Four Attacks and a Proof for Telegram*

Martin R. Albrecht[1], Lenka Mareková[2], Kenneth G. Paterson[3], and Igors Stepanovs[3]

[1] King's College London
martin.albrecht@kcl.ac.uk
[2] Information Security Group, Royal Holloway, University of London
lenka.marekova.2018@rhul.ac.uk
[3] Applied Cryptography Group, ETH Zurich
{kenny.paterson,istepanovs}@inf.ethz.ch

3 April 2023

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 that appeared at IEEE S&P 2022.

Contents

1 Introduction

Telegram is a chat platform that in January 2021 reportedly had 500M monthly users [Tel21a]. 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 [EHM17] and participants of protests [ABJM21]. In particular, it is reported in [EHM17,ABJM21] 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 [JO16] an attack against the IND-CCA security of MTProto 1.0 was reported, in response to which the protocol was updated. In [SK17] a replay attack based on improper validation in the Android client was reported. Similarly, [Kob18] reports input validation bugs in Telegram's Windows Phone client. Recently, in [MV21] 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 [JO16,Kob18] as opposed to cloud chats. In contrast, in [ABJM21] 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).

1.1 Contributions

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

Security model. Starting from the observation that MTProto entangles the keys of the two channel directions, in Section 3 we develop a bidirectional security model for two-party secure channels that allows an adversary full control over generating and delivering ciphertexts from/to either party (client or server). The model assumes that the two parties start with a shared key and use stateful algorithms. Our security definitions come in two flavours, one capturing confidentiality, the other integrity. 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 (neither of which we consider appropriate for secure messaging). This caters for situations where the secure channel operates over an unreliable transport protocol, but where the channel is designed to recover from accidental errors in message delivery as well as from certain permitted adversarial behaviours.

This is done technically by introducing the concept of *support functions*, inspired by the support predicates recently introduced by [FGJ20] 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, reordering or dropping 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 [Tel21b].

Formal model of MTProto. In Section 4, we provide a detailed formal model of Telegram's symmetric encryption. Our model is computational and does not abstract away the building blocks used in Telegram. This in itself is a non-trivial task as no formal specification exists and behaviour can only be derived from official (but incomplete) documentation and from observation; moreover different clients do not have the same behaviour.

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

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

In particular, we verified in practice that current implementations allow an attacker on the network to reorder messages from a client to the server, with the transcript on the client being updated later to reflect the attacker-altered server's view. We stress, though, that this trivial yet practical attack is not inherent in MTProto and can be avoided by updating the processing of message metadata in Telegram's servers. The consequences of such an attack can be quite severe, as we discuss further in Section 4.2.

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 drops, 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 5 that our slight variant of MTProto achieves channel confidentiality and integrity in our model, under certain assumptions on the components used in its construction. As described in Section 1.2, Telegram has implemented our proposed alterations so that there can be some assurances about MTProto as currently deployed.[4]

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 concrete security analysis. This results in security theorems that quantitatively relate the confidentiality and integrity of MTProto as a secure channel to the security of its underlying cryptographic components. Our main security results, Theorems 1 and 2 and Corollaries 1 and 2, provide confidentiality and integrity bounds containing terms equivalent to $\approx q/2^{64}$ where q is the number of queries an attacker makes. We discuss this further in Section 5.

However, our security proofs rely on several assumptions about cryptographic primitives that, while plausible, have not been considered in the literature. In more detail, due to the way Telegram makes use of SHA-256 as a MAC algorithm and as a KDF, we have to rely on the novel assumption that the block cipher SHACAL-2 underlying the SHA-256 compression function is a leakage-resilient PRF under related-key attacks, where "leakage-resilient" means that the adversary can choose a part of the key. Our proofs rely on two distinct variants of such an assumption. In Appendix F we show that these assumptions hold in the ideal cipher model, but further cryptanalysis is needed to validate

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

them for SHACAL-2. For similar reasons, we also require a dual-PRF assumption of SHACAL-2. We stress that such assumptions are likely necessary for our or any other computational security proofs for MTProto. This is due to the specifics of how MTProto uses SHA-256 and how it constructs keys and tags from public inputs and overlapping key bits of a master secret. Given the importance of Telegram, these assumptions provide new, significant cryptanalysis targets as well as motivate further research on related-key attacks.

Besides using SHA-256 as a MAC algorithm and a KDF, MTProto also uses SHA-1 to compute a key identifier. This does not lead to length-extension attacks because in each use case either the input is required to have a fixed length, or the output gets truncated. The latter technique was previously studied as ChopMD [CDMP05] and employed to build AMAC [BBT16]. But rather than applying these results to show that the design of the MAC algorithm prevents forgeries, our proofs rely on an observation that even if length-extension attacks were possible, it would still not lead to breaking the security of the overall scheme. This is true because the plaintext encoding format of MTProto mandates the presence of certain metadata in the first block of the encrypted payload.

Attacks. We present further implementation attacks against Telegram in Sections 6 and 7. 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.[5] The attack takes inspiration from an attack on SSH [APW09]. 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 recover m_1 and thereby enable our plaintext-recovery attack, in Section 7 we chain it with another attack on the server-side implementation of Telegram's key exchange protocol. This attack exploits how Telegram servers process RSA ciphertexts. While the exploited behaviour was confirmed by the Telegram developers, we did not verify it with an experiment.[6] It uses a combination of lattice reduction and Bleichenbacher-like techniques [Ble98]. This attack actually breaks server authentication – allowing a MiTM attack – assuming the attack can be completed before a session times out. But, more germanely, it also allows us to recover the id field. This essentially reduces the overall security of Telegram to guessing the 64-bit salt field. Details can be found in Section 7. We stress, though, that even if all the assumptions that we make in Section 7 are met, our exploit chain (Section 6, Section 7) – 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 8.

Remark 1. This is the full version of the paper published at IEEE S&P 2022 [AMPS22]. The proofs referred to in [AMPS22, Section V] are contained in full here, and can be found in Appendices E and F and in Section 5 (in particular Sections 5.5 and 5.6). We have also expanded the content of several other sections as follows. Section 3 defining bidirectional channels, orig. [AMPS22, Section III], was expanded with more context and illustrating examples. Section 6.1 on the timing attack, orig. [AMPS22, Section VI], contains the code samples for all affected Telegram clients. Section 7 on the key exchange attack, orig. [AMPS22, Appendix A], was significantly expanded and contains an overview of the key exchange protocol as well as the attack in detail. This work also contains several new appendices:

[5] We note that Telegram's TDLib [Tel20d] library manages to avoid this leak [Tel21g].

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

Appendices A to C expand and help to position our new channels framework, while Appendices D and G give more details about the Telegram protocol and the implementation of our attacks.

1.2 Disclosure

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

2 Preliminaries

2.1 Notational conventions

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 list 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. If tr is a list then $\mathsf{tr}[i]$ denotes the element of this list that is indexed by i, where the index is 0-based. We let $\perp \notin \{0,1\}^*$ be an error code that indicates rejection, and we may also use $\lightning \notin \{0,1\}^* \cup \{\perp\}$ when another distinct error code is needed. Uninitialised integers are assumed to be initialised to 0, Booleans to false, strings to ε, sets to $\emptyset$, and lists to $[]$. Each element of a table is assumed to be initialised to $\perp$, indicating that it is empty. 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.

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. 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 $\mathbf{abort}(x_1, \ldots)$ is used to immediately halt the algorithm with output $(x_1, \ldots)$. Adversaries are algorithms. Besides using $\perp$ as an error code, we also let oracles explicitly return $\perp$ if they would have otherwise terminated with no output. We require that adversaries never pass $\perp$ as input to their oracles. If any of inputs taken by an adversary A is $\perp$, then all of its outputs are $\perp$.

Security games and reductions. We use the code-based game-playing framework of [BR06]. (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 $\mathsf{Adv}(\mathcal{D}) = \Pr[\,d' = 1 \,|\, d = 1\,] - \Pr[\,d' = 1 \,|\, d = 0\,]$. As part of our reductions, the intermediary games (e.g. Fig. 39) use the following colour-coding: grey for equivalent but expanded code and green for the code added for the transitions between games; the adversaries constructed for the transitions (e.g. Fig. 36) use orange to mark the changes in the code of the simulated reduction games.

2.2 Standard definitions

Fundamental Lemma of Game Playing. In our game-hopping proofs, we frequently make use of the Fundamental Lemma of Game Playing [BR06]. Suppose that the games G_i and G_{i+1} are identical until the flag bad is set. Then we have

$$\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}] \leq \Pr[\mathsf{bad}^{\mathrm{G}_i}] = \Pr[\mathsf{bad}^{\mathrm{G}_{i+1}}],$$

where $\Pr[\mathsf{bad}^{\mathrm{G}}]$ denotes the probability of setting the flag bad in game G.

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.

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

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 $\mathsf{E.Inv}(\mathit{ek}, \mathsf{E.Ev}(\mathit{ek}, x)) = x$ for all $\mathit{ek} \in \mathsf{E.Keys}$ 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 $\mathsf{E.Ev}(\mathit{ek}, \cdot)$ and $\mathsf{E.Inv}(\mathit{ek}, \cdot)$ respectively.

One-time PRF-security of function family. Consider game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$ of Fig. 2, defined for a function family F and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the OTPRF-security of F is defined as $\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{F}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}\right] - 1$. 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 $\mathsf{F.Ev}(\mathit{fk}, x)$ (if $b = 1$) or a uniformly random element from $\{0,1\}^{\mathsf{F.ol}}$ (if $b = 0$). $\mathcal{D}$ wins if it returns a bit b' that is equal to the challenge bit.

```
Game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{F},\mathcal{D}}$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{D}^{\textsc{RoR}}$
Return $b' = b$

$\textsc{RoR}(x)$   // $x \in \mathsf{F.In}$
$\mathit{fk} \leftarrow\!\!\$\ \mathsf{F.Keys}$
$y_1 \leftarrow \mathsf{F.Ev}(\mathit{fk}, x)$
$y_0 \leftarrow\!\!\$\ \{0,1\}^{\mathsf{F.ol}}$
Return $y_b$
```

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

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

```
Game $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}}$
  $b \leftarrow\!\!\$\ \{0,1\}$
  $b' \leftarrow\!\!\$\ \mathcal{D}^{\mathrm{RoR}}$
  Return $b' = b$

$\mathrm{RoR}(m)$   // $m \in \mathsf{SE.MS}$
  $k \leftarrow\!\!\$\ \{0,1\}^{\mathsf{SE.kl}}$
  $c_1 \leftarrow \mathsf{SE.Enc}(k, m)$
  $c_0 \leftarrow\!\!\$\ \{0,1\}^{\mathsf{SE.cl}(|m|)}$
  Return $c_b$
```

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

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

CBC block cipher mode of operation. Let E be a block cipher. Define the Cipher Block Chaining (CBC) mode of operation as a deterministic symmetric encryption scheme $\mathsf{SE} = \mathsf{CBC}[\mathsf{E}]$ 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 $\mathsf{SE.cl}$ 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.

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

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

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

$\mathsf{IGE}[\mathsf{E}].\mathsf{Dec}(k, c)$
  $K \leftarrow k[0 : \mathsf{E.kl}]$
  $c_0 \leftarrow k[\mathsf{E.kl} : \mathsf{E.kl} + \mathsf{E.ol}]$
  $m_0 \leftarrow k[\mathsf{E.kl} + \mathsf{E.ol} : \mathsf{SE.kl}]$
  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 $\mathsf{CBC}[\mathsf{E}]$ and $\mathsf{IGE}[\mathsf{E}]$ from block cipher E. Consider t as the number of blocks of m (or c), i.e. $m = m_1 \,\|\, \ldots \,\|\, m_t$.

IGE block cipher mode of operation. Let E be a block cipher. Define the Infinite Garble Extension (IGE) mode of operation as $\mathsf{SE} = \mathsf{IGE}[\mathsf{E}]$ as in Fig. 4, 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 6.

IGE was first defined in [Cam78], which claims it has infinite error propagation and thus can provide integrity. This claim was disproved in an attack on Free-MAC [Jut00], which has the same specification as IGE. [Jut00] 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 [Knu00]. A chosen-plaintext attack on ABC that relied on IV reuse between encryptions was described in [BBKN12].

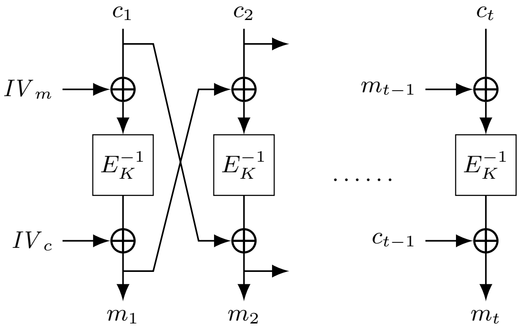


Figure 5. IGE mode decryption, where $c_0 = IV_c$ and $m_0 = IV_m$ are the initial values so decryption can be expressed as $m_i = E_K^{-1}(c_i \oplus m_{i-1}) \oplus c_{i-1}$.

MD transform. Fig. 6 defines the Merkle-Damgård transform as a function family $\mathsf{MD}[h]$ 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 $\mathsf{MD.ol} = \ell$.[7]

```
MD.Ev(k, $x_1 \| \ldots \| x_t$)   // $|x_i| = \ell'$
$H_0 \leftarrow$ k
For $i = 1, \ldots, t$ 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 $h\colon \{0,1\}^{\ell} \times \{0,1\}^{\ell'} \to \{0,1\}^{\ell}$. Right pane: SHA-pad pads SHA-1 or SHA-256 input x to a length that is a multiple of 512 bits.

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 [NIS15]. We refer to their compression functions as $h_{160} : \{0,1\}^{160} \times \{0,1\}^{512} \to \{0,1\}^{160}$ and $h_{256} : \{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$, and to their initial states as IV_{160} and IV_{256}. We can write

$$\mathsf{SHA\text{-}1}(x) = \mathsf{MD}[h_{160}].\mathsf{Ev}(\mathsf{IV}_{160}, \mathsf{SHA\text{-}pad}(x)), \text{ and}$$
$$\mathsf{SHA\text{-}256}(x) = \mathsf{MD}[h_{256}].\mathsf{Ev}(\mathsf{IV}_{256}, \mathsf{SHA\text{-}pad}(x))$$

where SHA-pad is defined in Fig. 6.

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 $z \leftarrow x \mathbin{\hat{+}} y$ splits x and y into 32-bit words and

[7] Traditionally, $\mathsf{MD}[h]$ 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 $\mathsf{MD}[h]$, the key is fixed to a specific IV value.

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 $\mathsf{SHACAL\text{-}1}$ [HN00] be the block cipher defined by $\mathsf{SHACAL\text{-}1.kl} = 512$, $\mathsf{SHACAL\text{-}1.ol} = 160$ such that $h_{160}(k, x) = k \hat{+} \mathsf{SHACAL\text{-}1.Ev}(x, k)$. Similarly, let $\mathsf{SHACAL\text{-}2}$ be the block cipher defined by $\mathsf{SHACAL\text{-}2.kl} = 512$, $\mathsf{SHACAL\text{-}2.ol} = 256$ such that $h_{256}(k, x) = k \hat{+} \mathsf{SHACAL\text{-}2.Ev}(x, k)$. See Fig. 7.

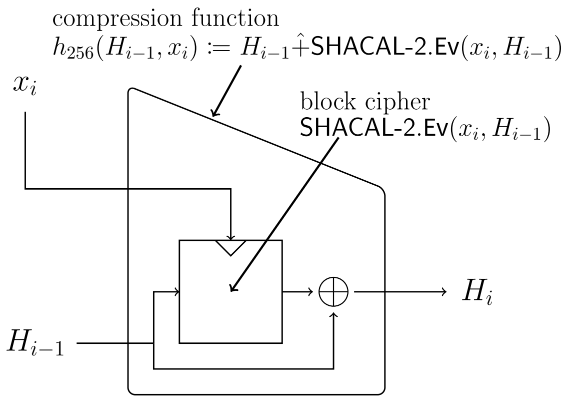

compression function
$h_{256}(H_{i-1}, x_i) \coloneqq H_{i-1} \hat{+} \mathsf{SHACAL\text{-}2.Ev}(x_i, H_{i-1})$
x_i
block cipher
$\mathsf{SHACAL\text{-}2.Ev}(x_i, H_{i-1})$
H_i
H_{i-1}

Figure 7. SHA-256 compression function h_{256} and its underlying block cipher SHACAL-2.

3 Bidirectional channels

3.1 Our formal model in the context of prior work

The choice of a cryptographic primitive. We model Telegram's MTProto protocol as a bidirectional cryptographic channel. A *channel* provides a method for two users to exchange messages, and it is called *bidirectional* [MP17] when each user can both send and receive messages. A *unidirectional channel* provides an interface between two users where only a single user can send messages, and only the opposite user can receive them. Two unidirectional channels can be composed to build a bidirectional channel, but some care needs to be taken to establish what level of security is inherited by the resulting channel [MP17]. A symmetric encryption scheme can be thought of as a special case of a unidirectional channel; it allows to achieve security notions stronger than unforgeability once its encryption and decryption algorithms are modelled as being stateful [BKN02,BKN04].

MTProto uses distinct but related secret keys to send messages in the opposite directions on the channel, so it would not be sufficient to model it as a unidirectional channel. Such an analysis could miss bad interactions between the two directions.

The choice of a security model. Cryptographic security models normally require that channels provide strict in-order delivery of all messages. In the *unidirectional* setting, this means that the receiver should only accept messages in the same order as they were dispatched by the sender. In particular, the channel must prevent all attempts to forge, replay, reorder or drop messages.[8] In the *bidirectional* setting, the in-order delivery is required to hold separately in either direction of communication.

The current version of MTProto 2.0 does not enforce strict in-order delivery. It determines whether a successfully decrypted ciphertext should be accepted based on a complex set of rules. In particular, it happens to allow message reordering, as we describe in Section 4.2. We consider that a vulnerability. So in Section 4.4 we define a slight variant of MTProto 2.0 that enforces strict in-order delivery. Our security analysis in Section 5 is then provided with respect to the fixed version of the protocol. Nevertheless, we set out to choose a formal model for channels that could also potentially be used to

[8] We note that any analysis of such cryptographic restrictions is orthogonal to whether a reliable transport protocol such as TCP is used; we study the security of MTProto against an active, on-path attacker as is standard in secure channel models.

analyse the current version of MTProto 2.0. In particular, we chose a model that could express *both* strict in-order delivery and the message delivery rules that are used in the current version.[9]

No prior work on bidirectional channels defines correctness and security notions that could be used to capture message delivery rules of varied strengths. In the unidirectional setting, [KPB03,BHMS16] each define a hierarchy of multiple security notions where the weakest notion requires only unforgeability and the strongest requires strict in-order delivery. [RZ18,FGJ20] define abstract definitional frameworks for unidirectional channels with fully parametrisable security notions. In this work we extend the *robust channel* framework of [FGJ20], lifting it to the bidirectional setting.

Extending the robust channel framework. The robust channel framework [FGJ20] defines unidirectional correctness and security notions with respect to an arbitrary *support predicate*. When a ciphertext is delivered to the receiver, the corresponding notion uses the support predicate to determine whether the channel is expected to accept this ciphertext or to reject it, i.e. whether this ciphertext is currently *supported*. For example, the notion of correctness in [FGJ20] requires that a channel accepts and correctly decrypts all supported ciphertexts, whereas their notion of integrity requires that a channel rejects all ciphertexts that are not supported. The correctness and security games in [FGJ20] maintain a sequence of ciphertexts that were sent by the sender, and a sequence of ciphertexts that were received and accepted by the receiver. A support predicate takes both sequences as input and it can use them to decide on whether an incoming ciphertext is supported. For completeness, we provide the core definitions of [FGJ20] in Appendix C.2.

We lift the robust channel framework [FGJ20] to the bidirectional setting, and we significantly extend it in other ways. Most importantly, our framework uses more information to determine whether an incoming ciphertext is supported. In particular, we define our correctness and security games to maintain a *support transcript* for each of the two users; this extends the idea of using sequences of sent and received ciphertexts in [FGJ20]. A user's support transcript represents a sequence of events, each entry describing an attempt to send or to receive a message. More precisely, each entry can be thought of as describing one of the following events (stated in terms of some specific plaintext m and/or ciphertext c): "sent c that encrypts m", "failed to send m", "received c, accepted it, and decrypted it as m", "received c and rejected it". In our framework the support transcripts are used by a *support function*; it extends the concept of the support predicate from [FGJ20]. Given the support transcripts of both users as input, a support function in our framework is meant to prescribe the exact behaviour of a channel when a new ciphertext is delivered to either user. Namely, a support function either determines that the incoming ciphertext must be rejected, or it determines that the incoming ciphertext must be accepted *and* a specific plaintext value must be obtained upon decrypting this ciphertext. For example, our notion of correctness is similar in spirit to that of [FGJ20], requiring that a channel accepts and correctly decrypts each plaintext that is not rejected by a support function. The core difference between our correctness notion and that of [FGJ20] is in how these definitions determine whether a specific ciphertext was decrypted "correctly". In our framework the output of a support function prescribes that a specific plaintext value must be obtained, whereas in [FGJ20] the correctness game builds a lookup table to determine that value.

The above example provides an intuition that by defining our support transcripts to contain plaintext messages, we obtain simpler correctness and security definitions when compared to [FGJ20]. But one could also see this as a trade-off between different parts of the formalism, because some complexity that is removed from the correctness and security games might simply be relegated to the step of specifying and analysing a support function. In order to better understand how our framework relates to the robust channel framework, in Appendix C we provide a thorough comparison between the unidirectional variants of our definitions and those of [FGJ20].

Relation to secure messaging models. A recent line of work uses channels to study the best achievable security of instant messaging between two users. A limited, unidirectional case was first considered by [BSJ$^+$17]; follow-up work uses bidirectional channels [JS18,JMM19,ACD19,CDV21]. The focus is on achieving strong forward security and post-compromise security guarantees in the presence of an attacker that can compromise secret states of the users. With the exception of [ACD19],

[9] One could then modify our construction of the MTProto-based channel from Section 4.4 so that it precisely models the current version of MTProto 2.0, and adjust our security analysis accordingly so that it holds with respect to the relaxed set of message delivery rules that is used in practice.

all of this work models channels that are required to provide strict in-order message delivery. In contrast, the *immediate decryption*-aware channel of [ACD19] effectively allows message drops but mandates that the dropped messages can later be delivered and retroactively assigned to their correct positions in the communication transcript. Any of these bidirectional models except [ACD19] could be simplified (to not require advanced security properties) and used for a formal analysis of our MTProto-based channel from Section 4.4. None of these models would be able to capture the correctness and security properties of MTProto 2.0 as it is currently implemented.

3.2 Syntax of channels

We refer to the two users of a channel as $\mathcal{I}$ and $\mathcal{R}$. These will map to client and server in the setting of MTProto. We use $u \in \{\mathcal{I}, \mathcal{R}\}$ as a variable to represent an arbitrary user and $\overline{u}$ to represent the other user, meaning $\overline{u}$ denotes the sole element of $\{\mathcal{I}, \mathcal{R}\} \setminus \{u\}$. We use st_u to represent the internal state of user u. A channel uses an initialisation algorithm to abstract away the key agreement; this matches the main focus of our work – to study the symmetric encryption of MTProto.

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

Figure 8. Syntax of the constituent algorithms of channel CH.

Definition 1. *A channel* CH *specifies algorithms* CH.Init, CH.Send *and* CH.Recv, *where* CH.Recv *is deterministic. The syntax used for the algorithms of* CH *is given in Fig. 8. Associated to* CH *is a plaintext space* $\mathsf{CH.MS} \subseteq \{0,1\}^* \setminus \{\varepsilon\}$ *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* $m \in \mathsf{CH.MS}$, *and auxiliary information* aux *to return the updated state* st_u *and a ciphertext* c, *where* $c = \bot$ *may be used to indicate a failure to send. We may surface random coins* $r \in \mathsf{CH.SendRS}$ *as an additional input to* CH.Send. *The receiving algorithm* CH.Recv *takes* st_u, c *and auxiliary information* aux *to return the updated state* st_u *and a plaintext* $m \in \mathsf{CH.MS} \cup \{\bot\}$, *where* $\bot$ *indicates a failure to recover a plaintext.*

Our channel definition reflects some unusual choices that are necessary to model the MTProto protocol. The abstract auxiliary information field aux will be used to associate timestamps to each sent and received message.[10] In this work we do not use the aux field to model associated data that would need to be authenticated; but our definitions in principle allow to use it that way. Also note that the sending algorithm CH.Send is randomised, but a stateful channel in general does not need randomness to achieve basic security notions. We only use randomness to faithfully model MTProto; it uses randomness to determine the length and contents of message padding. Our correctness and security notions will let an attacker choose arbitrary random coins, so we surface it as an optional input to the sending algorithm CH.Send.

3.3 Support transcripts and functions

In this section we extend the definitional framework for robust channels from [FGJ20]. In Section 3.1 we outlined the core differences between the two frameworks, and in Appendix C we provide a detailed comparison between them.

Support transcripts. We define a support transcript to represent the communication record of a single user. Each transcript entry describes an attempt to send or to receive a plaintext, ordered chronologically. A support transcript tr_u of user $u \in \{\mathcal{I}, \mathcal{R}\}$ contains two types of entries: $(\mathsf{sent}, m, \mathsf{label}, aux)$ and

[10] Our formal model of MTProto in Section 4 leaves the aux field unused. We only use the aux field in Appendix D where we expand our model to use message encoding that captures the real-world MTProto protocol more precisely.

$(\mathsf{recv}, m, \mathsf{label}, aux)$ for an event of sending or receiving some plaintext m respectively. In either case label is a *support label* whose purpose is to distinguish between different network messages each encrypting or encoding a specific plaintext m, and aux is auxiliary information such as the timestamp at the moment of sending or receiving the network message. Depending on the level of abstraction, our model uses ciphertexts or *message encodings* as support labels.[11]

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

A support transcript is not intended to surface the implementation details of the primitive that is used for communication. This is reflected in our abstract treatment of the support labels: an outside observer with no knowledge of the internal states of the two communicating users might not be able to interpret the (possibly encrypted) network messages that are being exchanged. So our framework treats each network message as a mere label that can be observed to be sent by a user in response to some plaintext input. One might subsequently observe the same label being taken as input by the opposite user, resulting in some plaintext output. If the scheme used for the two-user communication guarantees that all such labels are unique, an observer might be able to use the equality of exchanged labels across both support transcripts to determine whether a message replay, reordering or drop occurred. The MTProto-based scheme that we study in this paper produces distinct ciphertexts, and our framework uses ciphertexts as support labels when analysing a channel; this will allow us to rely on equality patterns that arise between them. In this work we use no information about support labels beyond their equality patterns.

Support transcripts can include entries of the form $(\mathsf{recv}, m, \mathsf{label}, aux)$ with the plaintext $m = \bot$ to indicate that the received network message was rejected. Support transcripts can also include entries of the form $(\mathsf{sent}, m, \mathsf{label}, aux)$ with the support label value $\mathsf{label} = \bot$, e.g. to indicate that a network message encrypting the plaintext m could not be sent over a terminated channel. Our support transcripts are therefore suitable for two-user communication primitives that implement a wide range of possible behaviours in the event of an error, from terminating after the first failure to full recovery.

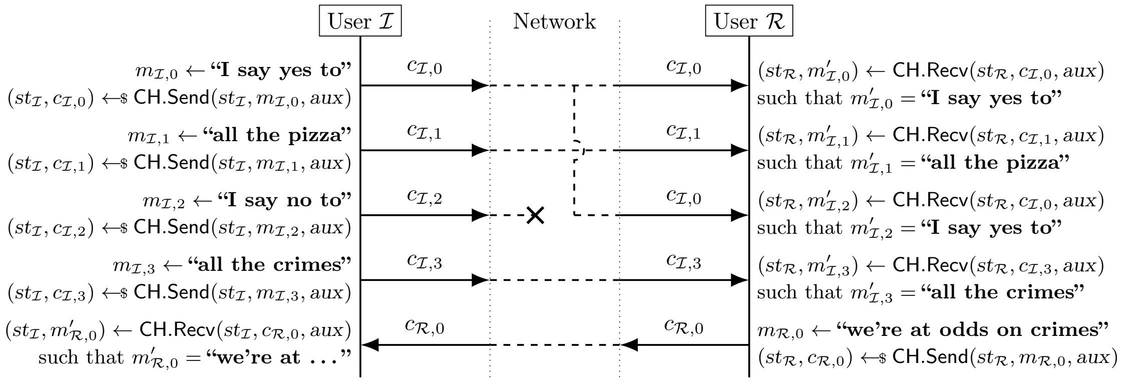
```
CH.Init()
  $k \leftarrow\!\$\ \{0,1\}^{\mathsf{SE.kl}}$
  $st_{\mathcal{I}} \leftarrow (0, k)$
  $st_{\mathcal{R}} \leftarrow (1, k)$
  Return $(st_{\mathcal{I}}, st_{\mathcal{R}})$

CH.Send($st_u, m, aux$)
  $(b, k) \leftarrow st_u$
  $p \leftarrow b \,\|\, m$
  $c \leftarrow\!\$\ \mathsf{SE.Enc}(k, p)$
  Return $(st_u, c)$

