding"). That is, the transcript will have Bob acknowledging a joke or criminal activity.

D. Message encoding scheme of MTProto

Figure 41 defines an approximation of the current ME construction in MTProto, where header fields have encodings of fixed size as in Section IV-A. Salt generation is modelled as an abstract call within ME.Init. We omit modelling containers or acknowledgement messages, though they are not properly separated from the main protocol logic in implementations. We stress that because implementations of MTProto differ even in protocol details, it would be impossible to define a single ME scheme, so Fig. 41 shows an approximation. For instance, the GenPadding function in Android has randomised padding length which is at most 240 bytes, whereas the same function on desktop does not randomise the padding length. Different client/server behaviour is captured by $u = \mathcal{I}$ representing the client and $u = \mathcal{R}$ representing the server, and we assume that $\mathcal{I}$ always sends the first message.

E. Games and proofs for standard primitives

Here we give the reductions referred to in Sections V-A and V-B.

1) OTWIND of MTP-HASH: Proposition 3 shows that MTP-HASH is a one-time weak indistinguishable function (Fig. 20) if SHACAL-1 is a one-time pseudorandom function (Fig. 2). At a high level, our proof uses that SHACAL-1 is called with random independent keys and thus produces random outputs if it is a PRF. The final SHACAL-1 call on a known constant (the padding) cannot improve the distinguishing advantage; this is a special case of the processing inequality.

Proposition 3. *Let $\mathcal{D}_{\text{OTWIND}}$ be an adversary against the* OTWIND*-security of* MTP-HASH*. Then we can build an adversary $\mathcal{D}_{\text{OTPRF}}$ against the* OTPRF*-security of* SHACAL-1 *such that*

$$\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\text{OTWIND}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\text{OTPRF}}).$$

Proof. Recall that SHA-1 operates on 512-bit input blocks. Padding is appended at the end of the last input block. If the message size is already a multiple of the block size (as it is in MTP-HASH), a new input block is added, which we denote by x_p for a message of length 2048. Define P as the public function $P(H) := h_{160}(H, x_p)$, i.e. the last iteration of SHA-1 over the padding block.

Let $\mathcal{D}_{\text{OTWIND}}$ be an adversary in the $G^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH},\mathcal{D}}$ game (Fig. 20). Using the definition of SHA-1, we first rewrite the game in a functionally equivalent way as G_0 in Fig. 42. The two last calls to the compression function h take as input two blocks from the secret input of MTP-HASH.Ev, i.e. $hk[32:1056]$, so they can be rewritten to use two invocations of SHACAL-1.Ev with random and independent keys. We then construct game G_1 in which these calls are replaced with a random value. In this game, $\mathcal{D}_{\text{OTWIND}}$ is given auth_key_id $= P(H_3 \dot{+} r_1)[96:160]$ for a random value r_1 which does not depend on the challenge bit b, so it cannot have an advantage in winning the game.

```
Games G_0–G_1
$b \leftarrow_\$ \{0,1\}$ ; $hk \leftarrow_\$ \{0,1\}^{\mathsf{HASH.kl}}$
$x_0 \leftarrow_\$ \mathsf{HASH.In}$ ; $x_1 \leftarrow_\$ \mathsf{HASH.In}$
$r_0 \leftarrow_\$ \{0,1\}^{\mathsf{SHACAL\text{-}1.ol}}$ ; $r_1 \leftarrow_\$ \{0,1\}^{\mathsf{SHACAL\text{-}1.ol}}$
$H_1 \leftarrow h_{160}(\mathsf{IV}_{160}, x_b[0:512])$
$H_2 \leftarrow h_{160}(H_1, x_b[512:672] \| hk[0:32] \| x_b[672:992])$
$H_3 \leftarrow H_2 \dot{+} \mathsf{SHACAL\text{-}1.Ev}(hk[32:544], H_2)$        // G_0
$H_4 \leftarrow H_3 \dot{+} \mathsf{SHACAL\text{-}1.Ev}(hk[544:1056], H_3)$      // G_0
$H_3 \leftarrow H_2 \dot{+} r_0$                                                // G_1
$H_4 \leftarrow H_3 \dot{+} r_1$                                                // G_1
auth_key_id $\leftarrow P(H_4)[96:160]$
$b' \leftarrow_\$ \mathcal{D}_{\text{OTWIND}}(x_0, x_1, \text{auth\_key\_id})$ ; Return $b' = b$
```

Figure 42: Games for the proof of Proposition 3. In $G_0 = G^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH},\mathcal{D}_{\text{OTWIND}}}$, auth_key_id $\leftarrow$ MTP-HASH.Ev(hk, x_b) is expanded and the last two h_{160} calls are expressed using SHACAL-1 (in gray). In G_1, changes from G_0 are in green.

We construct the adversary $\mathcal{D}_{\text{OTPRF}}$ for $G^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1},\mathcal{D}}$ as shown in Fig. 43 so that $\Pr[G_0] - \Pr[G_1] = \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\text{OTPRF}})$. Let d be the challenge bit in $G^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1},\mathcal{D}_{\text{OTPRF}}}$ and d' be the output of the adversary in that game. Then, if $d = 1$ in $G^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1},\mathcal{D}_{\text{OTPRF}}}$, calls to RoR made by $\mathcal{D}_{\text{OTPRF}}$ are SHACAL-1 invocations with random keys. If $d = 0$, calls to RoR both draw a random value and so $y = P(H)$ for some $H \leftarrow_\$ \{0,1\}^{\mathsf{SHACAL\text{-}1.ol}}$.

```
Adversary $\mathcal{D}^{\text{RoR}}_{\text{OTPRF}}$
$b \leftarrow_\$ \{0,1\}$ ; $hk' \leftarrow_\$ \{0,1\}^{32}$
$x_0 \leftarrow_\$ \mathsf{MTP\text{-}HASH.In}$ ; $x_1 \leftarrow_\$ \mathsf{MTP\text{-}HASH.In}$
$H_1 \leftarrow h_{160}(\mathsf{IV}_{160}, x_b[0:512])$
$H_2 \leftarrow h_{160}(H_1, x_b[512:672] \| hk' \| x_b[672:992])$
$H_3 \leftarrow H_2 \dot{+} \text{RoR}(H_2)$
$H_4 \leftarrow H_3 \dot{+} \text{RoR}(H_3)$
auth_key_id $\leftarrow P(H_4)[96:160]$
$b' \leftarrow_\$ \mathcal{D}_{\text{OTWIND}}(x_0, x_1, \text{auth\_key\_id})$
If $b' = b$ then return 1 else return 0
```

Figure 43: Adversary for the proof of Proposition 3.

We can write:

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\text{OTPRF}}) \\
&\quad = \Pr[d' = 1 \mid d = 1] - \Pr[d' = 1 \mid d = 0] \\
&\quad = \Pr[G_0] - \Pr[G_1] \\
&\quad = \frac{1}{2} \cdot \left(\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\text{OTWIND}}) + 1\right) - \frac{1}{2} \\
&\quad = \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\text{OTWIND}}).
\end{aligned}$$

The inequality follows. □

2) RKPRF of MTP-KDF: What complicates the construction of MTP-KDF, when expressed using calls to SHACAL-2, is the fact that a part of the key input to SHACAL-2 is a known constant (the SHA-256 padding) and a part of it is a variable input that can be manipulated by the adversary (the msg_key input to the MTP-KDF). This means that we can only prove security under a very strong assumption: in Proposition 4, we show that KDF = MTP-KDF is a PRF under related-key attacks (Fig. 21) restricted to ϕ_{KDF} (Fig. 18) if SHACAL-2 is a leakage-resilient PRF under related-key attacks (Fig. 44) restricted to ϕ_{KDF} composed with $\phi_{\mathsf{SHACAL\text{-}2}}$ (Fig. 45). 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 $\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) = 2 \cdot \Pr\left[\, \mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi,\mathcal{D}} \,\right] - 1$. At a high level, our proof proceeds analogously to the proof in Appendix E1.

```
Game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi,\mathcal{D}}$
$b \leftarrow\$ \{0,1\}$
$kk \leftarrow\$ \{0,1\}^{672}$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR}}$
Return $b' = b$

$\mathrm{RoR}(u, i, \mathsf{msg\_key})$   // $|\mathsf{msg\_key}| = 128$
$(sk_0, sk_1) \leftarrow \phi_{\mathsf{SHACAL\text{-}2}}(kk_u, \mathsf{msg\_key})$
$y_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(sk_i, \mathsf{IV}_{256})$
If $\mathsf{T}[u, i, \mathsf{msg\_key}] = \perp$ then
    $\mathsf{T}[u, i, \mathsf{msg\_key}] \leftarrow\$ \{0,1\}^{\ell}$
$y_0 \leftarrow \mathsf{T}[u, i, \mathsf{msg\_key}]$
Return $y_b$
```

Figure 44: Leakage-resilient PRF under related-key attacks with constant input IV for SHACAL-2, where $i \in \{0,1\}$ and msg_key is the chosen-key part. We abbreviate SHACAL-2.ol by ℓ and ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ by ϕ.

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

Figure 45: Related-key derivation function $\phi_{\mathsf{SHACAL\text{-}2}} \colon \mathsf{KDF.Keys} \times \mathsf{KDF.In} \to \mathsf{SHACAL\text{-}2.Keys} \times \mathsf{SHACAL\text{-}2.Keys}$.

Proposition 4. *Let $\mathcal{D}_{\mathrm{RKPRF}}$ be an adversary against the* RKPRF*-security of* KDF = MTP-KDF *under the related-key-deriving function ϕ_{KDF} from Fig. 18. Then we can build an adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ against the* LRKPRF*-security of* SHACAL-2 *under ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ (abbrev. with ϕ) such that*

$$\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) \le 2 \cdot \mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi}(\mathcal{D}_{\mathrm{LRKPRF}}).$$

Proof. Let $\mathcal{D}_{\mathrm{RKPRF}}$ be an adversary in the $\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}$ game (Fig. 21) against KDF. We first rewrite the game in a functionally equivalent way as G_0 in Fig. 46 using the definition of SHA-256 which is called twice on related input blocks, with padding. Then G_1 expresses this in terms of the related-key derivation function $\phi_{\mathsf{SHACAL\text{-}2}}$ (Fig. 45) and calls to SHACAL-2 on fixed input; game G_1 is equivalent to game G_0. Finally, the game G_2 replaces these calls with random values that are independent of the challenge bit, so $\mathcal{D}_{\mathrm{RKPRF}}$ can have no advantage better than guessing in this game.

We construct the adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ for $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi,\mathcal{D}}$ as shown in Fig. 47 so that $\Pr[\,\mathrm{G}_1\,] - \Pr[\,\mathrm{G}_2\,] = \mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi}(\mathcal{D}_{\mathrm{LRKPRF}})$. Let d be the challenge bit in $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi,\mathcal{D}_{\mathrm{LRKPRF}}}$ and d' be the output of the adversary in that game. Then, if $d = 1$ calls to RoR made by $\mathcal{D}_{\mathrm{LRKPRF}}$ are

```
Games $\mathrm{G}_0$–$\mathrm{G}_2$
$b \leftarrow\$ \{0,1\}$ ; $kk \leftarrow\$ \{0,1\}^{672}$ ; $(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR}}_{\mathrm{RKPRF}}$ ; Return $b' = b$

$\mathrm{RoR}(u, \mathsf{msg\_key})$
$(kk_0, kk_1) \leftarrow kk_u$                                                  // $\mathrm{G}_0$
$k_1^{(0)} \leftarrow h_{256}(\mathsf{IV}_{256}, \mathsf{SHA\text{-}pad}(\mathsf{msg\_key} \,\|\, kk_0))$        // $\mathrm{G}_0$
$k_1^{(1)} \leftarrow h_{256}(\mathsf{IV}_{256}, \mathsf{SHA\text{-}pad}(kk_1 \,\|\, \mathsf{msg\_key}))$        // $\mathrm{G}_0$
$(sk_0, sk_1) \leftarrow \phi_{\mathsf{SHACAL\text{-}2}}(kk_u, \mathsf{msg\_key})$                 // $\mathrm{G}_1$–$\mathrm{G}_2$
$k_1^{(0)} \leftarrow \mathsf{IV}_{256} \hat{+} \mathsf{SHACAL\text{-}2.Ev}(sk_0, \mathsf{IV}_{256})$               // $\mathrm{G}_1$
$k_1^{(1)} \leftarrow \mathsf{IV}_{256} \hat{+} \mathsf{SHACAL\text{-}2.Ev}(sk_1, \mathsf{IV}_{256})$               // $\mathrm{G}_1$
$r_0 \leftarrow\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$                                    // $\mathrm{G}_2$
$r_1 \leftarrow\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$                                    // $\mathrm{G}_2$
$k_1^{(0)} \leftarrow \mathsf{IV}_{256} \hat{+} r_0$ ; $k_1^{(1)} \leftarrow \mathsf{IV}_{256} \hat{+} r_1$                   // $\mathrm{G}_2$
$k_1 \leftarrow k_1^{(0)} \,\|\, k_1^{(1)}$
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 46: Games for the proof of Proposition 4. In $\mathrm{G}_0 = \mathrm{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\mathrm{RKPRF}}}$, $k_1 \leftarrow \mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$ is expanded. In G_1, calls to h_{256} are expressed using SHACAL-2 (shown in gray). In G_2, changes from G_1 are in green.

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathrm{LRKPRF}}$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{RoRSim}}_{\mathrm{RKPRF}}$
If $b' = b$ then return 1
Else return 0

$\mathrm{RoRSim}(u, \mathsf{msg\_key})$
$k_1^{(0)} \leftarrow \mathsf{IV}_{256} \hat{+} \mathrm{RoR}(u, 0, \mathsf{msg\_key})$
$k_1^{(1)} \leftarrow \mathsf{IV}_{256} \hat{+} \mathrm{RoR}(u, 1, \mathsf{msg\_key})$
$k_1 \leftarrow k^{(0)} \,\|\, k^{(1)}$
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 47: Adversary for the proof of Proposition 4.

SHACAL-2 invocations with related and partially-chosen keys. If $d = 0$, calls to RoR both draw a random value and so the output k is random and independent of the challenge bit. We write:

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi}(\mathcal{D}_{\mathrm{LRKPRF}}) \\
&\quad = \Pr[\, d' = 1 \mid d = 1 \,] - \Pr[\, d' = 1 \mid d = 0 \,] \\
&\quad = \Pr[\, \mathrm{G}_0 \,] - \Pr[\, \mathrm{G}_2 \,] \\
&\quad = \frac{1}{2}\left(\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) + 1\right) - \frac{1}{2} \\
&\quad = \frac{1}{2}\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}).
\end{aligned}$$

The inequality follows. □

3) UPRKPRF of MTP-MAC: We reduce UPRKPRF of MAC to the security of the Merkle-Damgård construction and SHACAL-2. To this end, we first prove a result about the Merkle-Damgård transform that is analogous to the basic cascade PRF security proved in [51], except that we only prove *one-time* security and hence we do not require prefix-free inputs.

Lemma 1. *Let h_{256} be the* SHA-256 *compression function, and let* H *be the corresponding function family with* $\mathsf{H.Ev} = h_{256}$, $\mathsf{H.kl} = \mathsf{H.ol} = 256$, $\mathsf{H.In} = \{0,1\}^{512}$. *Let* $\mathcal{D}_{\mathsf{MD}}$ *be an adversary against the* OTPRF*-security (Fig. 2) of the function family* $\mathsf{MD} = \mathsf{MD}[h_{256}]$ *that makes queries of length at most T blocks (i.e. at most $T \cdot 512$ bits). Then we can build an adversary* $\mathcal{D}_{\mathsf{H}}$ *against the* OTPRF*-security of* H *such that*

$$\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}}(\mathcal{D}_{\mathsf{MD}}) \le T \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{H}}(\mathcal{D}_{\mathsf{H}}).$$

Proof. In the $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{H},\mathcal{D}}$ game (Fig. 2), denote the oracle by $\mathrm{RoR}^{\mathsf{H}}$ and the challenge bit by b, and in the $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{MD},\mathcal{D}}$ game, denote by $\mathrm{RoR}^{\mathsf{MD}}$ and d respectively. Construct the adversary $\mathcal{D}_{\mathsf{H}}$ as in Fig. 48. We adopt the convention that $x_0 = x_{t+1} = \varepsilon$, $\mathrm{RoR}^{\mathsf{H}}(\varepsilon)$ returns $H_0 \leftarrow\!\!\$\ \{0,1\}^{\mathsf{H.kl}}$ and $\mathsf{MD.Ev}(H_t, \varepsilon)$ returns H_t. Consider the oracles shown in Fig. 49 for $i \in \{0,1,\ldots,T\}$, which correspond to a sequence of games G_i such that $\Pr[\,\mathrm{G}_i\,] = \Pr\left[\,\mathcal{D}^{\mathrm{RoR}_i}_{\mathsf{MD}} = 1\,\right]$. In the edge cases, RoR_0 behaves exactly like $\mathrm{RoR}^{\mathsf{MD}}$ if $d = 1$, and RoR_T behaves exactly like $\mathrm{RoR}^{\mathsf{MD}}$ if $d = 0$. So we can write $\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}}(\mathcal{D}_{\mathsf{MD}}) = \Pr[\,\mathrm{G}_T\,] - \Pr[\,\mathrm{G}_0\,]$. Next, fix $i \in \{0,1,\ldots,T\}$ and denote such $\mathcal{D}_{\mathsf{H}}$ by $\mathcal{D}_{\mathsf{H}}(i)$. Then

$$\Pr\left[\,\mathcal{D}^{\mathrm{RoR}^{\mathsf{H}}}_{\mathsf{H}}(i) = 1 \mid b = 0\,\right] = \Pr\left[\,\mathcal{D}^{\mathrm{RoRSim}_i}_{\mathsf{MD}} = 1 \mid b = 0\,\right] = \Pr[\,\mathrm{G}_i\,]$$

since if $t \le i - 1$, both RoRSim_i and RoR_i return a random value, and otherwise their code is the same. Similarly

$$\Pr\left[\,\mathcal{D}^{\mathrm{RoR}^{\mathsf{H}}}_{\mathsf{H}}(i) = 1 \mid b = 1\,\right] = \Pr\left[\,\mathcal{D}^{\mathrm{RoRSim}_i}_{\mathsf{MD}} = 1 \mid b = 1\,\right] = \Pr[\,\mathrm{G}_{i-1}\,]$$

since we have that $\mathsf{MD.Ev}(\mathsf{H.Ev}(H_{i-1}, x_i), x_{i+1} \,\|\, \ldots \,\|\, x_t) = \mathsf{MD.Ev}(H_{i-1}, x_i \,\|\, \ldots \,\|\, x_t)$ by the construction of MD.

Putting it together, we can write

$$\begin{aligned}\Pr\left[\,\mathcal{D}^{\mathrm{RoR}^{\mathsf{H}}}_{\mathsf{H}} = 1 \mid b = 0\,\right] &= \Pr\left[\bigvee_{j=1}^{T}(i = j \wedge \mathcal{D}^{\mathrm{RoR}^{\mathsf{H}}}_{\mathsf{H}}(j) = 1) \mid b = 0\right] \\ &= \frac{1}{T}\sum_{j=1}^{T}\Pr[\,\mathrm{G}_j\,]\end{aligned}$$

and similarly

$$\Pr\left[\,\mathcal{D}^{\mathrm{RoR}^{\mathsf{H}}}_{\mathsf{H}} = 1 \mid b = 1\,\right] = \frac{1}{T}\sum_{j=1}^{T}\Pr[\,\mathrm{G}_{j-1}\,]$$

so that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{H}}(\mathcal{D}_{\mathsf{H}}) &= \frac{1}{T}\left(\sum_{j=1}^{T}\Pr[\,\mathrm{G}_j\,] - \sum_{j=1}^{T}\Pr[\,\mathrm{G}_{j-1}\,]\right) \\ &= \frac{1}{T}(\Pr[\,\mathrm{G}_T\,] - \Pr[\,\mathrm{G}_0\,]) = \frac{1}{T}\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}}(\mathcal{D}_{\mathsf{MD}}).\end{aligned}$$

The inequality follows. □

```
Adversary $\mathcal{D}_{\mathsf{H}}^{\mathrm{RoR}^{\mathsf{H}}}$
$i \leftarrow\!\!\$\ \{0,1,\ldots,T\}$
$b' \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{MD}}^{\mathrm{RoRSim}_i}$
Return $b'$

$\mathrm{RoRSim}_i(x_1 \,\|\, \ldots \,\|\, x_t)$
If $t \le i - 1$ then $y \leftarrow\!\!\$\ \{0,1\}^{\mathsf{H.ol}}$
Else
    $H_i \leftarrow \mathrm{RoR}^{\mathsf{H}}(x_i)$
    $y \leftarrow \mathsf{MD.Ev}(H_i, x_{i+1} \,\|\, \ldots \,\|\, x_t)$
Return $y$
```

Figure 48: Adversary for the proof of Lemma 1.

```
Game $\mathrm{G}_i$
$d' \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{MD}}^{\mathrm{RoR}_i}$
Return $d'$

$\mathrm{RoR}_i(x_1 \,\|\, \ldots \,\|\, x_t)$
$H_i \leftarrow\!\!\$\ \{0,1\}^{\mathsf{H.kl}}$
$y \leftarrow \mathsf{MD.Ev}(H_i, x_{i+1} \,\|\, \ldots \,\|\, x_t)$
Return $y$
```

Figure 49: Intermediary games for the proof of Lemma 1.

We are ready to state the main result about the security of MTP-MAC, which we reduce to two assumptions in Proposition 5. As in the case of MTP-KDF, we use an unusual assumption on SHACAL-2 that involves related keys and the adversary's ability to choose a part of the key, but it is only evaluated on a fixed input (see Fig. 50). The advantage of $\mathcal{D}$ in breaking the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr\left[\,\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}\,\right] - 1$.

Overall, we require two assumptions: (a) that $\mathsf{SHACAL\text{-}2.Ev}(k, m)$ is a PRF under known fixed m, partially known k and key relations ϕ_{MAC} and (b) that $h_{256}(k, \cdot)$ is a one-time PRF. Concretely, $h_{256}(a, b) := a \mathbin{\hat{+}} \mathsf{SHACAL\text{-}2.Ev}(b, a)$ and thus we require both assumptions to hold for SHACAL-2.[26]

```
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 $b' = b$

$\mathrm{RoR}(u, p)$   // $|p| = 256$
$y_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(mk_u \,\|\, p, \mathsf{IV}_{256})$
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 50: Leakage-resilient PRF security of SHACAL-2 under related-key attacks with constant input IV_{256}, where p is the chosen-key part.

Proposition 5. *Let* $\mathcal{D}_{\mathrm{UPRKPRF}}$ *be an adversary against the* UPRKPRF*-security of* MAC = MTP-MAC *under the related-key-deriving function* ϕ_{MAC} *for inputs whose 256-bit prefixes are distinct from each other. Then we can build an adversary* $\mathcal{D}_{\mathrm{HRKPRF}}$ *against the* HRKPRF*-security of* SHACAL-2 *under* ϕ_{MAC} *and an adversary* $\mathcal{D}_{\mathrm{OTPRF}}$ *against the* OTPRF*-security of the Merkle–Damgård transform of* SHA-256, $\mathsf{MD} = \mathsf{MD}[h_{256}]$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}) \le\ & 2 \cdot \mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}}) \\ &+ 2 \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}}(\mathcal{D}_{\mathrm{OTPRF}}).\end{aligned}$$

[26]Note that $\mathsf{SHACAL\text{-}2.Ev}(m, k)$ for chosen m and random secret k is not a PRF since it comes endowed with a decryption function revealing k given $y = \mathsf{SHACAL\text{-}2.Ev}(m, k)$ and the chosen m. This does not rule out the "masked" construction $k \mathbin{\hat{+}} \mathsf{SHACAL\text{-}2.Ev}(m, k)$ being a PRF.

Proof. Consider the $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$ game (Fig. 23). Recall that $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192] = \mathsf{MD}[h_{256}].\mathsf{Ev}(\mathsf{IV}_{256}, \mathsf{SHA\text{-}pad}(mk_u \,\|\, p))[64:192]$. We first rewrite the game in a functionally equivalent way as G_0, splitting the MD.Ev call based on what happens to the first block of input. Since the first block contains a secret mk_u, it can be interpreted as providing security guarantees for a SHACAL-2 call keyed with the first block. G_1 thus captures that such a call result should be indistinguishable from random if SHACAL-2 is a leakage-resilient PRF under related keys. Similarly, G_2 replaces the MD call on the remaining input (if there is any) with a random value. This final game returns a random value regardless of the challenge bit, so $\mathcal{D}_{\mathrm{UPRKPRF}}$ cannot have a better than guessing advantage to win.