CH.Recv($st_u, c, aux$)
  $(b, k) \leftarrow st_u$
  $p \leftarrow \mathsf{SE.Dec}(k, c)$
  If $p = \bot$ then return $(st_u, \bot)$
  $b' \,\|\, m \leftarrow p$   // s.t. $|b'| = 1$
  If $b' \neq 1 - b$ then return $(st_u, \bot)$
  Return $(st_u, m)$
```

Figure 9. Construction of sample channel $\mathsf{CH} = \mathsf{SAMPLE\text{-}CH}[\mathsf{SE}]$ from symmetric encryption scheme SE. This channel ignores the auxiliary information aux.

We now provide the construction of a sample channel, along with an example of how communication over this channel can be captured using support transcripts. We will use this channel and its support transcripts to showcase more examples throughout this section. Let SE be an arbitrary symmetric encryption scheme that provides integrity and confidentiality (i.e. it provides authenticated encryption). Consider a sample channel $\mathsf{CH} = \mathsf{SAMPLE\text{-}CH}[\mathsf{SE}]$ as defined in Fig. 9. In addition to the security assurances inherited from SE, the channel CH is only designed to prevent forgeries that could occur by mirroring a ciphertext back to its sender. Fig. 10 provides a step-by-step example of communication between users $\mathcal{I}$ and $\mathcal{R}$ over CH. It shows $\mathcal{I}$'s and $\mathcal{R}$'s support transcripts at the end of the communication between them, where the channel's ciphertexts are used as labels. Note that the ciphertext $c_{\mathcal{I},2}$ was dropped and the ciphertext $c_{\mathcal{I},0}$ was replayed in its place. As a result, each user's transcript shows that the other user endorsed crimes.

[11] We use ciphertexts as support labels when channels are considered. We use message encodings as support labels when properties of message encoding schemes are considered (as defined in Section 3.5).

Initialisation: $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ \mathsf{CH.Init}()\ ;\ aux \leftarrow \varepsilon$


User $\mathcal{I}$
Network
User $\mathcal{R}$
$m_{\mathcal{I},0} \leftarrow$ **"I say yes to"**
$(st_{\mathcal{I}}, c_{\mathcal{I},0}) \leftarrow\$ \mathsf{CH.Send}(st_{\mathcal{I}}, m_{\mathcal{I},0}, aux)$
$c_{\mathcal{I},0}$
$c_{\mathcal{I},0}$
$(st_{\mathcal{R}}, m'_{\mathcal{I},0}) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{R}}, c_{\mathcal{I},0}, aux)$
such that $m'_{\mathcal{I},0} =$ **"I say yes to"**
$m_{\mathcal{I},1} \leftarrow$ **"all the pizza"**
$(st_{\mathcal{I}}, c_{\mathcal{I},1}) \leftarrow\$ \mathsf{CH.Send}(st_{\mathcal{I}}, m_{\mathcal{I},1}, aux)$
$c_{\mathcal{I},1}$
$c_{\mathcal{I},1}$
$(st_{\mathcal{R}}, m'_{\mathcal{I},1}) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{R}}, c_{\mathcal{I},1}, aux)$
such that $m'_{\mathcal{I},1} =$ **"all the pizza"**
$m_{\mathcal{I},2} \leftarrow$ **"I say no to"**
$(st_{\mathcal{I}}, c_{\mathcal{I},2}) \leftarrow\$ \mathsf{CH.Send}(st_{\mathcal{I}}, m_{\mathcal{I},2}, aux)$
$c_{\mathcal{I},2}$
$c_{\mathcal{I},0}$
$(st_{\mathcal{R}}, m'_{\mathcal{I},2}) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{R}}, c_{\mathcal{I},0}, aux)$
such that $m'_{\mathcal{I},2} =$ **"I say yes to"**
$m_{\mathcal{I},3} \leftarrow$ **"all the crimes"**
$(st_{\mathcal{I}}, c_{\mathcal{I},3}) \leftarrow\$ \mathsf{CH.Send}(st_{\mathcal{I}}, m_{\mathcal{I},3}, aux)$
$c_{\mathcal{I},3}$
$c_{\mathcal{I},3}$
$(st_{\mathcal{R}}, m'_{\mathcal{I},3}) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{R}}, c_{\mathcal{I},3}, aux)$
such that $m'_{\mathcal{I},3} =$ **"all the crimes"**
$(st_{\mathcal{I}}, m'_{\mathcal{R},0}) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{I}}, c_{\mathcal{R},0}, aux)$
such that $m'_{\mathcal{R},0} =$ **"we're at ..."**
$c_{\mathcal{R},0}$
$c_{\mathcal{R},0}$
$m_{\mathcal{R},0} \leftarrow$ **"we're at odds on crimes"**
$(st_{\mathcal{R}}, c_{\mathcal{R},0}) \leftarrow\$ \mathsf{CH.Send}(st_{\mathcal{R}}, m_{\mathcal{R},0}, aux)$

Support transcripts:

$$\begin{aligned}\mathsf{tr}_{\mathcal{I}} = \big[&(\mathsf{sent}, \text{“I say yes to”}, c_{\mathcal{I},0}, \varepsilon), (\mathsf{sent}, \text{“all the pizza”}, c_{\mathcal{I},1}, \varepsilon), (\mathsf{sent}, \text{“I say no to”}, c_{\mathcal{I},2}, \varepsilon),\\ &(\mathsf{sent}, \text{“all the crimes”}, c_{\mathcal{I},3}, \varepsilon), (\mathsf{recv}, \text{“we're at odds on crimes”}, c_{\mathcal{R},0}, \varepsilon)\big]\\ \mathsf{tr}_{\mathcal{R}} = \big[&(\mathsf{recv}, \text{“I say yes to”}, c_{\mathcal{I},0}, \varepsilon), (\mathsf{recv}, \text{“all the pizza”}, c_{\mathcal{I},1}, \varepsilon), (\mathsf{recv}, \text{“I say yes to”}, c_{\mathcal{I},0}, \varepsilon),\\ &(\mathsf{recv}, \text{“all the crimes”}, c_{\mathcal{I},3}, \varepsilon), (\mathsf{sent}, \text{“we're at odds on crimes”}, c_{\mathcal{R},0}, \varepsilon)\big]\end{aligned}$$

Figure 10. Communication between users $\mathcal{I}$ and $\mathcal{R}$ over the sample channel CH defined in Fig. 9. The resulting communication records of $\mathcal{I}$ and $\mathcal{R}$ are represented by support transcripts $\mathsf{tr}_{\mathcal{I}}$ and $\mathsf{tr}_{\mathcal{R}}$ respectively. The transcripts contradict each other due to an adversarial behaviour on the network.

Support functions. We now define the notion of a support function. We use a support function to prescribe the exact input-output behaviour of a receiver at any point in a two-user communication process (i.e. we use it to specify the expected behaviour of a channel's decryption algorithm or that of a message encoding scheme's decoding algorithm, the latter primitive defined in Section 3.5). More specifically, a support function supp determines whether a user $u \in \{\mathcal{I}, \mathcal{R}\}$ should accept an incoming network message – that is associated to a support label label – from the opposite user $\overline{u}$, based on the support transcripts $\mathsf{tr}_u, \mathsf{tr}_{\overline{u}}$ of both users. If the network message should be accepted, then supp must return a plaintext m^* to indicate that u is expected to recover m^* as a result of accepting it; otherwise supp must return $\perp$ to indicate that the network message must be rejected. We also let supp take the auxiliary information aux as input so that timestamps can be captured in our definitions.

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

In Section 3.4 we define the notions of channel correctness, integrity, and indistinguishability. Our correctness and integrity notions jointly require that the channel's receiving algorithm works exactly as prescribed by a specific support function. More precisely, both notions require that the channel's receiving algorithm consistently returns the same output as that returned by the support function, but each notion is defined with respect to an adversary that has different capabilities. In the correctness game the adversary gets the channel's state as input, and is only allowed to query the receiving algorithm on supported ciphertexts (i.e. those that are not rejected by the support function). In the integrity game the adversary does not get any secrets as input, and is allowed to query the receiving algorithm on all possible inputs, including attempted ciphertext forgeries or any gibberish inputs that aim to corrupt the channel's state. This definitional approach is similar in spirit to how correctness and integrity are defined for basic cryptographic primitives. For example, for a symmetric encryption scheme one often considers the notions of decryption correctness and ciphertext integrity, where the former should hold even when the adversary knows the secret key, whereas the latter requires the adversary to produce ciphertext forgeries without knowing the key. In comparison, a channel is a stateful primitive so its correctness and integrity conditions can be significantly more complex, depending on how it should treat forgeries, replays, reordering and drops. A support function allows us to capture these conditions in a modular way. Finally, the notion of indistinguishability that we define for a channel requires that the output of the channel's sending algorithm leaks no information about the encrypted plaintext; this security notion makes no use of a support function.

```
$\mathsf{SAMPLE\text{-}SUPP}_0(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$ / $\boxed{\mathsf{SAMPLE\text{-}SUPP}_1(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)}$
$\mathsf{forged} \leftarrow \nexists m, aux' \colon (\mathsf{sent}, m, \mathsf{label}, aux') \in \mathsf{tr}_{\overline{u}}$
If $\mathsf{forged}$ then return $\bot$
$\boxed{\mathsf{replayed} \leftarrow \exists m, aux' \colon (\mathsf{recv}, m, \mathsf{label}, aux') \in \mathsf{tr}_u}$
$\boxed{\text{If } \mathsf{replayed} \text{ then return } \bot}$
For each $(\mathsf{op}, m, \mathsf{label}', aux') \in \mathsf{tr}_{\overline{u}}$ do
    If $(\mathsf{op} = \mathsf{sent}) \wedge (\mathsf{label}' = \mathsf{label})$ then   // $aux'$ can be arbitrary
        Return $m$
```

Figure 11. Sample support functions $\mathsf{SAMPLE\text{-}SUPP}_0$ and $\mathsf{SAMPLE\text{-}SUPP}_1$. Support function $\mathsf{SAMPLE\text{-}SUPP}_1$ includes the boxed code, and support function $\mathsf{SAMPLE\text{-}SUPP}_0$ does not include it. Both support functions allow arbitrary auxiliary information, never checking the values of aux and aux'.

Consider a sample support function $\mathsf{SAMPLE\text{-}SUPP}_0$ in Fig. 11. It does not contain the boxed code. The support function prohibits forgeries by returning $\bot$ if the opposite user's support transcript $\mathsf{tr}_{\overline{u}}$ does not contain an entry indicating that $\overline{u}$ previously sent a network message associated to the support label label. If a forgery is not detected then the support function finds and returns a plaintext m such that $(\mathsf{sent}, m, \mathsf{label}, aux')$ belongs to $\mathsf{tr}_{\overline{u}}$ with any aux'. For any symmetric encryption scheme SE that provides authenticated encryption, recall algorithms $\mathsf{CH.Init}$ and $\mathsf{CH.Send}$ of the sample channel $\mathsf{CH} = \mathsf{SAMPLE\text{-}CH}[\mathsf{SE}]$ defined in Fig. 9; let us treat ciphertexts produced by $\mathsf{CH.Send}$ as support labels. Then the algorithm $\mathsf{CH.Recv}$ from Fig. 9 implements the functionality that is prescribed by $\mathsf{SAMPLE\text{-}SUPP}_0$: it rejects forgeries and otherwise recovers and returns the originally encrypted plaintext. Note that $\mathsf{SAMPLE\text{-}SUPP}_0$ grabs the first plaintext m that it finds associated to label in $\mathsf{tr}_{\overline{u}}$, without checking whether any other plaintext values are also associated to label. This does not produce ambiguity when used with algorithms $\mathsf{CH.Init}$ and $\mathsf{CH.Send}$; implicit in our example is that SE provides decryption correctness, and therefore two distinct plaintexts cannot be encrypted into the same ciphertext (and hence be mapped to the same support label). This serves to show that it is often beneficial to interpret a support function in the context of the properties that are known to be true about the used support labels, because that might provide a better intuition regarding how a receiver will behave.

Consider another sample support function $\mathsf{SAMPLE\text{-}SUPP}_1$ as defined in Fig. 11. In addition to the code from $\mathsf{SAMPLE\text{-}SUPP}_0$, this support function also contains the boxed code. The added code is designed to prevent replays by rejecting any network message associated to a support label label that is already present in the one of the entries of the receiver's support transcript tr_u. For example, consider the following intermediate support transcripts of users $\mathcal{I}$ and $\mathcal{R}$ that could have arisen at some point during the communication displayed in Fig. 10:

$$\mathsf{tr}_{\mathcal{I},3} = \big[(\mathsf{sent}, \text{“I say yes to”}, c_{\mathcal{I},0}, \varepsilon), (\mathsf{sent}, \text{“all the pizza”}, c_{\mathcal{I},1}, \varepsilon), (\mathsf{sent}, \text{“I say no to”}, c_{\mathcal{I},2}, \varepsilon)\big]$$
$$\mathsf{tr}_{\mathcal{R},2} = \big[(\mathsf{recv}, \text{“I say yes to”}, c_{\mathcal{I},0}, \varepsilon), (\mathsf{recv}, \text{“all the pizza”}, c_{\mathcal{I},1}, \varepsilon)\big]$$

These support transcripts represent the moment when $\mathcal{I}$ has already sent 3 network messages, but so far $\mathcal{R}$ has only received 2 of them. Following Fig. 10, let us assume that a replay attack happens next and $\mathcal{R}$ receives a network message containing the ciphertext $c_{\mathcal{I},0}$ with auxiliary information $aux = \varepsilon$. According to $\mathsf{SAMPLE\text{-}SUPP}_0$ this network message should be accepted (and should decrypt to $m^* =$ “I say yes to”), but according to $\mathsf{SAMPLE\text{-}SUPP}_1$ this network message should be rejected:

$$\mathsf{SAMPLE\text{-}SUPP}_0(\mathcal{R}, \mathsf{tr}_{\mathcal{R},2}, \mathsf{tr}_{\mathcal{I},3}, c_{\mathcal{I},0}, \varepsilon) = \text{“I say yes to”}$$
$$\mathsf{SAMPLE\text{-}SUPP}_1(\mathcal{R}, \mathsf{tr}_{\mathcal{R},2}, \mathsf{tr}_{\mathcal{I},3}, c_{\mathcal{I},0}, \varepsilon) = \bot$$

Note that the algorithm $\mathsf{CH.Recv}$ from Fig. 9 can be changed to simply reject duplicate ciphertexts in order to accommodate the specification of $\mathsf{SAMPLE\text{-}SUPP}_1$, without having to change algorithms $\mathsf{CH.Init}$ and $\mathsf{CH.Send}$. That would result in a contrived channel where the same plaintext can be encrypted and sent multiple times, but only the first of them is allowed to be received. A more appropriate change would require to also concatenate a distinct counter to each plaintext processed by $\mathsf{CH.Send}$, so that the same plaintext can be sent and received many times while still preventing replay attacks by a third party.

We now provide some observations about the power of support functions. This is irrelevant for the purpose of analysing MTProto, but is useful to highlight the strengths and limitations of our framework in general:

- A support function does not take as input any information about the internal state of the primitive that is used for communication (i.e. that of a channel or a message encoding scheme). But a communication primitive might use its internal state to interpret incoming network messages in a non-trivial way. For example, in some channels the same ciphertext (in our framework associated to the same support label) could be repeatedly decrypted to a different plaintext depending on some shared secret that is being synchronously evolved by both users. A support function might not be able to capture a receiver's behaviour in cases like this. Support functions are best suited for communication where the knowledge that "user u created a network message ξ to send a plaintext m" uniquely determines that the opposite user $\overline{u}$ can only recover m from ξ (or otherwise produce the error symbol $\perp$).
- Due to having access to user support transcripts, a support function can prescribe a receiver's behaviour that is not achievable by any implementation. For example, if two channel ciphertexts c_0, c_1 were sent by the user u prior to any of them being received by the user $\overline{u}$, then a support function can require $\overline{u}$ to recover both underlying plaintexts from the first ciphertext it receives. This is impossible if each ciphertext encrypted an independently sampled and uniformly random value.
- A support function prescribes a receiver's behaviour with respect to a pair of existing support transcripts. But our framework does not have a similar way to state complex requirements regarding a sender's behaviour. For example, our framework can require a channel user's receiving algorithm to perpetually return $\perp$ once the channel is considered closed (e.g. due to repeated errors while processing incoming ciphertexts), but it cannot require for the same user's sending algorithm to subsequently return $\perp$ in response to all attempts to send new plaintexts.

In Section 5.3 we define the support function SUPP with respect to which we will analyse the security of MTProto 2.0. In Appendix A we formalise two correctness-style properties of a support function, but we do not mandate that they must always be met. Both properties were also considered in [FGJ20]. The *integrity* of a support function requires that it always returns $\perp$ if the queried support label label does not appear in the opposite user's support transcript $\mathsf{tr}_{\overline{u}}$. The *order correctness* of a support function requires that it enforces in-order delivery for each direction between the two users separately, assuming that each network message is associated to a distinct support label.

3.4 Correctness and security of channels

In Section 3.3 we provided a high-level intuition regarding how we define channel correctness and security notions, here we formalise them. In all of the notions, we allow the adversary to control the randomness used by the channel's sending algorithm $\mathsf{CH.Send}$. Channels are stateful, so they can achieve strong notions of security even when the adversary can control the randomness used for encryption.

Correctness. Consider the correctness game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ in Fig. 12, defined for a channel CH, a support function supp and an adversary $\mathcal{F}$. 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 starts by calling the algorithm $\mathsf{CH.Init}$ to initialise users $\mathcal{I}$ and $\mathcal{R}$, and the adversary is given their initial states. The adversary $\mathcal{F}$ gets access to a sending oracle $\textsc{Send}$ and to a receiving oracle $\textsc{Recv}$. Calling $\textsc{Send}(u, m, aux, r)$ encrypts the plaintext m with auxiliary data aux and randomness r from the 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. Oracle $\textsc{Recv}$ can only be called on ciphertexts that should not produce a decryption error according to the behaviour prescribed by the support function supp (when queried on the current support transcripts), meaning $\textsc{Recv}$ immediately exits with $\perp$ when supp returns $m^* = \perp$. Calling $\textsc{Recv}(u, c, aux)$ thus recovers

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

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

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

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

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

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

Figure 12. Correctness of channel CH; integrity of channel CH. Both notions are defined with respect to support function supp.

the plaintext m^* from the support function, decrypts the queried ciphertext c into plaintext m and adds $(\mathsf{recv}, m, c, aux)$ to the receiver's transcript tr_u; the game verifies that the decrypted plaintext m is equal to m^*. If the adversary can cause the channel to output a different m, then the adversary wins. This game captures the *minimal* requirement one would expect from a communication channel: that it succeeds to decrypt incoming ciphertexts in accordance to its specification, with only a limited possible interference from an adversary. In particular, the adversary is not allowed to test that the channel appropriately identifies and handles any errors.

Note that the $\textsc{Recv}$ oracle always returns $\bot$, but $\mathcal{F}$ can use the support function to compute the value m on its own for as long as the condition $m = m^*$ has never been false yet.[12] Based on the same condition, $\mathcal{F}$ can also use the support function to distinguish whether $\bot$ was returned because $m^* = \bot$ or because the end of the code of $\textsc{Recv}$ was reached (i.e. its last instruction "Return $\bot$" was evaluated).

Consider the sample channel $\mathsf{CH} = \mathsf{SAMPLE\text{-}CH}[\mathsf{SE}]$ from Fig. 9 for any symmetric encryption scheme SE that has decryption correctness. Then CH provides correctness with respect to either sample support function $\mathsf{supp} \in \{\mathsf{SAMPLE\text{-}SUPP}_0, \mathsf{SAMPLE\text{-}SUPP}_1\}$ from Fig. 11. In particular, for all adversaries $\mathcal{F}$ we have $\mathsf{Adv}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}) = 0$.

Integrity. Consider the integrity game $G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ in Fig. 12, defined for a channel CH, a support function supp and an adversary $\mathcal{F}$. The advantage of $\mathcal{F}$ in breaking the INT-security of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}) = \Pr\left[G^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}\right]$. We define the integrity game in a very similar way to the correctness game above, but with two important distinctions. First, in the integrity game the adversary $\mathcal{F}$ no longer gets the initial states of users $\mathcal{I}$ and $\mathcal{R}$ as input. Second, the receiving oracle $\textsc{Recv}$ now allows all inputs from the adversary $\mathcal{F}$, including those that are meant to be rejected according to the support function supp. These changes reflect the intuition that the adversary $\mathcal{F}$ is now also allowed to win by producing an input such that the channel's receiving algorithm returns $m \neq \bot$ while the support function returned $m^* = \bot$, which is essentially a forgery. The adversary does not get the channel's initial states as input because that could trivialize its goal of producing a forgery.

According to the examples discussed in Section 3.3, the sample channel $\mathsf{CH} = \mathsf{SAMPLE\text{-}CH}[\mathsf{SE}]$ from Fig. 9 provides integrity with respect to the sample support function $\mathsf{SAMPLE\text{-}SUPP}_0$ from Fig. 11 if SE provides authenticated encryption. Here it is in fact sufficient for SE to only provide ciphertext

[12] The initial version of this work defined $\textsc{Recv}$ to always return m. This made the definition stronger, by allowing the adversary to detect the moment it won the game. Switching between the two alternative definitions does not affect our proofs, but returning m made it harder to reason about the joint security for channels in Appendix B. So for simplicity we chose to always return $\bot$.

integrity, without any assurances about the confidentiality of encrypted data. In contrast, no properties of SE would be sufficient for CH to provide integrity with respect to the sample support function $\mathsf{SAMPLE\text{-}SUPP}_1$ from Fig. 11; the construction of $\mathsf{SAMPLE\text{-}CH}$ itself would need to be changed to prevent replay attacks like the one displayed in Fig. 10.

Prior work on symmetric encryption formalises the intuition that a decryption oracle is useless to an adversary if all of its decryption queries can be simulated based on the live transcript of its encryption queries. This is captured as PA1 in [ABL$^+$14] (where "PA" stands for *plaintext awareness*) and as *decryption simulatability* in [DF18]. An important distinction is that our definition of integrity requires $\mathsf{CH.Recv}$ to behave exactly as prescribed by a *specific* support function, whereas the goal of [ABL$^+$14,DF18] is to draw implications from the existence of *any* algorithm that can simulate $\mathsf{CH.Recv}$.

```
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 13. Indistinguishability of channel CH.

Confidentiality. Consider the indistinguishability game $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ in Fig. 13, defined for a channel CH and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the IND-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 game samples a challenge bit b, and the adversary is required to guess it in order to win. The adversary $\mathcal{D}$ is provided with access to a challenge oracle $\textsc{Ch}$ and a receiving oracle $\textsc{Recv}$. The adversary can query the challenge oracle $\textsc{Ch}$ on inputs u, m_0, m_1, aux, r to obtain a ciphertext encrypting plaintext m_b with random coins r from user u to user $\overline{u}$, with auxiliary information aux. Here the two plaintexts m_0, m_1 are required to have the same length. The adversary can query the receiving oracle $\textsc{Recv}$ on inputs u, c, aux to make the user u decrypt the incoming ciphertext c from the user $\overline{u}$ with auxiliary information aux. The goal of this query is to update the receiving user's state st_u; this is important because the updated state is then used to compute future outputs of queries to the challenge oracle $\textsc{Ch}$ when user u is the sender. The receiving oracle always discards the decrypted plaintext m and returns $\perp$. Note that if channel CH has integrity with respect to any support function supp, then the indistinguishability adversary $\mathcal{D}$ can itself use supp to compute all outputs of the receiving oracle $\textsc{Recv}$ for either choice of the challenge bit b.

Consider the sample channel $\mathsf{CH} = \mathsf{SAMPLE\text{-}CH[SE]}$ from Fig. 9 for any symmetric encryption scheme SE that is IND-CPA secure. Then CH provides indistinguishability.

Authenticated encryption. In Appendix B we define the authenticated encryption security of a channel, which simultaneously captures the integrity and indistinguishability notions from above. We define the joint notion in the all-in-one style of [Shr04,RS06]. We prove that our two separate security notions together are equivalent to the authenticated encryption security. This serves as a sanity check for our definitional choices.

3.5 Message encoding schemes

We advocate for a modular approach when building cryptographic channels. At its core, a channel can be expected to have a mechanism that handles the process of encoding plaintexts into payloads and decoding payloads back into plaintexts. Such a mechanism might need to maintain counters that store the number of previously encoded and decoded messages. It might add padding to plaintexts, while possibly encoding their original lengths. It might also embed other metadata into the produced payloads. This mechanism does not need to provide any security assurances, and can be intended for use with a communication channel that already guarantees cryptographic integrity and confidentiality of all relayed payloads. We formalise it as a separate primitive called a *message encoding scheme*. Then a cryptographic channel can be built by composing a message encoding scheme with appropriate cryptographic primitives that would provide integrity and confidentiality.

We now formally define a message encoding scheme. The modular approach suggested above leads us to define syntax for message encoding that is similar to that of a cryptographic channel. In particular, a message encoding scheme needs to have stateful encoding and decoding algorithms. Auxiliary information can be used to relay and verify metadata such as timestamps. One could expect all algorithms of a message encoding scheme to be deterministic; our definition uses randomness purely because it is necessary when modelling Telegram (i.e. because in MTProto 2.0 the length of padding used for payloads is randomised).

$$\boxed{\begin{array}{l} (st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{ME.Init}() \\ (st_u, p) \leftarrow \mathsf{ME.Encode}(st_u, m, aux; \nu) \\ (st_u, m) \leftarrow \mathsf{ME.Decode}(st_u, p, aux) \end{array}}$$

Figure 14. Syntax of message encoding scheme ME.

Definition 4. *A message encoding scheme* ME *specifies algorithms* $\mathsf{ME.Init}$, $\mathsf{ME.Encode}$, $\mathsf{ME.Decode}$, *where* $\mathsf{ME.Decode}$ *is deterministic. Associated to* ME *is a message space* $\mathsf{ME.MS} \subseteq \{0,1\}^* \setminus \{\varepsilon\}$, *a payload space* $\mathsf{ME.Out}$, *a randomness space* $\mathsf{ME.EncRS}$ *of* $\mathsf{ME.Encode}$, *and a payload length function* $\mathsf{ME.pl}\colon \mathbb{N} \times \mathsf{ME.EncRS} \to \mathbb{N}$. *The initialisation algorithm* $\mathsf{ME.Init}$ *returns* $\mathcal{I}$*'s and* $\mathcal{R}$*'s initial states* $st_{\mathcal{I}}$ *and* $st_{\mathcal{R}}$. *The encoding algorithm* $\mathsf{ME.Encode}$ *takes* st_u *for* $u \in \{\mathcal{I}, \mathcal{R}\}$, *a message* $m \in \mathsf{ME.MS}$, *and auxiliary information aux to return the updated state* st_u *and a payload* $p \in \mathsf{ME.Out}$.[13] *We may surface random coins* $\nu \in \mathsf{ME.EncRS}$ *as an additional input to* $\mathsf{ME.Encode}$*; then a message m should be encoded into a payload p of length* $|p| = \mathsf{ME.pl}(|m|, \nu)$. *The decoding algorithm* $\mathsf{ME.Decode}$ *takes* st_u, p, *and auxiliary information aux to return the updated state* st_u *and a message* $m \in \mathsf{ME.MS} \cup \{\bot\}$. *The syntax used for the algorithms of* ME *is given in Fig. 14.*

We now define two properties of a message encoding scheme: *encoding correctness* and *encoding integrity*. We formalise each property with respect to a support function, in a similar way to how we formalised correctness and integrity for a channel in Section 3.4. The encoding correctness and integrity notions both roughly require that the decoding algorithm of a message encoding scheme always returns outputs that are consistent with the support function. The two notions differ in that the encoding correctness only requires the outputs to be consistent until the first error occurs (i.e. until the support function returns $\bot$), whereas the encoding integrity also requires the decoding algorithm to recover from errors and keep returning consistent outputs throughout. We formalise both notions in the setting where the message encoding scheme is being run over an authenticated channel. This reflects the intuition that the message encoding scheme does not have to provide any cryptographic properties, but it is expected to be composed with a primitive that guarantees the integrity of communication. In contrast, the message encoding scheme itself is responsible for providing all properties that are required by a support function and are not implied by integrity. This may include the impossibility to replay, reorder and drop messages.

We use the games in Fig. 15 to formalise the encoding correctness and integrity notions of a message encoding scheme ME with respect to a support function supp. The advantage of an adversary $\mathcal{F}$ in breaking the encoding correctness of ME with respect to supp is defined as $\mathsf{Adv}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{F}) = \Pr[\mathrm{G}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}]$. The advantage of an adversary $\mathcal{F}$ in breaking the encoding integrity (EINT-security)

[13] For full generality, the algorithm $\mathsf{ME.Encode}$ could also be allowed to return $p = \bot$, denoting a failure to encode a message. However, the message encoding schemes we define in this work never fail to encode messages from each scheme's corresponding message space $\mathsf{ME.MS}$. So for simplicity we do not define $p = \bot$ to be a valid output of $\mathsf{ME.Encode}$.

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

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

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

Figure 15. Encoding correctness and encoding integrity of message encoding scheme ME with respect to support function supp. Game $\mathrm{G}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$ includes the boxed code and game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$ does not.

of ME with respect to supp is defined as $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{F}) = \Pr[\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}]$. The encoding correctness game $\mathrm{G}^{\mathsf{ecorr}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$ contains the boxed code while the encoding integrity game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$ does not. The encoding correctness requires that ME manages to "correctly" decode all payloads that are deemed to be admissible by the support function supp, while the inadmissible payloads are ignored by the game; here the support function itself is used to determine what constitutes a "correct" decoding. The encoding integrity requires that ME rejects inadmissible payloads while maintaining its baseline correctness; this in particular means that the processing of inadmissible payloads should not corrupt the state of ME in unexpected ways. As a result of processing inadmissible payloads, the receiver's transcript will contain $(\mathsf{recv}, \bot, p, aux)$-type entries. The support function supp might process various conditions involving these entries (e.g. depending on the number of errors that occurred), and the encoding scheme ME still has to provide outputs that are consistent with supp.

The two core differences from the corresponding channel notions in Section 3.4 are as follows. First, the message encoding scheme is meant to be run within an integrity-protected communication channel, so the Recv oracle in both games now starts by checking that the queried payload p was returned by a prior call to the opposite user's Send oracle (in response to some message m and auxiliary information aux). Second, the message encoding is not meant to serve any cryptographic purpose, so the initial states $st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}$ should not contain any secret information and are given as inputs to adversary $\mathcal{F}$ in both games. This means that the encoding integrity is a strictly stronger notion than the encoding correctness, and the latter has limited value.[14]

In Section 5.3 we define three more properties of message encoding that will be necessary for our security analysis of MTProto 2.0. None of these properties are defined with respect to a support function. Our modular approach of building a channel from a message encoding scheme serves to localise the number of times we need to consider the specifics of a support function: the integrity proof (in Section 5.6) of the channel that we study is reduced to the encoding integrity of the underlying message encoding scheme, and the latter is then proved in Appendix E.5.

4 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 4.1 we give an informal description of MTProto based on Telegram documentation and client implementations. Next, in Section 4.2 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 4.3. We conclude with Section 4.4 where we give our formal model for a fixed version of the protocol.

[14] In Section 5.4, rather than arguing that a message encoding scheme has encoding correctness, we point out that it is implied by the proof of its encoding integrity.

4.1 Telegram description

We studied MTProto 2.0 as described in the online documentation [Tel21c] and as implemented in the official desktop[15] and Android[16] clients. 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 16 and 17 give a visual summary of the following description.

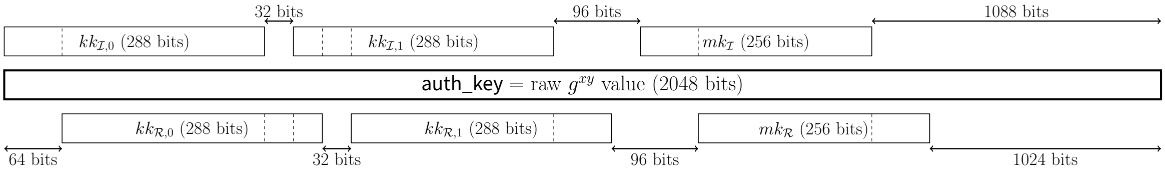

32 bits
96 bits
1088 bits
$kk_{\mathcal{I},0}$ (288 bits)
$kk_{\mathcal{I},1}$ (288 bits)
$mk_{\mathcal{I}}$ (256 bits)
auth_key = raw g^{xy} value (2048 bits)
$kk_{\mathcal{R},0}$ (288 bits)
$kk_{\mathcal{R},1}$ (288 bits)
$mk_{\mathcal{R}}$ (256 bits)
64 bits
32 bits
96 bits
1024 bits

Figure 16. 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.

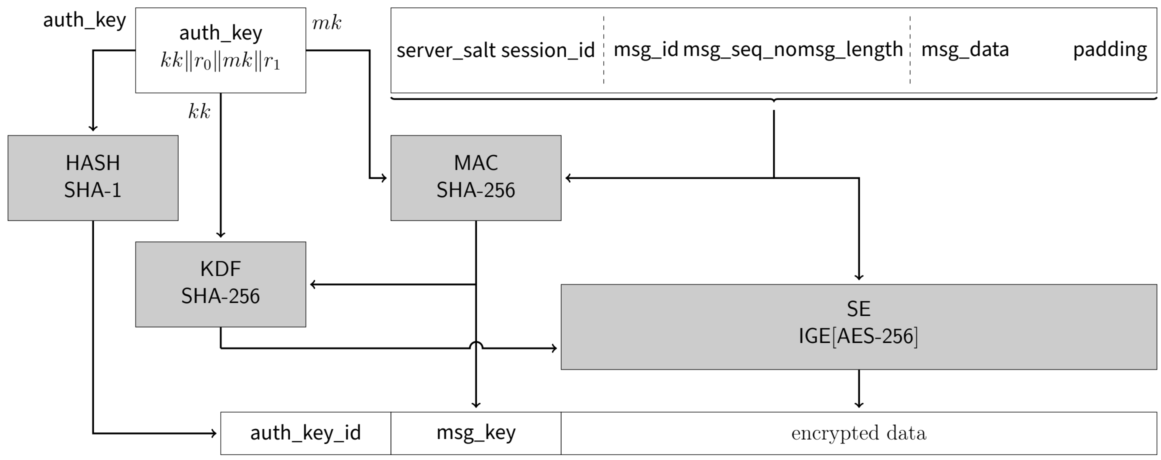


Figure 17. Overview of message processing in MTProto 2.0.

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 Section 7. 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 [Tel20c].
2. The API requests and responses are serialised according to the type language (TL) [Tel20e] and embedded in the msg_data field of a payload p, shown in Table 1. 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.