We first build an adversary $\mathcal{D}_{\mathrm{HRKPRF}}$ for the $\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ game (Fig. 50) so that we obtain $\Pr[\,\mathrm{G}_0\,] - \Pr[\,\mathrm{G}_1\,] = \mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}})$, as shown in Fig. 51. The uniqueness of prefixes of p is used to ensure that the RoR oracle of $\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ is never called on the same input twice. Next, we build an adversary $\mathcal{D}_{\mathrm{OTPRF}}$ for the $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{MD},\mathcal{D}}$ game (Fig. 2) so that $\Pr[\,\mathrm{G}_1\,] - \Pr[\,\mathrm{G}_2\,] = \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}}(\mathcal{D}_{\mathrm{OTPRF}})$, as shown in Fig. 52. Note that the $\mathrm{RoR}^{\mathsf{otprf}}$ oracle is only called if $\mathcal{D}_{\mathrm{RKPRF}}$ calls RoRSim on large enough inputs. However, if this never happened, it could have no distinguishing advantage better than guessing because we already swapped out the first block at this point. Denote the advantages by $a_{\mathsf{hrkprf}} = \mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}})$ and $a_{\mathsf{otprf}} = \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}}(\mathcal{D}_{\mathrm{OTPRF}})$. Then using both adversaries we can write

$$\begin{aligned}\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}) &= 2\cdot a_{\mathsf{hrkprf}} - 1 + 2\cdot \Pr[\,\mathrm{G}_1\,]\\ &= 2\cdot a_{\mathsf{hrkprf}} + 2\cdot a_{\mathsf{otprf}}\end{aligned}$$

by substituting $\Pr[\,\mathrm{G}_0\,] = \frac{1}{2}\left(\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}) + 1\right)$ in $a_{\mathsf{hrkprf}} = \Pr[\,\mathrm{G}_0\,] - \Pr[\,\mathrm{G}_1\,]$ and substituting $\Pr[\,\mathrm{G}_2\,] = \frac{1}{2}$ in $a_{\mathsf{otprf}} = \Pr[\,\mathrm{G}_1\,] - \Pr[\,\mathrm{G}_2\,]$. The inequality follows.

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathrm{HRKPRF}}$
$b \leftarrow\!\$ \{0,1\}$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$
$b' \leftarrow\!\$ \mathcal{D}^{\mathrm{RoRSim}}_{\mathrm{UPRKPRF}}$
If $b' = b$ then return 1
Else return 0

$\mathrm{RoRSim}(u, p)$
If $|p| < 256$ then return $\bot$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then return $\bot$
$X_u \leftarrow X_u \cup \{p_0\}$
$p \leftarrow \mathsf{SHA\text{-}pad}(p)$ ; $p_1 \leftarrow p[256:|p|]$
$H \leftarrow \mathsf{IV}_{256} \hat{+} \mathrm{RoR}(u, p_0)$
If $|p_1| > 0$ then
    $\mathsf{msg\_key}_1 \leftarrow \mathsf{MD.Ev}(H, p_1)$
Else
    $\mathsf{msg\_key}_1 \leftarrow H$
$\mathsf{msg\_key}_0 \leftarrow\!\$ \{0,1\}^{\mathsf{MAC.ol}}$
Return $\mathsf{msg\_key}_b[64:192]$
```

Figure 51: First adversary for the proof of Proposition 5.

∎

4) OTIND\$ of IGE: We will show that IGE encryption can be expressed in terms of CBC. Using this, Proposition 6 shows that MTP-SE, which uses IGE mode, is one-time indistinguishable

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathrm{OTPRF}}$
$b \leftarrow\!\$ \{0,1\}$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$
$b' \leftarrow\!\$ \mathcal{D}^{\mathrm{RoRSim}}_{\mathrm{UPRKPRF}}$
If $b' = b$ then return 1
Else return 0

$\mathrm{RoRSim}(u, p)$
If $|p| < 256$ then return $\bot$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then return $\bot$
$X_u \leftarrow X_u \cup \{p_0\}$
$p \leftarrow \mathsf{SHA\text{-}pad}(p)$ ; $p_1 \leftarrow p[256:|p|]$
$H \leftarrow\!\$ \{0,1\}^{256}$
If $|p_1| > 0$ then
    $\mathsf{msg\_key}_1 \leftarrow \mathrm{RoR}(u, p_1)$
Else
    $\mathsf{msg\_key}_1 \leftarrow H$
$\mathsf{msg\_key}_0 \leftarrow\!\$ \{0,1\}^{\mathsf{MAC.ol}}$
Return $\mathsf{msg\_key}_b[64:192]$
```

Figure 52: Second adversary for the proof of Proposition 5.

```
Games $\mathrm{G}_0$–$\mathrm{G}_2$
$b \leftarrow\!\$ \{0,1\}$ ; $mk \leftarrow\!\$ \{0,1\}^{320}$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$ ; $b' \leftarrow\!\$ \mathcal{D}^{\mathrm{RoR}}_{\mathrm{UPRKPRF}}$
Return $b' = b$

$\mathrm{RoR}(u, p)$
If $|p| < 256$ then return $\bot$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then return $\bot$
$X_u \leftarrow X_u \cup \{p_0\}$
$p \leftarrow \mathsf{SHA\text{-}pad}(p)$ ; $p_1 \leftarrow p[256:|p|]$
$H \leftarrow \mathsf{IV}_{256} \hat{+} \mathsf{SHACAL\text{-}2.Ev}(mk_u \,\|\, p_0, \mathsf{IV}_{256})$        // $\mathrm{G}_0$
$H \leftarrow\!\$ \{0,1\}^{256}$                                                 // $\mathrm{G}_1$
If $|p_1| > 0$ then
    $\mathsf{msg\_key}_1 \leftarrow \mathsf{MD.Ev}(H, p_1)$                       // $\mathrm{G}_0$–$\mathrm{G}_1$
    $\mathsf{msg\_key}_1 \leftarrow\!\$ \{0,1\}^{256}$                             // $\mathrm{G}_2$
Else
    $\mathsf{msg\_key}_1 \leftarrow H$
$\mathsf{msg\_key}_0 \leftarrow\!\$ \{0,1\}^{\mathsf{MAC.ol}}$
Return $\mathsf{msg\_key}_b[64:192]$
```

Figure 53: Games for the proof of Proposition 5. $\mathrm{G}_0 = \mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$ expands $\mathsf{msg\_key}_1 \leftarrow \mathsf{MAC.Ev}(mk_u, p)$ into two calls (in gray). The changes made to G_1 and G_2 are in green.

if CBC mode is (game in Fig. 3). We refer to known CBC bounds in the literature [52], [53] but note that, since we are in the one-time setting, in which each key is used to encrypt only a limited amount of data, we can obtain a sharper bound for CBC mode than would be obtained if a single key were used to encrypt all the messages.

Proposition 6. *Let* E *be a block cipher. Consider the symmetric encryption schemes* $\mathsf{SE}_{\mathsf{IGE}} = \mathsf{IGE}[\mathsf{E}]$ *and* $\mathsf{SE}_{\mathsf{CBC}} = \mathsf{CBC}[\mathsf{E}]$*. Let* $\mathcal{D}_{\mathsf{IGE}}$ *be an adversary against the* OTIND\$*-security of* $\mathsf{SE}_{\mathsf{IGE}}$*. Then we can build an adversary* $\mathcal{D}_{\mathsf{CBC}}$ *against the* OTIND\$*-security of* $\mathsf{SE}_{\mathsf{CBC}}$ *such that*

$$\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{IGE}}}(\mathcal{D}_{\mathsf{IGE}}) \le \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{CBC}}}(\mathcal{D}_{\mathsf{CBC}}).$$

Proof. Construct the adversary $\mathcal{D}_{\mathsf{CBC}}$ as in Fig. 54. If $b = 0$ in $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{CBC}},\mathcal{D}_{\mathsf{CBC}}}$, $\mathrm{RoRSim}(m)$ returns a random value as c', which is preserved under XOR. If $b = 1$, we get $c' = \mathsf{SE}_{\mathsf{CBC}}.\mathsf{Enc}(k, m')$

which implies that $c'_i = \mathsf{E.Ev}(k[0 : \mathsf{E.kl}], m_i \oplus m_{i-2} \oplus c'_{i-1})$ for $i \geq 2$. Since $c_i = c'_i \oplus m_{i-1}$, we get $c_i = \mathsf{E.Ev}(k[0 : \mathsf{E.kl}], m_i \oplus c_{i-1}) \oplus m_{i-1}$ and so $c = \mathsf{SE}_{\mathsf{IGE}}.\mathsf{Enc}(k \| m_0, m)$. In both cases RoRSim simulates RoR perfectly, so $\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{CBC}}}(\mathcal{D}_{\mathsf{CBC}}) = \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{IGE}}}(\mathcal{D}_{\mathsf{IGE}})$.