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

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

3. The payload is encrypted using AES-256 in IGE mode. The ciphertext c is a part of an MTProto ciphertext $\mathsf{auth\_key\_id} \| \mathsf{msg\_key} \| c$, where (recalling that $z[a:b]$ denotes bits a to $b-1$, inclusive, of string z):

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

   Here, the first two fields form an *external header*. The IGE[AES-256] keys and IVs are computed via:

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

   In the above steps, $x = 0$ for messages from the client and $x = 64$ from the server. Telegram clients use the BoringSSL implementation [Goo18] of IGE, which has 2-block IVs.
4. MTProto ciphertexts are encapsulated in a "transport protocol". The MTProto documentation defines multiple such protocols [Tel20a], but the default is the *abridged* format that begins the stream with a fixed value of `0xefefefef` and then wraps each MTProto ciphertext c_{MTP} in a transport packet as:
   - $\mathsf{length} \| c_{\mathsf{MTP}}$ where 1-byte length contains the c_{MTP} length divided by 4, if the resulting packet length is < 127, or
   - $\texttt{0x7f} \| \mathsf{length} \| c_{\mathsf{MTP}}$ where length is encoded in 3 bytes.
5. All the resulting packets are obfuscated by default using AES-128 in CTR mode. The key and IV are transmitted at the beginning of the stream, so the obfuscation provides no cryptographic protection and we ignore it henceforth.[17]
6. Communication is over TCP (port 443) or HTTP. Clients attempt to choose the best available connection. There is support for TLS in the client code, but it does not seem to be used.

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

Table 1. 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.

[17] 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.

4.2 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).

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.

Message 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 [Tel21f]. 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 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 [Lud17] 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.

Note that such reordering attacks are not possible against e.g. Signal or MTProto's closest "competitor" TLS. TLS-like protocols over UDP such as DTLS [RTM21] or QUIC [IT21] either leave it to the application to handle packet reordering (DTLS, i.e. reordering is possible against DTLS itself) or have built-in mechanisms to handle these (QUIC, i.e. reordering is not possible against QUIC itself).

Other types of reordering. 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. This is sometimes referred to as *global transcript* in the literature [UDB$^+$15], 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. Note that in the context of group messaging, there is another related but weaker property: the notion of *causality preservation* [EMP18]. However, when restricted to the two-party case, the property becomes equivalent to in-order delivery (as exhibited by the support function SUPP defined in Fig. 32).

Message drops. MTProto makes it possible to silently drop a message both when the client is the sender[18] and when it is the receiver, but it is difficult to exploit in practice. Clients and the server attempt to resend messages for which they did not get acknowledgements. 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 [Tel21j]. 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 message drops in the latter case, there is likely no practical attack.

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 drops, it enables attacks in the IND-CPA setting, as we explain next.

As a motivation, consider a local passive adversary that tries to establish whether $\mathcal{R}$ responded to $\mathcal{I}$ when looking at a transcript of three ciphertexts $(c_{\mathcal{I},0}, c_{\mathcal{R}}, c_{\mathcal{I},1})$, where c_u represents a ciphertext sent from u. In particular, it aims to establish whether $c_{\mathcal{R}}$ encrypts an automatically generated acknowledgement, denoted by "✓", 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 $c_{\mathcal{R}}$.[19]

Suppose we have a channel CH that models the MTProto protocol as described in Section 4.1 and uses the payload format given in Table 1.[20] To sketch a model for acknowledgement messages for the purpose of explaining this attack, 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 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. 18. If $b = 0$, $c_{\mathcal{R},i}$ encrypts an ✓ and so $c_{\mathcal{I},i+1}$ 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 $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ for some j, k, where $c^{(i)}$ denotes the i-th block of c, with probability 1 whenever $\mathsf{msg\_key}_j = \mathsf{msg\_key}_k$. This is true because the payloads of $c_{\mathcal{I},j}$ and $c_{\mathcal{I},k}$ share the same header fields, in particular including the 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 padding, the condition $\mathsf{msg\_key}_j = \mathsf{msg\_key}_k$ can be made to always hold and thus $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ holds with probability 1. As a consequence two sets of queries (i.e. a total of six queries) to the oracles suffice. When the adversary does not control the randomness, we can 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 $3 \cdot 2^{64}$ queries we expect a collision with constant probability (note that the adversary can check when a collision is found).

[18] There are scenarios where message drops 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. [ABJM21] for the significance of history deletion in some settings).

[19] 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.

[20] We give a formal definition of the channel in Section 4.4, but it is not necessary to outline the attack.

```
Adversary $\mathcal{D}^{\textsc{Ch},\textsc{Recv}}_{\text{IND},q}$
Let $aux = \varepsilon$. Choose any $m_0, m_1 \in \mathsf{CH.MS} \setminus \{\checkmark\}$.
Require $\forall i \in \mathbb{N}$: $r_{\mathcal{I},i}, r_{\mathcal{R},i} \in \mathsf{CH.SendRS}$.
For $i = 1, \ldots, q$ do
  $c_{\mathcal{I},i} \leftarrow \textsc{Ch}(\mathcal{I}, m_0, m_0, aux, r_{\mathcal{I},i})$
  $c_{\mathcal{R},i} \leftarrow \textsc{Ch}(\mathcal{R}, \checkmark, m_1, aux, r_{\mathcal{R},i})$ ; $\textsc{Recv}(\mathcal{I}, c_{\mathcal{R},i}, aux)$
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 $\bot$
```

Figure 18. 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.

Finally, in either setting, when $b = 0$ we have $c^{(2)}_{\mathcal{I},j} = c^{(2)}_{\mathcal{I},k}$ with probability 0 since the underlying payloads differ, the key is the same and AES is a permutation for a fixed key.

To allow a security proof to go through, the cleanest solution is to remove the re-encryption capability from the model. If a message resend facility is needed, applications can do this either by resending the original ciphertext at the underlying transport level (without involvement of the channel) or using the secure channel (in which case each resending would take place using an updated, unique state of the channel).

4.3 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 took that depart from the current version of MTProto 2.0 and justify each change.

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.[21] 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" [Tel21d]. In practice the clients receive salts that change every hour and which overlap with each other.[22] For client differences, consider padding generation: on desktop [Tel21j], a given message length will always result in the same padding length, whereas on Android [Tel21h], the padding length is randomised.

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.

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

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

Client/server roles. The client and the server are not considered equal in MTProto. For instance, the server is trusted to timestamp TL messages for history, while the clients are not, which is why our reordering attacks only work in the client to server direction. The client chooses the session_id, the server generates the server_salt. The server accepts any session_id given in the first message and then expects that value, while the client checks the session_id but may accept any server_salt given.[23] 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 [Tel20b] 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.

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.

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

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

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 message replay and drop attacks, which the implementation of MTProto as studied avoided in some cases only due to application-level processing.[24]

Re-encryption. Because of the attacks in Section 4.2, 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.

Message encoding. Some of the previous points outline changes to message encoding. We simplify the scheme, keeping to the format of Table 1 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.

[23] The Android client accepts any value in the place of server_salt, and the desktop client [Tel21k] 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.

[24] Secret chats implement more elaborate measures against replay/reordering [Tel21f], however this complexity is not required when in-order delivery is required for each direction separately.

4.4 MTProto-based channel

Our model of the MTProto channel is given in Definition 5 and Fig. 19. The users $\mathcal{I}$ and $\mathcal{R}$ represent the client and the server. We abstract the individual keyed primitives into function families.[25]

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. 19, with* $\mathsf{CH.MS} = \mathsf{ME.MS}$ *and* $\mathsf{CH.SendRS} = \mathsf{ME.EncRS}$*.*

```
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 $(st_{\mathcal{I}}, st_{\mathcal{R}})$

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 $(st_u, c)$

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

Figure 19. Construction of MTProto-based channel $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$ from message encoding scheme ME, function families HASH, MAC and KDF, related-key-deriving functions ϕ_{MAC} and ϕ_{KDF}, and from deterministic symmetric encryption scheme SE.

$\mathsf{CH.Init}$ generates the keys for both users and initialises the message encoding scheme. Note that $\mathsf{auth\_key}$ as described in Section 4.1 does not appear in the code in Fig. 19, since each part of $\mathsf{auth\_key}$ that is used for keying the primitives can be generated independently. These parts are denoted by hk, kk and mk.[26] The function ϕ_{KDF} (resp. ϕ_{MAC}) is then used to derive the (related) keys for each user from kk (resp. mk).

$\mathsf{CH.Send}$ proceeds by first using ME to encode a message m into a payload p. The MAC is computed on this payload to produce a $\mathsf{msg\_key}$, and the KDF is called on the $\mathsf{msg\_key}$ 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 $\mathsf{auth\_key\_id}$, $\mathsf{msg\_key}$ and the symmetric ciphertext c_{se}.

[25] 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.

[26] The comments in Fig. 21 show how the exact 2048-bit value of $\mathsf{auth\_key}$ 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 $\mathsf{auth\_key}$ that are *not* used for KDF and MAC keys kk, mk.

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 6.

The message encoding scheme MTP-ME is specified in Definition 6 and Fig. 20. It is a simplified scheme for strict in-order delivery without replays (see Appendix D for the actual MTProto scheme that permits reordering as outlined in Section 4.2).

Definition 6. *Let session_id* $\in \{0,1\}^{64}$ *and let pb,* bl $\in \mathbb{N}$. *Denote by* ME = MTP-ME[*session_id, pb,* bl] *the message-encoding scheme given in Fig. 20, with* $\mathsf{ME.MS} = \bigcup_{i=1}^{2^{24}} \{0,1\}^{8 \cdot i}$*,* $\mathsf{ME.Out} = \bigcup_{i \in \mathbb{N}} \{0,1\}^{\mathsf{bl} \cdot i}$ *and* $\mathsf{ME.pl}(\ell, \nu) = 256 + \ell + |\mathsf{GenPadding}(\ell; \nu)|$.[27]

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

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

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

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

Figure 20. Construction of simplified message encoding scheme for strict in-order delivery ME = MTP-ME[session_id, pb, bl] for session identifier session_id, maximum padding length (in full blocks) pb, and output block length bl.

As justified in Section 4.3, MTP-ME follows the header format of Table 1, but it does not use the server_salt field (we define salt as filled with zeros to preserve the field order) and we merge the 64-bit msg_id and 32-bit msg_seq_no fields into a single 96-bit seq_no field. Note that the internal counters of MTP-ME wrap around when seq_no "overflows" modulo 2^{96}, and an attacker can start replaying old payloads as soon as this happens. So when proving the encoding integrity of MTP-ME in Appendix E.5 with respect to a support function that prohibits replays, we will consider adversaries that make at most 2^{96} message encoding queries.[28]

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

[27] 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. 20 and instead handle this implicitly.

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

Definition 7. MTP-HASH *is the function family with* MTP-HASH.Keys $= \{0,1\}^{1056}$, MTP-HASH.In $= \{0,1\}^{992}$, MTP-HASH.ol $= 128$ *and* MTP-HASH.Ev *given in Fig. 21.*

```
MTP-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$[1024:2048]$
auth_key $\leftarrow kk \,\|\, r_0 \,\|\, mk \,\|\, r_1$
auth_key_id $\leftarrow$ SHA-1(auth_key)$[96:160]$
Return auth_key_id
```

Figure 21. Construction of function family MTP-HASH.

Definition 8. MTP-MAC *is the function family defined by* MTP-MAC.Keys $= \{0,1\}^{256}$, MTP-MAC.In $= \{0,1\}^{*}$, MTP-MAC.ol $= 128$ *and* MTP-MAC.Ev *given in Fig. 22.*

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

Figure 22. Construction of function family MTP-MAC.

Definition 9. MTP-KDF *is the function family defined by* MTP-KDF.Keys $= \{0,1\}^{288} \times \{0,1\}^{288}$, MTP-KDF.In $= \{0,1\}^{128}$, MTP-KDF.ol $= 2 \cdot$ SHA-256.ol *and* MTP-KDF.Ev *given in Fig. 23.*

```
MTP-KDF.Ev($kk_u$, msg_key)   // |msg_key| $= 128$
$(kk_0, kk_1) \leftarrow kk_u$
$k_0 \leftarrow$ SHA-256(msg_key $\|\, kk_0$)
$k_1 \leftarrow$ SHA-256($kk_1 \,\|$ msg_key)
$k \leftarrow k_0 \,\|\, k_1$
Return $k$
```

Figure 23. Construction of function family 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 [BK03], but define our related-key-deriving function ϕ_{KDF} (resp. ϕ_{MAC}) to output both keys at once, as a function of kk (resp. mk). See Fig. 24 for precise details of ϕ_{KDF} and ϕ_{MAC}.

Finally, we define the deterministic symmetric encryption scheme.

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

5 Formal security analysis

In this section, we define the security notions that we require to hold for each of the underlying primitives of MTP-CH, and then use these notions to justify its correctness and prove its security properties.

```
$\underline{\phi_{\mathsf{KDF}}(kk)}$   // $|kk| = 672$
$kk_{\mathcal{I},0} \leftarrow kk[0:288]$
$kk_{\mathcal{R},0} \leftarrow kk[64:352]$
$kk_{\mathcal{I},1} \leftarrow kk[320:608]$
$kk_{\mathcal{R},1} \leftarrow kk[384:672]$
$kk_{\mathcal{I}} \leftarrow (kk_{\mathcal{I},0}, kk_{\mathcal{I},1})$
$kk_{\mathcal{R}} \leftarrow (kk_{\mathcal{R},0}, kk_{\mathcal{R},1})$
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 24. Related-key-deriving functions $\phi_{\mathsf{KDF}} \colon \{0,1\}^{672} \rightarrow \mathsf{MTP\text{-}KDF.Keys} \times \mathsf{MTP\text{-}KDF.Keys}$ and $\phi_{\mathsf{MAC}} \colon \{0,1\}^{320} \rightarrow \mathsf{MTP\text{-}MAC.Keys} \times \mathsf{MTP\text{-}MAC.Keys}$.

We start by defining the security notions we require from the standard primitives in Section 5.1 (i.e. from the MTProto-based instantiations of HASH, KDF, MAC, SE); in Section 5.2 we then define two novel assumptions about SHACAL-2 that will be used in Appendix E to justify some of the aforementioned security notions. In Section 5.3 we define the security notions that will be required from the MTProto-based message encoding scheme; these notions are likewise justified in Appendix E. We prove that channel MTP-CH satisfies correctness, indistinguishability and integrity in Sections 5.4,5.5 and 5.6 respectively. We conclude by providing an interpretation of our formal results in Section 5.7.

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.

5.1 Security requirements on standard primitives

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. 25. The security game $\mathrm{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}$ in Fig. 25 evaluates the function family HASH on a challenge input x_b using a secret uniformly random function key hk. Adversary $\mathcal{D}$ is given x_0, x_1 and the output of HASH; it is required to guess the challenge bit $b \in \{0,1\}$. The game samples inputs x_0, x_1 uniformly at random rather than allowing $\mathcal{D}$ to choose them, so this security notion requires HASH to provide only a *weak* form of one-time indistinguishability. The advantage of $\mathcal{D}$ in breaking the OTWIND-security of HASH is defined as $\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{HASH}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}}\right] - 1$. Appendix E.1 provides a formal reduction from the OTWIND-security of MTP-HASH to the one-time PRF-security of SHACAL-1 (as defined in Section 2.2).

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

Figure 25. One-time weak indistinguishability of function family HASH.

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. 26. The security game $\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}$ in Fig. 26 defines a variant of the standard PRF notion allowing the adversary $\mathcal{D}$ to use its RoR oracle to evaluate the function family KDF on either of the two secret, related function keys $kk_{\mathcal{I}}, kk_{\mathcal{R}}$ (both computed using related-key-deriving function ϕ_{KDF}). The advantage of $\mathcal{D}$ in breaking the RKPRF-security of KDF with respect to ϕ_{KDF} is defined as $\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}}\right] - 1$.

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

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

Figure 26. Related-key PRF-security of function family KDF with respect to related-key-deriving function ϕ_{KDF}.

In Section 5.2 we define a novel security notion for SHACAL-2 that roughly requires it to be a leakage-resilient PRF under related-key attacks; in Appendix E.2 we provide a formal reduction from the RKPRF-security of MTP-KDF to the new security notion. In this context, "leakage resilience" means that the adversary can adaptively choose a part of the SHACAL-2 key. However, we limit the adversary to being able to evaluate SHACAL-2 only on a single known, constant input (which is IV_{256}, the initial state of SHA-256). The new security notion is formalised as the LRKPRF-security of SHACAL-2 with respect to a pair of related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ (the latter is defined in Section 5.2).

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. 27. The security game $\mathrm{G}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{F}}$ in Fig. 27 gives the adversary $\mathcal{F}$ two related function keys $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ (created by the related-key-deriving function ϕ_{MAC}), and requires it to produce two payloads p_0, p_1 (for either user u) such that there is a collision in the corresponding outputs $\mathsf{msg\_key}_0, \mathsf{msg\_key}_1$ of the function family MAC. The advantage of $\mathcal{F}$ in breaking the RKCR-security of MAC with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{F}}\right]$. It is clear by inspection that the RKCR-security of $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$ (with respect to ϕ_{MAC} from Fig. 24) reduces to the collision resistance of truncated-output SHA-256.

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

Figure 27. Related-key collision resistance of function family MAC with respect to related-key-deriving function ϕ_{MAC}.

MTP-MAC **is a PRF under RKA for unique-prefix inputs.** We require that MTP-MAC behaves like a pseudorandom function in the RKA setting when it is evaluated on a set of inputs that have unique 256-bit prefixes (UPRKPRF), as defined in Fig. 28. The security game $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 28 extends the standard PRF notion to use two related ϕ_{MAC}-derived function keys $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ for the function family MAC (similar to the RKPRF-security notion we defined above); but it also enforces that the adversary $\mathcal{D}$ cannot query its oracle RoR on two inputs (u, p_0) and (u, p_1) for any $u \in \{\mathcal{I}, \mathcal{R}\}$ such that p_0, p_1 share the same 256-bit prefix. The unique-prefix condition means that the game does not need to maintain a PRF table to achieve output consistency. Note that this security game only allows to call the oracle RoR with inputs of length $|p| \geq 256$; this is sufficient for our purposes, because in MTP-CH the function family MTP-MAC is only used with payloads that are longer than

256 bits. The advantage of $\mathcal{D}$ in breaking the UPRKPRF-security of MAC with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}}\right] - 1$.

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

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

Figure 28. Related-key PRF-security of function family MAC for inputs with unique 256-bit prefixes, with respect to key derivation function ϕ_{MAC}.

In Section 5.2 we define a novel security notion that requires $\mathsf{SHACAL\text{-}2}$ to be a leakage-resilient, related-key PRF when evaluated on a fixed input; in Appendix E.3 we show that the UPRKPRF-security of $\mathsf{MTP\text{-}MAC}$ reduces to this security notion and to the one-time PRF-security (OTPRF) of the $\mathsf{SHA\text{-}256}$ compression function h_{256}. The new security notion is similar to the notion discussed in Section 5.1 and defined in Section 5.2, in that it only allows the adversary to evaluate $\mathsf{SHACAL\text{-}2}$ on the fixed input IV_{256}. However, the underlying security game derives the related $\mathsf{SHACAL\text{-}2}$ keys differently, partially based on the function ϕ_{MAC} defined in Fig. 24 (as opposed to ϕ_{KDF}). The new notion is formalised as the HRKPRF-security of $\mathsf{SHACAL\text{-}2}$ with respect to ϕ_{MAC}.

$\mathsf{MTP\text{-}SE}$ is a one-time indistinguishable SE scheme. For any block cipher E, Appendix E.4 shows that $\mathsf{IGE}[\mathsf{E}]$ as used in MTProto is OTIND\$-secure (defined in Fig. 3) if $\mathsf{CBC}[\mathsf{E}]$ is OTIND\$-secure. This enables us to use standard results [BDJR97,Rog04] on CBC in our analysis of MTProto.

5.2 Novel assumptions about SHACAL-2

In this section we define two novel assumptions about $\mathsf{SHACAL\text{-}2}$. Both assumptions require $\mathsf{SHACAL\text{-}2}$ to be a related-key PRF when evaluated on the fixed input IV_{256} (i.e. on the initial state of $\mathsf{SHA\text{-}256}$), meaning that the adversary can obtain the values of $\mathsf{SHACAL\text{-}2.Ev}(\cdot, \mathsf{IV}_{256})$ for a number of different but related keys. We formalise the two assumptions as security notions, called LRKPRF and HRKPRF, each defined with respect to different related-key-deriving functions; this reflects the fact that these security notions allow the adversary to choose the keys in substantially different ways. The notion of LRKPRF-security derives the $\mathsf{SHACAL\text{-}2}$ keys partially based on the function ϕ_{KDF}, whereas the notion of HRKPRF-security derives $\mathsf{SHACAL\text{-}2}$ keys partially based on the function ϕ_{MAC} (both functions are defined in Fig. 24). Both security notions also have different flavours of leakage resilience: (1) the security game defining LRKPRF allows the adversary to directly choose 128 bits of the 512-bit long $\mathsf{SHACAL\text{-}2}$ key, with another 96 bits of this key fixed and known (due to being chosen by the SHA padding function $\mathsf{SHA\text{-}pad}$), and (2) the security game defining HRKPRF allows the adversary to directly choose 256 bits of the 512-bit long $\mathsf{SHACAL\text{-}2}$ key.

We use the notion of LRKPRF-security to justify the RKPRF-security of $\mathsf{MTP\text{-}KDF}$ with respect to ϕ_{KDF} (as explained in Section 5.1, with the security reduction in Appendix E.2), which is needed in both the IND-security and the INT-security proofs of $\mathsf{MTP\text{-}CH}$. We use the notion of HRKPRF-security to justify the UPRKPRF-security of $\mathsf{MTP\text{-}MAC}$ with respect to ϕ_{MAC} (as explained in Section 5.1, with the security reduction in Appendix E.3), which is needed in the IND-security proof of $\mathsf{MTP\text{-}CH}$.

We stress that we have to assume properties of $\mathsf{SHACAL\text{-}2}$ that have not been studied in the literature. Related-key attacks on reduced-round $\mathsf{SHACAL\text{-}2}$ have been considered [KKL+04,LKKD06], but they ordinarily work with a *known difference* relation between unknown keys. In contrast, our LRKPRF-security notion uses keys that differ by random, unknown parts. Both of our security notions consider keys that are partially chosen or known by the adversary. In Appendix F we show that both the LRKPRF-security and the HRKPRF-security of $\mathsf{SHACAL\text{-}2}$ hold in the ideal cipher model (i.e. when $\mathsf{SHACAL\text{-}2}$ is modelled as the ideal cipher); we provide concrete upper bounds for breaking each of them. 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.

SHACAL-2 is a PRF with ϕ_{KDF}-based related keys. Our LRKPRF-security notion for SHACAL-2 is defined with respect to related-key-deriving functions ϕ_{KDF} (from Fig. 24) and $\phi_{\mathsf{SHACAL\text{-}2}}$ from Fig. 29. The latter mirrors the design of MTP-KDF that (in Definition 9) is defined to return $\mathsf{SHA\text{-}256}(\mathsf{msg\_key} \,\|\, kk_0) \,\|\, \mathsf{SHA\text{-}256}(kk_1 \,\|\, \mathsf{msg\_key})$ for the target key $kk_u = (kk_0, kk_1)$, except that $\phi_{\mathsf{SHACAL\text{-}2}}$ only needs to produce the corresponding SHA-padded inputs. We note that LRKPRF-security of SHACAL-2 could instead be defined with respect to a single related-key-deriving function that would merge ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$, which could lead to a cleaner formalisation of LRKPRF-security; however, we chose to avoid introducing an additional abstraction level here.

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

Figure 29. Related-key-deriving function $\phi_{\mathsf{SHACAL\text{-}2}}$: $(\mathsf{MTP\text{-}KDF.Keys} \times \mathsf{MTP\text{-}KDF.Keys}) \times \{0,1\}^{128} \rightarrow \{0,1\}^{512}$.

Consider the game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ in Fig. 30. Adversary $\mathcal{D}$ is given access to the RoR oracle that takes $u, i, \mathsf{msg\_key}$ as input; all inputs to the oracle serve as parameters for the SHACAL-2 key derivation, used to determine the challenge key sk_i. The adversary gets back either the output of $\mathsf{SHACAL\text{-}2.Ev}(sk_i,\, \mathsf{IV}_{256})$ (if $b = 1$), or a uniformly random value (if $b = 0$), and is required to guess the challenge bit. The PRF table T is used to ensure consistency, so that a single random value is sampled and remembered for each set of used key derivation parameters $u, i, \mathsf{msg\_key}$. The advantage of $\mathcal{D}$ in breaking the LRKPRF-security of SHACAL-2 with respect to ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ is defined as $\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}\right] - 1$.

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

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

Figure 30. Leakage-resilient, related-key PRF-security of function family SHACAL-2 on fixed input IV_{256} with respect to related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$.

SHACAL-2 is a PRF with ϕ_{MAC}-based related keys. Consider the game $\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ in Fig. 31. Adversary $\mathcal{D}$ is given access to RoR oracle, and is required to choose the 256-bit suffix p of each challenge key used for evaluating $\mathsf{SHACAL\text{-}2.Ev}(\cdot, \mathsf{IV}_{256})$. The value of mk_u is then used to set the 256-bit prefix of the challenge key, where u is also chosen by the adversary, but the $mk_{\mathcal{I}}, mk_{\mathcal{R}}$ values themselves are related secrets that are not known to $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} is defined as $\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}\right] - 1$.

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

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

Figure 31. Leakage-resilient, related-key PRF-security of function family SHACAL-2 on fixed input IV_{256} with respect to related-key-deriving function ϕ_{MAC}.

5.3 Security requirements on message encoding

In Section 3.5 we defined encoding integrity of a message encoding scheme ME with respect to any support function supp. We now define the support function supp = SUPP that will be used for our security proofs. We also define three ad-hoc notions that must be met by the MTProto-based message encoding scheme MTP-ME in order to be compatible with our security proofs.

MTP-ME ensures in-order delivery. We require that MTP-ME is EINT-secure (Fig. 15) with respect to the support function SUPP defined in Fig. 32. We define SUPP to enforce strict in-order delivery for each user's sent messages, thus preventing message forgeries, replays, (unidirectional) reordering and drops.

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

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

Figure 32. Support function SUPP for strict in-order delivery.

The formalisation of the support function SUPP uses a helper function find(op, tr, label) that searches a transcript tr for an op-type entry (where op ∈ {sent, recv}) containing a specific label label. This code relies on an assumption that all support labels are unique, which is true for payloads of MTP-ME and for ciphertexts of MTP-CH as long as at most 2^{96} plaintexts are sent.[29] Function find also determines the order number of the target entry among all valid entries (denoted N_{op}); if the entry was not found then N_{op} is set to the number of all valid entries in the transcript. The support function SUPP on inputs $u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}$ requires that (i) an entry with label label is found in the sender's transcript $\mathsf{tr}_{\overline{u}}$, and (ii) an entry with label label is not found in the receiver's transcript tr_u, and (iii) the number of valid entries in the receiver's transcript is one fewer than the order number of the entry found in the sender's transcript, i.e. $N_{\mathsf{sent}} = N_{\mathsf{recv}} + 1$. Here the condition (i) prevents message forgery, the condition (ii) prevents message replays, whereas the condition (iii) prevents message reordering and drops. As outlined in Section 4.2, the message encoding scheme ME in MTProto we studied (cf. Appendix D) allowed reordering so it was not EINT-secure with respect to SUPP; instead we use the simplified message encoding scheme MTP-ME (cf. Definition 6) for our formal analysis of MTProto.[30] In Appendix E.5 we show that $\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{MTP\text{-}ME},\mathsf{SUPP}}(\mathcal{F}) = 0$ for any $\mathcal{F}$ making at most 2^{96} queries to SEND.

[29] In MTP-CH the first 2^{96} plaintexts are encoded into distinct payloads using MTP-ME, whereas distinct payloads are then encrypted into distinct ciphertexts according to the RKCR-security of MAC with respect to ϕ_{MAC}. The latter is used for transition from G_5 to G_6 of the integrity proof for MTP-CH in Section 5.6.

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

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

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

Figure 33. Prefix uniqueness of message encoding scheme ME.

Prefix uniqueness of MTP-ME. We require that payloads produced by MTP-ME have distinct prefixes of size 256 bits (independently for each user $u \in \{\mathcal{I}, \mathcal{R}\}$), as defined by the security game in Fig. 33. The advantage of an adversary $\mathcal{F}$ in breaking the UPREF-security of a message encoding scheme ME is defined as $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{upref}}_{\mathsf{ME},\mathcal{F}}\right]$. Given the fixed prefix size, this notion cannot be satisfied against unbounded adversaries. Our MTP-ME scheme ensures unique prefixes using the 96-bit counter seq_no that contains the number of messages sent by user u, so we have $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{MTP\text{-}ME}}(\mathcal{F}) = 0$ for any $\mathcal{F}$ making at most 2^{96} queries, and otherwise there exists an adversary $\mathcal{F}$ such that $\mathsf{Adv}^{\mathsf{upref}}_{\mathsf{MTP\text{-}ME}}(\mathcal{F}) = 1$. Note that MTP-ME always has payloads larger than 256 bits. The MTProto implementation of message encoding we analysed was not UPREF-secure as it allowed repeated msg_id (cf. Section 4.2).

```
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 $\bot$
```

Figure 34. Encoding robustness of message encoding scheme ME.

Encoding robustness of MTP-ME. We require that decoding in MTP-ME should not affect its state in such a way that would be visible in future encoded payloads, as defined by the security game in Fig. 34. The advantage of an adversary $\mathcal{D}$ in breaking the ENCROB-security of a message encoding scheme ME is defined as $\mathsf{Adv}^{\mathsf{encrob}}_{\mathsf{ME}}(\mathcal{D}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}}\right] - 1$. This advantage is trivially zero for both MTP-ME and the original MTProto message encoding scheme (cf. Appendix D). Note, however, that this property prevents a message encoding scheme from building payloads that include the number of previously received messages. It is thus incompatible with stronger notions of resistance against reordering attacks such as the global transcript (cf. Section 4.2).