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathsf{CBC}}$      RoRSim($m$)
$b' \leftarrow\$ \mathcal{D}^{\mathrm{RoRSim}}_{\mathsf{IGE}}$        $m_0 \leftarrow\$ \{0,1\}^{\mathsf{E.ol}}$ ; $m'_1 \leftarrow m_1$
Return $b'$                 For $i = 2, \ldots, t$ do
                                $m'_i \leftarrow m_i \oplus m_{i-2}$
                            $c' \leftarrow \mathrm{RoR}(m')$
                            For $i = 1, \ldots, t$ do
                                $c_i \leftarrow c'_i \oplus m_{i-1}$
                            Return $c$
```
Figure 54: Adversary for the proof of Proposition 6.

□

5) EINT of MTP-ME with respect to SUPP: Here we prove that MTP-ME matches SUPP for strict in-order delivery.

Proposition 7. *Let* ME = MTP-ME *be the message encoding defined in Fig. 19 and* supp = SUPP *be the support function defined in Fig. 25. Then for all $\mathcal{F}$ against* EINT*-security of* ME *with respect to* supp *making $q < 2^{96}$* SEND *queries, we have*

$$\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{F}) = 0.$$

Proof. Consider the $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$ game (Fig. 10).

First, notice that p as produced by Send can act as a unique label within each transcript tr_u. For sent entries, ME.Encode ensures that every payload includes seq_no that is incremented with every new message, which holds as long as less than 2^{96} Send queries are made for the same message m. Assume that this is the case. Then, Recv only adds recv entries for honestly produced payloads. If ME.Decode is called twice on the same p, it cannot output $m \neq \perp$ more than once because we would have to have $\mathsf{seq\_no} = N_{\mathsf{recv}} + 1 = N'_{\mathsf{recv}} + 1$ for $N_{\mathsf{recv}} \neq N'_{\mathsf{recv}}$ (since $m \neq \perp$ implies that the counter was incremented). So each p cannot appear in a recv entry in some tr_u more than once. It is clear by inspection that ME.Encode never outputs $p = \perp$ and that ME.Decode only outputs a changed state if it also outputs a message $m \neq \perp$.

We examine what happens during a call to $\mathrm{Recv}(u, p, aux)$. We can assume that there was a $\mathrm{Send}(\overline{u}, m', aux', r) \rightarrow p$ call in the past, otherwise there would be no $(\mathsf{sent}, m', p, aux')$ entry in $\mathsf{tr}_{\overline{u}}$ and the win condition could not be satisfied. From ME.Encode we get that $p = \mathsf{salt} \,\|\, \mathsf{session\_id} \,\|\, \mathsf{seq\_no} \,\|\, \mathsf{length} \,\|\, m' \,\|\, \mathsf{padding}$. Let $st_{\mathsf{ME},u} = (\mathsf{session\_id}, \cdot, N_{\mathsf{recv},u})$ be the state before ME.Decode is called on p:

- Suppose there was a $\mathrm{Recv}(u, p, aux'')$ call in the past such that tr_u contains $(\mathsf{recv}, m, p, aux'')$ for some $m \neq \perp$. As shown earlier, ME.Decode does not output successfully more than once on the same p, so in the current call it has to output $\perp$. The support function $\mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, p, aux)$ returns $m^* = \perp$, because $\mathsf{find}(\mathsf{recv}, \mathsf{tr}_u, p)$ iterates over all recv entries in tr_u and finds a match for p such that its $m \neq \perp$. So we always have $m = m^*$ and $\mathcal{F}$ cannot win in this case.
- Suppose there was no $\mathrm{Recv}(u, p, \cdot)$ call in the past, or for all $(\mathsf{recv}, m, p, \cdot)$ in tr_u we have $m = \perp$. The support function $\mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, p, aux)$ first makes a call to $\mathsf{find}(\mathsf{recv}, \mathsf{tr}_u, p)$ which outputs $(n_u, \perp)$ where n_u is the number of entries of tr_u of the form $(\mathsf{recv}, m, p', \cdot)$ for $m \neq \perp$ and $p' \neq p$. Next, it calls $\mathsf{find}(\mathsf{sent}, \mathsf{tr}_{\overline{u}}, p)$ which outputs $(n_{\overline{u}}, m')$ because $\mathsf{tr}_{\overline{u}}$ contains the entry $(\mathsf{sent}, m', p, aux')$, where $n_{\overline{u}}$ is the number of entries of $\mathsf{tr}_{\overline{u}}$ that were sent before and including that entry. Then it checks whether $n_{\overline{u}} = n_u + 1$.

Let us compute both counts. Whenever an entry $(\mathsf{recv}, m, p', \cdot)$ for $m \neq \perp$ is added to tr_u, it means that the output of ME.Decode included a changed state that incremented the number of received messages by one. Hence $n_u = N_{\mathsf{recv},u}$. Similarly, an entry $(\mathsf{sent}, m, \cdot, \cdot)$ is only added to $\mathsf{tr}_{\overline{u}}$ when ME.Encode was called and its output included a changed state that incremented the number of sent messages by one and saved it in the sequence number field. We get that $n_{\overline{u}} = \mathsf{seq\_no}$ as long as $n_{\overline{u}} < 2^{96}$, which we assumed at the beginning. Then the support function check is the same as the check performed by $\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$, whether $\mathsf{seq\_no} = N_{\mathsf{recv},u} + 1$. Hence the support function outputs m' if and only if ME.Decode does, and $\mathcal{F}$ cannot win.

For completeness, let us now deal with the case of the overflow and show that the adversary can win then. Suppose that $\mathcal{F}$ repeatedly queries $\mathrm{Send}(\overline{u}, m, \cdot) \rightarrow p_i$ for the same m and $i = 0, 1, \ldots, 2^{96}$. Because seq_no is of fixed size, $p_0 = p_{2^{96}}$. The first $\mathrm{Recv}(u, p_0, \cdot)$ call returns m as expected since both ME.Decode and supp interpret it as the honestly sent first message. Suppose that $\mathcal{F}$ then queries $\mathrm{Recv}(u, p_i, \cdot)$ for $i = 1, \ldots, 2^{96} - 1$. These will be honestly processed. Then a $\mathrm{Recv}(u, p_{2^{96}}, \cdot)$ query causes a mismatch: in ME.Decode the seq_no check passes because the counter wraps ($N_{\mathsf{recv},u} = \mathsf{seq\_no} = 1$) and so it returns m, but in supp we get $\mathsf{find}(\mathsf{recv}, \mathsf{tr}_u, p) \rightarrow m \neq \perp$ so it returns $\perp$ (despite it being honestly produced, which violates a different property which is defined in Fig. 35). □

6) UNPRED of MTP-SE and MTP-ME: In Proposition 8, we consider MTP-SE without instantiating it with a particular block cipher. We show that it is hard for $\mathcal{F}$ to find c_{se} such that its decryption under a random key begins with $p' = \mathsf{salt} \,\|\, \mathsf{session\_id}$, where session_id is a value chosen by the adversary via st_{ME} and salt is arbitrary. Note that the proof is not tight, i.e. the advantage could potentially be lower if we also considered the seq_no and length fields in the second block. However, this would complicate analysis and possibly overstate the security of MTProto as implemented, given that we made the modelling choice to check more fields in MTP-ME upon decoding. Note that the bound could easily be improved if MTP-ME checked the salt in the first block, however this would deviate even further from the current MTProto implementation and so we did not include this in our definition.

Proposition 8. *Let $\mathcal{F}$ be an adversary against the* UNPRED-*security of* SE = MTP-SE *and* ME = MTP-ME *which makes q_{CH} queries to* CH. *Then*

$$\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}) \leq \frac{q_{\text{CH}}}{2^{64}}.$$

Proof. We have that MTP-SE = IGE[AES-256], but for the purposes of this proof we can treat AES-256 as an abstract block cipher E. We rewrite the $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}$ game (Fig. 27) as G_0 in Fig. 55 with the following relaxation on ME: we omit the seq_no and length checks so that we can focus only on the first plaintext block. This makes the game easier to win for the adversary, but does not change it otherwise as CH does not return any output.

```
Game G_0
win ← false ; $\mathcal{F}^{\text{Expose},\text{Ch}}$ ; Return win
Expose($u$, msg_key)
S[$u$, msg_key] ← true ; Return T[$u$, msg_key]
Ch($u$, msg_key, $c_{se}$, $st_{\mathsf{ME}}$, $aux$)
If ¬S[$u$, msg_key] then
    If T[$u$, msg_key] = ⊥ then T[$u$, msg_key] ←$ $\{0,1\}^{\mathsf{SE.kl}}$
    $k$ ← T[$u$, msg_key] ; $(K, c_0, p_0) \leftarrow k$
    $c_1 \leftarrow c_{se}[0:128]$
    $p_1 \leftarrow \mathsf{E.Inv}(K, c_1 \oplus p_0) \oplus c_0$
    (session_id, $N_{\mathsf{sent}}$, $N_{\mathsf{recv}}$) ← $st_{\mathsf{ME}}$
    If $p_1[64:128]$ = session_id then
        win ← true
Return ⊥
```

Figure 55: Game $\mathrm{G}_0 = \mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}$, where MTP-SE and MTP-ME calls are expanded up to the first block of input in gray. Keys k are parsed such that $|K| = \mathsf{E.kl}$, $|c_0| = |p_0| = \mathsf{E.ol}$.

The adversary $\mathcal{F}$ can only win in G_0 if $p_1[64:128]$ = session_id for some p_1 that is defined by the equation $p_1 = \mathsf{E.Inv}(K, c_1 \oplus p_0) \oplus c_0$. We can rewrite this winning condition as $\mathsf{E.Inv}(K, c_1 \oplus p_0)[64:128] \oplus \mathsf{session\_id} = c_0[64:128]$. Here $c_0[64:128]$ is a bit string that is sampled uniformly at random for each pair $(u, \mathsf{msg\_key})$ and that is unknown to the adversary.

Consider for a moment a particular pair $(u, \mathsf{msg\_key})$; suppose that $\mathcal{F}$ makes $q_{u,\mathsf{msg\_key}}$ queries to CH relating to this pair. These queries result in some specific set of values $X_{u,\mathsf{msg\_key}}$ for $\mathsf{E.Inv}(K, c_1 \oplus p_0)[64:128] \oplus \mathsf{session\_id}$ arising in the game. Moreover, $\mathcal{F}$ wins for one of these queries if and only if some element of the set $X_{u,\mathsf{msg\_key}}$ matches $c_0[64:128]$. Note also that $\mathcal{F}$ learns nothing about $c_0[64:128]$ from each such query (since the CH oracle always returns ⊥). Combining these facts, we see that $\mathcal{F}$'s winning probability for this set of $q_{u,\mathsf{msg\_key}}$ queries is no larger than $q_{u,\mathsf{msg\_key}}/2^{64}$ (in essence, $\mathcal{F}$ can do no better than random guessing of distinct values for the unknown 64 bits). Moreover, while the adversary can learn c_0 for any $(u, \mathsf{msg\_key})$ pair after-the-fact using EXPOSE, it cannot continue querying CH for this value once the query is made, which makes the output of that oracle useless in winning the game.

Considering all pairs $(u, \mathsf{msg\_key})$ involved in $\mathcal{F}$'s queries and using the union bound, we obtain that $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}) \leq \Pr[\,\mathrm{G}_0\,] \leq q_{\text{CH}} \cdot 2^{-64}$. □

Remark 2. *We could formalise the above using an additional hop to a game which should make it obvious that the adversary can do nothing better than guessing. Consider the game* G_1 *in Fig. 56. We claim that given $\mathcal{F}_0$ that wins in* G_0*, we can build an adversary $\mathcal{F}_1$ that wins in* G_1*. This is because $\mathcal{F}_1$ can simulate the original oracles of* G_0 *by choosing all key material except the second half of c_0 (here c^*), which is chosen by its* CH *oracle and constitutes the challenge. Hence* $\Pr[\,\mathrm{G}_0\,] \leq \Pr[\,\mathrm{G}_1\,] \leq q_{\text{CH}} \cdot 2^{-64}$.