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

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

$\textsc{Ch}(u, \mathsf{msg\_key}, c_{se}, st_{\mathsf{ME}}, aux)$
// $\mathsf{msg\_key} \in \{0,1\}^*$
If $\neg\mathsf{S}[u, \mathsf{msg\_key}]$ then
    If $\mathsf{T}[u, \mathsf{msg\_key}] = \bot$ 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 $m \neq \bot$ then $\mathsf{win} \leftarrow \mathsf{true}$
Return $\bot$
```

Figure 35. Unpredictability of deterministic symmetric encryption scheme SE with respect to message encoding scheme ME.

Combined security of $\mathsf{MTP\text{-}SE}$ and $\mathsf{MTP\text{-}ME}$. We require that decryption in $\mathsf{MTP\text{-}SE}$ with uniformly random keys has unpredictable outputs with respect to $\mathsf{MTP\text{-}ME}$, as defined in Fig. 35. The security game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}$ in Fig. 35 gives adversary $\mathcal{F}$ access to two oracles. For any user $u \in \{\mathcal{I}, \mathcal{R}\}$ and message key $\mathsf{msg\_key}$, oracle $\textsc{Ch}$ decrypts a given ciphertext c_{se} of deterministic symmetric encryption scheme SE under a uniformly random key $k \in \{0,1\}^{\mathsf{SE.kl}}$, and then decodes it using the given message encoding state st_{ME} of message encoding scheme ME, returning no output. The adversary is allowed to choose arbitrary values of c_{se} and st_{ME}; it is allowed to repeatedly query oracle $\textsc{Ch}$ on inputs that contain the same values for $u, \mathsf{msg\_key}$ in order to reuse a fixed, secret SE key k with different choices of c_{se}. Oracle $\textsc{Expose}$ lets $\mathcal{F}$ learn the SE key corresponding to the given u and $\mathsf{msg\_key}$; the table S is then used to disallow the adversary from querying $\textsc{Ch}$ with this pair of u and $\mathsf{msg\_key}$ values again. $\mathcal{F}$ wins if it can cause $\mathsf{ME.Decode}$ to output a valid $m \neq \bot$. Note that $\mathsf{msg\_key}$ in this game merely serves as a label for the tables, so we allow it to be an arbitrary string $\mathsf{msg\_key} \in \{0,1\}^*$. The advantage of $\mathcal{F}$ in breaking the UNPRED-security of SE with respect to ME is defined as $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}}\right]$. In Appendix E.6 we show that $\mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{MTP\text{-}SE},\mathsf{MTP\text{-}ME}}(\mathcal{F}) \leq q_{\textsc{Ch}}/2^{64}$ for any $\mathcal{F}$ making $q_{\textsc{Ch}}$ queries.

5.4 Correctness of MTP-CH

We claim that our MTProto-based channel satisfies our correctness definition. Consider any adversary $\mathcal{F}$ playing in the correctness game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ (Fig. 12) for channel $\mathsf{CH} = \mathsf{MTP\text{-}CH}$ (Fig. 19) and support function $\mathsf{supp} = \mathsf{SUPP}$ (Fig. 32). Due to the definition of SUPP, the $\textsc{Recv}$ oracle in game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{MTP\text{-}CH},\mathsf{SUPP},\mathcal{F}}$ rejects all CH ciphertexts that were not previously returned by the $\textsc{Send}$ oracle. The encryption and decryption algorithms of channel $\mathsf{MTP\text{-}CH}$ rely in a modular way on the message encoding scheme $\mathsf{MTP\text{-}ME}$, deterministic function families $\mathsf{MTP\text{-}KDF}, \mathsf{MTP\text{-}MAC}$, and deterministic symmetric encryption scheme $\mathsf{MTP\text{-}SE}$; the latter provides decryption correctness, so any valid ciphertext processed by oracle $\textsc{Recv}$ correctly yields the originally encrypted payload p. Thus we need to show that $\mathsf{MTP\text{-}ME}$ always recovers the expected plaintext m from payload p, meaning m matches the corresponding output of SUPP. In Section 3.5 we formalised this requirement as the encoding correctness of $\mathsf{MTP\text{-}ME}$ with respect to SUPP, and discussed that it is also implied by the encoding integrity of $\mathsf{MTP\text{-}ME}$ with respect to SUPP. We prove the latter in Appendix E.5 for adversaries that make at most 2^{96} queries.

5.5 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 $\mathsf{CH} = \mathsf{MTP\text{-}CH}$ (i.e. $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\mathrm{IND}}}$ in Fig. 13), 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 $\bot$, and the only functionality of this oracle is to update the state of receiver's channel by calling $\mathsf{CH.Recv}$. We assume that calls to $\mathsf{CH.Recv}$ never affect the ciphertexts that are returned by future calls to $\mathsf{CH.Send}$ (more precisely, we use the ENCROB property of ME that reasons about payloads rather than ciphertexts). This allows us to completely disregard the $\textsc{Recv}$ oracle, making it immediately return $\bot$ without calling $\mathsf{CH.Recv}$.

(2) We use the UPRKPRF-security of MAC to show that the ciphertexts returned by oracle $\textsc{Ch}$ contain $\mathsf{msg\_key}$ values that look uniformly random and are independent of each other. Roughly, this security notion requires that MAC can only be evaluated on a set of inputs with unique prefixes. To ensure this, we assume that the payloads produced by ME meet this requirement (as formalised by the UPREF property of ME).

(3) In order to prove that oracle $\textsc{Ch}$ does not leak the challenge bit, it remains to show that ciphertexts returned by $\textsc{Ch}$ contain c_{se} values that look uniformly random and independent of each other. This follows from the OTIND\$-security of SE. We invoke the OTWIND-security of HASH to show that $\mathsf{auth\_key\_id}$ does not leak any information about the KDF keys; we then use the RKPRF-security of KDF to show that the keys used for SE are uniformly random. Finally, we use the birthday bound to argue that the uniformly random values of $\mathsf{msg\_key}$ are unlikely to collide, and hence the keys used for SE are also one-time.

Formally, we prove the following.

Theorem 1. *Let* ME*,* HASH*,* MAC*,* KDF*,* ϕ_{MAC}*,* ϕ_{KDF}*,* SE *be any primitives that meet the requirements stated in Definition 5 of channel* $\mathsf{MTP\text{-}CH}$*. Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$*. Let* $\mathcal{D}_{\mathrm{IND}}$ *be any adversary against the* IND*-security of* CH*, making* q_{CH} *queries to its* CH *oracle. Then we can build 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*

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

Proof. This proof uses games G_0–G_3 in Fig. 39 and G_4–G_8 in Fig. 40, in which the code added for the transitions between games is highlighted in green. The adversaries for transitions between games are referenced throughout the proof. Each constructed adversary simulates one or two subsequent games of the security reduction for adversary $\mathcal{D}_{\mathrm{IND}}$. The highlighted instructions mark the changes in the code of the simulated games.

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 $\mathsf{CH.Init}$, $\mathsf{CH.Send}$ and $\mathsf{CH.Recv}$; the expanded instructions are highlighted in grey. 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 the value of $\mathsf{auth\_key\_id}$ depends on the raw KDF and MAC keys (i.e. kk and mk), and adversary $\mathcal{D}_{\mathrm{IND}}$ can learn it from any ciphertext returned by oracle CH. To invoke PRF-style security notions for either primitive in later steps, we appeal to the OTWIND-security of HASH (Fig. 25), which essentially guarantees that $\mathsf{auth\_key\_id}$ leaks no information about KDF and MAC keys. Game G_1 is the same as game G_0, except $\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, \cdot)$ 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.

```
Adversary $\mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, \mathsf{auth\_key\_id})$
$kk \,\|\, mk \leftarrow x_1$    // s.t. $|kk| = 672$, $|mk| = 320$
$b \leftarrow\!\$ \{0,1\}$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$ \mathsf{ME.Init}()$
$b' \leftarrow\!\$ \mathcal{D}_{\mathrm{IND}}^{\mathrm{CHSIM},\mathrm{RECVSIM}}$
If $b' = b$ then return 1 else return 0

$\mathrm{CHSIM}(u, m_0, m_1, aux, r)$,
$\mathrm{RECVSIM}(u, c, aux)$
// Identical to oracles $\mathrm{CH}$ and $\mathrm{RECV}$
in games $\mathrm{G}_0$, $\mathrm{G}_1$ of Fig. 39.
```

Figure 36. Adversary $\mathcal{D}_{\mathrm{OTWIND}}$ against the OTWIND-security of HASH for the transition between games G_0–G_1.

More formally, given $\mathcal{D}_{\mathrm{IND}}$, in Fig. 36 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 $\mathcal{D}_{\mathrm{IND}}$ (i.e. kk and mk are derived from the input to $\mathsf{HASH.Ev}(hk, \cdot)$), and otherwise it simulates game G_1 (i.e. kk and mk are independent from the input to $\mathsf{HASH.Ev}(hk, \cdot)$). It follows that $\Pr[\mathrm{G}_0] = \Pr[\, d' = 1 \,|\, d = 1\,]$ and $\Pr[\mathrm{G}_1] = \Pr[\, d' = 1 \,|\, d = 0\,]$, and hence

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

$\mathbf{G}_1 \rightarrow \mathbf{G}_2$. In the transition between games G_1 and G_2 (Fig. 39), we use the RKPRF-security of KDF (Fig. 26) with respect to ϕ_{KDF} in order to replace $\mathsf{KDF.Ev}(kk_u,\ \mathsf{msg\_key})$ with a uniformly random value from $\{0,1\}^{\mathsf{KDF.ol}}$ (and for consistency store the latter in $\mathsf{T}[u, \mathsf{msg\_key}]$). Similarly to the above, in Fig. 37 we build an adversary $\mathcal{D}_{\text{RKPRF}}$ attacking the RKPRF-security of KDF that simulates G_1 or G_2 for adversary $\mathcal{D}_{\text{IND}}$, depending on the challenge bit in game $\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\text{RKPRF}}}$. We have

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

```
Adversary $\mathcal{D}^{\text{RoR}}_{\text{RKPRF}}$
$b \leftarrow\$ \{0,1\}$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
$mk \leftarrow\$ \{0,1\}^{320}$
$x \leftarrow\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
$b' \leftarrow\$ \mathcal{D}^{\text{ChSim},\text{RecvSim}}_{\text{IND}}$
If $b' = b$ then return 1 else return 0

ChSim$(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
$k \leftarrow \text{RoR}(u, \mathsf{msg\_key})$
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
Return $c$

RecvSim$(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
$k \leftarrow \text{RoR}(\overline{u}, \mathsf{msg\_key})$
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
$\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
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 $\perp$
```

Figure 37. Adversary $\mathcal{D}_{\text{RKPRF}}$ against the RKPRF-security of KDF for the transition between games G_1–G_2.

```
Adversary $\mathcal{D}^{\text{Send},\text{Recv}}_{\text{ENCROB}}$
$b \leftarrow\$ \{0,1\}$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
$mk \leftarrow\$ \{0,1\}^{320}$
$x \leftarrow\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$b' \leftarrow\$ \mathcal{D}^{\text{ChSim},\text{RecvSim}}_{\text{IND}}$
If $b' = b$ then return 1 else return 0

ChSim$(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$p \leftarrow \text{Send}(u, m_b, aux, r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
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 $c$

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
    Recv$(u, p, aux)$
Return $\perp$
```

Figure 38. Adversary $\mathcal{D}_{\text{ENCROB}}$ against the ENCROB-security of ME for the transition between games G_2–G_3.

$\mathrm{G}_2 \to \mathrm{G}_3$. We invoke the ENCROB property of ME (Fig. 34) to transition from G_2 to G_3 (Fig. 39). 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. 38 we build an adversary $\mathcal{D}_{\text{ENCROB}}$ against ENCROB of ME that simulates either G_2 or G_3 for $\mathcal{D}_{\text{IND}}$, depending on the challenge bit in game $\mathrm{G}^{\mathsf{encrob}}_{\mathsf{ME},\mathcal{D}_{\text{ENCROB}}}$, such that

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

```
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)
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$ \mathsf{ME.Init}()$
$b' \leftarrow\!\$ \mathcal{D}_{\text{IND}}^{\text{CH},\text{RECV}}$
Return $b' = b$

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

RECV$(u, c, aux)$   // $u \in \{\mathcal{I}, \mathcal{R}\}$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \bot$ then $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$                  // $G_0$–$G_1$
$k \leftarrow \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$                          // $G_2$–$G_3$ (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 $\bot$
```

Figure 39. Games G_0–G_3 for the proof of Theorem 1. The code added by expanding the algorithms of CH in game $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}_{\text{IND}}}$ is highlighted in grey.

$\mathrm{G}_3 \to \mathrm{G}_4$. Game G_4 (Fig. 40) differs from G_3 (Fig. 39) 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 $\bot$ 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 256-bit prefixes of payloads that were produced by calling the specific user's CH oracle. Every time a new payload p is generated, the added code inside oracle CH checks whether its prefix $p[0:256]$ is already contained inside X_u, which would mean that another previously seen payload had the same prefix. Then, regardless of

```
Games $\mathrm{G}_4$–$\mathrm{G}_8$
$b \leftarrow\!\!\$ \{0,1\}$
$hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\!\!\$ \{0,1\}^{320}$
$x \leftarrow\!\!\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$ \mathsf{ME.Init}()$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$
$b' \leftarrow\!\!\$ \mathcal{D}_{\mathrm{IND}}^{\mathrm{CH,RECV}}$
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 $p[0:256] \in X_u$ then
    $\mathsf{bad}_0 \leftarrow \mathsf{true}$
    $\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$ // $\mathrm{G}_4$
    $\mathsf{msg\_key} \leftarrow\!\!\$ \{0,1\}^{\mathsf{MAC.ol}}$      // $\mathrm{G}_5$–$\mathrm{G}_8$ (UPREF of $\mathsf{ME}$)
Else
    $\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$ // $\mathrm{G}_4$–$\mathrm{G}_5$
    $\mathsf{msg\_key} \leftarrow\!\!\$ \{0,1\}^{\mathsf{MAC.ol}}$      // $\mathrm{G}_6$–$\mathrm{G}_8$ (UPRKPRF of $\mathsf{MAC}$)
$X_u \leftarrow X_u \cup \{p[0:256]\}$
$k \leftarrow\!\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
If $\mathsf{T}[u, \mathsf{msg\_key}] \neq \perp$ then
    $\mathsf{bad}_1 \leftarrow \mathsf{true}$
    $k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$             // $\mathrm{G}_4$–$\mathrm{G}_6$ (Birthday bound)
$\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow k$
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$                    // $\mathrm{G}_4$–$\mathrm{G}_7$
$c_{se} \leftarrow\!\!\$ \{0,1\}^{\mathsf{SE.cl}(\mathsf{ME.pl}(|m_b|, r))}$   // $\mathrm{G}_8$ (OTIND\$ of $\mathsf{SE}$)
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
Return $c$

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

Figure 40. Games G_4–G_8 for the proof of Theorem 1. The code highlighted in grey was rewritten in a way that is functionally equivalent to the corresponding code in G_3.

whether this condition passes, the new prefix $p[0:256]$ 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

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

$\mathrm{G}_4 \rightarrow \mathrm{G}_5$. The transition from game G_4 to G_5 replaces the value assigned to $\mathsf{msg\_key}$ when the newly added unique-prefixes condition is satisfied; the value of $\mathsf{msg\_key}$ changes from $\mathsf{MAC.Ev}(mk_u, p)$ to a uniformly random string from $\{0,1\}^{\mathsf{MAC.ol}}$. Games G_4 and G_5 are identical until bad_0 is set. We have

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

The UPREF property of ME (Fig. 33) states that it is hard to find two payloads returned by $\mathsf{ME.Encode}$ 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. 41 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 set in game G_4, this corresponds to adversary $\mathcal{F}_{\mathrm{UPREF}}$ setting flag win to true in its own game $\mathrm{G}^{\mathsf{upref}}_{\mathsf{ME},\mathcal{F}_{\mathrm{UPREF}}}$. It follows that

$$\Pr[\mathsf{bad}_0^{\mathrm{G}_4}] \leq \mathsf{Adv}^{\mathsf{upref}}_{\mathsf{ME}}(\mathcal{F}_{\mathrm{UPREF}}).$$

```
Adversary $\mathcal{F}_{\mathrm{UPREF}}^{\mathrm{SEND}}$
$b \leftarrow\!\!\$ \{0,1\}$
$hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
$mk \leftarrow\!\!\$ \{0,1\}^{320}$
$x \leftarrow\!\!\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$b' \leftarrow\!\!\$ \mathcal{D}_{\mathrm{IND}}^{\mathrm{ChSim,RecvSim}}$

$\mathrm{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$p \leftarrow \mathrm{Send}(u, m_b, aux, r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
$k \leftarrow\!\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
If $\mathsf{T}[u, \mathsf{msg\_key}] \neq \bot$ 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 $c$

$\mathrm{RecvSim}(u, c, aux)$
Return $\bot$
```

Figure 41. Adversary $\mathcal{F}_{\mathrm{UPREF}}$ against the UPREF-security of ME for the transition between games G_4–G_5.

$\mathbf{G}_5 \to \mathbf{G}_6$. We use the UPRKPRF-security of MAC (Fig. 28) with respect to ϕ_{MAC} in order to replace the value of $\mathsf{msg\_key}$ 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 (Fig. 40). Note that the notion of UPRKPRF-security only guarantees the indistinguishability from random when MAC is evaluated on inputs with unique prefixes, whereas games $\mathrm{G}_5, \mathrm{G}_6$ ensure that this prerequisite is satisfied by only evaluating MAC if $p[0:256] \notin X_u$. In Fig. 42 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 $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$. It follows that

$$\Pr[\mathrm{G}_5] - \Pr[\mathrm{G}_6] = \mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}).$$

```
Adversary $\mathcal{D}_{\mathrm{UPRKPRF}}^{\mathrm{RoR}}$
$b \leftarrow\!\!\$ \{0,1\}$
$hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
$x \leftarrow\!\!\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$ \mathsf{ME.Init}()$
$b' \leftarrow\!\!\$ \mathcal{D}_{\mathrm{IND}}^{\mathrm{ChSim,RecvSim}}$
If $b' = b$ then return 1 else return 0

$\mathrm{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathrm{RoR}(u, p)$
If $\mathsf{msg\_key} = \bot$ then $\mathsf{msg\_key} \leftarrow\!\!\$ \{0,1\}^{\mathsf{MAC.ol}}$
$k \leftarrow\!\!\$ \{0,1\}^{\mathsf{KDF.ol}}$
If $\mathsf{T}[u, \mathsf{msg\_key}] \neq \bot$ 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 $c$

$\mathrm{RecvSim}(u, c, aux)$
Return $\bot$
```

Figure 42. Adversary $\mathcal{D}_{\mathrm{UPRKPRF}}$ against the UPRKPRF-security of MAC for the transition between games G_5–G_6.

$\mathbf{G}_6 \to \mathbf{G}_7$. Games G_6 and G_7 are identical until bad_1 is set; as above, we have

$$\Pr[\mathrm{G}_6] - \Pr[\mathrm{G}_7] \leq \Pr[\mathsf{bad}_1^{\mathrm{G}_6}].$$

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:

$$\Pr[\mathsf{bad}_1^{\mathrm{G}_6}] \leq \frac{q_{\mathrm{CH}} \cdot (q_{\mathrm{CH}} - 1)}{2 \cdot 2^{\mathsf{MAC.ol}}}.$$

$\mathbf{G}_7 \rightarrow \mathbf{G}_8$. In the transition from G_7 to G_8 (Fig. 40), 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 $\mathsf{ME.Encode}$ on any message of length $|m_b|$ and on random coins r, whereas $\mathsf{SE.cl}(\cdot)$ maps the payload length to the resulting ciphertext length when encrypted with SE. In Fig. 43 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 $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE},\mathcal{D}_{\mathrm{OTIND\$}}}$. It follows that

$$\Pr[\mathrm{G}_7] - \Pr[\mathrm{G}_8] = \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}}(\mathcal{D}_{\mathrm{OTIND\$}}).$$

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathrm{OTIND\$}}$
$b \leftarrow\$ \{0,1\}$
$hk \leftarrow\$ \{0,1\}^{\mathsf{HASH.kl}}$
$x \leftarrow\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\$ \mathsf{ME.Init}()$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{ChSim},\mathrm{RecvSim}}_{\mathrm{IND}}$
If $b' = b$ then return 1 else return 0
```

```
$\mathrm{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\perp$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m_b, aux; r)$
$\mathsf{msg\_key} \leftarrow\$ \{0,1\}^{\mathsf{MAC.ol}}$
$c_{se} \leftarrow \mathrm{RoR}(p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
Return $c$

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

Figure 43. Adversary $\mathcal{D}_{\mathrm{OTIND\$}}$ against the OTIND\$-security of SE for the transition between games G_7–G_8.

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

$$\Pr[\mathrm{G}_8] = \frac{1}{2}.$$

The theorem statement follows. □

Proof alternatives. Our security reduction relies on the RKPRF-security of KDF with respect to ϕ_{KDF}. We note that it would suffice to instead define and use a related-key *weak*-PRF notion here. It could be used in the penultimate step of this security reduction: right before appealing to the OTIND\$-security of SE.

Further, in this security reduction we consider a generic function family MAC and rely on it being related-key PRF-secure with respect to unique-prefix inputs. Recall that MTProto uses $\mathsf{MAC} = \mathsf{MTP\text{-}MAC}$ such that $\mathsf{MTP\text{-}MAC.Ev}(mk_u,\ p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$. It discards half of the $\mathsf{SHA\text{-}256}$ output bits, so we could alternatively model it as an instance of Augmented MAC (AMAC) and prove it to be related-key PRF-secure based on [BBT16]. However, using the results from [BBT16] would have required us to show that the $\mathsf{SHA\text{-}256}$ compression function is a secure PRF when half of its key is leaked to the adversary. We achieve a simpler and tighter security reduction by relying on the unique-prefix property of ME that is already guaranteed in MTProto.

5.6 INT-security of MTP-CH

The first half of our integrity proof shows that it is hard to forge ciphertexts; in order to justify this, we rely on security properties of the cryptographic primitives that are used to build the channel $\mathsf{MTP\text{-}CH}$ (i.e. HASH, KDF, SE, and MAC). Once ciphertext forgery is ruled out, we are guaranteed that $\mathsf{MTP\text{-}CH}$ broadly matches an intuition of an *authenticated channel*: it prevents an attacker from modifying or creating its own ciphertexts but still allows to intercept and subsequently replay, reorder or drop honestly produced ciphertexts. So in the second part of the proof we 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* $session\_id \in \{0,1\}^{64}$, $pb \in \mathbb{N}$, *and* $\mathsf{bl} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[session\_id, pb, \mathsf{bl}]$ *be the message encoding scheme as defined in Definition 6. Let* $\mathsf{SE} = \mathsf{MTP\text{-}SE}$ *be the deterministic* *symmetric encryption scheme as defined in Definition 10. Let* HASH*,* MAC*,* KDF*,* ϕ_{MAC}*,* ϕ_{KDF} *be any primitives that, together with* ME *and* SE*, meet the requirements stated in Definition 5 of channel* $\mathsf{MTP\text{-}CH}$*. Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{HASH}, \mathsf{MAC}, \mathsf{KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{SE}]$*. Let* $\mathsf{supp} = \mathsf{SUPP}$ *be the support function as defined in Fig. 32. Let* $\mathcal{F}_{\text{INT}}$ *be any adversary against the* INT*-security of* CH *with respect to* supp*. Then we can build adversaries* $\mathcal{D}_{\text{OTWIND}}$*,* $\mathcal{D}_{\text{RKPRF}}$*,* $\mathcal{F}_{\text{UNPRED}}$*,* $\mathcal{F}_{\text{RKCR}}$*,* $\mathcal{F}_{\text{EINT}}$ *such that*

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

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

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. 32. 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 properties are also simple enough that we chose not to define them in a game-based style (the one we require from ME is distinct from, and simpler than, the encoding correctness notion that we defined in Section 3.5). Our security reduction nonetheless introduces and justifies a game hop for each of the these properties. This necessitates the use of 14 security reduction 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 properties). However, some of the reduction steps require a detailed analysis.

Theorem 2 could be stated in a more general way, fully formalising the aforementioned correctness notions and phrasing 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.

Proof phase I: Forging a ciphertext is hard. Let $\mathcal{F}_{\text{INT}}$ be an adversary playing in the INT-security game against channel $\mathsf{MTP\text{-}CH}$. Consider an arbitrary call made by $\mathcal{F}_{\text{INT}}$ to its oracle 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 verify $\mathsf{msg\_key}$ 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 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 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 $m \neq \bot$). 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 captured by the notion of SE's unpredictability (UNPRED) with respect to ME (Section 5.3).

The UNPRED-security of SE, ME 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 decryption algorithm $\mathsf{SE.Dec}(k, \cdot)$ 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 E.6 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 E.6 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 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 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; $\mathsf{MTP\text{-}SE}$ satisfies this property as well.

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, the channel decrypts c to $m = \bot$ 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's state st_u of user u does not change when $\mathcal{F}_{\mathrm{INT}}$ queries its oracle $\textsc{Recv}$ on inputs u, c, aux that get decrypted to $m = \bot$. This could only happen in Case A above, assuming that the $\mathsf{msg\_key}$ verification succeeds but then the $\mathsf{ME.Decode}$ call returns $m = \bot$ and changes the message encoding scheme's state $st_{\mathsf{ME},u}$ of user 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 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 Recv oracle. More specifically, oracle Recv 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 Recv can fetch this payload and immediately call ME.Decode to decode it.

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 $\mathcal{F}_{\text{INT}}$ wins the game iff the message m recovered by ME.Decode inside oracle Recv is not equal to the corresponding output m^* of supp). However, the support function supp uses the MTP-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 $m^* = \bot$ when it is called on a ciphertext that cannot be found in the opposite user's transcript $\mathsf{tr}_{\overline{u}}$.[31]
(2) Robustness of a support function requires that adding failed decryption events (i.e. $m = \bot$) to a transcript does not affect the future outputs of 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 (i.e. $u \rightarrow \overline{u}$ and $\overline{u} \rightarrow u$).

For the last property, we observe that in our game $p_0 = p_1$ iff the corresponding MTP-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 supp = 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 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 MTP-CH to the encoding integrity (EINT) property of ME, supp (see Fig. 15) 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 E.5 we show that this property holds for MTP-ME with respect to SUPP.

Proof of Theorem 2. This proof uses games G_0–G_2 in Fig. 44, games G_3–G_8 in Fig. 46 and games G_9–G_{13} in Fig. 49. The code added for the transitions between games is highlighted in green. The adversaries for transitions between games are provided throughout the proof. The instructions that are highlighted inside adversaries mark the changes in the code of the simulated security reduction games.

Games G_0–G_2 and the transitions between them ($\mathrm{G}_0 \rightarrow \mathrm{G}_1$ based on the OTWIND-security of HASH, and $\mathrm{G}_1 \rightarrow \mathrm{G}_2$ 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.

$\mathbf{G}_0$. Game G_0 is equivalent to game $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\text{INT}}}$. It expands the code of algorithms CH.Init, CH.Send and CH.Recv. The expanded instructions are highlighted in grey. It follows that

$$\mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}_{\text{INT}}) = \Pr[\mathrm{G}_0].$$

$\mathbf{G}_0 \rightarrow \mathbf{G}_1$. The value of auth_key_id in game G_0 depends on the initial KDF key kk and MAC key mk. In contrast, game G_1 computes auth_key_id by evaluating HASH on a uniformly random input x that is independent of kk and mk. We invoke the OTWIND-security of HASH (Fig. 25) in order to claim that adversary $\mathcal{F}_{\text{INT}}$ cannot distinguish between playing in G_0 and G_1. In Fig. 45a we build an adversary $\mathcal{D}_{\text{OTWIND}}$ against the OTWIND-security of HASH. When adversary $\mathcal{D}_{\text{OTWIND}}$ plays in game $\mathrm{G}^{\mathsf{otwind}}_{\mathsf{HASH},\mathcal{D}_{\text{OTWIND}}}$ 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

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

[31] Integrity of a support function is formalised in Appendix A.

```
Games $\mathrm{G}_0$–$\mathrm{G}_2$
$\mathsf{win} \leftarrow \mathsf{false}$
$hk \leftarrow\!\!\$\ \{0,1\}^{\mathsf{HASH.kl}}$ ; $kk \leftarrow\!\!\$\ \{0,1\}^{672}$ ; $mk \leftarrow\!\!\$\ \{0,1\}^{320}$
$x \leftarrow kk \,\|\, mk$                          // $\mathrm{G}_0$
$x \leftarrow\!\!\$\ \{0,1\}^{992}$                        // $\mathrm{G}_1$–$\mathrm{G}_2$ (OTWIND of $\mathsf{HASH}$)
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{ME.Init}()$
$\mathcal{F}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$
Return $\mathsf{win}$

$\textsc{Send}(u, m, aux, r)$    // $u \in \{\mathcal{I}, \mathcal{R}\}$, $m \in \mathsf{CH.MS}$, $r \in \mathsf{CH.SendRS}$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \bot$ then $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_u, \mathsf{msg\_key})$   // $\mathrm{G}_0$–$\mathrm{G}_1$
$k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$                // $\mathrm{G}_2$ (RKPRF of $\mathsf{KDF}$)
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$
Return $c$

$\textsc{Recv}(u, c, aux)$    // $u \in \{\mathcal{I}, \mathcal{R}\}$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \bot$ then $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$   // $\mathrm{G}_0$–$\mathrm{G}_1$
$k \leftarrow \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$                // $\mathrm{G}_2$ (RKPRF of $\mathsf{KDF}$)
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
$\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
$m \leftarrow \bot$
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 $\bot$
```

Figure 44. Games G_0–G_2 for the proof of Theorem 2. The code added by expanding the algorithms of CH in game $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\mathrm{INT}}}$ is highlighted in grey.

$\mathbf{G}_1 \to \mathbf{G}_2$. Going from G_1 to G_2, we switch the outputs of $\mathsf{KDF.Ev}$ 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 a 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. 45b we build an adversary $\mathcal{D}_{\mathrm{RKPRF}}$ against the RKPRF-security of KDF (Fig. 26) with respect to ϕ_{KDF}. When adversary $\mathcal{D}_{\mathrm{RKPRF}}$ plays in game $\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\mathrm{RKPRF}}}$ with challenge bit $d \in \{0,1\}$, it simulates game G_1 (when $d = 1$) or game G_2 (when $d = 0$) for adversary $\mathcal{F}_{\mathrm{INT}}$. Adversary $\mathcal{D}_{\mathrm{RKPRF}}$ returns $d' = 1$ iff $\mathcal{F}_{\mathrm{INT}}$ sets win, so we have

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

$\mathbf{G}_2 \to \mathbf{G}_3$. Game G_3 (Fig. 46) differs from G_2 (Fig. 44) 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 $\textsc{Send}$. We set $\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$ to remember that user u produced $\mathsf{msg\_key}$ when sending (to user $\overline{u}$) an SE ciphertext c_{se}, that encrypts payload p.

```
Adversary $\mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, \mathsf{auth\_key\_id})$
$kk \,\|\, mk \leftarrow x_1$   // s.t. $|kk| = 672$, $|mk| = 320$.
$\mathsf{win} \leftarrow \mathsf{false}$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{KDF}}(kk)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\$\ \mathsf{ME.Init}()$
$\mathcal{F}_{\mathrm{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$
If win then return 1 else return 0

$\textsc{SendSim}(u, m, aux, r)$, $\textsc{RecvSim}(u, c, aux)$
// Identical to oracles $\textsc{Send}$ and $\textsc{Recv}$ in games $\mathrm{G}_0$, $\mathrm{G}_1$ of Fig. 44.
```

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

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

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

$\textsc{RecvSim}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
$k \leftarrow \textsc{RoR}(\overline{u}, \mathsf{msg\_key})$
$p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
$\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$
$m \leftarrow \bot$
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 $\bot$
```

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

Figure 45. The adversaries for games G_0–G_2 of the proof of Theorem 2. Each constructed adversary simulates one or two subsequent games of the security reduction for adversary $\mathcal{F}_{\mathrm{INT}}$.

```
Games $\mathrm{G}_3$–$\mathrm{G}_8$
$\mathsf{win} \leftarrow \mathsf{false}$
$hk \leftarrow\!\!\$\ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\!\!\$\ \{0,1\}^{320}$ ; $x \leftarrow\!\!\$\ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{ME.Init}()$
$\mathcal{F}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$ ; Return $\mathsf{win}$

$\textsc{Send}(u, m, aux, r)$
$st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
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 $\mathsf{S}[u, \mathsf{msg\_key}] \neq \bot$ then
    $(p', c'_{se}) \leftarrow \mathsf{S}[u, \mathsf{msg\_key}]$
    If $p \neq p'$ then
        $\mathsf{bad}_2 \leftarrow \mathsf{true}$
        $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\bot$          // $\mathrm{G}_6$–$\mathrm{G}_8$ (RKCR of $\mathsf{MAC}$)
If $\mathsf{SE.Dec}(k, c_{se}) \neq p$ then
    $\mathsf{bad}_3 \leftarrow \mathsf{true}$
    $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$ ; Return $\bot$              // $\mathrm{G}_7$–$\mathrm{G}_8$ ($\mathsf{SE} = \mathsf{MTP\text{-}SE}$)
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$ ; $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$ ; Return $c$

$\textsc{Recv}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \bot$ then $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$ ; $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$
$\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ ; $m \leftarrow \bot$
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 $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \bot$ then
        If $(m = \bot) \wedge (st_{\mathsf{ME},u} \neq st^*_{\mathsf{ME},u})$ then
            $\mathsf{bad}_0 \leftarrow \mathsf{true}$
            $st_{\mathsf{ME},u} \leftarrow st^*_{\mathsf{ME},u}$                  // $\mathrm{G}_4$–$\mathrm{G}_8$ ($\mathsf{ME} = \mathsf{MTP\text{-}ME}$)
        If $m \neq \bot$ then
            $\mathsf{bad}_1 \leftarrow \mathsf{true}$
            $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \bot)$          // $\mathrm{G}_5$–$\mathrm{G}_8$ (UNPRED of $\mathsf{SE}, \mathsf{ME}$)
    If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] \neq \bot$ then
        $(p', c'_{se}) \leftarrow \mathsf{S}[\overline{u}, \mathsf{msg\_key}]$
        If $p \neq p'$ then
            $\mathsf{bad}_2 \leftarrow \mathsf{true}$
            $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \bot)$          // $\mathrm{G}_6$–$\mathrm{G}_8$ (RKCR of $\mathsf{MAC}$)
        Else if $c_{se} \neq c'_{se}$ then
            $\mathsf{bad}_4 \leftarrow \mathsf{true}$
            $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \bot)$          // $\mathrm{G}_8$ ($\mathsf{SE} = \mathsf{MTP\text{-}SE}$)
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$ ; If $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$ ; Return $\bot$
```

Figure 46. Games G_3–G_8 for the proof of Theorem 2.

(3) Oracle Recv in game G_3, prior to calling $\mathsf{ME.Decode}$, 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 ($G_3 \to G_4$, $G_4 \to G_5$, $G_5 \to G_6$, $G_7 \to G_8$) will do the following. Each of them 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 the 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 encoding scheme'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

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

$\mathbf{G_3 \to G_4}$. Games G_3 and G_4 (Fig. 46) are identical until bad_0 is set. We have

$$\Pr[G_3] - \Pr[G_4] \le \Pr[\mathsf{bad}_0^{G_3}].$$

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 $\mathsf{MTP\text{-}ME}$. But the latter never modifies its state $st_{\mathsf{ME},u}$ when the decoding fails (i.e. $m = \bot$), so

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

```
Adversary $\mathcal{F}_{\text{UNPRED}}^{\text{Expose},\text{Ch}}$
$hk \leftarrow\!\!\$\ \{0,1\}^{\mathsf{HASH.kl}}$
$mk \leftarrow\!\!\$\ \{0,1\}^{320}$
$x \leftarrow\!\!\$\ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{ME.Init}()$
$\mathcal{F}_{\text{INT}}^{\text{SendSim},\text{RecvSim}}$
```

```
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 $\mathsf{S}[u, \mathsf{msg\_key}] = \bot$ then
    $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow$ Expose$(u, \mathsf{msg\_key})$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \bot$ then
    $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KDF.ol}}$
$k \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$
$c_{se} \leftarrow \mathsf{SE.Enc}(k, p)$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
Return $c$
```

```
RecvSim$(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \bot$ then
    Ch$(\overline{u}, \mathsf{msg\_key}, c_{se}, st_{\mathsf{ME},u}, aux)$
If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] \neq \bot$ then
    If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \bot$ 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
            $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
Return $\bot$
```

Figure 47. Adversary $\mathcal{F}_{\text{UNPRED}}$ against the UNPRED-security of SE, ME for the transition between games G_4–G_5.

$\mathbf{G}_4 \to \mathbf{G}_5$. Games G_4 and G_5 (Fig. 46) are identical until bad_1 is set. We have

$$\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] \leq \Pr[\mathsf{bad}_1^{\mathrm{G}_5}].$$

When the bad_1 flag is set in G_5, we know that the SE key $k = \mathsf{T}[\overline{u}, \mathsf{msg\_key}]$ was sampled uniformly at random and never used inside the SEND oracle before (because $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \bot$). Yet the adversary $\mathcal{F}_{\text{INT}}$ 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 $\mathcal{F}_{\text{INT}}$ 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 $\mathcal{F}_{\text{INT}}$ 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 fraction of them). We use the unpredictability notion of SE with respect to ME, which captures this intuition. In Fig. 47 we build an adversary $\mathcal{F}_{\text{UNPRED}}$ against the UNPRED-security of SE, ME (Fig. 35) as follows. When adversary $\mathcal{F}_{\text{UNPRED}}$ plays in game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}_{\text{UNPRED}}}$, it simulates game G_5 for adversary $\mathcal{F}_{\text{INT}}$. Adversary $\mathcal{F}_{\text{UNPRED}}$ wins in its own game whenever $\mathcal{F}_{\text{INT}}$ sets bad_1, so we have

$$\Pr[\mathsf{bad}_1^{\mathrm{G}_5}] \leq \mathsf{Adv}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME}}(\mathcal{F}_{\text{UNPRED}}).$$

We now explain the ideas behind the construction of $\mathcal{F}_{\text{UNPRED}}$. Adversary $\mathcal{F}_{\text{UNPRED}}$ 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 RECV oracle. This is because supp's outputs do not affect the input-output behaviour of the simulated oracles SEND and RECV, and because this reduction step does not rely on whether adversary $\mathcal{F}_{\text{INT}}$ manages 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 transcripts.

Adversary $\mathcal{F}_{\text{UNPRED}}$ splits the simulation of game G_5's RECV oracle into two cases:

(1) If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \bot$, then $\mathcal{F}_{\text{UNPRED}}$ does not modify $st_{\mathsf{ME},u}$; this is consistent with the behaviour of oracle RECV in game G_5. In addition, adversary $\mathcal{F}_{\text{UNPRED}}$ also makes a call to its oracle CH. The CH oracle simulates all instructions that would have been evaluated by RECV when $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \bot$, except it omits the condition checking $(\mathsf{msg\_key}' = \mathsf{msg\_key}) \wedge (\mathsf{auth\_key\_id} = \mathsf{auth\_key\_id}')$. The omitted condition is a prerequisite to setting flag bad_1 in game G_5; this change is fine because adversary $\mathcal{F}_{\text{UNPRED}}$ will nonetheless set the win flag in its game $\mathrm{G}^{\mathsf{unpred}}_{\mathsf{SE},\mathsf{ME},\mathcal{F}_{\text{UNPRED}}}$ whenever the simulated adversary $\mathcal{F}_{\text{INT}}$ would have set the bad_1 flag in G_5.

(2) If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] \neq \bot$, then $\mathcal{F}_{\text{UNPRED}}$ honestly simulates all instructions that would have been evaluated by RECV.

Finally, adversary $\mathcal{F}_{\text{UNPRED}}$ uses its EXPOSE oracle to learn the values from the PRF table that is maintained by the UNPRED-security game, and synchronises them with its own PRF table T inside the simulated oracle SEND (intuitively, this appears unnecessary, but it helps us avoid further analysis to show that $\mathcal{F}_{\text{UNPRED}}$ perfectly simulates game G_5).

$\mathbf{G}_5 \to \mathbf{G}_6$. Games G_5 and G_6 (Fig. 46) are identical until bad_2 is set. We have

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

Game G_5 sets the bad_2 flag in two different places: one inside oracle SEND, and one inside oracle 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 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 (Fig. 27) with respect to ϕ_{MAC}. In Fig. 48 we build an adversary $\mathcal{F}_{\text{RKCR}}$ against the RKCR-security of MAC with respect to ϕ_{MAC} as follows. When adversary $\mathcal{F}_{\text{RKCR}}$ plays in game $\mathrm{G}^{\mathsf{rkcr}}_{\mathsf{MAC},\phi_{\mathsf{MAC}},\mathcal{F}_{\text{RKCR}}}$, it simulates game G_5 for adversary $\mathcal{F}_{\text{INT}}$. Adversary $\mathcal{F}_{\text{RKCR}}$ wins in its own game whenever $\mathcal{F}_{\text{INT}}$ sets bad_2, so we have

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

$\mathbf{G}_6 \to \mathbf{G}_7$. Games G_6 and G_7 (Fig. 46) are identical until bad_3 is set. We have

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

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 $\mathsf{MTP\text{-}SE}$ 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 (Fig. 46) are identical until bad_4 is set. We have

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

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 $\textsc{Recv}$ call, and (2) $c'_{se} \leftarrow \mathsf{SE.Enc}(k,p)$ was computed during an earlier call to $\textsc{Send}$, 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

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

```
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 $\mathsf{out}$
```

```
$\textsc{SendSim}(u, m, aux, r)$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \bot$ 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 $\mathsf{S}[u, \mathsf{msg\_key}] \neq \bot$ then
    $(p', c'_{se}) \leftarrow \mathsf{S}[u, \mathsf{msg\_key}]$
    If $p \neq p'$ then $\mathsf{out} \leftarrow (u, p, p')$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
Return $c$

$\textsc{RecvSim}(u, c, aux)$
$(\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se}) \leftarrow c$
If $\mathsf{T}[\overline{u}, \mathsf{msg\_key}] = \bot$ 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
        $st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
        $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
        If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] = \bot$ then
            $(st_{\mathsf{ME},u}, m) \leftarrow (st^*_{\mathsf{ME},u}, \bot)$
        If $\mathsf{S}[\overline{u}, \mathsf{msg\_key}] \neq \bot$ then
            $(p', c'_{se}) \leftarrow \mathsf{S}[\overline{u}, \mathsf{msg\_key}]$
            If $p \neq p'$ then $\mathsf{out} \leftarrow (\overline{u}, p, p')$
Return $\bot$
```

Figure 48. Adversary $\mathcal{F}_{\mathrm{RKCR}}$ against the RKCR-security of MAC for the transition between games G_5–G_6.

```
Games $\mathrm{G}_9$–$\mathrm{G}_{13}$
$\mathsf{win} \leftarrow \mathsf{false}$
$hk \leftarrow\!\!\$\ \{0,1\}^{\mathsf{HASH.kl}}$ ; $mk \leftarrow\!\!\$\ \{0,1\}^{320}$ ; $x \leftarrow\!\!\$\ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{ME.Init}()$
$\mathcal{F}_{\mathrm{INT}}^{\textsc{Send},\textsc{Recv}}$ ; Return $\mathsf{win}$

$\textsc{Send}(u, m, aux, r)$
$st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
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 $\bot$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, c, aux)$          // $\mathrm{G}_9$–$\mathrm{G}_{11}$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, p, aux)$          // $\mathrm{G}_{12}$–$\mathrm{G}_{13}$ $(\mathsf{supp} = \mathsf{SUPP})$
$\mathsf{P}[u, c] \leftarrow p$
Return $c$

$\textsc{Recv}(u, c, aux)$
If $\mathsf{P}[\overline{u}, c] \neq \bot$ then   // $\exists m', aux' : (\mathsf{sent}, m', c, aux') \in \mathsf{tr}_{\overline{u}}$
    $p \leftarrow \mathsf{P}[\overline{u}, c]$
    $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
    $m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$ }
    $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$    }  // $\mathrm{G}_9$–$\mathrm{G}_{11}$
    $m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, p, aux)$ }
    $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, p, aux)$    }  // $\mathrm{G}_{12}$–$\mathrm{G}_{13}$ $(\mathsf{supp} = \mathsf{SUPP})$
Else
    $m \leftarrow \bot$ ; $m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
    If $m^* \neq \bot$ then
        $\mathsf{bad}_5 \leftarrow \mathsf{true}$
        $m^* \leftarrow \bot$                         // $\mathrm{G}_{10}$–$\mathrm{G}_{13}$ $(\mathsf{supp} = \mathsf{SUPP})$
    $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$      // $\mathrm{G}_9$–$\mathrm{G}_{10}$ $(\mathsf{supp} = \mathsf{SUPP})$
If $m \neq m^*$ then
    $\mathsf{bad}_6 \leftarrow \mathsf{true}$
    $\mathsf{win} \leftarrow \mathsf{true}$                    // $\mathrm{G}_9$–$\mathrm{G}_{12}$ (EINT of $\mathsf{ME}, \mathsf{supp}$)
Return $\bot$
```

Figure 49. Games G_9–G_{13} for the proof of Theorem 2. The code highlighted in grey is functionally equivalent to the corresponding code in G_8.

$\mathbf{G}_8 \to \mathbf{G}_9$. While discussing this and subsequent transitions, we say that a ciphertext c belongs to (or appears in) a support transcript tr if and only if $\exists m', aux'\colon (\mathsf{sent}, m', c, aux') \in \mathsf{tr}$.

Consider oracle Recv in game G_8 (Fig. 46). Let $st^*_{\mathsf{ME},u}$ contain the value of variable $st_{\mathsf{ME},u}$ at the start of the ongoing call to Recv on inputs (u, c, aux). We start by showing that Recv evaluates $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u},\, p,\, aux)$ *and* does not subsequently roll back the values of $(st_{\mathsf{ME},u}, m)$ to $(st^*_{\mathsf{ME},u}, \bot)$ 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 MTP-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 $(st^*_{\mathsf{ME},u}, \bot)$.

Game G_9 (Fig. 49) differs from game G_8 (Fig. 46) 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 MTP-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 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 $m \leftarrow \bot$. 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 $\mathsf{P}[\overline{u}, c] \neq \bot$. For simplicity, we do the latter; and if the condition is satisfied, then we set $p \leftarrow \mathsf{P}[\overline{u}, c]$ and run ME.Decode with this payload as input. As discussed above, the MTP-CH ciphertext c that is issued by user $\overline{u}$ always encrypts a unique payload p, and hence we can rely on the fact 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 (here we rely on the fact that values $u, \mathsf{msg\_key}, p$ uniquely determine c_{se}). It also adds one new conditional statement to oracle Recv (checking $m^* \neq \bot$), 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} (Fig. 49) enforces that $m^* = \bot$ whenever 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. We have

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

If bad_5 is set in game G_9 then the support function supp returned $m^* \neq \bot$ in response to an MTP-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 $m^* = \bot$ when its input label does not appear in $\mathsf{tr}_{\overline{u}}$, so

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

We refer to this property as the *integrity* of support function supp. We formalise it in Appendix A.

$\mathbf{G}_{10} \to \mathbf{G}_{11}$. Game G_{11} (Fig. 49) stops adding entries of the form $(\mathsf{recv}, \bot, c, aux)$ 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 $(\mathsf{recv}, \bot, c, aux)$, so removing the instruction $\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$ for $m = \bot$ will not affect the outputs of any future calls to this support function. We have

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

Earlier in this section we referred to this property as the *robustness* of support function supp.

$\mathbf{G}_{11} \to \mathbf{G}_{12}$. When discussing the differences between games G_8 and G_9, we showed that for each pair of sender $u \in \{\mathcal{I}, \mathcal{R}\}$ and $\mathsf{MTP\text{-}CH}$ 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 $\mathsf{MTP\text{-}CH}$ ciphertext c that encrypts p in the direction from u to $\overline{u}$. It follows that in games G_{11} and G_{12} (Fig. 49) for any fixed user $u \in \{\mathcal{I}, \mathcal{R}\}$ there is a 1-to-1 correspondence between payloads and $\mathsf{MTP\text{-}CH}$ ciphertexts that could be successfully sent from u to $\overline{u}$ (note that this property does not hold if SE does not have decryption correctness, but the code added for the transition $\mathrm{G}_6 \to \mathrm{G}_7$ 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 checks only its equality with every z^* such that $(\mathsf{sent}, m, z^*, aux) \in \mathsf{tr}_u$ or $(\mathsf{recv}, m, z^*, aux) \in \mathsf{tr}_{\overline{u}}$ across all 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 support transcripts from c to p, and replace the label inputs to the support function SUPP in the same way; this does not change the outputs of the support function. We have

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

$\mathbf{G}_{12} \to \mathbf{G}_{13}$. Games G_{12} and G_{13} are identical until bad_6 is set. We have

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

Games G_{12} and G_{13} (Fig. 49) 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, replay, reorder and drop the payloads; but it is not allowed to forge them. This description roughly corresponds to the definition of EINT-security of ME with respect to supp (Fig. 15). In games G_{12}–G_{13} the oracle SEND still runs cryptographic algorithms in order to generate and return $\mathsf{MTP\text{-}CH}$ ciphertexts, but we will build an EINT-security adversary that simulates these instructions for $\mathcal{F}_{\mathrm{INT}}$. In Fig. 50 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. □

```
Adversary $\mathcal{F}_{\text{EINT}}^{\textsc{Send},\textsc{Recv}}(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
$hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{HASH.kl}}$
$mk \leftarrow\!\!\$ \{0,1\}^{320}$
$x \leftarrow\!\!\$ \{0,1\}^{992}$
$\mathsf{auth\_key\_id} \leftarrow \mathsf{HASH.Ev}(hk, x)$
$(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$\mathcal{F}_{\text{INT}}^{\textsc{SendSim},\textsc{RecvSim}}$
```

```
$\textsc{SendSim}(u, m, aux, r)$
$st^*_{\mathsf{ME},u} \leftarrow st_{\mathsf{ME},u}$
$(st_{\mathsf{ME},u}, p) \leftarrow \mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux; r)$
$\mathsf{msg\_key} \leftarrow \mathsf{MAC.Ev}(mk_u, p)$
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 $\bot$
$\mathsf{S}[u, \mathsf{msg\_key}] \leftarrow (p, c_{se})$
$c \leftarrow (\mathsf{auth\_key\_id}, \mathsf{msg\_key}, c_{se})$
$\textsc{Send}(u, m, aux, r)$
$\mathsf{P}[u, c] \leftarrow p$
Return $c$

$\textsc{RecvSim}(u, c, aux)$
If $\mathsf{P}[\overline{u}, c] \neq \bot$ then
  $p \leftarrow \mathsf{P}[\overline{u}, c]$
  $(st_{\mathsf{ME},u}, m) \leftarrow \mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$
  $\textsc{Recv}(u, p, aux)$
Return $\bot$
```

Figure 50. Adversary $\mathcal{F}_{\text{EINT}}$ against the EINT-security of $\mathsf{ME}, \mathsf{supp}$ for the transition between games G_{12}–G_{13} in the proof of Theorem 2.

Proof alternatives. 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 does not rely on such a property of ME.

In Case A adversary $\mathcal{F}_{\text{INT}}$ calls its oracle $\textsc{Recv}(u, c, aux)$ on $c = (\mathsf{auth\_key\_id}', \mathsf{msg\_key}, c_{se})$ with a msg_key value that was never previously returned by oracle $\textsc{Send}$ as a part of a ciphertext produced by user $\overline{u}$. Let us modify our initial goal for Case A as follows: we want to show that evaluating $k \leftarrow \mathsf{KDF.Ev}(kk_{\overline{u}}, \mathsf{msg\_key})$, $p \leftarrow \mathsf{SE.Dec}(k, c_{se})$ and $\mathsf{msg\_key}' \leftarrow \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$ is very unlikely to result in $\mathsf{msg\_key}' = \mathsf{msg\_key}$. In fact, it is sufficient to focus on the last instruction here: we require that it is hard to forge any input-output pair $(p, \mathsf{msg\_key})$ such that $\mathsf{msg\_key} = \mathsf{MAC.Ev}(mk_{\overline{u}}, p)$. This property is guaranteed if MAC is related-key PRF-secure.

Theorem 2 is currently stated for a generic function family MAC, but it could be narrowed down to use MAC = MTP-MAC where $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) = \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$. Crucially, the algorithm MTP-MAC.Ev is defined to drop half of the output bits of SHA-256; this prevents length extension attacks. We could model MTP-MAC as the Augmented MAC (AMAC), and use the results from [BBT16] to show that it is related-key PRF-secure. Technically, this would require proving three claims as follows:

(1) Output of the first compression function within $\mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64:192]$ looks uniformly random when used with related keys; we already formalise and analyse this property in Section 5.2, phrased as the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC}.

(2) The SHA-256 compression function h_{256} is OTPRF-secure.

(3) The SHA-256 compression function is (roughly) PRF-secure even in the presence of some leakage on its key, i.e. an attacker receives $k[64:192]$ when trying to break the PRF-security of $h_{256}(k, \cdot)$; we do not formalise or analyse this property in our work.

Here (1) and (2) could be chained together to show that MTP-MAC is a secure PRF even for variable-length inputs; then (3) would suffice to show that MTP-MAC is resistant to length extension attacks.

Adopting the above proof strategy would have allowed us to omit the following two steps from the current security reduction. The UNPRED-security of SE, ME would get directly replaced with a new related-key PRF-security assumption for MAC = MTP-MAC, following the results for AMAC from [BBT16]. The RKPRF-security of KDF (with respect to ϕ_{KDF}) would no longer be needed, because currently its only use is to transform the security game prior to appealing to the UNPRED-security of SE, ME.

5.7 Instantiation and interpretation

We are now ready to combine the theorems from the previous two sections with the notions defined in Section 5.1 and Section 5.3 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* $\mathit{session\_id} \in \{0,1\}^{64}$, $\mathit{pb} \in \mathbb{N}$ *and* $\mathsf{bl} = 128$. *Let* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathit{session\_id}, \mathit{pb}, \mathsf{bl}]$, MTP-HASH, MTP-MAC, MTP-KDF, ϕ_{MAC}, ϕ_{KDF}, MTP-SE *be the primitives of MTProto defined in Section 4.4. Let* $\mathsf{CH} = \mathsf{MTP\text{-}CH}[\mathsf{ME}, \mathsf{MTP\text{-}HASH}, \mathsf{MTP\text{-}MAC}, \mathsf{MTP\text{-}KDF}, \phi_{\mathsf{MAC}}, \phi_{\mathsf{KDF}}, \mathsf{MTP\text{-}SE}]$. *Let* $\phi_{\mathsf{SHACAL\text{-}2}}$ *be the related-key-deriving function defined in Fig. 29. Let* h_{256} *be the* SHA-256 *compression function, and let* H *be the corresponding function family with* $\mathsf{H.Ev} = h_{256}$, $\mathsf{H.kl} = \mathsf{H.ol} = 256$ *and* $\mathsf{H.In} = \{0,1\}^{512}$. *Let* $\ell \in \mathbb{N}$. *Let* $\mathcal{D}_{\mathrm{IND}}$ *be any adversary against the* IND*-security of* CH, *making* $q_{\mathrm{CH}} \leq 2^{96}$ *queries to its* CH *oracle, each query made for a message of length at most* $\ell \leq 2^{27}$ *bits.*[32] *Then we can build adversaries* $\mathcal{D}^{\mathsf{shacal}}_{\mathrm{OTPRF}}$, $\mathcal{D}_{\mathrm{LRKPRF}}$, $\mathcal{D}_{\mathrm{HRKPRF}}$, $\mathcal{D}^{\mathsf{compr}}_{\mathrm{OTPRF}}$, $\mathcal{D}_{\mathrm{OTIND\$}}$ *such that*

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

Corollary 1 follows from Theorem 1 together with Proposition 5, Proposition 6, Proposition 7 with Lemma 1 and Proposition 8. The two terms in Theorem 1 related to ME are zero for ME = MTP-ME when an adversary is restricted to making $q_{\mathrm{CH}} \leq 2^{96}$ queries. Qualitatively, Corollary 1 shows that the confidentiality of the MTProto-based channel depends on whether SHACAL-1 and SHACAL-2 can be considered as pseudorandom functions in a variety of modes: with keys used only once, related keys, partially chosen-keys when evaluated on fixed inputs and when the key and input switch positions. Especially the related-key assumptions (LRKPRF and HRKPRF given in Section 5.2) are highly unusual; in Appendix F we show that both assumptions hold in the ideal cipher model, but both of them 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.

[32] The length of plaintext m in MTProto is $\ell := |m| \leq 2^{27}$ bits. To build a payload p, algorithm ME.Encode prepends a 256-bit header, and appends at most $\mathsf{bl} \cdot (\mathsf{pb}+1)$-bit padding. Further evaluation of MAC on p might append at most 512 additional bits of SHA padding. So this corollary uses Lemma 1 with the maximum number of blocks $T = \lfloor (256 + \ell + \mathsf{bl} \cdot (\mathsf{pb}+1) + 512)/512 \rfloor$ minus the first 512-bit block that is processed separately in Proposition 7.

Corollary 2. *Let* $\textsf{\textit{session\_id}} \in \{0,1\}^{64}$*,* $\textsf{\textit{pb}} \in \mathbb{N}$ *and* $\textsf{bl} = 128$*. Let* $\textsf{ME} = \textsf{MTP-ME}[\textsf{\textit{session\_id}}, \textsf{\textit{pb}}, \textsf{bl}]$*,* $\textsf{MTP-HASH}$*,* $\textsf{MTP-MAC}$*,* $\textsf{MTP-KDF}$*,* $\phi_{\textsf{MAC}}$*,* $\phi_{\textsf{KDF}}$*,* $\textsf{MTP-SE}$ *be the primitives of MTProto defined in Section 4.4. Let* $\textsf{CH} = \textsf{MTP-CH}[\textsf{ME}, \textsf{MTP-HASH}, \textsf{MTP-MAC}, \textsf{MTP-KDF}, \phi_{\textsf{MAC}}, \phi_{\textsf{KDF}}, \textsf{MTP-SE}]$*. Let* $\phi_{\textsf{SHACAL-2}}$ *be the related-key-deriving function defined in Fig. 29. Let* $\textsf{SHA-256}'$ *be* $\textsf{SHA-256}$ *with its output truncated to the middle 128 bits. Let* $\textsf{supp} = \textsf{SUPP}$ *be the support function as defined in Fig. 32. Let* $\mathcal{F}_{\text{INT}}$ *be any adversary against the* INT*-security of* $\textsf{CH}$ *with respect to* $\textsf{supp}$*, making* $q_{\text{SEND}} \leq 2^{96}$ *queries to its* SEND *oracle. Then we can build adversaries* $\mathcal{D}_{\text{OTPRF}}$*,* $\mathcal{D}_{\text{LRKPRF}}$*,* $\mathcal{F}_{\text{CR}}$ *such that*

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

Corollary 2 follows from Theorem 2 together with Proposition 5, Proposition 6 and Proposition 11. The term $\textsf{Adv}^{\textsf{eint}}_{\textsf{MTP-ME},\textsf{SUPP}}(\mathcal{F}_{\text{EINT}})$ from Theorem 2 resolves to 0 for adversaries making $q_{\text{SEND}} \leq 2^{96}$ queries according to Proposition 9. Qualitatively, Corollary 2 shows that also the integrity of the MTProto-based channel depends on SHACAL-1 and SHACAL-2 behaving as PRFs. Due to the way MTP-MAC is constructed, the result also depends on the collision resistance of truncated-output SHA-256 (as discussed in Section 5.1). Quantitatively, the advantage is again bounded by $q_{\text{SEND}} < 2^{64}$. This bound follows from the fact that the first block of payload contains a 64-bit constant session_id which has to match upon decoding. If the MTProto message encoding scheme consistently checked more fields during decoding (especially in the first block), the bound could be improved.

6 Timing side-channel attack

The formal model and proof we gave in the previous sections do not provide full guarantees about the security of MTProto, which we illustrate in this section. Going beyond the model, 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 [APW09]: 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 [APW09]. We begin by describing a generic way to overcome this obstacle in Section 6.1. We describe the side channels found in the implementations of several Telegram clients in Section 6.2 and experimentally demonstrate the existence of a timing side channel in the desktop client in Section 6.3.

6.1 Manipulating IGE

Suppose we intercept an IGE ciphertext c consisting of t blocks (for any block cipher E): $c_1 \mid c_2 \mid \ldots \mid c_t$ where $\mid$ denotes a block boundary. Further, suppose we have a side channel that enables us to learn some bits of m_2, the second plaintext block.[33] 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 2). 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).

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

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. Section 7 gives an attack to recover these values. Note that these values are the same for all ciphertexts within a single session, so if they were recovered, then we could carry out the attack on each of the ciphertexts in turn. This allows the basic attack above to be iterated when the target m_i is fixed across all the ciphertexts, e.g. in order to amplify the total information learned about m_i when a single ciphertext allows to infer only a partial or noisy information about it (cf. [APW09]).

6.2 Leaky length field

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

The msg_length field occupies the last four bytes of the second block of every MTProto cloud message plaintext (see Section 4.1). 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.

Android. The field msg_length is referred to as `messageLength` here. The check is performed in `decryptServerResponse` of `Datacenter.cpp` [Tel21i], 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;
```

Desktop. The method `handleReceived` of `session_private.cpp` [Tel21l] performs the length check, comparing the `messageLength` field with a fixed value of `kMaxMessageLength` $= 2^{24}$. When this check fails, the connection is closed and no MAC check is performed, providing a potentially large timing difference. Because of the fixed value 2^{24}, this check would leak the 8 most significant bits of the target block m_i with probability 2^{-8}, i.e. the eight most significant bits of the 32-bit length field, allowing those bits to be recovered after about 2^8 attempts on average.[34]

```
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
```