```
Game G_1
win ← false ; $\mathcal{F}_1^{\text{Expose},\text{Ch}}$ ; Return win
Expose($i$)
S[$i$] ← true ; Return T[$i$]
Ch($i$, $a$)
If ¬S[$i$] then
    If T[$i$] = ⊥ then T[$i$] ←$ $\{0,1\}^{64}$
    $c^*$ ← T[$i$]
    If $a = c^*$ then
        win ← true
Return ⊥
--------------------------------------------
$\mathcal{F}_1^{\text{Expose},\text{Ch}}$
$\mathcal{F}_0^{\text{ExposeSim},\text{ChSim}}$ ; Return
ExposeSim($u$, msg_key)
$i$ ← $u$ || msg_key ; S[$i$] ← true
$c^*$ ← Expose($i$) ; $(K, C, p_0)$ ← T[$i$] ; Return $(K, C \| c^*, p_0)$
ChSim($u$, msg_key, $c_{se}$, $st_{\mathsf{ME}}$, $aux$)
$i$ ← $u$ || msg_key
If ¬S[$i$] then
    If T[$i$] = ⊥ then
        $K$ ←$ $\{0,1\}^{\mathsf{E.kl}}$ ; $C$ ←$ $\{0,1\}^{\mathsf{E.ol}/2}$ ; $p_0$ ←$ $\{0,1\}^{\mathsf{E.ol}}$
        T[$i$] ← $(K, C, p_0)$
    $(K, C, p_0)$ ← T[$i$]
    $c_1 \leftarrow c_{se}[0:128]$
    (session_id, $N_{\mathsf{sent}}$, $N_{\mathsf{recv}}$) ← $st_{\mathsf{ME}}$
    $a \leftarrow \mathsf{E.Inv}(K, c_1 \oplus p_0)[64:128] \oplus$ session_id
    err ← Ch($i$, $a$)
Return ⊥
```

Figure 56: Game G_1 and adversary $\mathcal{F}_1$.

F. Attacking the key exchange

Recall that our attack in Section VI relies on knowledge of m_1 which in MTProto contains a 64-bit salt and a 64-bit session ID. In Appendix F1, we present a strategy for recovering the 64-bit salt. We then use it in a simple guess and confirm approach to recover the session ID in Appendix F2.

We stress, however, that the attack in Appendix F1 only applies in a short period after a key exchange between a client and a server.[27] Furthermore, the attack critically relies

[27]Telegram will perform roughly one key exchange per day, aiming for forward secrecy.

on observing small timing differences which is unrealistic in practice, especially over a wide network. That is, our attack relies on a timing side channel when Telegram's servers decrypt RSA ciphertexts and verify their integrity. While – in response to our disclosure – the Telegram developers confirmed the presence of non-constant code in that part of their implementation and hence confirmed our attack, they did not share source code or other details with us. That is, since Telegram does not publish source code for its servers in contrast to its clients the only option to verify the precise server behaviour is to test it. This would entail sending millions if not billions of requests to Telegram's servers, from a host that is geographically and topologically close to one of Telegram's data centres, observing the response time. Such an experiment would have been at the edge of our capabilities but is clearly feasible for a dedicated, well-resourced attacker.

In Appendix F3, we then discuss how the attack in Appendix F1 enables to break server authentication and thus enables an attacker-in-the-middle (MitM) attack on the Diffie-Hellman key exchange.

1) Recovering the salt: At a high level, our strategy exploits the fact that during the initial key exchange, Telegram integrity-protects RSA ciphertexts by including a hash of the underlying message contents in the encrypted payload *except for the random padding* which necessitates parsing the data which in turn establishes the potential for a timing side-channel.[28] In what follows, we assume the presence of such a side channel and show how it enables the recovery of the encrypted message, solving noisy linear equations via lattice reduction. We refer the reader to [54], [55] for an introduction to the application of lattice reduction in side-channel attacks and the state of the art respectively.

In Fig. 57 we display Telegram's Diffie-Hellman key exchange instantiation [56] at the level of detail required for our attack, omitting TL schema encoding. In Fig. 57, we let $n := \mathsf{nonce}$, $s := \mathsf{server\_nonce}$, $n' := \mathsf{new\_nonce}$ be nonces; $\mathcal{S}$ be the set of public server fingerprints, $F \in \mathcal{S}$ be the fingerprint of the key selected by the client, $t_s := \mathsf{server\_time}$ be a timestamp for the server; let $\mathcal{F}(\cdot,\cdot)$ be some function used to derive keys;[29] let p_r, p_s, p_c be random padding of appropriate length; and $ak := \mathsf{auth\_key}$ be the final key. The initial salt used by Telegram is then computed as $\mathsf{server\_salt} := n'[0:64] \oplus s[0:64]$. Since s is sent in the clear during the key exchange protocol, recovering the salt is equivalent to recovering $n'[0:64]$. We will let N', e denote the public RSA key (modulus and exponent) used to perform RSA encryption by the client in the key exchange and will let d denote the private RSA exponent used by the server to perform RSA decryption.[30] We assume N' has exactly 2048 bits which holds for the values used by Telegram.

[28]We note that this issue mirrors the one reported in [4].

[29]This consists of SHA-1 calls but we omit the details here.

[30]Note that N' is distinct from the proof-of-work value N that is sent by the server during the protocol and whose factors p, q are returned by the client.

Further, we have

$$h_{n'} := \mathsf{SHA\text{-}1}\left(n' \| \mathtt{0x0}i \| \mathsf{SHA\text{-}1}(ak)[0:64]\right)[32:160]$$

in Fig. 57 where $i = 1$, 2 or 3 depending on whether the key exchange terminated successfully and h_r, h_s, h_c are SHA-1 hashes over the rest of the RSA payload *except* for the padding p_r, p_s, p_c. In particular, we have

$$h_r := \mathsf{SHA\text{-}1}\left(N, p, q, n, s, n'\right).$$

The critical observation in this section is that while n, s and n' have fixed lengths of 128, 128 and 256 bits respectively, the same is not true for N, p and q. This implies that the content to be fed to SHA-1 after RSA decryption and during verification must first be parsed by the server. This opens up the possibility of a timing side channel. In particular, at a byte level SHA-1 is called on

$$hd \;\|\; \mathcal{L}(N)\|N\|\mathcal{P}(N) \;\|\; \mathcal{L}(p)\|p\|\mathcal{P}(p) \;\|\; \mathcal{L}(q)\|q\|\mathcal{P}(q) \;\|\; n\|s\|n'$$

where $\mathcal{L}(x)$ encodes the length of x in one byte;[31] x is stored in big endian byte order and $\mathcal{P}(x)$ is up to three zero bytes so that length of $\mathcal{L}(x)\|x\|\mathcal{P}(x)$ is divisible by 4; $hd = \mathtt{0xec5ac983}$.

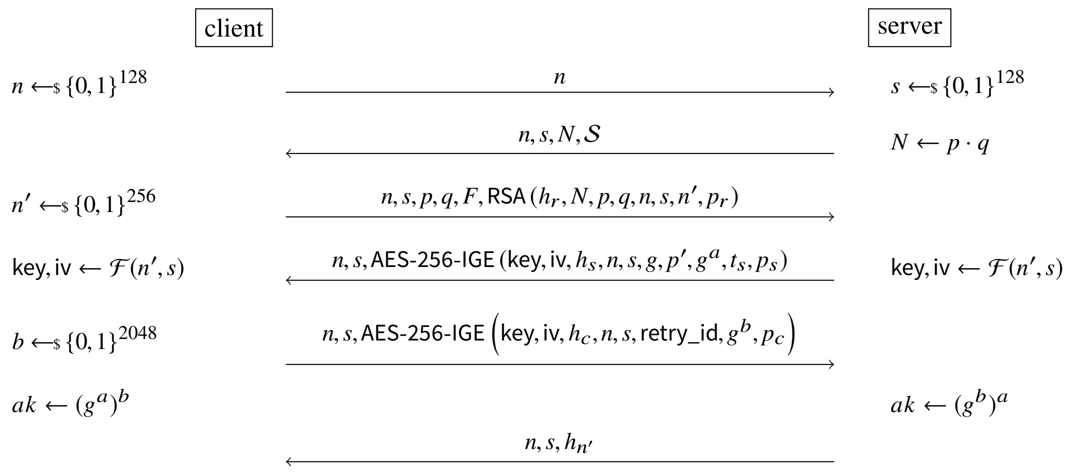
We verified the following behaviour of the Telegram server, where "is checked" and "expects" means the key exchange aborts if the payload deviates from the expectation.

- The header $hd = \mathtt{0xec5ac983}$ is checked;
- the server expects $1 \leq \mathcal{L}(N) \leq 16$ and $\mathcal{L}(p)$, $\mathcal{L}(q) = 4$ (different valid encodings, e.g. by prefixing zeroes, of valid values are not accepted);
- the value of N is *not* checked, p, q are checked against the value of N stored on the server and the server expects $p < q$;
- the contents of $\mathcal{P}(\cdot)$ are *not* checked;
- both n, s are checked.

While we do not know in what order the Telegram server performs these checks, we recall that the payload must be parsed before being integrity checked and that the number of bytes being fed to SHA-1 depends on this parsing. This is because the random padding must be removed from the payload before calling SHA-1.

Recall that the Telegram developers acknowledged the attack presented here but did not provide further details on their implementation. Therefore, below we will assume that the Telegram server code follows a similar pattern to Telegram's flagship TDLib library, which is used e.g. to implement the Telegram Bot API [10]. While TDLib does not implement RSA decryption, it does implement message parsing during the handshake. In particular, the library returns early when the header does not match its expected value. In our case the header is 0xec5ac983 but we stress that this behaviour does not seem to be problematic in TDLib and we do not know if the Telegram servers follow the same pattern also for RSA decryption. We will discuss other leakage patterns below, but for now we will assume the Telegram servers return early

[31]Longer inputs are supported by $\mathcal{L}(\cdot)$ but would not fit into ≤ 255 bytes of RSA payload.



client
server
$n \leftarrow_\$ \{0,1\}^{128}$
n
$s \leftarrow_\$ \{0,1\}^{128}$
$n, s, N, \mathcal{S}$
$N \leftarrow p \cdot q$
$n' \leftarrow_\$ \{0,1\}^{256}$
$n, s, p, q, F, \mathsf{RSA}(h_r, N, p, q, n, s, n', p_r)$
$\mathsf{key}, \mathsf{iv} \leftarrow \mathcal{F}(n', s)$
$n, s, \mathsf{AES\text{-}256\text{-}IGE}(\mathsf{key}, \mathsf{iv}, h_s, n, s, g, p', g^a, t_s, p_s)$
$\mathsf{key}, \mathsf{iv} \leftarrow \mathcal{F}(n', s)$
$b \leftarrow_\$ \{0,1\}^{2048}$
$n, s, \mathsf{AES\text{-}256\text{-}IGE}\left(\mathsf{key}, \mathsf{iv}, h_c, n, s, \mathsf{retry\_id}, g^b, p_c\right)$
$ak \leftarrow (g^a)^b$
$ak \leftarrow (g^b)^a$
$n, s, h_{n'}$

Figure 57: Telegram Key Exchange.

whenever there is a header mismatch, skipping the SHA-1 call in this case. This produces a timing side channel.

Thus, we consider a textbook RSA ciphertext $c = m^e \bmod N'$ with

$$m = h_r \| hd \| \mathcal{L}(N) \| N \| \mathcal{P}(N) \| \mathcal{L}(p) \| p \| \mathcal{P}(p) \| \mathcal{L}(q) \| q \| \mathcal{P}(q) \| n \| s \| n' \| p_r$$

of length 255 bytes. First, observe that an attacker knows all contents of the payload (including their encodings) except for h_r, n' and p_r and we can write:

$$x = 2^{\ell(p_r)} \cdot n' + p_r < 2^{256+\ell(p_r)}$$
$$m = (2^{1880} \cdot h_r + 2^{256+\ell(p_r)} \cdot \gamma + x)$$

where γ is a known constant derived from n, s, p, q, N and where $\ell(p_r)$ is the known length of p_r. This relies on knowing that $|n'| = 256$ and $|m| - |h_r| = 1880$.

Under our assumption on header checking, we can detect whether the bits in positions $8 \cdot 255 - 160 - 32$ to $8 \cdot 255 - 160 - 1$ (big endian, SHA-1 outputs 160 bits) of $m' := (c')^d$ match `0xec5ac983` for any c' we submit to the Telegram servers. Thus, inspired by [13], we submit $s_i^e \cdot c$, for several chosen s_i to the server and receive back an answer whether the bits 1848 to 1879 of $s_i \cdot m$ match the expected header. If the s_i are chosen sufficiently randomly, this event will have probability $\approx 2^{-32}$. Writing ζ = `0xec5ac983`, we consider

$$\begin{aligned} e_i &= \left((s_i \cdot m \bmod N') - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\ &= \left(\left(s_i \cdot \left(2^{1880} \cdot h_r + 2^{256+\ell(p_r)} \cdot \gamma + x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\ &= \left(\left(\left(s_i \cdot 2^{1880} \cdot h_r + s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma + s_i \cdot x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \\ &\quad \bmod 2^{1880}. \end{aligned}$$

That is, we pick random s_i (we will discuss how to pick those below) and submit $s_i^e \cdot c$ to the Telegram servers. Using the timing side channel we then detect when the bits in the header position match ζ. When this happens, we store s_i. Overall, we find μ such s_i (we discuss below how to pick μ) and suppose the event happens for some set of s_i, with $i = 0, \ldots, \mu - 1$.

a) Recovering h_r: Note that $e_i < 2^{1880-32}$ by construction and $x < 2^{256+\ell(p_r)} \ll 2^{1848}$. Thus, picking sufficiently small s_i an attacker can make $e_i' := (e_i - s_i \cdot x) \bmod 2^{1880} < 2^{1848}$, i.e.

$$\begin{aligned} e_i' &= \left(\left(\left(s_i \cdot 2^{1880} \cdot h_r + s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \\ &\quad \bmod 2^{1880} < 2^{1848}. \end{aligned}$$

We rewrite e_i' as

$$e_i' = \left(s_i \cdot 2^{1880} \cdot h_r + s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma - \zeta \cdot 2^{1848} - \sigma_i \cdot 2^{1880}\right) \bmod N'$$

for $\sigma_i < 2^{160}$ and use lattice reduction to recover h_r. Writing

$$t_i = \left(s_i \cdot 2^{256+\ell(p_r)} \cdot \gamma - \zeta \cdot 2^{1848}\right) \bmod N',$$

we consider the lattice spanned by the rows of L_1 with

$$L_1 := \begin{pmatrix} 2^{1688} & 0 & 0 & 0 & 2^{1880} \cdot s_0 & \cdots & 2^{1880} \cdot s_{\mu-1} & 0 \\ 0 & 2^{1688} & 0 & 0 & 2^{1880} & \cdots & 0 & 0 \\ & & \ddots & & & \ddots & & \\ 0 & 0 & \ddots & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 2^{1688} & 0 & \cdots & 2^{1880} & 0 \\ 0 & 0 & 0 & 0 & N' & \cdots & 0 & 0 \\ & & & & & \ddots & & \\ 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & N' & 0 \\ 0 & 0 & 0 & 0 & t_0 & \cdots & t_{\mu-1} & 2^{1848} \end{pmatrix}.$$