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

iOS. The field msg_length is referred to as messageDataLength here. The check is performed in _decryptIncomingTransportData of MTProto.m [Tel21m], which compares messageDataLength 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[35], 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;
```

Discussion. Note that all three of the above implementations were in violation of Telegram's own security guidelines [Tel21e] which state: "If an error is encountered before this check could be performed, the client must perform the msg_key check anyway before returning any result. Note that the response to any error encountered before the msg_key check must be the same as the response to a failed msg_key check." In contrast, TDLib [Tel21g], the cross-platform library for building Telegram clients, does avoid timing leaks by running the MAC check first.

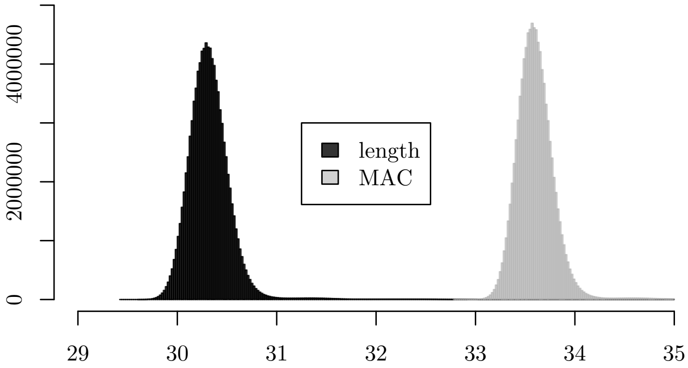
Remark 2. Recall that in Section 4.4, we define a simplified message encoding scheme which uses a constant in place of session_id and server_salt. This change would make the above attack more practical. However, the attack is enabled by a misplaced msg_key check and the mitigation offered by those values 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.

6.3 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 G.1. 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. 51. This should be sufficiently large for a remote attacker to detect, even with network and other noise sources (cf. [AP13], where sub-microsecond timing differences were successfully resolved over a LAN).

[35] 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.

Figure 51. 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

7 Attacking the key exchange

Recall that our attack in Section 6 relies on knowledge of m_1 which in MTProto contains a 64-bit salt and a 64-bit session ID. In Section 7.1, 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 Section 7.2.

We stress, however, that the attack in Section 7.1 only applies in a short period after a key exchange between a client and a server.[36] Furthermore, the attack critically relies 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 Section 7.3, we then discuss how the attack in Section 7.1 enables to break server authentication and thus enables an attacker-in-the-middle (MitM) attack on the Diffie-Hellman key exchange.

7.1 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.[37] 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 [MH20,AH21] for an introduction to the application of lattice reduction in side-channel attacks and the state of the art respectively.

In Fig. 52 we show Telegram's instantiation of the Diffie-Hellman key exchange [Tel21n] at the level of detail required for our attack, omitting TL schema encoding. In Fig. 52, we let $n :=$ nonce, $s :=$ server_nonce, $n' :=$ 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, $t_s :=$ server_time be a timestamp for the server; let $\mathcal{F}(\cdot,\cdot)$ be some function used to derive keys;[38] let p_r, p_s, p_c be random padding of appropriate

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

[37] We note that this issue mirrors the one reported in [JO16].

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

length; and $ak \coloneqq \mathsf{auth\_key}$ be the final key. The initial salt used by Telegram is then computed as $\mathsf{server\_salt} \coloneqq n'[0:64] \oplus s[0:64]$. Since s is sent in the clear during the key exchange protocol, recovering the salt is equivalent to recovering $n'[0:64]$. We let N', e denote the public RSA key (modulus and exponent) used to perform RSA encryption by the client in the key exchange and we let d denote the private RSA exponent used by the server to perform RSA decryption.[39] We assume N' has exactly 2048 bits which holds for the values used by Telegram.

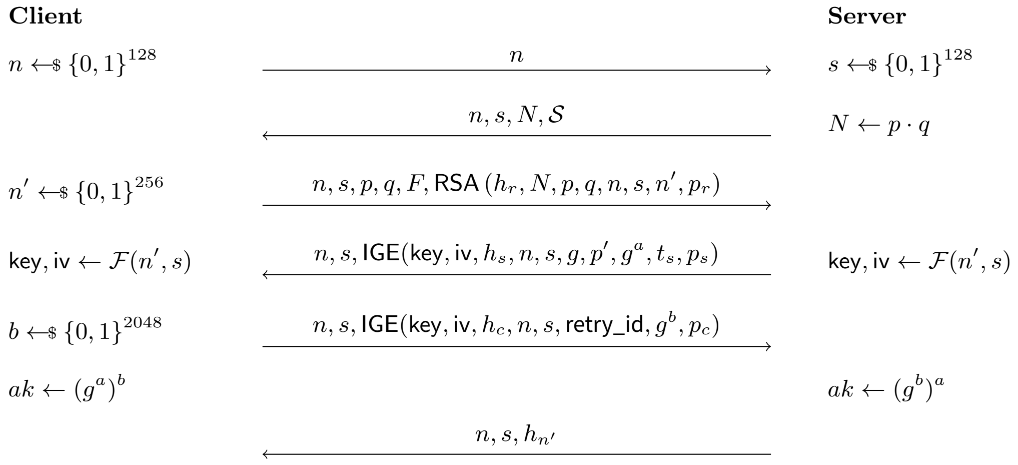

Client
Server
$n \leftarrow\!\!\$ \{0,1\}^{128}$
n
$s \leftarrow\!\!\$ \{0,1\}^{128}$
$n, s, N, \mathcal{S}$
$N \leftarrow p \cdot q$
$n' \leftarrow\!\!\$ \{0,1\}^{256}$
$n, s, p, q, F, \mathsf{RSA}\,(h_r, N, p, q, n, s, n', p_r)$
$\mathsf{key}, \mathsf{iv} \leftarrow \mathcal{F}(n', s)$
$n, s, \mathsf{IGE}(\mathsf{key}, \mathsf{iv}, h_s, n, s, g, p', g^a, t_s, p_s)$
$\mathsf{key}, \mathsf{iv} \leftarrow \mathcal{F}(n', s)$
$b \leftarrow\!\!\$ \{0,1\}^{2048}$
$n, s, \mathsf{IGE}(\mathsf{key}, \mathsf{iv}, h_c, n, s, \mathsf{retry\_id}, g^b, p_c)$
$ak \leftarrow (g^a)^b$
$ak \leftarrow (g^b)^a$
$n, s, h_{n'}$

Figure 52. Telegram Key Exchange, where $\mathsf{IGE} = \mathsf{IGE}[\mathsf{AES\text{-}256}]$.

Further, we have

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

in Fig. 52 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 corresponding payloads *except* for the padding p_r, p_s, p_c. In particular, we have

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

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;[40] 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.

[39] 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.

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

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 [Tel20d]. 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 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:

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

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 returns 160 bits) of $m' := (c')^d$ match `0xec5ac983` for any c' we submit to the Telegram servers. Thus, inspired by [Ble98], 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 $\zeta =$ `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 $i = 0, \ldots, \mu - 1$.

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.

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

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

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

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

$$(2^{1688} \cdot h_r,\ -2^{1688}\,\sigma_0,\ \ldots,\ -2^{1688}\,\sigma_{\mu-1},\ e'_0, \ldots,\ e'_{\mu-1},\ 2^{1848}) \tag{1}$$

where all entries are bounded by $2^{1848} = 2^{1688+160}$. Thus that vector has Euclidean norm $\leq \sqrt{2\,\mu + 2} \cdot 2^{1848}$.[41] On the other hand, the Gaussian heuristic predicts the shortest vector in the lattice to have norm

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

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.2). Thus, under our assumptions, recovering h_r requires about $6 \cdot 2^{32}$ queries to Telegram's servers and a trivial amount of computation.

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

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

where the s'_i are again chosen randomly and we collect s'_i for $i = 0, \ldots, \mu' - 1$ 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.[42] 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.[43] 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

[41] This estimate is pessimistic for the attacker. Applying the techniques summarised in [AH21] for constructing such lattices, we can save a factor of roughly two. We forgo these improvements here to keep the presentation simple.

[42] 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 [AH21]. However, for simplicity we forgo a discussion of this variant here.

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

part of the lower-order bit noise. Thus, we pick random $s'_i \approx 2^{1848-\ell(p_r)}$ and consider $d'_i \coloneqq d_i - s'_i \cdot p_r$ 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

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

As before, multiplying L_2 from the left by

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

shows that this lattice contains a vector

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

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.

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 2.2) 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. [AP13,MBA+20] 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 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 [AH21]. 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.

7.2 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 6. Here, we simply run our attack from Section 6 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 such that

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

passes the length check.[44] 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).

7.3 Breaking server authentication

Recall from Fig. 52 that the key, iv pair used to encrypt g^a and g^b are derived from s (sent in the clear) and n'. Since the attack in Section 7.1 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 Section 7.1, 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'.

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

8 Discussion

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

Attacks. Our work also presents attacks against the symmetric encryption in Telegram. These highlight the gap between the variant of MTProto 2.0 that we model and Telegram's implementations. While the reordering attack in Section 4.2 and the attack on IND-CPA security in Section 4.2 were possible against implementations that we studied, they can easily be avoided without making changes to the on-the-wire format of MTProto, i.e. by only changing processing in clients and servers. After disclosing our findings, Telegram informed us that they have changed this processing accordingly.

Our attacks in Section 6 are attacks on the implementation. As such, they can be considered outside the model: our model only shows that there *can* be secure instantiations of MTProto but does not cover the actual implementations; in particular, we do not model timing differences. That said, protocol design has a significant impact on the ease with which secure implementations can be achieved. Here, the decision in MTProto to adopt Encrypt & MAC results in the potential for a leak that we can exploit in specific implementations. This "brittleness" of MTProto is of particular relevance due to the surfeit of implementations of the protocol, and the fact that security advice may not be heeded by all authors, as we showed with our msg_length attack in Section 6. Here Telegram's apparent ambition to provide TDLib as a one-stop solution for clients across platforms will allow security researchers to focus their efforts. We thus recommend that Telegram replaces the low-level cryptographic processing in all official clients with a carefully vetted library.

Note that the security of the Telegram ecosystem does not stop with official clients. As the recent work of [vAP22] shows, many third-party client implementations are also vulnerable to attacks.

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

Future work. Our attack in Section 7 is against the implementation of Telegram's key exchange and is thus outside of our model for two reasons: as before, we do not consider timing side channels in our model and, critically, we only model the symmetric part of MTProto. This highlights a second significant caveat for our results that large parts of Telegram's design remain unstudied: multi-user security, the key exchange, the higher-level message processing, secret chats, forward secrecy, control messages, bot APIs, CDNs, cloud storage, the Passport feature, to name but a few. These are pressing topics for future work.

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

Telegram. While we prove security of the symmetric part of MTProto at a protocol level, we recall that by default communication via Telegram must trust the Telegram servers, i.e. end-to-end encryption is optional and not available for group chats. We thus, on the one hand, (a) recommend that Telegram open-sources the cryptographic processing on their servers and (b) recommend to avoid referencing Telegram as an "encrypted messenger" which – post-Snowden – has come to mean end-to-end encryption. On the other hand, discussions about end-to-end encryption aside, echoing [EHM17,ABJM21] we note that many higher-risk users *do* rely on MTProto and Telegram and shun Signal. This emphasises the need to study these technologies and how they serve those who rely on them.

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. The bulk of this work was done while Albrecht was at Royal Holloway, University of London.

References

ABJM21. Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: the case of hong kong. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 3363–3380. USENIX Association, August 2021.

ABL$^+$14. Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. How to securely release unverified plaintext in authenticated encryption. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 105–125. Springer, Heidelberg, December 2014.

ACD19. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Ishai and Rijmen [IR19], pages 129–158.

AH21. Martin R. Albrecht and Nadia Heninger. On bounded distance decoding with predicate: Breaking the "lattice barrier" for the hidden number problem. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 528–558. Springer, Heidelberg, October 2021.

AMPS22. Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for telegram. In *2022 IEEE Symposium on Security and Privacy*, pages 87–106. IEEE Computer Society Press, May 2022.

AP13. Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society Press, May 2013.

APW09. Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *2009 IEEE Symposium on Security and Privacy*, pages 16–26. IEEE Computer Society Press, May 2009.

BBKN12. Mihir Bellare, Alexandra Boldyreva, Lars R. Knudsen, and Chanathip Namprempre. On-line ciphers and the hash-CBC constructions. *Journal of Cryptology*, 25(4):640–679, October 2012.

BBT16. Mihir Bellare, Daniel J. Bernstein, and Stefano Tessaro. Hash-function based PRFs: AMAC and its multi-user security. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 566–595. Springer, Heidelberg, May 2016.

BCK96. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th FOCS*, pages 514–523. IEEE Computer Society Press, October 1996.

BDJR97. Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403. IEEE Computer Society Press, October 1997.

BHMS16. Colin Boyd, Britta Hale, Stig Frode Mjølsnes, and Douglas Stebila. From stateless to stateful: Generic authentication and authenticated encryption constructions with application to TLS. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 55–71. Springer, Heidelberg, February / March 2016.

BK03. Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 491–506. Springer, Heidelberg, May 2003.

BKN02. Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 1–11. ACM Press, November 2002.

BKN04. Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):206–241, 2004.

Ble98. Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 1–12. Springer, Heidelberg, August 1998.

BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Vaudenay [Vau06], pages 409–426.

BSJ+17. Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650. Springer, Heidelberg, August 2017.

Cam78. C. Campbell. Design and specification of cryptographic capabilities. *IEEE Communications Society Magazine*, 16(6):15–19, 1978.

CDMP05. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, Heidelberg, August 2005.

CDV21. Andrea Caforio, F. Betül Durak, and Serge Vaudenay. Beyond security and efficiency: On-demand ratcheting with security awareness. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 649–677. Springer, Heidelberg, May 2021.

DF18. Jean Paul Degabriele and Marc Fischlin. Simulatable channels: Extended security that is universally composable and easier to prove. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 519–550. Springer, Heidelberg, December 2018.

EHM17. Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. Can Johnny build a protocol? co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. In *European Workshop on Usable Security*, 2017.

EMP18. Patrick Eugster, Giorgia Azzurra Marson, and Bertram Poettering. A cryptographic look at multi-party channels. In Steve Chong and Stephanie Delaune, editors, *CSF 2018 Computer Security Foundations Symposium*, pages 31–45. IEEE Computer Society Press, 2018.

FGJ20. Marc Fischlin, Felix Günther, and Christian 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.

Goo18. Google. BoringSSL AES IGE implementation. https://github.com/DrKLO/Telegram/blob/d073b80063c568f31d81cc88c927b47c01a1dbf4/TMessagesProj/jni/boringssl/crypto/fipsmodule/aes/aes_ige.c, Jul 2018.

HN00. Helena Handschuh and David 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.

IR19. Yuval Ishai and Vincent Rijmen, editors. *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*. Springer, Heidelberg, May 2019.

IT21. Jana Iyengar and Martin Thomson. QUIC: A UDP-based multiplexed and secure transport. https://datatracker.ietf.org/doc/draft-ietf-quic-transport/, March 2021. Draft Version 34.

JMM19. Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Ishai and Rijmen [IR19], pages 159–188.

JO16. Jakob Jakobsen and Claudio 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.

JS18. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

Jut00. Charanjit Jutla. Attack on free-mac, sci.crypt. https://groups.google.com/forum/#!topic/sci.crypt/4bkzm_n7UGA, Sep 2000.

KKL+04. Jongsung Kim, Guil Kim, Sangjin Lee, Jongin Lim, and Jung Hwan Song. Related-key attacks on reduced rounds of SHACAL-2. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT 2004*, volume 3348 of *LNCS*, pages 175–190. Springer, Heidelberg, December 2004.

Knu00. Lars R Knudsen. Block chaining modes of operation. 2000.

Kob18. Nadim Kobeissi. *Formal Verification for Real-World Cryptographic Protocols and Implementations*. Theses, INRIA Paris ; Ecole Normale Supérieure de Paris - ENS Paris, December 2018. https://hal.inria.fr/tel-01950884.

KPB03. Tadayoshi Kohno, Adriana Palacio, and John Black. Building secure cryptographic transforms, or how to encrypt and MAC. Cryptology ePrint Archive, Report 2003/177, 2003. https://eprint.iacr.org/2003/177.

LKKD06. Jiqiang Lu, Jongsung Kim, Nathan Keller, and Orr Dunkelman. Related-key rectangle attack on 42-round SHACAL-2. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC 2006*, volume 4176 of *LNCS*, pages 85–100. Springer, Heidelberg, August / September 2006.

Lud17. Kelby Ludwig. Trudy - Transparent TCP proxy, 2017. https://github.com/praetorian-inc/trudy.

MBA+20. Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon Attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). https://raccoon-attack.com/RacoonAttack.pdf, September 2020. accessed 11 September 2020.

MH20. Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. Cryptology ePrint Archive, Report 2020/1506, 2020. https://eprint.iacr.org/2020/1506.

MP17. Giorgia Azzurra Marson and Bertram Poettering. Security notions for bidirectional channels. *IACR Trans. Symm. Cryptol.*, 2017(1):405–426, 2017.

MV21. Marino Miculan and Nicola Vitacolonna. Automated symbolic verification of Telegram's MTProto 2.0. In Sabrina De Capitani di Vimercati and Pierangela Samarati, editors, *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021*, pages 185–197. SciTePress, 2021.

NIS15. NIST. FIPS 180-4: Secure Hash Standard. 2015. http://dx.doi.org/10.6028/NIST.FIPS.180-4.

Rog04. Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 348–359. Springer, Heidelberg, February 2004.

RS06. Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Vaudenay [Vau06], pages 373–390.

RTM21. Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. The Datagram Transport Layer Security (DTLS) protocol version 1.3. https://datatracker.ietf.org/doc/draft-ietf-tls-dtls13/, February 2021. Draft Version 41.

RZ18. Phillip Rogaway and Yusi Zhang. Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2018.

Sha49. Claude E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

Shr04. Tom Shrimpton. A characterization of authenticated-encryption as a form of chosen-ciphertext security. Cryptology ePrint Archive, Report 2004/272, 2004. https://eprint.iacr.org/2004/272.

SK17. Tomáš Sušánka and Josef Kokeš. Security analysis of the Telegram IM. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, pages 1–8, 2017.

Tel20a. Telegram. MTProto transports. http://web.archive.org/web/20200527124125/https://core.telegram.org/mtproto/mtproto-transports, May 2020.

Tel20b. Telegram. 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.

Tel20c. Telegram. Schema. https://core.telegram.org/schema, Sep 2020.

Tel20d. Telegram. tdlib. https://github.com/tdlib/td, Sep 2020.

Tel20e. Telegram. TL language. https://core.telegram.org/mtproto/TL, Sep 2020.

Tel21a. Telegram. 500 million users. https://t.me/durov/147, Feb 2021.

Tel21b. 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.

Tel21c. Telegram. Mobile protocol: Detailed description. http://web.archive.org/web/20210126200309/https://core.telegram.org/mtproto/description, Jan 2021.

Tel21d. Telegram. Mobile protocol: Detailed description – server salt. http://web.archive.org/web/20210221134408/https://core.telegram.org/mtproto/description#server-salt, Feb 2021.

Tel21e. Telegram. Security guidelines for client developers. http://web.archive.org/web/20210203134436/https://core.telegram.org/mtproto/security_guidelines#mtproto-encrypted-messages, Feb 2021.

Tel21f. Telegram. Sequence numbers in secret chats. http://web.archive.org/web/20201031115541/https://core.telegram.org/api/end-to-end/seq_no, Jan 2021.

Tel21g. Telegram. tdlib – Transport.cpp. https://github.com/tdlib/td/blob/v1.7.0/td/mtproto/Transport.cpp#L272, Apr 2021.

Tel21h. Telegram. Telegram Android – Datacenter.cpp. https://github.com/DrKLO/Telegram/blob/release-7.4.0_2223/TMessagesProj/jni/tgnet/Datacenter.cpp#L1171, Feb 2021.

Tel21i. Telegram. Telegram Android – Datacenter.cpp. https://github.com/DrKLO/Telegram/blob/release-7.6.0_2264/TMessagesProj/jni/tgnet/Datacenter.cpp#L1250, Apr 2021.

Tel21j. 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.

Tel21k. Telegram. Telegram Desktop – session_private.cpp. https://github.com/telegramdesktop/tdesktop/blob/v2.6.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1338, Mar 2021.

Tel21l. Telegram. Telegram Desktop – session_private.cpp. https://github.com/telegramdesktop/tdesktop/blob/v2.7.1/Telegram/SourceFiles/mtproto/session_private.cpp#L1258, Apr 2021.

Tel21m. Telegram. Telegram iOS – MTProto.m. https://github.com/TelegramMessenger/Telegram-iOS/blob/release-7.6.2/submodules/MtProtoKit/Sources/MTProto.m#L2144, Apr 2021.

Tel21n. Telegram. Telegram MTProto – creating an authorization key. http://web.archive.org/web/20210112084225/https://core.telegram.org/mtproto/auth_key, Jan 2021.

UDB$^+$15. Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249. IEEE Computer Society Press, May 2015.

vAP22. Theo von Arx and Kenneth G. Paterson. On the cryptographic fragility of the Telegram ecosystem. Cryptology ePrint Archive, Paper 2022/595, 2022. to appear at AsiaCCS 2023.

Vau06. Serge Vaudenay, editor. *EUROCRYPT 2006*, volume 4004 of *LNCS*. Springer, Heidelberg, May / June 2006.

A Correctness-style properties of a support function

In this section we formalise two basic correctness-style properties of a support function that we call *integrity* and *order correctness*. Both properties were specified and required (but not named) in the robust channel framework of [FGJ20]. For our formal analysis of the MTProto protocol in Section 5 we use the support function SUPP defined in Fig. 32 that mandates strict in-order delivery; it happens to satisfy both of the properties formalised in this section. However, we do not mandate that every support function must satisfy these properties so as not to constrain the range of possible channel behaviours. We provide formalisations of these two properties to better clarify the relation to prior work.

The INT-security proof of our MTProto-based channel $\mathsf{MTP\text{-}CH}$ in Section 5.6 relied on the integrity property, which we formalise here, and on two other basic properties of SUPP. The two latter properties were informally introduced in Section 5.6. We do not formalise them in this work in order to avoid introducing additional complexity.

Recall that any support function supp can be interpreted as a specification regarding how a channel should behave. In particular, the correctness definition for channels from Section 3.4 matches the implementation of any channel CH against its desired functionality as specified by supp. However, there can be many different channel implementations that adhere to a fixed specification defined by supp. Therefore, in this work all definitions of support functions and all formalisations of support function properties are agnostic to (i.e. not parametrised with) any specific channel implementation CH.

Integrity of a support function. This property roughly requires that only the messages that were genuinely sent by another user on the channel are delivered. In particular, the support function should return $\bot$ whenever it is evaluated on an input tuple $(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$ such that the label label does not appear in the opposite user's transcript $\mathsf{tr}_{\overline{u}}$. The game in Fig. 53 captures this requirement by allowing an adversary $\mathcal{F}$ to choose an arbitrary input tuple for the support function supp. The advantage of $\mathcal{F}$ in breaking the integrity of supp is defined as $\mathsf{Adv}^{\mathsf{sint}}_{\mathsf{supp}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{sint}}_{\mathsf{supp},\mathcal{F}}\right]$.

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

Figure 53. Integrity of support function supp.

Order correctness of a support function. This property roughly requires that in-order delivery is enforced separately in each direction on the channel, assuming that a distinct label is assigned to each network message. In particular, when evaluated on an input tuple $(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$, we require that the support function should return m if the opposite user's transcript $\mathsf{tr}_{\overline{u}}$ contains a tuple $(\mathsf{sent}, m, \mathsf{label}, aux)$ and if all prior messages from $\overline{u}$ to u were delivered and accepted strictly in-order.

The notion of order correctness is captured by the game in Fig. 54. It provides an adversary $\mathcal{F}$ with an access to oracles Send and Recv. The oracles can be used to send and receive messages in either direction between the two users. When querying the Send oracle, the adversary is allowed to associate any plaintext m with an arbitrary label of its choice, as long as all labels are distinct. The adversary controls the network, and can call its Recv oracle to deliver an arbitrary plaintext and label pair to either user of its choice; the support function is used to determine how to resolve the delivered network message. The adversary wins if it manages to create a situation that the support function fails to recover a plaintext that was delivered strictly in-order in either direction. The game in Fig. 54 uses an auxiliary function $\mathsf{buildList}$ to build a list of messages sent or received by a particular user, and it uses $\mathcal{L}_0 \preccurlyeq \mathcal{L}_1$ to denote that a list $\mathcal{L}_0$ is a prefix of another list $\mathcal{L}_1$. The advantage of $\mathcal{F}$ in breaking the order correctness of supp is defined as $\mathsf{Adv}^{\mathsf{ord}}_{\mathsf{supp}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{ord}}_{\mathsf{supp},\mathcal{F}}\right]$.

```
Game $\mathrm{G}^{\mathsf{ord}}_{\mathsf{supp},\mathcal{F}}$
$\mathsf{win} \leftarrow \mathsf{false}$
$X \leftarrow \emptyset$
$\mathcal{F}^{\textsc{Send},\textsc{Recv}}$
Return $\mathsf{win}$

$\textsc{Send}(u, m, \mathsf{label}, aux)$
// $u \in \{\mathcal{I}, \mathcal{R}\}$
If $\mathsf{label} \in X$ then return $\perp$
$X \leftarrow X \cup \{\mathsf{label}\}$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{sent}, m, \mathsf{label}, aux)$
Return $\perp$
```

```
$\textsc{Recv}(u, m, \mathsf{label}, aux)$   // $u \in \{\mathcal{I}, \mathcal{R}\}$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, \mathsf{label}, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, \mathsf{label}, aux)$
$\mathsf{inOrder} \leftarrow \mathsf{buildList}(u, \mathsf{recv}) \preccurlyeq \mathsf{buildList}(\overline{u}, \mathsf{sent})$
If $\mathsf{inOrder} \wedge (m \neq m^*)$ then $\mathsf{win} \leftarrow \mathsf{true}$
Return $\perp$

$\mathsf{buildList}(u, \mathsf{op})$
// $u \in \{\mathcal{I}, \mathcal{R}\}$, $\mathsf{op} \in \{\mathsf{sent}, \mathsf{recv}\}$
$\mathsf{list} \leftarrow []$
For each $(\mathsf{op}', m, \mathsf{label}, aux) \in \mathsf{tr}_u$ do
    If $\mathsf{op}' = \mathsf{op}$ then
        $\mathsf{list} \leftarrow \mathsf{list} \,\|\, (m, \mathsf{label}, aux)$
Return $\mathsf{list}$
```

Figure 54. Order correctness of support function supp.

Recall that our definition of a support transcript allows entries of the form $(\mathsf{sent}, m, \perp, aux)$ and $(\mathsf{recv}, \perp, \mathsf{label}, aux)$, denoting a failure to send and receive a message respectively. Such entries cannot appear in the user transcripts of our order correctness game because we require that adversaries never pass $\perp$ as input to their oracles (cf. Section 2.1). This conveniently provides us with a weak notion of order correctness that does not prescribe the behaviour of the support function in the presence of channel errors. Our definition can be strengthened by giving the adversary a choice to create support transcript entries that contain $\perp$; the updated game could then mandate whether the in-order delivery should still be required after an error is encountered.

The order correctness property is defined implicitly in [FGJ20]; it is required to hold as a part of their channel correctness game. In our framework, the channel correctness game (cf. Section 3.4) is defined with respect to an arbitrary support function, without mandating order correctness. Thus, expressing order correctness is delegated to the support function itself: if the property is required to hold in some setting, the support function must be required to satisfy the game in Fig. 54 (or some stronger variant of it).

B Combined security of bidirectional channels

In this section we define a security notion for channels that simultaneously captures the integrity and indistinguishability definitions from Section 3. We call it *authenticated encryption*. It follows the all-in-one definitional style of [Shr04,RS06]. We will prove that integrity and indistinguishability together are equivalent to authenticated encryption.

Security definition. Consider the authenticated encryption game $\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. 55, defined for a channel CH, a support function supp and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the AE-security of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}\right] - 1$. Adversary $\mathcal{A}$ is given access to the challenge encryption oracle $\textsc{Ch}$ and to the receiving oracle $\textsc{Recv}$. The $\textsc{Ch}$ oracle is a copy of the $\textsc{Send}$ oracle from the channel integrity game $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ (Fig. 12), except it is amended for the left-or-right setting. Note that the $\textsc{Ch}$ oracle can be queried with $m_0 = m_1$ in order to recover the functionality of the $\textsc{Send}$ oracle. The $\textsc{Recv}$ oracle is likewise based on the corresponding oracle of $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$, but it is amended as follows. Instead of always returning $\perp$, the $\textsc{Recv}$ oracle can now alternatively return another error code $\lightning$ (here $\lightning \neq \perp$ as per Section 2.1). In the spirit of [Shr04,RS06], oracle $\textsc{Recv}$ returns $\perp$ whenever the challenge bit b is equal to 0. If $b = 1$ *and* adversary $\mathcal{A}$ violated the integrity of the channel (i.e. $m \neq m^*$ is true), then the $\textsc{Recv}$ oracle returns $\lightning$. Returning $\lightning$ here signals that $b = 1$, and the adversary can immediately use this to win the game.[45] Finally, if $b = 1$ and $m = m^*$, then $\textsc{Recv}$ returns $\perp$; this ensures that the adversary cannot trivially win by requesting the decryption of a challenge ciphertext. Note that adversary $\mathcal{A}$ can use the

[45] We emphasise that it is sufficient to return *any* non-$\perp$ value when $\mathcal{A}$ violates the integrity of CH. For example, one could instead return an arbitrary constant string.

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

$\textsc{Ch}(u, m_0, m_1, aux, r)$   // $u \in \{\mathcal{I}, \mathcal{R}\}$, $m_i \in \mathsf{CH.MS}$, $r \in \mathsf{CH.SendRS}$
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 $c$

$\textsc{Recv}(u, c, aux)$   // $u \in \{\mathcal{I}, \mathcal{R}\}$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
$(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
If $(m \neq m^*) \wedge (b = 1)$ then return $\lightning$
Return $\bot$
```

Figure 55. Authenticated encryption security of channel CH with respect to support function supp.

support function supp to itself compute each plaintext value m that is obtained in the $\textsc{Recv}$ oracle (separately for either possible challenge bit $b \in \{0,1\}$) for as long as $m = m^*$ has never been false yet.

AE **is equivalent to** INT + IND**.** In the following two propositions, we show that our notions of channel integrity and indistinguishability from Section 3 together are equivalent to the above notion of authenticated encryption security.

Proposition 1. *Let* CH *be a channel. Let* supp *be a support function. Let* $\mathcal{A}$ *be any 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. This proof uses games G_0–G_1 in Fig. 56. Game G_0 is functionally equivalent to the authenticated encryption game $\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$, only adding a single instruction that sets the bad flag, so by definition we have

$$\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) = 2 \cdot \Pr[\,\mathrm{G}_0\,] - 1.$$

We obtain game G_1 by removing the only instruction from game G_0 that could have returned the non-$\bot$ output; we denote this part of the code by setting $\mathsf{bad} \leftarrow \mathsf{true}$. In this proof we will first use the IND-security of CH to show that $\mathcal{A}$ cannot win if the bad flag is never set (i.e. as captured by game G_1, which does not contain the line commented with G_0). And we will then show that the bad flag cannot be set if CH has INT-security with respect to supp.

We build the adversaries $\mathcal{F}$ for $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ (Fig. 12) and $\mathcal{D}$ for $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ (Fig. 13) as shown in Fig. 57. First, consider the IND-security adversary $\mathcal{D}$. By inspection, it perfectly simulates the oracles of game G_1 for the AE-security adversary $\mathcal{A}$, so we can write

$$\Pr[\,\mathrm{G}_1\,] = \Pr\left[\,\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}\,\right] = \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) + \frac{1}{2}.$$

Second, we have

$$\Pr[\,\mathrm{G}_0\,] - \Pr[\,\mathrm{G}_1\,] \leq \Pr\left[\,\mathsf{bad}^{\mathrm{G}_1}\,\right].$$

Now consider the INT-security adversary $\mathcal{F}$. It perfectly simulates the oracles of game G_1 for the AE-security adversary $\mathcal{A}$, sampling its own challenge bit $b \in \{0,1\}$ and using it to consistently encrypt the appropriate challenge plaintext when simulating the challenge oracle $\textsc{Ch}$ of game G_1. Whenever the bad flag is set in game G_1, the $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ game likewise sets $\mathsf{win} = \mathsf{true}$, so we have

$$\Pr[\,\mathsf{bad}\,] \leq \Pr\left[\,\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}\,\right] = \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}).$$

```
Games $\mathrm{G}_0$–$\mathrm{G}_1$
$b \leftarrow\!\!\$ \{0,1\}$
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$ \mathsf{CH.Init}()$
$b' \leftarrow\!\!\$ \mathcal{A}^{\textsc{Ch},\textsc{Recv}}$
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 $c$

$\textsc{Recv}(u, c, aux)$
$m^* \leftarrow \mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, c, aux)$
$(st_u, m) \leftarrow \mathsf{CH.Recv}(st_u, c, aux)$
$\mathsf{tr}_u \leftarrow \mathsf{tr}_u \,\|\, (\mathsf{recv}, m, c, aux)$
If $(m \neq m^*) \wedge (b = 1)$ then
    $\mathsf{bad} \leftarrow \mathsf{true}$
    Return $\lightning$                                  // $\mathrm{G}_0$
Return $\bot$
```

Figure 56. Games G_0–G_1 for the proof of Proposition 1. The code added for the transition between games is highlighted in green.

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

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

$\textsc{RecvSim}(u, c, aux)$
$\textsc{Recv}(u, c, aux)$
Return $\bot$

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

$\textsc{ChSim}(u, m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$c \leftarrow \textsc{Send}(u, m_b, aux, r)$
Return $c$

$\textsc{RecvSim}(u, c, aux)$
$\textsc{Recv}(u, c, aux)$
Return $\bot$
```

Figure 57. Adversaries $\mathcal{D}$ and $\mathcal{F}$ for the proof of Proposition 1.

Combining all of the above, we can write

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) &= 2 \cdot (\Pr[\,\mathrm{G}_1\,] + (\Pr[\,\mathrm{G}_0\,] - \Pr[\,\mathrm{G}_1\,])) - 1 \\ &\leq 2 \cdot \left(\frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) + \frac{1}{2} + \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F})\right) - 1 = \\ &= \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}) + 2 \cdot \mathsf{Adv}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}).\end{aligned}$$

This concludes the proof. □

Proposition 2. *Let* CH *be a channel. Let* supp *be a support function. Let* $\mathcal{F}$ *be any adversary against the* INT*-security of* CH *with respect to* supp. *Let* $\mathcal{D}$ *be any 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*

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

Proof. We build adversaries $\mathcal{A}_{\mathrm{INT}}$ and $\mathcal{A}_{\mathrm{IND}}$ as shown in Fig. 58.

First, let us consider the AE-security adversary $\mathcal{A}_{\mathrm{INT}}$. It perfectly simulates the integrity game $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ for the INT-security adversary $\mathcal{F}$. When adversary $\mathcal{F}$ queries its oracle $\textsc{Send}$ with plaintext m as input, adversary $\mathcal{A}_{\mathrm{INT}}$ calls its $\textsc{Ch}$ oracle with challenge plaintexts $m_0 = m_1 = m$ as input, and forwards the response back to $\mathcal{F}$. When $\mathcal{F}$ queries its oracle $\textsc{Recv}$, adversary $\mathcal{A}_{\mathrm{INT}}$ first calls its own $\textsc{Recv}$ oracle on the same inputs and receives back an error code $\mathsf{err} \in \{\bot, \lightning\}$. If $\mathsf{err} = \lightning$ then $b = 1$ in the AE-security game where $\mathcal{A}_{\mathrm{INT}}$ is playing, so it calls **abort**(1) (defined in Section 2.1) to immediately halt with the return value 1, causing $\mathcal{A}_{\mathrm{INT}}$ to win the game. Alternatively, if $\mathcal{F}$ terminates without triggering this condition, then $\mathcal{A}_{\mathrm{INT}}$ returns 0. We now derive the advantage of $\mathcal{A}_{\mathrm{INT}}$. Let b be the challenge bit in game $\mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{INT}}}$. If $b = 1$ then $\mathcal{A}_{\mathrm{INT}}$ returns $b' = 1$ whenever $\mathcal{F}$ sets $\mathsf{win} = \mathsf{true}$

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

SendSim$(u, m, aux, r)$
    $c \leftarrow$ Ch$(u, m, m, aux, r)$
    Return $c$

RecvSim$(u, c, aux)$
    $\mathsf{err} \leftarrow$ Recv$(u, c, aux)$
    If $\mathsf{err} \neq \bot$ then abort(1)
    Return $\bot$
```

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

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

RecvSim$(u, c, aux)$
    $\mathsf{err} \leftarrow$ Recv$(u, c, aux)$
    If $\mathsf{err} \neq \bot$ then abort(1)
    Return $\bot$
```

Figure 58. Adversaries $\mathcal{A}_{\text{INT}}$ and $\mathcal{A}_{\text{IND}}$ for the proof of Proposition 2.

in the simulated game $\mathrm{G}^{\mathsf{int}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$. If $b = 0$ then $\mathcal{A}_{\text{INT}}$ never returns $b' = 1$. We can write

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

Next, consider the AE-security adversary $\mathcal{A}_{\text{IND}}$ as shown in Fig. 58. It perfectly simulates the indistinguishability game $\mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}}$ for the IND-security adversary $\mathcal{D}$. In particular, $\mathcal{A}_{\text{IND}}$'s oracles run the same code that $\mathcal{D}$ would expect from its own oracles, except for the additional processing of transcripts and the support function that happens in the oracles of the AE-security game. The latter does not affect the state of the channel, and can only cause $\mathcal{A}_{\text{IND}}$'s Recv oracle to occasionally return a non-$\bot$ output (i.e. $\mathsf{err} = \lightning$). Adversary $\mathcal{A}_{\text{IND}}$ checks the error code err obtained from its Recv oracle; it calls **abort**(1) to halt with the return value $b' = 1$ whenever $\mathsf{err} \neq \bot$, causing it to immediately win in the AE-security game. However, our formal statement below does not reflect the potential improvement in the advantage that $\mathcal{A}_{\text{IND}}$ might gain by doing this. Overall, if $\mathcal{D}$ returns the correct challenge bit, then so does $\mathcal{A}_{\text{IND}}$. Therefore, we can write

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\text{IND}}) &= 2 \cdot \Pr\left[\, \mathrm{G}^{\mathsf{ae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\text{IND}}} \,\right] - 1 \\ &\geq 2 \cdot \Pr\left[\, \mathrm{G}^{\mathsf{ind}}_{\mathsf{CH},\mathcal{D}} \,\right] - 1 \\ &= \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{CH}}(\mathcal{D}).\end{aligned}$$

This concludes the proof. □

C Comparison to the robust channel framework of [FGJ20]

Our definitional framework in Section 3 is designed to analyse the correctness and security of *bidirectional* channels with respect to relaxed requirements for message delivery. The robust channel framework of [FGJ20] aims to capture the same goal, but it is defined only for *unidirectional* channels. Even though our definitions are more general in this sense, the support functions in our framework can be clearly seen as extending and strengthening the concept of support predicates from [FGJ20]. Beyond that, it is not necessarily obvious that the correctness and security notions defined across the two frameworks capture the same – or even a comparable – intuition.

In order to be able to make meaningful claims about how the two frameworks compare to each other, in Appendix C.1 we define unidirectional variants of our notions for correctness and authenticated encryption security. We obtain them by simplifying our bidirectional definitions from Section 3.4 and Appendix B in a straightforward way. Then in Appendix C.2 we define the notions of correctness and combined security from [FGJ20] in the syntax of this work; we minimally modify the definitions of [FGJ20] in order to make them comparable to ours. Finally, in Appendix C.3 we state and prove formal claims showing roughly that correctness and security in our framework implies correctness and security in the framework of [FGJ20]. We also provide some informal discussion. The main takeaway is that our frameworks capture the same intuition. But our support transcripts are defined to contain more information than the support transcripts used by [FGJ20]; this allows us to capture a broader channel functionality in a somewhat simpler way.

C.1 Our definitions of unidirectional correctness and security

Consider the definitions of channel, support transcript, and support function from Section 3. These definitions specify syntax for *bidirectional* communication, meaning they allow to capture the case where the users $\mathcal{I}$ and $\mathcal{R}$ are both able to send and receive messages. Without loss of generality, in this section we will use the same definitions to formalise the *unidirectional* notions of correctness and authenticated encryption. Even though syntactically the definitions from Section 3 allow to model bidirectional communication, the notions we state in this section only guarantee correctness and security when messages are sent from $\mathcal{I}$ to $\mathcal{R}$ and never in the opposite direction.

We do not provide any high-level intuition for the unidirectional notions that are defined below. We instead refer the reader to Section 3 and Appendix B for detailed discussion on the bidirectional variants of these notions.

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

$\textsc{Send}(m, aux, r)$
$(st_{\mathcal{I}}, c) \leftarrow \mathsf{CH.Send}(st_{\mathcal{I}}, m, aux; r)$
$\mathsf{tr}_{\mathcal{I}} \leftarrow \mathsf{tr}_{\mathcal{I}} \parallel (\mathsf{sent}, m, c, aux)$
Return $c$

$\textsc{Recv}(c, aux)$
$m^* \leftarrow \mathsf{supp}(\mathcal{R}, \mathsf{tr}_{\mathcal{R}}, \mathsf{tr}_{\mathcal{I}}, c, aux)$
If $m^* = \bot$ then return $\bot$
$(st_{\mathcal{R}}, m) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{R}}, c, aux)$
$\mathsf{tr}_{\mathcal{R}} \leftarrow \mathsf{tr}_{\mathcal{R}} \parallel (\mathsf{recv}, m, c, aux)$
If $m \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
Return $\bot$
```

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

$\textsc{Ch}(m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$(st_{\mathcal{I}}, c) \leftarrow \mathsf{CH.Send}(st_{\mathcal{I}}, m_b, aux; r)$
$\mathsf{tr}_{\mathcal{I}} \leftarrow \mathsf{tr}_{\mathcal{I}} \parallel (\mathsf{sent}, m_b, c, aux)$
Return $c$

$\textsc{Recv}(c, aux)$
$m^* \leftarrow \mathsf{supp}(\mathcal{R}, \mathsf{tr}_{\mathcal{R}}, \mathsf{tr}_{\mathcal{I}}, c, aux)$
$(st_{\mathcal{R}}, m) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{R}}, c, aux)$
$\mathsf{tr}_{\mathcal{R}} \leftarrow \mathsf{tr}_{\mathcal{R}} \parallel (\mathsf{recv}, m, c, aux)$
If $(m \neq m^*) \wedge (b = 1)$ then return $\lightning$
Return $\bot$
```

Figure 59. Unidirectional correctness of channel CH and unidirectional authenticated encryption security of channel CH, both with respect to support function supp.

Unidirectional correctness. Consider the unidirectional correctness game $\mathrm{G}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}$ in Fig. 59, defined for a channel CH, a support function supp and an adversary $\mathcal{F}$. The advantage of $\mathcal{F}$ in breaking the unidirectional correctness of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}) = \Pr\left[\mathrm{G}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}}\right]$. The UCORR game closely mirrors the bidirectional correctness game from Fig. 12, except it allows only $\mathcal{I}$ to send messages and only $\mathcal{R}$ to receive them. This means that oracles Send and Recv no longer take a user variable $u \in \{\mathcal{I}, \mathcal{R}\}$ as input. Instead, $\mathcal{I}$ is hardcoded as the user that sends messages in Send (i.e. this oracle is defined to always use the channel state $st_{\mathcal{I}}$ and the support transcript $\mathsf{tr}_{\mathcal{I}}$), and $\mathcal{R}$ is similarly hardcoded as the user that receives messages in Recv.

Unidirectional authenticated encryption. Consider the unidirectional authenticated encryption game $\mathrm{G}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}$ in Fig. 59, defined for a channel CH, a support function supp and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the UAE-security of CH with respect to supp is defined as $\mathsf{Adv}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}}\right] - 1$. The UAE game closely mirrors the bidirectional authenticated encryption game from Fig. 55, except $\mathcal{I}$ is now hardcoded as the sender in oracle Ch and $\mathcal{R}$ is now hardcoded as the receiver in oracle Recv.

C.2 The robust channel framework of [FGJ20]

In this section we specify the core definitions from [FGJ20]. We formalise these definitions using our channel syntax from Section 3.[46] This syntax allows to model bidirectional communication, but in Appendix C.1 we explain that we can also use it to study unidirectional notions. The syntax used by [FGJ20] treats the auxiliary information differently: it is required to be recoverable from ciphertexts, so their receiving algorithm $\mathsf{CH.Recv}$ does not take aux as input. In particular, [FGJ20] define a helper algorithm $\mathsf{aux}(\cdot)$ that takes a ciphertext c as input and returns the auxiliary information aux that is recovered from c. Note that channels with recoverable auxiliary information can also be captured using our syntax, e.g. by considering a receiving algorithm $\mathsf{CH.Recv}(st_{\mathcal{R}}, c, aux)$ that extracts $aux' \leftarrow \mathsf{aux}(c)$ and then overwrites $aux \leftarrow aux'$. So we do not lose generality by using our syntax to state the definitions of [FGJ20].

Predicate-based support transcripts and functions. The robust channel framework [FGJ20] uses deterministic *support predicates* as the counterpart of our support functions. A support predicate $\mathsf{suppPred}$ (when adjusted to our syntax) takes the following inputs: receiver's identity $\mathcal{R}$, receiver's support transcript $\mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}$, sender's support transcript $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}$, ciphertext c, and auxiliary information aux. It returns a *support decision* d. We write $d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}, \mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}, c, aux)$. We will define the exact format of the support transcripts $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}, \mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}$ below. A *Boolean* support predicate returns $d \in \{\mathsf{true}, \mathsf{false}\}$ to indicate whether c, aux should be accepted (i.e. whether it is *supported*). An *index-recovering* support predicate returns $d \in \{\mathsf{false}\} \cup \{0, 1, \ldots\}$. Here $d = \mathsf{false}$ indicates that c, aux should be rejected, whereas $d \in \{0, 1, \ldots\}$ indicates that it should be accepted *and* that d is the index of the entry containing c, aux in the sender's transcript $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}$. In the latter case the correctness of $\mathsf{suppPred}$ guarantees that $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}[d]$ indeed contains c, aux.

The robust channel framework uses support transcripts that are defined as follows (again adjusted to use our syntax). The sender's support transcript $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}$ is a list of (sent, c, aux)-type entries, each indicating that the sender $\mathcal{I}$ sent a ciphertext c with auxiliary information aux. The receiver's support transcript $\mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}$ is a list of $(\mathsf{recv}, d, c, aux)$-type entries, each indicating that c, aux was delivered to the receiver $\mathcal{R}$, and that $\mathcal{R}$ subsequently accepted them based on obtaining a positive support decision d. We emphasise that the receiver's support transcript in [FGJ20] is defined to only contain information about supported ciphertexts, i.e. $\mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}$ should never contain an entry with $d = \mathsf{false}$.

Summary of the notions defined in [FGJ20]. The robust channel framework [FGJ20] defines correctness and multiple security notions for a channel, each of them with respect to an arbitrary support predicate $\mathsf{suppPred}$. These notions capture roughly the following intuition. Correctness requires that for any sequence of ciphertexts that was sent by the sender $\mathcal{I}$ on an authenticated channel, and for any way – *that is supported by* $\mathsf{suppPred}$ – in which these ciphertexts could have been replayed, reordered or dropped (while in transit), each of them is correctly decrypted upon arriving to the receiver $\mathcal{R}$. Integrity (INT) requires that if $\mathsf{suppPred}$ requires to reject a ciphertext, then $\mathcal{R}$ rejects it. Robustness (ROB) requires that if $\mathsf{suppPred}$ requires to accept a ciphertext, then $\mathcal{R}$ correctly decrypts it; this should happen even if $\mathcal{R}$ previously rejected other ciphertexts, meaning $\mathcal{R}$'s channel state should not get corrupted regardless of what ciphertexts $\mathcal{R}$ attempts to decrypt. Robust integrity ($\mathsf{ROB\text{-}INT}$) combines ROB and INT essentially requiring that the channel behaves exactly as specified by $\mathsf{suppPred}$ (likewise correctly decrypting the supported ciphertexts even after rejecting maliciously formed ciphertexts). Finally, the master notion $\mathsf{ROB\text{-}INT\text{-}IND\text{-}CCA}$ combines $\mathsf{ROB\text{-}INT}$ with the standard notion of $\mathsf{IND\text{-}CCA}$ security; it is recognized by [FGJ20] as the "ultimate target" to be achieved by real-world protocols. In this section we will state formal definitions for the correctness and the $\mathsf{ROB\text{-}INT\text{-}IND\text{-}CCA}$ security notions of [FGJ20] in the syntax of our work, minimally strengthening both of them. For simplicity and consistency with other definitions in this paper, we will refer to $\mathsf{ROB\text{-}INT\text{-}IND\text{-}CCA}$ as the *unidirectional predicate-based authenticated encryption.*

Unidirectional predicate-based correctness. Consider the unidirectional predicate-based correctness game $\mathrm{G}_{\mathsf{CH},\mathsf{suppPred},\mathcal{F}}^{\mathsf{predcorr}}$ in Fig. 60, defined for a channel CH, a support predicate $\mathsf{suppPred}$ and an

[46] We need both frameworks to study the same object so that in Appendix C.3 we can state and prove relations between their correctness and security notions.

```
Game $\mathrm{G}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{F}}$
$\mathsf{win} \leftarrow \mathsf{false}$
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ \mathsf{CH.Init}()$
$i \leftarrow 0$  // Number of messages sent.
$\mathcal{F}^{\textsc{Send},\textsc{Recv}}(st_{\mathcal{I}}, st_{\mathcal{R}})$
Return $\mathsf{win}$

$\textsc{Send}(m, aux, r)$
$(st_{\mathcal{I}}, c) \leftarrow \mathsf{CH.Send}(st_{\mathcal{I}}, m, aux; r)$
$\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \parallel (\mathsf{sent}, c, aux)$
$\mathsf{T}[i] \leftarrow (m, c, aux)$ ; $i \leftarrow i + 1$
Return $c$

$\textsc{Recv}(j)$
If $\neg(0 \leq j < i)$ then return $\bot$
$(m, c, aux) \leftarrow \mathsf{T}[j]$
$d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, c, aux)$
If $(d = \mathsf{false})$ $\boxed{\vee (d \neq j)}$ then return $\bot$
$\mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \parallel (\mathsf{recv}, d, c, aux)$
$(st_{\mathcal{R}}, m^c) \leftarrow \mathsf{CH.Recv}(st_{\mathcal{R}}, c, aux)$
If $m^c \neq m$ then $\mathsf{win} \leftarrow \mathsf{true}$
Return $m^c$
```

```
Game $\mathrm{G}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{A}}$
$b \leftarrow\$ \{0, 1\}$
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ \mathsf{CH.Init}()$
$st^r_{\mathcal{R}} \leftarrow st^c_{\mathcal{R}} \leftarrow st_{\mathcal{R}}$
$b' \leftarrow\$ \mathcal{A}^{\textsc{Ch},\textsc{Recv}}$
Return $b' = b$

$\textsc{Ch}(m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$(st_{\mathcal{I}}, c) \leftarrow \mathsf{CH.Send}(st_{\mathcal{I}}, m_b, aux; r)$
$\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \parallel (\mathsf{sent}, c, aux)$
Return $c$

$\textsc{Recv}(c, aux)$
$(st^r_{\mathcal{R}}, m^r) \leftarrow \mathsf{CH.Recv}(st^r_{\mathcal{R}}, c, aux)$
$m^c \leftarrow \bot$
If $b = 0$ then
  $m^r \leftarrow \bot$
Else
  $d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, c, aux)$
  If $d \neq \mathsf{false}$ then
    $(st^c_{\mathcal{R}}, m^c) \leftarrow \mathsf{CH.Recv}(st^c_{\mathcal{R}}, c, aux)$
    $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \parallel (\mathsf{recv}, d, c, aux)$
  If $m^r = m^c$ then
    $m^r \leftarrow \bot$
  Else  // Set $m^r$ to non-$\bot$ so $\mathcal{A}$ can win.
    $m^r \leftarrow \lightning$
Return $m^r$
```

Figure 60. Unidirectional predicate-based correctness of channel CH and unidirectional predicate-based authenticated encryption security of channel CH (called ROB-INT-IND-CCA in [FGJ20], both with respect to support predicate suppPred. The $\boxed{\text{boxed}}$ code is used only when suppPred is an index-recovering support predicate. Marked in grey are the main differences from the corresponding definitions in [FGJ20].

adversary $\mathcal{F}$. The boxed code is used only when suppPred is an index-recovering support predicate. The advantage of $\mathcal{F}$ in breaking the unidirectional predicate-based correctness of CH with respect to suppPred is defined as $\mathsf{Adv}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{F}) = \Pr[\mathrm{G}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{F}}]$. The game allows adversary $\mathcal{F}$ to relay arbitrary messages from user $\mathcal{I}$ to user $\mathcal{R}$, calling oracles Send and Recv to respectively send and receive them. The Recv oracle takes an integer j as input, instructing it to receive the j-th ciphertext that was produced by the oracle Send. This effectively models an authenticated channel where adversary $\mathcal{F}$ cannot forge messages but it can replay, reorder and drop them. The game maintains two ordered transcripts: $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}$ is a list that contains ciphertexts (and is passed to the support predicate), and T is a table that contains plaintext-ciphertext pairs. More precisely, the table contains entries of the form $\mathsf{T}[j] = (m, c, aux)$ to indicate that the j-th call to oracle Send encrypted (m, aux) into c. When adversary queries $\textsc{Recv}(j)$ and the support predicate suppPred determines that c, aux from $\mathsf{T}[j]$ is a supported input, then the channel should decrypt it as m. Otherwise, the adversary wins the game.

We now discuss the intuition behind the index-recovering support predicates, and explain the purpose of the boxed code in the correctness game. We will roughly show that if a channel CH can encrypt two distinct plaintexts into the same ciphertext (e.g. with respect to different sender's states), then there exists an adversary that can break the correctness of CH with respect to any Boolean support predicate suppPred. We note that the output of CH.Send depends on the continuously evolving state of the sender $\mathcal{I}$, so even some real-world channels display such behaviour (this includes QUIC and DTLS 1.3, which motivated the authors of [FGJ20] to define their framework).

More precisely, consider any hypothetical sequence of oracle queries made by adversary $\mathcal{F}$ in game $\mathrm{G}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{F}}$ that resulted in creating (intermediate) transcripts $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, \mathsf{T}[\cdot]$ that satisfy the following conditions:

(1) There exist distinct indices j_0, j_1 and distinct plaintexts m_0, m_1 such that $\mathsf{T}[j_0] = (m_0, c, aux)$ and $\mathsf{T}[j_1] = (m_1, c, aux)$ for some c, aux.

(2) $d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, c, aux)$ returns $d = \mathsf{true}$ when queried on c, aux from $\mathsf{T}[j_0], \mathsf{T}[j_1]$.[47]

At this point $\mathcal{F}$ can choose to query either $\textsc{Recv}(j_0)$ or $\textsc{Recv}(j_1)$ in order to make the game check that $\mathsf{CH.Recv}$ decrypts (c, aux) into m_0 or m_1 respectively. But $\mathsf{CH.Recv}$ would in both cases take the same pair (c, aux) as input so it would fail to correctly recover at least one of these plaintexts. This illustrates a limitation in the correctness game when it is considered with respect to a Boolean support predicate. The use of an *index-recovering* support predicate resolves this definitional issue with the help of the boxed code in the correctness game. In particular, an index-recovering support predicate returns j to instruct the correctness game that the supported input pair (c, aux) corresponds to the j-th ciphertext that was produced by $\textsc{Send}$. So at any point in time there is a unique value j for which $\textsc{Recv}(j)$ would be expected to test the correctness of $\mathsf{CH.Recv}$, because otherwise the condition checking $d \neq j$ would cause the oracle to exit early.

The original correctness game defined in [FGJ20] is called $\mathsf{Expt}^{\mathsf{correct(suppPred)}}_{\mathsf{CH},\mathcal{F}}$. Our definition of $\mathrm{G}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{F}}$ closely resembles it. The differences we introduced are as follows. The original game does not provide the initial channel states as input to its adversary, and it does not allow the adversary to choose random coins when calling its oracle $\textsc{Send}$. Our game strengthens their definition in both ways; this is necessary for our analysis in Appendix C.3. Beyond that, the original game also verifies that the support predicate $\mathsf{suppPred}$ allows to deliver messages strictly in-order. We suggest to instead formalise it as a stand-alone property of a support predicate (e.g. in the bidirectional setting we formalise this property as the *order correctness* of a support function in Appendix A).

Unidirectional predicate-based authenticated encryption. We now define the notion of $\mathsf{ROB\text{-}INT\text{-}IND\text{-}CCA}$-security from [FGJ20], referring to it as *unidirectional predicate-based authenticated encryption.* Consider the unidirectional predicate-based authenticated encryption game $\mathrm{G}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{A}}$ in Fig. 60, defined for a channel CH, a support predicate $\mathsf{suppPred}$ and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the PREDAE-security of CH with respect to $\mathsf{suppPred}$ is defined as $\mathsf{Adv}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{A}) = 2 \cdot \Pr[\mathrm{G}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{A}}] - 1$. The game allows adversary $\mathcal{A}$ to relay arbitrary messages from the sender $\mathcal{I}$ to the receiver $\mathcal{R}$ by calling its oracles $\textsc{Send}$ and $\textsc{Recv}$. The sending oracle $\textsc{Send}$ presents a left-or-right style challenge, encrypting one of the two provided plaintexts. For the receiving oracle $\textsc{Recv}$, the game creates a "real" receiver's channel state $st^r_{\mathcal{R}}$ and a "correct" receiver's channel state $st^c_{\mathcal{R}}$ that are initially equal. When adversary calls $\textsc{Recv}$, the "real" state is always updated by running $\mathsf{CH.Recv}$ on the received input, whereas the "correct" state is only updated if the oracle's input is determined to be supported according to $\mathsf{suppPred}$ (and only if the challenge bit is $b = 1$). So the "real" state $st^r_{\mathcal{R}}$ could in principle get corrupted by maliciously formed ciphertexts, causing the channel to malfunction; whereas the same cannot happen to the "correct" state $st^c_{\mathcal{R}}$. If the challenge bit is $b = 0$ then $\textsc{Recv}$ always returns $\perp$. If the challenge bit is $b = 1$ then $\textsc{Recv}$ will also return $\perp$ unless the "real" and the "correct" states produce different outcomes (including the case when both of them accept the ciphertext, but the decrypted plaintexts m^r and m^c are distinct). If the latter occurs, then $\textsc{Recv}$ returns $\lightning$ instead of $\perp$, which unambiguously signals to $\mathcal{A}$ that $b = 1$ and thus enables it to immediately win the game by halting with output value $b' = 1$.

The original game defining the $\mathsf{ROB\text{-}INT\text{-}IND\text{-}CCA}$-security of CH within the robust channel framework [FGJ20] is called $\mathsf{Expt}^{\mathsf{ROB\text{-}INT\text{-}IND\text{-}CCA(suppPred)}}_{\mathsf{CH},\mathcal{A}}$. Our definition of $\mathrm{G}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{A}}$ closely resembles it. The differences we introduced are as follows. Similarly to the correctness notion above, the original game did not let its adversary choose random coins for encrypting challenge plaintexts; our definition allows that. In the original game, if the challenge bit is $b = 1$ and the adversary queries its receiving oracle $\textsc{Recv}$ on inputs that produce $m^r \neq m^c$ then the game returns the first of the two values m^r, m^c that is non-$\perp$. We simplified this by instead making $\textsc{Recv}$ immediately return $\lightning$; these behaviours are equivalent, because both of them unambiguously indicate to $\mathcal{A}$ that $b = 1$. Finally, in the original game the $\textsc{Recv}$ oracle does not take aux as input. In general, our definition gives more power to the adversary by allowing it to independently choose both c and aux. But this change can be negated by defining both $\mathsf{CH.Recv}$ and $\mathsf{suppPred}$ to ignore the aux value they receive as input, and instead recover the auxiliary information from c.

[47] Note that this attack fails for contrived support predicates that would *never* recognize (c, aux) as being supported.

C.3 Relations between our framework and the framework of [FGJ20]

Richer support transcripts help capturing more functionality. The robust channel framework [FGJ20] explicitly defines the receiver's support transcript to contain only those ciphertexts that were determined to be supported (according to the support predicate $\mathsf{suppPred}$ that is used in the corresponding correctness or security game). If a ciphertext is rejected, then it is done silently, without modifying the receiver's support transcript. In contrast, in our framework every attempt to receive a ciphertext is documented in the receiver's transcript. This includes rejected ciphertexts, i.e. if (c, aux) is delivered to the receiver and subsequently rejected by it then the entry $(\mathsf{recv}, \bot, c, aux)$ should be added to the receiver's support transcript. It is possible to define support functions that use this information in order to determine when the channel should be permanently closed, e.g. upon rejecting a specific number of delivered ciphertexts. Such functionality cannot be captured by the correctness and security games of [FGJ20] because they keep no record of rejected ciphertexts. We emphasise that this difference arises only due to the support transcripts of [FGJ20] containing less information, and not due to our framework using support functions instead of support predicates (or having stronger correctness or security notions).

Note that our framework also requires the *sender's* transcript to contain information about all failed encryption events (i.e. when $\mathsf{CH.Send}$ returns $\bot$), each described by a transcript entry of the type $(\mathsf{sent}, m, \bot, aux)$. In comparison, the channel's sending algorithm in [FGJ20] is likewise allowed to return $\bot$, but their correctness and security definitions do not explicitly consider the possibility of encryption failures (and so it is not obvious whether their definitions are well-defined in this respect). However, it is also not clear whether such entries could serve any purpose in either framework, so the difference here is moot.

Support functions aid in reducing the definitional complexity. We now argue that our correctness definition captures the same intuition as that of [FGJ20], except that the use of a support function in our definition allows us to remove some of the complexity that appeared in the definition of [FGJ20]. But the syntax of a function on its own would not be sufficient to achieve that. In this regard, it is again crucial that our definitions use support transcripts that are richer than those of [FGJ20], i.e. that our support transcripts in addition contain plaintext messages.

Consider the unidirectional (function-based) correctness definition in Fig. 59 and the unidirectional predicate-based correctness definition in Fig. 60. The predicate-based definition essentially maintains two different transcripts: $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}$ and $\mathsf{T}[\cdot]$ (the latter defined as a table with sequential integer indices). Here $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}$ is the sender's support transcript that is used as an input to the support predicate $\mathsf{suppPred}$ in order to determine whether (c, aux) is supported. If $\mathsf{suppPred}$ determines this to be true, then the game implicitly uses $\mathsf{T}[\cdot]$ to map (c, aux) to the corresponding plaintext m. If such a mapping cannot be unambiguously inferred based just on the pair (c, aux) (as we discussed in Appendix C.2), then $\mathsf{suppPred}$ is required to be index-recovering so that it can point the correctness game to the appropriate index j inside table T. The process of mapping (c, aux) to j and then to m – that is done jointly by $\mathsf{suppPred}$ and the correctness game itself – is arguably complicated and counter-intuitive.[48] Having observed this, now consider our function-based correctness definition. One can think of it as having essentially delegated the task of mapping (c, aux) to m entirely to its support function. In order to be able to do this, the support function now needs to take some information about the encrypted plaintexts as input. This is accomplished by having our support transcript itself contain the plaintexts. Finally, note that even though our function-based correctness definition could potentially allow its adversary to forge ciphertexts, this entirely depends on what support function is considered. Outside of some exotic applications, one would expect any support function to return $m^* = \bot$ whenever c was not previously produced by the sender (in the bidirectional setting we formalise this property as the *integrity* of a support function in Appendix A).

Is our framework at least as expressive as the framework of [FGJ20]? We now make partial progress towards showing the following: any channel that is correct and secure in the robust channel

[48] For example, the first two instructions of oracle RECV in the predicate-based correctness game (enforcing $0 \leq j < i$ and then fetching $(m, c, aux) \leftarrow \mathsf{T}[j]$) only serve the goal of implicitly mapping (c, aux) directly to the corresponding plaintext m. Later on, this mapping could still be determined invalid if $d \neq j$. This condition is actually *not* inherently necessary for the purpose of disallowing the adversary from forging ciphertexts because the support predicate could itself have enforced that.

framework of [FGJ20] is also correct and secure in our framework. When trying to prove such claim, ideally for any support predicate $\mathsf{suppPred}$ one would like to build a support function supp such that the following is true: any channel CH that is correct and secure with respect to $\mathsf{suppPred}$ in the [FGJ20] framework is also correct and secure with respect to supp in our framework. The result we provide below is weaker in two ways: (1) we only consider an *index-recovering* support predicate $\mathsf{suppPred}$ as the starting point[49], and (2) the support function supp that we build is simultaneously based on $\mathsf{suppPred}$ *and* CH, and not all combinations of $\mathsf{suppPred}, \mathsf{CH}$ are permitted. It is not necessarily obvious why it is valuable to prove any implications between the two frameworks. In our opinion, formalising and proving at least *some* relations between these frameworks serves as a sanity check for both of them.

Building a support function. As outlined above, our goal is to build a support function supp from an arbitrary index-recovering support predicate $\mathsf{suppPred}$ such that unidirectional predicate-based correctness and security of a channel CH with respect to $\mathsf{suppPred}$ would imply unidirectional (function-based) correctness and security of CH with respect to supp. This inevitably requires to build supp from $\mathsf{suppPred}$ in a black-box way, and so the support function supp also needs to be able to convert its own supports transcripts into the format that is used by $\mathsf{suppPred}$. For this purpose we will assume there exists a *transcript conversion algorithm* $\mathsf{convertTr}$ that takes the support transcripts $\mathsf{tr}_{\mathcal{I}}, \mathsf{tr}_{\mathcal{R}}$ of our format (i.e. as defined in Definition 2) and converts them into support transcripts $\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}, \mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}$ that adhere to the format used by the robust channel framework (i.e. as defined in Appendix C.2). This essentially requires to convert $(\mathsf{sent}, m, c, aux)$-type entries into (sent, c, aux)-type entries, and $(\mathsf{recv}, m, c, aux)$-type entries into $(\mathsf{recv}, d, c, aux)$-type entries. Here $d \in \{0, 1, \ldots\}$ is the support decision that would have been returned by $\mathsf{suppPred}$; it should point at the entry of the sender's transcript that documents the event of sending c, aux. We emphasise that the transcript conversion algorithm $\mathsf{convertTr}$ takes $\mathsf{tr}_{\mathcal{I}}, \mathsf{tr}_{\mathcal{R}}$ as input. So for any $(\mathsf{recv}, m, c, aux) \in \mathsf{tr}_{\mathcal{R}}$ it can trivially determine d by searching $\mathsf{tr}_{\mathcal{I}}$ for the unique entry containing $(\mathsf{sent}, m, c, \cdot)$[50] – *unless* $\mathsf{tr}_{\mathcal{I}}$ could contain more than one such entry. This problem does not arise if CH is guaranteed to produce unique plaintext-ciphertext pairs; below we will define a correctness condition describing what exactly is required from $\mathsf{convertTr}$. If a "correct" transcript conversion algorithm $\mathsf{convertTr}$ exists, then we define a support function supp based on $\mathsf{suppPred}$ and $\mathsf{convertTr}$ as defined in Fig. 61. It calls $\mathsf{convertTr}$ to convert the transcripts and uses them to evaluate $\mathsf{suppPred}$ on c, aux. If $d \neq \mathsf{false}$ then supp simply gets its output m^* from the d-th entry of the sender's support transcript $\mathsf{tr}_{\mathcal{I}}$.