Multiplying L_1 from the left by

$$(h_r,\ -\sigma_0,\ \ldots,\ -\sigma_{\mu-1},\ *, \ldots, *, 1)$$

where $*$ stands for modular reduction by N', shows that this lattice contains a vector

$$(2^{1688} \cdot h_r,\ -2^{1688}\sigma_0,\ \ldots,\ -2^{1688}\sigma_{\mu-1},\ e_0', \ldots,\ e_{\mu-1}',\ 2^{1848}) \quad (1)$$

where all entries are bounded by $2^{1848} = 2^{1688+160}$. Thus that vector has Euclidean norm $\leq \sqrt{2\mu + 2} \cdot 2^{1848}$.[32] On the other hand, the Gaussian heuristic predicts the shortest vector in the lattice to have norm

$$\approx \sqrt{\frac{2\mu + 2}{2\pi e}} \cdot \left(2^{1688 \cdot (\mu+1)} \cdot (N')^{\mu} \cdot 2^{1848}\right)^{1/(2\mu+2)}. \quad (2)$$

[32]This estimate is pessimistic for the attacker. Applying the techniques summarised in [55] for constructing such lattices, we can save a factor of roughly two. We forgo these improvements here to keep the presentation simple.

Finding a shortest vector in the lattice spanned by the rows of L_1 is expected to recover our target vector and thus h_r when the norm of expression (1) is smaller than the expression (2) which is satisfied for $\mu = 6$.

We experimentally verified that LLL on a $(2 \cdot 6 + 2)$-dimensional lattice constructed as L_1 indeed succeeds (cf. Appendix G). Thus, under our assumptions, recovering h_r requires about $6 \cdot 2^{32}$ queries to Telegram's servers and a trivial amount of computation.

b) Recovering n': Once we have recovered h_r, we can target n'. Writing $\gamma' = 2^{1880-256-\ell(p_r)} \cdot h_r + \gamma$, we obtain

$$\begin{aligned}
d_i &= \left(\left(s'_i \cdot m \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\
&= \left(\left(s'_i \cdot \left(2^{256+\ell(p_r)} \cdot \gamma' + x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\
&= \left(\left(\left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot x\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \bmod 2^{1880} \\
&= \left(\left(\left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot (2^{\ell(p_r)} \cdot n' + p_r)\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \\
&\quad \bmod 2^{1880}
\end{aligned}$$

where the s'_i are again chosen randomly and we collect s'_i for $i = 0, \dots, \mu' - 1$ where the bits in the header position match ζ. We discuss how to choose s'_i and μ' below. Thus, we assume that $d_i < 2^{1848}$ for s'_i. Information theoretically, each such inequality leaks 32 bits. Considering that $x = 2^{\ell(p_r)}n' + p_r$ has $256 + \ell(p_r)$ bits, we thus require at least $(256 + \ell(p_r))/32$ such inequalities to recover x.[33] Yet, $\ell(p_r) \gg 256$ and the content of p_r is of no interest to us, i.e. we seek to recover n' without "wasting entropy" on p_r.[34] In other words, we wish to pick s'_i sufficiently large so that all bits of $s'_i \cdot 2^{\ell(p_r)} \cdot n'$ affect the 32 bits starting at 2^{1848} but sufficiently small to still allow us to consider "most of" $s'_i \cdot p_r$ as part of the lower-order bit noise. Thus, we pick random $s'_i \approx 2^{1848-\ell(p_r)}$ and consider $d'_i := d_i - s'_i \cdot p_r$ with

$$\begin{aligned}
d'_i &= \left(\left(\left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot 2^{\ell(p_r)} \cdot n'\right) \bmod N'\right) - \zeta \cdot 2^{1848}\right) \\
&\quad \bmod 2^{1880} \\
&= \left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' + s'_i \cdot 2^{\ell(p_r)} \cdot n' - \zeta \cdot 2^{1848} - \sigma'_i \cdot 2^{1880}\right) \bmod N'.
\end{aligned}$$

Writing

$$t'_i = \left(s'_i \cdot 2^{256+\ell(p_r)} \cdot \gamma' - \zeta \cdot 2^{1848}\right) \bmod N',$$

we consider the lattice spanned by the rows of L_2 with

$$L_2 := \begin{pmatrix}
2^{1592} & 0 & 0 & 0 & 2^{\ell(p_r)} \cdot s'_0 & \cdots & 2^{\ell(p_r)} \cdot s'_{\mu'-1} & 0 \\
0 & 2^{1688} & 0 & 0 & 2^{1880} & \cdots & 0 & 0 \\
 & & \ddots & & & \ddots & & \\
0 & 0 & & 0 & 0 & & 0 & 0 \\
0 & 0 & 0 & 2^{1688} & 0 & \cdots & 2^{1880} & 0 \\
0 & 0 & 0 & 0 & N' & \cdots & 0 & 0 \\
 & & & & & \ddots & & \\
0 & 0 & 0 & 0 & 0 & & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \cdots & N' & 0 \\
0 & 0 & 0 & 0 & t'_0 & \cdots & t'_{\mu'-1} & 2^{1848}
\end{pmatrix}.$$

As before, multiplying L_2 from the left by

$$(n',\ -\sigma'_0,\ \dots,\ -\sigma'_{\mu'-1},\ *, \dots, *, 1)$$

shows that this lattice contains a vector

$$(2^{1592} \cdot n',\ -2^{1688}\,\sigma'_0,\ \dots,\ -2^{1688}\,\sigma'_{\mu'-1},\ d'_0, \dots,\ d'_{\mu'-1},\ 2^{1848})$$

where all entries are $\approx 2^{1848}$ and thus has Euclidean norm $\approx \sqrt{2\,\mu' + 2} \cdot 2^{1848}$. We write "$\approx$" instead of "$\leq$" because $s'_i \cdot p_r$ may overflow 2^{1848}. Picking $\mu' = 256/32 + 1 = 9$ gives an instance where the target vector is expected to be shorter than the Gaussian heuristic predicts. However, due to our choice of s'_i, finding a shortest vector might not recover n' exactly but only the top $256 - \varepsilon$ bits for some small ε. We verified this behaviour with our proof of concept implementation which consistently recovers all but $\varepsilon \approx 4$ bits. To recover the remaining bits, we simply perform exhaustive search by computing $\mathsf{SHA\text{-}1}(N, p, q, n, s, n' + \Delta n')$ for all candidates for $\Delta n'$ and comparing against h_r. Overall, under our assumptions, using $\approx (6 + 9) \cdot 2^{32}$ noise-free queries and a trivial amount of computation we can recover n' from Telegram's key exchange. This in turn allows to compute the initial salt. Of course, timing side channels are noisy, suggesting a potentially significantly larger number of queries would be needed to recover sufficiently clean signals for the lattice reduction stage.

c) Extension to other leakage patterns: Our approach can be adapted to check other leakage patterns, e.g. targeting the values in the $\mathcal{L}(\cdot)$ fields. For example, recall that the Telegram servers require $1 \leq \mathcal{L}(N) \leq 16$. We do not know what the servers do when this condition is violated, but discuss possible behaviours:

- Assume the code terminates early, skipping the SHA-1 call. This would result in a timing side channel leaking that the three most significant bits of $\mathcal{L}(N)$ are zero when the SHA-1 call is triggered.
- Assume the code does not terminate early but the Telegram servers feed between 88 and 104 bytes to SHA-1. This would not produce a timing leak. That is, SHA-1 hashes data in blocks with its running time depending on the number of blocks processed. It has a block size of 64 bytes, and its padding algorithm (i.e. see algorithm SHA-pad in Section II-B) insists on adding at least 8 bytes of length and 1 byte of padding. Thus up to 55 full bytes are hashed as one block, then 119, 183, and 247, cf. [47], [57] for works exploiting this. Telegram's format checking restricts accepted length to between 88 and 104 bytes, i.e. all valid payloads lead to calls to the SHA-1 compression function on two blocks.
- Assume the code performs a dummy SHA-1 call on all data received, say, minus the received digest. This would lead to calls to the SHA-1 compression function on three blocks and a timing side channel leaking the three most significant bits of $\mathcal{L}(N)$, by distinguishing between $\mathcal{L}(N) > 16$ and $\mathcal{L}(N) \leq 16$.

Now, suppose Telegram's servers do leak whether the three most significant bits of $\mathcal{L}(N)$ are zero without first checking the header. On the one hand, this would reduce the query complexity because the target event is now expected to happen with probability 2^{-3}. On the other hand, this increases the cost of lattice reduction, as we now need to find shortest vectors in lattices of larger dimension. Information theoretically, we

[33]Technically, given the knowledge of h_r and that it is a hash of the remaining inputs save p_r the information theory limit does not apply and algorithms exist to exploit this additional information [55]. However, for simplicity we forgo a discussion of this variant here.

[34]Indeed, we are only interested in 64 bits of n': $n'[0 : 64]$.

need at least $m = 160/3$ samples to recover h_r and thus need to consider finding shortest vectors in a lattice of dimension 110, which is feasible [55]. For n' we can use the same tactic as above for "slicing up" x into n' and p_r to slice up n' into sufficiently small chunks. Alternatively, noting that we only need to recover 64 bits of n' we can simply consider a lattice of dimension ≈ 45, where finding shortest vectors is easy.

2) Recovering the session id: Given the salt, we can recover the session ID using a simple guess and verify approach exploiting the same timing side channel as in Section VI. Here, we simply run our attack from Section VI but this time we use a known plaintext block m_i in order to validate our guesses about the value of m_1 (which is now partially unknown). That is, for all 2^{64} choices of the session ID, and given the recovered salt value, we can construct a candidate for m_1. Then for known m_{i-1}, m_i, we construct $c_1 \mid c^\star$ as before, with $c^\star = m_{i-1} \oplus c_i \oplus m_1$. If our guess for the session ID was correct, then decrypting $c_1 \mid c^\star$ results in a plaintext having a second block of the form:

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

We can then check if the observed behaviour on processing the ciphertext is consistent with the known value $m_i \oplus c_{i-1} \oplus c_1$. If our choice of the session ID (and therefore m_1) is correct, this will always be the case. If our guess is incorrect then $m^\star$ can be assumed to be uniformly random.

In more detail, assume our timing side channel leaks 32 bits of plaintext from the length field check. Let $m_i^{(j)}$ and $c_i^{(j)}$ be the i-th block in the j-th plaintext and ciphertext respectively. Collect three plaintext-ciphertext pairs s.t.

$$m_i^{(j)} \oplus c_{i-1}^{(j)} \oplus c_1^{(j)}, \ (0 \le j < 3)$$

passes the length check.[35] For each guess of the session ID submit three ciphertexts containing $c^{\star,(j)} = m_{i-1}^{(j)} \oplus c_i^{(j)} \oplus m_1^{(j)}$ as the second block. If our guess for m_1 was correct then all three will pass the length check which is leaked to us by the timing side channel. If our guess for m_1 was incorrect then $E_K^{-1}(c^{\star,(j)} \oplus m_1)$ will output a random block, i.e. such that $E_K^{-1}(c^{\star,(j)} \oplus m_1) \oplus c_1$ passes the length check with probability 2^{-32}. Thus, all three length checks will pass with probability 2^{-96}. In other words, the probability of a false positive is upper-bounded by $2^{64} \cdot 2^{-96} = 2^{-32}$ (i.e. in the worst case we will check and discard $2^{64} - 1$ possible values of session ID before finding the correct one).

3) Breaking server authentication: Recall from Fig. 57 that the $\mathsf{key}, \mathsf{iv}$ pair used to encrypt g^a and g^b are derived from s (sent in the clear) and n'. Since the attack in Appendix F1 recovers n', it can be immediately extended into an attacker-in-the-middle (MitM) attack on the Diffie-Hellman key exchange. That is, knowing n' the attacker can compose the appropriate IGE ciphertext containing some $g^{a'}$ of its choice where it knows a' (and similarly replace g^b coming from the client with $g^{b'}$ for some b' it knows). Both client and server will thus complete their respective key exchanges with the adversary rather than each other, allowing the adversary to break confidentiality and integrity of their communication. However, even in the presence of the side channel that enabled the attack in Appendix F1, the MitM attack is more complicated due to the need to complete it before the session between client and server times out. This may be feasible under some of the alternative leakage patterns discussed earlier but unlikely to be realistic when $> 2^{32}$ requests are required to recover n'.

[35]A different index i can be used within each ciphertext.

G. Proof of concept implementation

```
#!/usr/bin/env sage
"""
"""
from sage.all import ZZ, matrix, set_random_seed, log, pi, e, sqrt, RR, ceil
from fpylll import IntegerMatrix, BKZ, FPLLL
from fpylll.algorithms.bkz2 import BKZReduction as BKZ2

"""
Configuration
"""

header_len = 32  # 0xec5ac983
N_len = 16 * 8 + 8  # length field
p_len = 8 * 8 + 8  # length field
q_len = 8 * 8 + 8  # length field
nonce_len = 128
server_nonce_len = 128
new_nonce_len = 256
sha1_len = 20 * 8
total_len = 255 * 8
pad_len = total_len - (
    sha1_len + header_len + N_len + p_len + q_len + nonce_len + server_nonce_len + new_nonce_len
)
leak_bits = 32
leak_pos = total_len - sha1_len - leak_bits

# https://github.com/DrKLO/Telegram/blob/f41b228a111e304c2505a86c7cc8b448eaecaf6f/TMessagesProj/jni/tgnet/Handshake.cpp#L398
# import rsa  ## pip install rsa
# for pubkey in pubkeys:
#    N = ZZ(rsa.PublicKey.load_pkcs1(pubkey).n)
#    print(hex(N))

N_ = ZZ(
    "0xaeec36c8ffc109cb099624685b9781"
    "5415657bd76d8c9c3e398103d7ad16c9"
    "bba6f525ed0412d7ae2c2de2b44e77d7"
    "2cbf4b7438709a4e646a05c43427c7f1"
    "84debf72947519680e651500890c6832"
    "796dd11f772c25ff8f576755afe055b0"
    "a3752c696eb7d8da0d8be1faf38c9bdd"
    "97ce0a77d3916230c4032167100edd0f"
    "9e7a3a9b602d04367b689536af0d64b6"
    "13ccba7962939d3b57682beb6dae5b60"
    "8130b2e52aca78ba023cf6ce806b1dc4"
    "9c72cf928a7199d22e3d7ac84e47bc94"
    "27d0236945d10dbd15177bab413fbf0e"
    "dfda09f014c7a7da088dde9759702ca7"
    "60af2b8e4e97cc055c617bd74c3d9700"
    "8635b98dc4d621b4891da9fb04730479"
    "27"
)

N_ = ZZ(
    "0xbdf2c77d81f6afd47bd30f29ac76e5"
    "5adfe70e487e5e48297e5a9055c9c07d"
    "2b93b4ed3994d3eca5098bf18d978d54"
    "f8b7c713eb10247607e69af9ef44f38e"
    "28f8b439f257a11572945cc0406fe3f3"
    "7bb92b79112db69eedf2dc71584a6616"
    "38ea5becb9e23585074b80d57d9f5710"
    "dd30d2da940e0ada2f1b878397dc1a72"
    "b5ce2531b6f7dd158e09c828d03450ca"
    "0ff8a174deacebcaa22dde84ef66ad37"
    "0f259d18af806638012da0ca4a70baa8"
    "3d9c158f3552bc9158e69bf332a45809"
    "e1c36905a5caa12348dd57941a482131"
    "be7b2355a5f4635374f3bd3ddf5ff925"
    "bf4809ee27c1e67d9120c5fe08a9de45"
    "8b1b4a3c5d0a428437f2beca81f4e2d5"
    "ff"
)

N_ = ZZ(
    "0xb3f762b739be98f343eb1921cf0148"
    "cfa27ff7af02b6471213fed9daa00989"
    "76e667750324f1abcea4c31e43b7d11f"
    "1579133f2b3d9fe27474e462058884e5"
    "e1b123be9cbbc6a443b2925c08520e73"
    "25e6f1a6d50e117eb61ea49d2534c8bb"
    "4d2ae4153fabe832b9edf4c5755fdd8b"
    "19940b81d1d96cf433d19e6a22968a85"
    "dc80f0312f596bd2530c1cfb28b5fe01"
    "9ac9bc25cd9c2a5d8a0f3a1c0c79bcca"
    "524d315b5e21b5c26b46babe3d75d06d"
    "1cd33329ec782a0f22891ed1db42a1d6"
    "c0dea431428bc4d7aabdcf3e0eb6fda4"
    "e23eb7733e7727e9a1915580796c5518"
    "8d2596d2665ad1182ba7abf15aaa5a8b"
    "779ea996317a20ae044b820bff35b6e8"
    "a1"
)

N_ = ZZ(
    "0xbe6a71558ee577ff03023cfa17aab4e"
    "6c86383cff8a7ad38edb9fafe6f323f2"
    "d5106cbc8cafb83b869cffd1ccf121cd"
    "743d509e589e68765c96601e813dc5b9"
    "dfc4be415c7a6526132d0035ca33d6d6"
    "075d4f535122a1cdfe017041f1088d14"
    "19f65c8e5490ee613e16dbf662698c0f"
    "54870f0475fa893fc41eb55b08ff1ac2"
    "11bc045ded31be27d12c96d8d3cfc6a7"
    "ae8aa50bf2ee0f30ed507cc2581e3dec"
    "56de94f5dc0a7abee0be990b893f2887"
    "bd2c6310a1e0a9e3e38bd34fded25415"
    "08dc102a9c9b4c95effd9dd2dfe96c29"
    "be647d6c69d66ca500843cfaed6e4401"
    "96f1dbe0e2e22163c61ca48c79116fa7"
    "7216726749a976a1c4b0944b5121e8c0"
    "1"
)


def sample_c(stage=1):
    """
    Sample a fresh challenge ciphertext and return private and public part.
    """
    header = 0xEC5AC983
    N = ZZ.random_element(2 ** N_len)
    p = ZZ.random_element(2 ** p_len)
    q = ZZ.random_element(2 ** q_len)
    nonce = ZZ.random_element(2 ** nonce_len)
    server_nonce = ZZ.random_element(2 ** server_nonce_len)
    new_nonce = ZZ.random_element(2 ** new_nonce_len)
    pad = ZZ.random_element(2 ** pad_len)
    sha1 = ZZ.random_element(2 ** sha1_len)

    x = new_nonce * 2 ** pad_len + pad
    x_len = new_nonce_len + pad_len
    y = sha1
    y_len = sha1_len

    gamma, gamma_len = 0, 0
    for v, s in (
        (server_nonce, server_nonce_len),
        (nonce, nonce_len),
        (q, q_len),
        (p, p_len),
        (N, N_len),
        (header, header_len),
    ):
        gamma += v * 2 ** gamma_len
        gamma_len += s

    if stage == 2:
        gamma += 2 ** (total_len - y_len - x_len) * y
        y = 0

    c = 2 ** (total_len - y_len) * y + 2 ** x_len * gamma + x

    return c, gamma


def leak(c, s_len):
    """
    Simulate RSA decryption leak
    """
    s = ZZ.random_element(2 ** s_len)
    d = s * c % N_
    d = (d // 2 ** leak_pos) % 2 ** leak_bits
    return s, d


def instancef(s_len, nleaks=(160 // leak_bits) + 1, stage=1):
    c, gamma = sample_c(stage=stage)
    leaks = []

    for _ in range(nleaks):
        s, d = leak(c, s_len=s_len)
        leaks.append((s, d))

    return c, (gamma, tuple(leaks))


def latticef(gamma, leaks, stage=1):
    m = len(leaks)
    d = 2 * m + 2
    A = matrix(ZZ, d, d)
    if stage == 1:
        A[0, 0] = 2 ** (leak_pos - sha1_len)
    else:
        A[0, 0] = 2 ** (leak_pos - new_nonce_len)
    A[-1, -1] = 2 ** (leak_pos - 2)
    for i, (si, li) in enumerate(leaks):
        if stage == 1:
            A[0, m + i + 1] = (si * 2 ** (total_len - sha1_len)) % N_  # noqa: E201
        else:
            A[0, m + i + 1] = (si * 2 ** pad_len) % N_  # noqa: E201
        A[i + 1, i + 1] = 2 ** (2 * leak_pos + leak_bits - ceil(log(N_, 2)))  # noqa: E201
        A[i + 1, m + i + 1] = 2 ** (leak_pos + leak_bits)  # noqa: E201
        A[m + i + 1, m + i + 1] = N_
        A[-1, m + i + 1] = (
            si * 2 ** (new_nonce_len + pad_len) * gamma % N_  # noqa: E201
            - 2 ** leak_pos * li
            - 2 ** (leak_pos - 1)
        ) % N_  # balance mod 2**leak_pos

    return A


def cut(A, log_factor):
    for i in range(A.nrows()):
        for j in range(A.ncols()):
            A[i, j] = A[i, j] // 2 ** log_factor
    return A


def estimate(gamma, leaks, stage=1):
    logN_ = log(N_, 2)
    m = len(leaks)
    d = 2 * m + 2
    if stage == 1:
        log_vol = (
            (leak_pos - sha1_len)
            + m * (2 * leak_pos + leak_bits - logN_)
            + m * logN_
            + (leak_pos - 2)
        )
    else:
        log_vol = (
            (leak_pos - new_nonce_len)
            + m * (2 * leak_pos + leak_bits - logN_)
            + m * logN_
            + (leak_pos - 2)
        )

    gh = RR(log(sqrt(d / 2 / pi / e), 2) + (log_vol / d))
    nm = RR(log(sqrt(d), 2) + leak_pos - 1)

    return (gh, nm, gh - nm)


def extract_y(c):
    return c // 2 ** (total_len - sha1_len)


def extract_x(c):
    return (c // 2 ** (pad_len)) % 2 ** new_nonce_len


def benchmark(seed, nleaks, block_size=2, stage=1):
    set_random_seed(seed)

    if stage == 1:
        s_len = 256
    else:
        s_len = leak_pos - pad_len
    print(s_len)

    c, (gamma, leaks) = instancef(s_len=s_len, nleaks=nleaks, stage=stage)
    gh, nm, df = estimate(gamma, leaks, stage=stage)
    A = latticef(gamma, leaks, stage=stage)

    if stage == 1:
        log_factor = leak_pos - sha1_len - 64
        A = cut(A, log_factor)
    else:
        log_factor = leak_pos - new_nonce_len - 64
        A = cut(A, log_factor)

    scale = A[0, 0]
    target = A[-1, -1]

    L = A.LLL()
    if block_size > 2:
        FPLLL.set_random_seed(ZZ.random_element(2 ** 64))
        L = IntegerMatrix.from_matrix(L)
        BKZ2(L)(BKZ.EasyParam(block_size, flags=BKZ.VERBOSE))
        L = L.to_matrix(matrix(A.nrows(), A.ncols()))

    print(
        (
            "nrows: {nrows:3d}, lf: {lf:3d}, tv: {tv:4d}, GH: 2^{gh:.1f}, E[|v|]: 2^{nm:.1f}, "
            "|v|: 2^{rs:.1f}, GH/E[|v|]: 2^{df:.1f}"
        ).format(
            tv=log(target, 2),
            gh=float(gh),
            nm=float(nm),
            df=float(df),
            lf=log_factor,
            nrows=A.nrows(),
            rs=float(log_factor + log(L[0].norm(), 2)),
        )
    )

    if stage == 1:
        extract = extract_y
    else:
        extract = extract_x

    for i in range(L.nrows()):
        # print(hex(abs(L[i][-1])), hex(abs(target)), hex(abs(L[i][0] // scale)), hex(extract_y(c)))
        if abs(L[i][-1]) == target:
            return hex(abs(L[i][0] // scale)), hex(extract(c)), L

    print("Not found")
    return L[0][0] // scale, extract(c), L


# Local Variables:
# conda-project-env-path: "sagemath"
# fill-column: 100
# End:
```

H. Timing experiment code

Assume Telegram desktop version 2.4.11.[36] The experiment code (`experiment.h` and `experiment.cpp`, also attached to the electronic version of the document) was added to `Telegram/SourceFiles/core/` and called from `Application::run()` inside `application.cpp`. We use `cpucycles`[37] to measure the running time.

```
//
//  experiment.cpp
//  not part of Telegram codebase
//

#include "experiment.h"

#include <chrono>
#include "base/bytes.h"
#include <openssl/rand.h>
#include <iostream>
```

[36]https://github.com/telegramdesktop/tdesktop/tree/v2.4.11
[37]https://www.ecrypt.eu.org/ebats/cpucycles.html

```
#include <fstream>
#include "cpucycles.h"

#include "mtproto/session_private.h"
#include "mtproto/details/mtproto_bound_key_creator.h"
#include "mtproto/details/mtproto_dcenter.h"
#include "mtproto/details/mtproto_dump_to_text.h"
#include "mtproto/details/mtproto_rsa_public_key.h"
#include "mtproto/session.h"
#include "mtproto/mtproto_rpc_sender.h"
#include "mtproto/mtproto_dc_options.h"
#include "mtproto/connection_abstract.h"
#include "base/openssl_help.h"
#include "base/qthelp_url.h"
#include "base/unixtime.h"
#include "zlib.h"

int _numTrials = 10000;
int _msgLength = 1024;
bool _samePacket = true;
bool _runOnInit = false;
bool _cpucycles = false;

namespace MTP {
namespace details {

constexpr auto kMaxMessageLength = 16 * 1024 * 1024;
constexpr auto kIntSize = static_cast<int>(sizeof(mtpPrime));
AuthKeyPtr _encryptionKey;
MTP::AuthKey::Data _authKey;
uint64 _keyId;
ConnectionPointer _connection;

// adapted from DcKeyCreator::dhClientParamsSend
/* generate random authKey and set corresponding encryption key and id */
void generateEncryptionKey() {
    auto key = bytes::vector(256);
    bytes::set_random(key);
    AuthKey::FillData(_authKey, bytes::make_span(key));
    _encryptionKey = std::make_shared<AuthKey>(_authKey);
    _keyId = _encryptionKey->keyId();
}

// plain copy of SessionPrivate::ConstTimeIsDifferent
/* used for SHA checks */
[[nodiscard]] bool ConstTimeIsDifferent(
        const void *a,
        const void *b,
        size_t size) {
    auto ca = reinterpret_cast<const char*>(a);
    auto cb = reinterpret_cast<const char*>(b);
    volatile auto different = false;
    for (const auto ce = ca + size; ca != ce; ++ca, ++cb) {
        different = different | (*ca != *cb);
    }
    return different;
}

// copy from SerializedRequest, only MTProto version 2.0 and version 0 of transport protocol
/* generate padding size in units (1U = 4B) */
uint32 CountPaddingPrimesCount(uint32 requestSize) {
    auto result = ((8 + requestSize) & 0x03)
        ? (4 - ((8 + requestSize) & 0x03))
        : 0;

    // At least 12 bytes of random padding.
    if (result < 3) {
        result += 4;
    }

    return result;
}

// next 3 methods adapted from SessionPrivate::sendSecureRequest, only MTProto version 2.0

/* helper method to generate random plaintext w/ padding */
bytes::span preparePlaintext(uint32_t msgLength) {
    Expects(msgLength >= 4 && msgLength % 4 == 0);

    auto padLength = CountPaddingPrimesCount(msgLength/4) * 4;
    // 24B external header = 8B auth_key_id + 16B msg_key
    // 32B internal header = 8B salt + 8B session_id + 8B msg_id + 4B seq_no + 4B msg_length
    auto length = 24 + 32 + msgLength + padLength;
    //LOG(("Generated msgLength = %1, padLength = %2, length = %3.").arg(msgLength).arg(padLength).arg(length));

    // random plaintext = internal header + message + padding
    auto plaintext = bytes::vector(32 + msgLength + padLength);
    bytes::set_random(plaintext);
    return plaintext;
}

/* helper method to prepare packet from given plaintext
   msgLength field will be overriden according to valid value */
mtpBuffer preparePacket(bool valid, uint32_t msgLength, bytes::span plaintext) {
    int plaintextLength = plaintext.size();
    Expects(plaintextLength >= 48 && plaintextLength % 16 == 0);

    // msg_key = SHA-256(auth_key[96:128] || message)[8:24]

    uchar encryptedSHA256[32];
    MTPint128 &msgKey(*(MTPint128*)(encryptedSHA256 + 8));

    SHA256_CTX msgKeyLargeContext;
    SHA256_Init(&msgKeyLargeContext);
    SHA256_Update(&msgKeyLargeContext, _encryptionKey->partForMsgKey(false), 32);  // encrypt to self
    SHA256_Update(&msgKeyLargeContext, plaintext.data(), plaintext.size());
    SHA256_Final(encryptedSHA256, &msgKeyLargeContext);

    if (!valid) {
        msgLength = kMaxMessageLength + 1;  // over the limit
    }
    memcpy(plaintext.data() + 28, &msgLength, 4);

    auto fullSize = plaintext.size() / sizeof(mtpPrime);  // should equal length/4 - 6
    auto packet = _connection->prepareSecurePacket(_encryptionKey->keyId(), msgKey, fullSize);
    const auto prefix = packet.size();  // 8 due to tcp prefix and resizing
    packet.resize(prefix + fullSize);

    // adapted from aesIgeEncrypt(plaintext.data(), &packet[prefix], fullSize * sizeof(mtpPrime), _encryptionKey, msgKey) call
    MTPint256 aesKey, aesIV;
    _encryptionKey->prepareAES(msgKey, aesKey, aesIV, false);  // encrypt to self
    aesIgeEncryptRaw(plaintext.data(), &packet[prefix], fullSize * sizeof(mtpPrime),
                     static_cast<const void*>(&aesKey), static_cast<const void*>(&aesIV));

    return packet;
}

/* generate packet with given msgLength (w/o TCP prefix) that can be processed client-side
   2 cases to distinguish:
   valid = msgLength check passes but SHA check fails
   !valid = msgLength check doesn't pass */
mtpBuffer preparePacket(bool valid, uint32_t msgLength) {
    return preparePacket(valid, msgLength, preparePlaintext(msgLength));
}

// copy of SessionPrivate::handleReceived, only MTProto version 2.0, network connection calls commented out
/* process received packet */
void handlePacket(mtpBuffer intsBuffer) {
    Expects(_encryptionKey != nullptr);

    /* network connection management */
    //onReceivedSome();

    /* assume packets come in one by one (usually the case) */
    //while (!_connection->received().empty()) {
    //    auto intsBuffer = std::move(_connection->received().front());
    //    _connection->received().pop_front();

    constexpr auto kExternalHeaderIntsCount = 6U; // 2 auth_key_id, 4 msg_key
    constexpr auto kEncryptedHeaderIntsCount = 8U; // 2 salt, 2 session, 2 msg_id, 1 seq_no, 1 length
    constexpr auto kMinimalEncryptedIntsCount = kEncryptedHeaderIntsCount + 4U; // + 1 data + 3 padding
    constexpr auto kMinimalIntsCount = kExternalHeaderIntsCount + kMinimalEncryptedIntsCount;
    auto intsCount = uint32(intsBuffer.size());
    auto ints = intsBuffer.constData();
    if ((intsCount < kMinimalIntsCount) || (intsCount > kMaxMessageLength / kIntSize)) {
        LOG(("TCP Error: bad message received, len %1").arg(intsCount * kIntSize));
        TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(ints, intsCount * kIntSize).str()));

        // return restart();
        return;
    }
    if (_keyId != *(uint64*)ints) {
        LOG(("TCP Error: bad auth_key_id %1 instead of %2 received").arg(_keyId).arg(*(uint64*)ints));
        TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(ints, intsCount * kIntSize).str()));

        // return restart();
        return;
    }

    auto encryptedInts = ints + kExternalHeaderIntsCount;
    auto encryptedIntsCount = (intsCount - kExternalHeaderIntsCount) & ~0x03U;
    auto encryptedBytesCount = encryptedIntsCount * kIntSize;
    auto decryptedBuffer = QByteArray(encryptedBytesCount, Qt::Uninitialized);
    auto msgKey = *(MTPint128*)(ints + 2);

    // version 2.0 only
    aesIgeDecrypt(encryptedInts, decryptedBuffer.data(), encryptedBytesCount, _encryptionKey, msgKey);

    auto decryptedInts = reinterpret_cast<const mtpPrime*>(decryptedBuffer.constData());
    auto serverSalt = *(uint64*)&decryptedInts[0];
    auto session = *(uint64*)&decryptedInts[2];
    auto msgId = *(uint64*)&decryptedInts[4];
    auto seqNo = *(uint32*)&decryptedInts[6];
    auto needAck = ((seqNo & 0x01) != 0);

    auto messageLength = *(uint32*)&decryptedInts[7];
    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();
        return;
    }
    auto fullDataLength = kEncryptedHeaderIntsCount * kIntSize + messageLength; // Without padding.

    // Can underflow, but it is an unsigned type, so we just check the range later.
    auto paddingSize = static_cast<uint32>(encryptedBytesCount) - static_cast<uint32>(fullDataLength);

    constexpr auto kMinPaddingSize = 12U;
    constexpr auto kMaxPaddingSize = 1024U;
    auto badMessageLength = (paddingSize < kMinPaddingSize || paddingSize > kMaxPaddingSize);

    std::array<uchar, 32> sha256Buffer = { { 0 } };

    SHA256_CTX msgKeyLargeContext;
    SHA256_Init(&msgKeyLargeContext);
    SHA256_Update(&msgKeyLargeContext, _encryptionKey->partForMsgKey(false), 32);
    SHA256_Update(&msgKeyLargeContext, decryptedInts, encryptedBytesCount);
    SHA256_Final(sha256Buffer.data(), &msgKeyLargeContext);

    constexpr auto kMsgKeyShift = 8U;
    if (ConstTimeIsDifferent(&msgKey, sha256Buffer.data() + kMsgKeyShift, sizeof(msgKey))) {
        LOG(("TCP Error: bad SHA256 hash after aesDecrypt in message"));
        TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(encryptedInts, encryptedBytesCount).str()));

        // return restart();
        return;
    }

    if (badMessageLength || (messageLength & 0x03)) {
        LOG(("TCP Error: bad msg_len received %1, data size: %2").arg(messageLength).arg(encryptedBytesCount));
        TCP_LOG(("TCP Error: bad message %1").arg(Logs::mb(encryptedInts, encryptedBytesCount).str()));
        // return restart();
        return;
    }

    // rest of code cut, should never reach here
    LOG(("EXP: Something went wrong."));
}

}
} // namespace MTP::details

/* write the timing data to log file
   settings -> typing "viewlogs" shows the folder */
void writeToFile(std::string createTime, std::string msg) {
    std::ofstream timeFile;
    std::string c_string;
    if (getCpucycles()) {
        c_string = "_c";
    } else {
        c_string = "";
    }
    std::string path = cWorkingDir().toStdString() + createTime + "_" + std::to_string(_msgLength)
        + "_" + std::to_string(_samePacket) + "_" + std::to_string(_numTrials) + c_string + ".csv";
    timeFile.open(path.data(), std::ios_base::app);
    timeFile << msg.data();
    timeFile.close();
}

/* set experiment parameters */
void setNumTrials(int numTrials) {
    _numTrials = numTrials;
}

void setMsgLength(int msgLength) {
    _msgLength = msgLength;
}

void setSamePacket(bool samePacket) {
    _samePacket = samePacket;
}

void setRunOnInit(bool runOnInit) {
    _runOnInit = runOnInit;
}

void setCpucycles(bool cpucycles) {
    _cpucycles = cpucycles;
}

int getNumTrials() {
    return _numTrials;
}

int getMsgLength() {
    return _msgLength;
}

bool getSamePacket() {
    return _samePacket;
}

bool getRunOnInit() {
    return _runOnInit;
}

bool getCpucycles() {
    return _cpucycles;
}

/* generate a number of packets to process client-side
   and time processing to first error (in microseconds) */
std::string doExperiment() {
    const auto createTime = QDateTime::currentDateTime();
    auto timeFile = createTime.toString("yyyy-MM-dd_hh-mm-ss-zzz");
    LOG(("EXP: %1: Do %2 trials with message length %3B.").arg(timeFile).arg(_numTrials).arg(_msgLength));

    MTP::details::generateEncryptionKey();
    bytes::span plaintext;
    mtpBuffer packet;

    if (_samePacket) {
        //LOG(("EXP: Using a single plaintext."));
        plaintext = MTP::details::preparePlaintext(_msgLength);
    }

    for (int i = 0; i < 2 * _numTrials; i++) {
        bool valid = i < _numTrials;
        if (_samePacket) {
            if (i == 0 || i == _numTrials) {
                packet = MTP::details::preparePacket(valid, _msgLength, plaintext);
            }
        } else {
            packet = MTP::details::preparePacket(valid, _msgLength);
        }

        // shuffling data around between the two methods
        auto bufferSize = packet.size() - 2; // w/o tcp prefix
        auto buffer = mtpBuffer(bufferSize);
        memcpy(buffer.data(), packet.data() + 2, bufferSize * sizeof(mtpPrime));

        std::string diff_str;
        if (getCpucycles()) {
            auto t1 = cpucycles();
            MTP::details::handlePacket(buffer);
            auto t2 = cpucycles();
            auto diff = t2 - t1;
            diff_str = std::to_string(diff);
        } else {
            auto t1 = std::chrono::steady_clock::now();
            MTP::details::handlePacket(buffer);
            auto t2 = std::chrono::steady_clock::now();
            std::chrono::duration<double, std::micro> diff = t2 - t1;
            diff_str = std::to_string(diff.count());
        }

        writeToFile(timeFile.toStdString(), std::to_string(valid)+","+diff_str+"\n");
    }

    if (getRunOnInit()) {
        exit(0);
    }

    return timeFile.toStdString();
}
```