```
$\mathsf{supp}(\mathcal{R}, \mathsf{tr}_{\mathcal{R}}, \mathsf{tr}_{\mathcal{I}}, c, aux)$
// Convert both support transcripts from our format to the format of [FGJ20].
$(\mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}, \mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}) \leftarrow \mathsf{convertTr}(\mathsf{tr}_{\mathcal{I}}, \mathsf{tr}_{\mathcal{R}})$
// Evaluate the support predicate, and return $m^* = \bot$ if $(c, aux)$ is not supported.
$d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}_{\mathcal{R}}^{\mathsf{pred}}, \mathsf{tr}_{\mathcal{I}}^{\mathsf{pred}}, c, aux)$
If $d = \mathsf{false}$ then return $\bot$
// Index $d$ points at the entry in the sender's transcript $\mathsf{tr}_{\mathcal{I}}$ that contains $(c, aux)$.
$(\mathsf{sent}, m^*, c^*, aux^*) \leftarrow \mathsf{tr}_{\mathcal{I}}[d]$
Return $m^*$
```

Figure 61. Construction of $\mathsf{supp} = \mathsf{SUPP\text{-}FUNC\text{-}FROM\text{-}PRED}[\mathsf{suppPred}, \mathsf{convertTr}]$ building support function supp from index-recovering support predicate $\mathsf{suppPred}$ and transcript conversion algorithm $\mathsf{convertTr}$.

Correctness of a transcript conversion algorithm. Consider the correctness game $\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH}, \mathsf{suppPred}, \mathsf{convertTr}, \mathcal{D}}$ in Fig. 62, defined for a channel CH, an index-recovering support predicate $\mathsf{suppPred}$, a transcript

[49] The analysis of QUIC and DTLS 1.3 in [FGJ20] required to use index-recovering support predicates. As we discuss in Appendix C.2, only such predicates can be used to analyse channels with non-unique ciphertexts. So we focus on them here.

[50] We allow the sender and the receiver to use distinct aux values with respect to the same m, c, e.g. when aux contains a timestamp for the event of creating or delivering a ciphertext. So in general we cannot require aux values to be equal. This detail is only relevant because we extended the syntax of [FGJ20] to surface aux.

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

$\textsc{Send}(m, aux, r)$
$(st_{\mathcal{I}}, c) \leftarrow \mathsf{CH.Send}(st_{\mathcal{I}}, m, aux; r)$
If $b = 1$ then
    $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \parallel (\mathsf{sent}, c, aux)$
Else
    $\mathsf{tr}_{\mathcal{I}} \leftarrow \mathsf{tr}_{\mathcal{I}} \parallel (\mathsf{sent}, m, c, aux)$
Return $c$

$\textsc{Recv}(c, aux)$
If $b = 1$ then
    $d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, c, aux)$
    If $d = \mathsf{false}$ then return $\mathsf{false}$
    $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \parallel (\mathsf{recv}, d, c, aux)$
Else
    $(\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}) \leftarrow \mathsf{convertTr}(\mathsf{tr}_{\mathcal{I}}, \mathsf{tr}_{\mathcal{R}})$
    $d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, c, aux)$
    If $d = \mathsf{false}$ then $m^* \leftarrow \bot$
    Else $(\mathsf{sent}, m^*, c^*, aux^*) \leftarrow \mathsf{tr}_{\mathcal{I}}[d]$
    $\mathsf{tr}_{\mathcal{R}} \leftarrow \mathsf{tr}_{\mathcal{R}} \parallel (\mathsf{recv}, m^*, c, aux)$
Return $d$
```

Figure 62. Correctness of transcript conversion algorithm $\mathsf{convertTr}$ with respect to channel CH and index-recovering support predicate $\mathsf{suppPred}$.

conversion algorithm $\mathsf{convertTr}$ and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the correctness of $\mathsf{convertTr}$ with respect to CH, $\mathsf{suppPred}$ is defined as

$$\mathsf{Adv}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{suppPred},\mathsf{convertTr}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{suppPred},\mathsf{convertTr},\mathcal{D}}] - 1.$$

We say that $\mathsf{convertTr}$ is correct with respect to CH, $\mathsf{suppPred}$ if

$$\mathsf{Adv}^{\mathsf{corr}}_{\mathsf{CH},\mathsf{suppPred},\mathsf{convertTr}}(\mathcal{D}) = 0$$

for all adversaries $\mathcal{D}$. The correctness game captures the intuition that $\mathsf{convertTr}$ must be able to perfectly reconstruct support transcripts for the robust channel framework based on the support transcripts used in our framework. Here the benchmark that we use for “perfect reconstruction” is that $\mathsf{suppPred}$ would never return different support decisions based on whether it is run on real support transcripts of [FGJ20] or those that are converted from the support transcripts used in our framework. In particular, the game uses real transcripts when its challenge bit is $b = 1$, and it uses converted transcripts when $b = 0$. The sender's support transcript $\mathsf{tr}_{\mathcal{I}}$ is populated with ciphertexts that can be created by $\mathsf{CH.Send}$, whereas the receiver's support transcript $\mathsf{tr}_{\mathcal{R}}$ is formed based on the support decisions that can be returned by $\mathsf{suppPred}$. Here the use of $\mathsf{CH.Send}$ means that a simpler algorithm $\mathsf{convertTr}$ would be able to satisfy the correctness requirement when CH has, for example, unique ciphertexts. Note that if $b = 1$ then the receiver's transcript contains only the entries for supported c, aux; but if $b = 0$ then the receiver's transcript can also contain the entries with c, aux that were rejected. This reflects the observation that our support transcripts contain more information than the transcripts in the robust channel framework; this information might help when trying to convert them. Finally, observe that the $b = 0$ branch in the $\textsc{Recv}$ oracle could be replaced with the following two instructions (for $\mathsf{supp} = \mathsf{SUPP\text{-}FUNC\text{-}FROM\text{-}PRED}[\mathsf{suppPred}, \mathsf{convertTr}]$ as defined in Fig. 61):

$$m^* \leftarrow \mathsf{supp}(\mathcal{R}, \mathsf{tr}_{\mathcal{R}}, \mathsf{tr}_{\mathcal{I}}, c, aux),$$
$$\mathsf{tr}_{\mathcal{R}} \leftarrow \mathsf{tr}_{\mathcal{R}} \parallel (\mathsf{recv}, m^*, c, aux).$$

So the correctness of $\mathsf{convertTr}$ can be interpreted as requiring that $\mathsf{suppPred}$ and supp can be used interchangeably (each using support transcripts of a different format).

Proposition 3. *Let* CH *be a channel. Let* $\mathsf{suppPred}$ *be an index-recovering support predicate. Let* $\mathsf{convertTr}$ *be a transcript conversion algorithm that is correct with respect to* $\mathsf{CH}, \mathsf{suppPred}$. *Let* $\mathsf{supp} = \mathsf{SUPP\text{-}FUNC\text{-}FROM\text{-}PRED}[\mathsf{suppPred}, \mathsf{convertTr}]$ *be the support function as defined in Fig. 61. Let* $\mathcal{F}_{\text{PREDCORR}}$ *be any adversary against the unidirectional predicate-based correctness of* CH *with respect to* $\mathsf{suppPred}$. *Then we can build an adversary* $\mathcal{F}_{\text{UCORR}}$ *against the unidirectional (function-based) correctness of* CH *with respect to* supp *such that*

$$\mathsf{Adv}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{F}_{\text{PREDCORR}}) \leq \mathsf{Adv}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{F}_{\text{UCORR}}).$$

Proof. This proof uses games G_0–G_2 in Fig. 63. Game G_0 is equivalent to game $\mathrm{G}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{F}_{\text{PREDCORR}}}$ so we have

$$\mathsf{Adv}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{F}_{\text{PREDCORR}}) = \Pr[\mathrm{G}_0].$$

```
Games $\mathrm{G}_0$–$\mathrm{G}_2$
$\mathsf{win} \leftarrow \mathsf{false}$
$(st_\mathcal{I}, st_\mathcal{R}) \leftarrow\!\!\$\ \mathsf{CH.Init}()$
$i \leftarrow 0$
$\mathcal{F}_{\text{PREDCORR}}^{\text{SEND,RECV}}(st_\mathcal{I}, st_\mathcal{R})$
Return $\mathsf{win}$

$\text{SEND}(m, aux, r)$
$(st_\mathcal{I}, c) \leftarrow \mathsf{CH.Send}(st_\mathcal{I}, m, aux; r)$
$\mathsf{tr}_\mathcal{I}^{\mathsf{pred}} \leftarrow \mathsf{tr}_\mathcal{I}^{\mathsf{pred}} \parallel (\mathsf{sent}, c, aux)$          // $\mathrm{G}_0$
$\mathsf{tr}_\mathcal{I} \leftarrow \mathsf{tr}_\mathcal{I} \parallel (\mathsf{sent}, m, c, aux)$          // $\mathrm{G}_1$–$\mathrm{G}_2$
$\mathsf{T}[i] \leftarrow (m, c, aux)$ ; $i \leftarrow i + 1$
Return $c$

$\text{RECV}(j)$
If $\neg(0 \le j < i)$ then return $\perp$
$(m, c, aux) \leftarrow \mathsf{T}[j]$
$(\mathsf{tr}_\mathcal{I}^{\mathsf{pred}}, \mathsf{tr}_\mathcal{R}^{\mathsf{pred}}) \leftarrow \mathsf{convertTr}(\mathsf{tr}_\mathcal{I}, \mathsf{tr}_\mathcal{R})$ // $\mathrm{G}_1$–$\mathrm{G}_2$
$d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}_\mathcal{R}^{\mathsf{pred}}, \mathsf{tr}_\mathcal{I}^{\mathsf{pred}}, c, aux)$
If $(d = \mathsf{false}) \vee (d \ne j)$ then return $\perp$
$\mathsf{tr}_\mathcal{R}^{\mathsf{pred}} \leftarrow \mathsf{tr}_\mathcal{R}^{\mathsf{pred}} \parallel (\mathsf{recv}, d, c, aux)$          // $\mathrm{G}_0$
$(\mathsf{sent}, m^*, c^*, aux^*) \leftarrow \mathsf{tr}_\mathcal{I}[d]$          // $\mathrm{G}_1$–$\mathrm{G}_2$
$\mathsf{tr}_\mathcal{R} \leftarrow \mathsf{tr}_\mathcal{R} \parallel (\mathsf{recv}, m^*, c, aux)$          // $\mathrm{G}_1$–$\mathrm{G}_2$
$(st_\mathcal{R}, m^c) \leftarrow \mathsf{CH.Recv}(st_\mathcal{R}, c, aux)$
If $m^c \ne m$ then $\mathsf{win} \leftarrow \mathsf{true}$          // $\mathrm{G}_0$-$\mathrm{G}_1$
If $m^c \ne m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$          // $\mathrm{G}_2$
Return $m^c$
```

Figure 63. Games G_0–G_2 for the proof of Proposition 3. The code added for the transitions between games is highlighted in green.

Game G_1 is obtained from game G_0 by replacing the support transcripts of the robust channel framework [FGJ20] with the support transcripts of our framework. The transcript conversion algorithm $\mathsf{convertTr}$ is used to convert the transcripts, and its assumed correctness with respect to CH, $\mathsf{suppPred}$ guarantees that the two games are equivalent. It follows that

$$\Pr[\mathrm{G}_0] = \Pr[\mathrm{G}_1].$$

We do not provide an adversary against the correctness of $\mathsf{convertTr}$, but one could in principle do that as follows. Such an adversary gets both channel states as input, whereas the oracles SEND and RECV are fully deterministic in both G_0 and G_1. So the adversary against the correctness of $\mathsf{convertTr}$ could simultaneously simulate both games for $\mathcal{F}_{\text{PREDCORR}}$ and only use its own oracles to accordingly update the support transcripts in the game it is playing in. If it ever detects that the support decisions differ between G_0 and G_1, it can immediately query its own RECV oracle and use the knowledge of the d values in the simulated games in order to trivially determine the challenge bit.

Game G_2 is obtained from game G_1 by changing the win condition from $m^c \ne m$ to $m^c \ne m^*$. Here the plaintext m it taken from $\mathsf{T}[j]$, whereas m^* is taken from $\mathsf{tr}_\mathcal{I}[d]$. But $j = d$ is guaranteed to be true, so $m = m^*$ and the two win conditions are equivalent. We have

$$\Pr[\mathrm{G}_1] = \Pr[\mathrm{G}_2].$$

In Fig. 64 we build adversary $\mathcal{F}_{\text{UCORR}}$ against the unidirectional (function-based) correctness of CH with respect to supp. This adversary perfectly simulates game G_2 for adversary $\mathcal{F}_{\text{PREDCORR}}$. The simulation is trivial, because both oracles in game G_2 are deterministic and because $\mathcal{F}_{\text{UCORR}}$ gets the initial channel states as input. Adversary $\mathcal{F}_{\text{UCORR}}$ also calls its own oracles SEND and RECV during the simulation of the corresponding oracles of G_2, but the responses from these calls are ignored (i.e. these calls do not affect the simulation of G_2). We claim that at the end of each oracle call that is simulated by $\mathcal{F}_{\text{UCORR}}$, the values of $\mathsf{tr}_\mathcal{I}, \mathsf{tr}_\mathcal{R}, st_\mathcal{I}, st_\mathcal{R}$ in game $\mathrm{G}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\text{UCORR}}}$ are the same as the corresponding values in the simulated game G_2. More precisely, this is true at least until the win flag is set in game G_2; we will justify this below. Based on this claim, it follows that setting the win flag in the simulated game G_2 also results in setting it in game $\mathrm{G}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\text{UCORR}}}$. We have

$$\Pr[\mathrm{G}_2] \le \Pr[\mathrm{G}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\text{UCORR}}}].$$

The above claim is trivially true with respect to $\mathsf{tr}_\mathcal{I}, st_\mathcal{I}$. For the analysis of $\mathsf{tr}_\mathcal{R}, st_\mathcal{R}$ consider what happens when $\mathcal{F}_{\text{UCORR}}$ calls its own oracle RECV from the simulated oracle RECVSIM. Observe

```
Adversary $\mathcal{F}_{\text{UCORR}}^{\text{Send,Recv}}(st_\mathcal{I}, st_\mathcal{R})$
    $i \leftarrow 0$
    $\mathcal{F}_{\text{PREDCORR}}^{\text{SendSim,RecvSim}}(st_\mathcal{I}, st_\mathcal{R})$

SendSim$(m, aux, r)$
    Send$(m, aux, r)$
    $(st_\mathcal{I}, c) \leftarrow \mathsf{CH.Send}(st_\mathcal{I}, m, aux; r)$
    $\mathsf{tr}_\mathcal{I} \leftarrow \mathsf{tr}_\mathcal{I} \,\|\, (\mathsf{sent}, m, c, aux)$
    $\mathsf{T}[i] \leftarrow (m, c, aux)\,;\ i \leftarrow i + 1$
    Return $c$

RecvSim$(j)$
    If $\neg(0 \leq j < i)$ then return $\bot$
    $(m, c, aux) \leftarrow \mathsf{T}[j]$
    $(\mathsf{tr}_\mathcal{I}^{\mathsf{pred}}, \mathsf{tr}_\mathcal{R}^{\mathsf{pred}}) \leftarrow \mathsf{convertTr}(\mathsf{tr}_\mathcal{I}, \mathsf{tr}_\mathcal{R})$
    $d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}_\mathcal{R}^{\mathsf{pred}}, \mathsf{tr}_\mathcal{I}^{\mathsf{pred}}, c, aux)$
    If $(d = \mathsf{false}) \vee (d \neq j)$ then return $\bot$
    Recv$(c, aux)$
    $(\mathsf{sent}, m^*, c^*, aux^*) \leftarrow \mathsf{tr}_\mathcal{I}[d]$
    $\mathsf{tr}_\mathcal{R} \leftarrow \mathsf{tr}_\mathcal{R} \,\|\, (\mathsf{recv}, m^*, c, aux)$
    $(st_\mathcal{R}, m^c) \leftarrow \mathsf{CH.Recv}(st_\mathcal{R}, c, aux)$
    If $m^c \neq m^*$ then $\mathsf{win} \leftarrow \mathsf{true}$
    Return $m^c$
```

Figure 64. Adversary $\mathcal{F}_{\text{UCORR}}$ for the proof of Proposition 3. Highlighted are the locations in the pseudocode where adversary $\mathcal{F}_{\text{UCORR}}$ calls its own oracles.

that the value of m^* derived in Recv is equal to the corresponding value in RecvSim. This is true because RecvSim can be thought of as expanding and evaluating the code of the construction $\mathsf{supp} = \mathsf{SUPP\text{-}FUNC\text{-}FROM\text{-}PRED}[\mathsf{suppPred}, \mathsf{convertTr}]$ from Fig. 61. However, note that RecvSim adds m^* to $\mathsf{tr}_\mathcal{R}$ whereas Recv instead adds the real output of $\mathsf{CH.Recv}(st_\mathcal{R}, \cdot)$ to $\mathsf{tr}_\mathcal{R}$. It follows that the receiver's transcript and state will remain consistent across $\mathrm{G}^{\mathsf{ucorr}}_{\mathsf{CH},\mathsf{supp},\mathcal{F}_{\text{UCORR}}}$ and G_2 for as long as win is not set.

The inequality stated in the proposition follows from the above steps. This concludes the proof. □

Proposition 4. *Let* CH *be a channel. Let* suppPred *be an index-recovering support predicate. Let* convertTr *be a transcript conversion algorithm that is correct with respect to* CH, suppPred. *Let* $\mathsf{supp} = \mathsf{SUPP\text{-}FUNC\text{-}FROM\text{-}PRED}[\mathsf{suppPred}, \mathsf{convertTr}]$ *be the support function as defined in Fig. 61. Let* $\mathcal{A}_{\text{PREDAE}}$ *be any adversary against the* PREDAE*-security of* CH *with respect to* suppPred. *Then we can build an adversary* $\mathcal{F}_{\text{PREDCORR}}$ *against the unidirectional predicate-based correctness of* CH *with respect to* suppPred, *and an adversary* $\mathcal{A}_{\text{UAE}}$ *against the* UAE*-security of* CH *with respect to* supp *such that*

$$\mathsf{Adv}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{A}_{\text{PREDAE}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{F}_{\text{PREDCORR}}) + \mathsf{Adv}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\text{UAE}}).$$

Proof. This proof uses games G_0–G_2 in Fig. 65. Game G_0 is designed to be equivalent to game $\mathrm{G}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{A}_{\text{PREDAE}}}$. It rewrites the code of oracle Recv as follows. Consider the original definition of oracle Recv in game $\mathrm{G}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{A}_{\text{PREDAE}}}$ from Fig. 60. First, observe that the original oracle always returns $\bot$ or $\lightning$, and the latter is returned iff $(m^r \neq m^c) \wedge (b = 1)$. The last two lines of the rewritten oracle reflect this observation. Next, observe in the original oracle that the block of code computing suppPred and checking $d \neq \mathsf{false}$ could in principle be evaluated regardless of the challenge bit's value. So in the rewritten oracle we move these instructions up, to the very beginning of the oracle's code. Game G_0 also contains code marked in green that builds a table $\mathsf{T}[\cdot]$ in the same way as in the unidirectional (predicate-based) correctness game, but this code does not affect the functionality of G_0. We can conclude that the games are equivalent and hence

$$\mathsf{Adv}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{A}_{\text{PREDAE}}) = 2 \cdot \Pr[\mathrm{G}_0] - 1.$$

Game G_1 is obtained by adding a single instruction to oracle Recv in game G_0 that sets $m^c \leftarrow m$ whenever $m^c \neq m$. This basically ensures that m^c is always equal to the plaintext value from $\mathsf{T}[d]$. We argue that adversary $\mathcal{A}$ is unlikely to cause $m^c \neq m$, and hence it would not be able to distinguish between G_0 and G_1. In particular, observe that the "correct" instance of the channel (represented by state $st^c_\mathcal{R}$) is evaluated only on supported ciphertexts, so the correctness of CH with respect to suppPred would require its output m^c to be equal to the plaintext from $\mathsf{T}[d]$. Formally, games G_0 and G_1 are identical until bad is set. We have

$$\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] \leq \Pr[\mathsf{bad}^{\mathrm{G}_0}].$$

```
Games $\mathrm{G}_0$–$\mathrm{G}_2$
$b \leftarrow\!\!\$ \{0,1\}$
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$ \mathsf{CH.Init}()$
$st^r_{\mathcal{R}} \leftarrow st^c_{\mathcal{R}} \leftarrow st_{\mathcal{R}}$
$i \leftarrow 0$
$b' \leftarrow\!\!\$ \mathcal{A}^{\textsc{Ch},\textsc{Recv}}_{\textsc{predae}}$
Return $b' = b$

$\textsc{Ch}(m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$(st_{\mathcal{I}}, c) \leftarrow \mathsf{CH.Send}(st_{\mathcal{I}}, m_b, aux; r)$
$\mathsf{T}[i] \leftarrow (m_b, c, aux)$ ;  $i \leftarrow i + 1$
$\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \parallel (\mathsf{sent}, c, aux)$   // $\mathrm{G}_0$–$\mathrm{G}_1$
$\mathsf{tr}_{\mathcal{I}} \leftarrow \mathsf{tr}_{\mathcal{I}} \parallel (\mathsf{sent}, m_b, c, aux)$   // $\mathrm{G}_2$
Return $c$
```

```
$\textsc{Recv}(c, aux)$
$(\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}) \leftarrow \mathsf{convertTr}(\mathsf{tr}_{\mathcal{I}}, \mathsf{tr}_{\mathcal{R}})$ // $\mathrm{G}_2$
$d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, c, aux)$
If $d \neq \mathsf{false}$ then
    $(st^c_{\mathcal{R}}, m^c) \leftarrow \mathsf{CH.Recv}(st^c_{\mathcal{R}}, c, aux)$
    $(m, c, aux) \leftarrow \mathsf{T}[d]$
    If $m^c \neq m$ then
        $\mathsf{bad} \leftarrow \mathsf{true}$
        $m^c \leftarrow m$                                  // $\mathrm{G}_1$–$\mathrm{G}_2$
    $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \parallel (\mathsf{recv}, d, c, aux)$     // $\mathrm{G}_0$–$\mathrm{G}_1$
    $(\mathsf{sent}, m^*, c^*, aux^*) \leftarrow \mathsf{tr}_{\mathcal{I}}[d]$          // $\mathrm{G}_2$
Else
    $m^c \leftarrow \bot$
    $m^* \leftarrow \bot$                                   // $\mathrm{G}_2$
$\mathsf{tr}_{\mathcal{R}} \leftarrow \mathsf{tr}_{\mathcal{R}} \parallel (\mathsf{recv}, m^*, c, aux)$          // $\mathrm{G}_2$
$(st^r_{\mathcal{R}}, m^r) \leftarrow \mathsf{CH.Recv}(st^r_{\mathcal{R}}, c, aux)$
If $(m^r \neq m^c) \wedge (b = 1)$ then return $\lightning$
Return $\bot$
```

Figure 65. Games G_0–G_2 for the proof of Proposition 4. The code added for the transitions between games is highlighted in green.

We bound the probability of $\Pr[\mathsf{bad}^{\mathrm{G}_0}]$ by building adversary $\mathcal{F}_{\textsc{predcorr}}$ in Fig. 66 against the unidirectional predicate-based correctness of CH with respect to $\mathsf{suppPred}$ such that

$$\Pr[\mathsf{bad}^{\mathrm{G}_0}] \leq \mathsf{Adv}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{F}_{\textsc{predcorr}}).$$

This adversary perfectly simulates game G_0 for $\mathcal{A}_{\textsc{predae}}$ and wins in game $\mathrm{G}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred},\mathcal{F}_{\textsc{predcorr}}}$ whenever $\mathcal{A}_{\textsc{predae}}$ sets the bad flag in game G_0.

```
Adversary $\mathcal{F}^{\textsc{Send},\textsc{Recv}}_{\textsc{predcorr}}(st_{\mathcal{I}}, st^r_{\mathcal{R}})$
$b \leftarrow\!\!\$ \{0,1\}$
$b' \leftarrow\!\!\$ \mathcal{A}^{\textsc{ChSim},\textsc{RecvSim}}_{\textsc{predae}}$

$\textsc{ChSim}(m_0, m_1, aux, r)$
If $|m_0| \neq |m_1|$ then return $\bot$
$c \leftarrow \textsc{Send}(m_b, aux, r)$
$\mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}} \parallel (\mathsf{sent}, c, aux)$
Return $c$

$\textsc{RecvSim}(c, aux)$
$d \leftarrow \mathsf{suppPred}(\mathcal{R}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}, \mathsf{tr}^{\mathsf{pred}}_{\mathcal{I}}, c, aux)$
If $d \neq \mathsf{false}$ then
    $m^c \leftarrow \textsc{Recv}(d)$
    $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \leftarrow \mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}} \parallel (\mathsf{recv}, d, c, aux)$
Else $m^c \leftarrow \bot$
$(st^r_{\mathcal{R}}, m^r) \leftarrow \mathsf{CH.Recv}(st^r_{\mathcal{R}}, c, aux)$
If $(m^r \neq m^c) \wedge (b = 1)$ then return $\lightning$
Return $\bot$
```

```
Adversary $\mathcal{A}^{\textsc{Ch},\textsc{Recv}}_{\textsc{uae}}$
$b' \leftarrow\!\!\$ \mathcal{A}^{\textsc{ChSim},\textsc{RecvSim}}_{\textsc{predae}}$
Return $b'$

$\textsc{ChSim}(m_0, m_1, aux, r)$
$c \leftarrow \textsc{Ch}(m_0, m_1, aux, r)$
Return $c$

$\textsc{RecvSim}(c, aux)$
$\mathsf{err} \leftarrow \textsc{Recv}(c, aux)$
If $\mathsf{err} = \lightning$ then **abort**$(1)$
Return $\mathsf{err}$
```

Figure 66. Adversaries $\mathcal{F}_{\textsc{predcorr}}$ and $\mathcal{A}_{\textsc{uae}}$ for the proof of Proposition 4. The highlighted instructions of $\mathcal{F}_{\textsc{predcorr}}$ mark the changes in the code of the simulated game G_0.

Game G_2 is obtained from game G_1 by replacing the support transcripts of the robust channel framework [FGJ20] with the support transcripts of our framework. The transcript conversion algorithm $\mathsf{convertTr}$ is used to convert the transcripts, and its assumed correctness with respect to CH, $\mathsf{suppPred}$ guarantees that the two games are equivalent. It follows that

$$\Pr[\mathrm{G}_1] = \Pr[\mathrm{G}_2].$$

We do not provide an adversary against the correctness of $\mathsf{convertTr}$; its construction is straightforward.

Finally, we argue that game G_2 is equivalent to game $\mathrm{G}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{PREDAE}}}$ with only one minor difference, namely until $(m^r \neq m^c) \wedge (b = 1)$ becomes true in G_2. First, observe that in game G_2 the table entry $\mathsf{T}[d]$ and the transcript entry $\mathsf{tr}_{\mathcal{I}}[d]$ contain the same plaintext, i.e. $m^c = m^*$. Next, the $\textsc{Recv}$ oracle in G_2 can be thought of having been obtained by expanding the code of $\mathsf{supp} = \mathsf{SUPP\text{-}FUNC\text{-}FROM\text{-}PRED}[\mathsf{suppPred}, \mathsf{convertTr}]$ from Fig. 61 inside the $\textsc{Recv}$ oracle of game $\mathrm{G}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{PREDAE}}}$. The only distinction is that G_2 populates the receiver's transcript $\mathsf{tr}^{\mathsf{pred}}_{\mathcal{R}}$ with entries containing m^* whereas $\mathrm{G}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{PREDAE}}}$ instead uses the plaintexts returned by algorithm $\mathsf{CH.Recv}$. This does not affect the input-output distribution of either oracle when the challenge bit is $b = 0$. In contrast, if $b = 1$ then the input-output distribution of $\textsc{Recv}$ in games G_2 and $\mathrm{G}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{PREDAE}}}$ is only identical until $(m^r \neq m^c) \wedge (b = 1)$ becomes true for the first time. We define adversary $\mathcal{A}_{\mathrm{UAE}}$ in Fig. 66 against the UAE-security of CH with respect to supp that simulates game G_2 for adversary $\mathcal{A}_{\mathrm{PREDAE}}$ by calling its own corresponding oracles and returning the received responses back to $\mathcal{A}_{\mathrm{PREDAE}}$ without any additional processing, except it uses **abort**(1) to halt with output 1 as soon as it detects that $\mathcal{A}_{\mathrm{PREDAE}}$ triggered the aforementioned condition in G_2. We have

$$\Pr[\mathrm{G}_2] \leq \Pr\left[\, \mathrm{G}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp},\mathcal{A}_{\mathrm{UAE}}} \,\right].$$

Combining all of the above steps, we can write

$$\begin{aligned}\mathsf{Adv}^{\mathsf{predae}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{A}_{\mathrm{PREDAE}}) &= 2 \cdot \left(\sum_{i=0}^{1} (\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}]) + \Pr[\mathrm{G}_2] \right) - 1 \\ &\leq 2 \cdot \mathsf{Adv}^{\mathsf{predcorr}}_{\mathsf{CH},\mathsf{suppPred}}(\mathcal{F}_{\mathrm{PREDCORR}}) + \mathsf{Adv}^{\mathsf{uae}}_{\mathsf{CH},\mathsf{supp}}(\mathcal{A}_{\mathrm{UAE}}).\end{aligned}$$

This concludes the proof. □

D Message encoding scheme of MTProto

Fig. 67 defines an approximation of the current ME construction in MTProto, where header fields have encodings of fixed size as in Section 4.1. Salt generation is modelled as an abstract call within $\mathsf{ME.Init}$. Table S contains 64-bit $\mathsf{server\_salt}$ values, each associated to some time period; algorithm $\mathsf{GenerateSalts}$ generates this table; algorithms $\mathsf{GetSalt}$ and $\mathsf{ValidSalts}$ are used to choose and validate salt values depending on the current timestamp. M is a fixed-size set that stores $(\mathsf{msg\_id}, \mathsf{msg\_seq\_no})$ for each of recently received messages; when M reaches its maximum size, the entries with the smallest $\mathsf{msg\_id}$ are removed first. $\mathsf{M.IDs}$ is the set of $\mathsf{msg_id}$s 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. We assume all strings are byte-aligned.

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. 67 shows an approximation. For instance, the $\mathsf{GenPadding}$ 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 Proofs for the underlying MTProto primitives

In this appendix we provide the reductions referred to in Sections 5.1 and 5.3. The code added for the transition between games is highlighted in green. In the adversaries, the highlighted instructions mark the changes in the code of the simulated games.

E.1 OTWIND of MTP-HASH

Proposition 5 shows that $\mathsf{MTP\text{-}HASH}$ is a one-time weak indistinguishable function (Fig. 25) if $\mathsf{SHACAL\text{-}1}$ is a one-time pseudorandom function (Fig. 2). At a high level, our proof uses the fact that

```
$\underline{\mathsf{ME.Init}()}$
$N_{\mathsf{sent}} \leftarrow 0$
$\mathsf{session\_id} \leftarrow 0$
$\mathsf{last\_sent\_msg\_id} \leftarrow 0$
$\mathsf{S} \leftarrow\!\!\$\ \mathsf{GenerateSalts}()$
$\mathsf{M} \leftarrow \emptyset$
For each $u \in \{\mathcal{I}, \mathcal{R}\}$ do
   $st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}, \mathsf{session\_id}, \mathsf{last\_sent\_msg\_id}, \mathsf{S}, \mathsf{M})$
Return $(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$

$\underline{\mathsf{ME.Encode}(st_{\mathsf{ME},u}, m, aux)}$
$(N_{\mathsf{sent}}, \mathsf{session\_id}, \mathsf{last\_sent\_msg\_id}, \mathsf{S}, \mathsf{M}) \leftarrow st_{\mathsf{ME},u}$
If $u = \mathcal{I} \land N_{\mathsf{sent}} = 0$ then
   $\mathsf{session\_id} \leftarrow\!\!\$\ \{0,1\}^{64}$
$\mathsf{server\_salt} \leftarrow \mathsf{GetSalt}(\mathsf{S}, aux)$
$\mathsf{msg\_id} \leftarrow \mathsf{GetMsgId}(u, aux, \mathsf{last\_sent\_msg\_id})$
$\mathsf{msg\_seq\_no} \leftarrow \langle 2 \cdot N_{\mathsf{sent}} + 1 \rangle_{32}$
$\mathsf{msg\_length} \leftarrow \langle |m|/8 \rangle_{32}$
$\mathsf{padding} \leftarrow\!\!\$\ \mathsf{GenPadding}(|m|)$
$p_0 \leftarrow \mathsf{server\_salt} \,\|\, \mathsf{session\_id}$
$p_1 \leftarrow \mathsf{msg\_id} \,\|\, \mathsf{msg\_seq\_no} \,\|\, \mathsf{msg\_length}$
$p_2 \leftarrow m \,\|\, \mathsf{padding}$
$p \leftarrow p_0 \,\|\, p_1 \,\|\, p_2$
$N_{\mathsf{sent}} \leftarrow N_{\mathsf{sent}} + 1$
$\mathsf{last\_sent\_msg\_id} \leftarrow \mathsf{msg\_id}$
$st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}, \mathsf{session\_id}, \mathsf{last\_sent\_msg\_id}, \mathsf{S}, \mathsf{M})$
Return $(st_{\mathsf{ME},u}, p)$

$\underline{\mathsf{GetMsgId}(u, aux, \mathsf{last\_sent\_msg\_id})}$
$\mathsf{msg\_id} \leftarrow aux \ll 32$
If $\mathsf{msg\_id} \leq \mathsf{last\_sent\_msg\_id}$ then
   $\mathsf{msg\_id} \leftarrow \mathsf{last\_sent\_msg\_id} + 1$
$i_{\mathcal{I}} \leftarrow 0$
$i_{\mathcal{R}} \leftarrow 1$
$t \leftarrow (i_u - \mathsf{msg\_id}) \bmod 4$
Return $\langle \mathsf{msg\_id} + t \rangle_{64}$

$\underline{\mathsf{GenPadding}(\ell)}$   $/\!/\ \ell \in \bigcup_{i=1}^{2^{24}} \{0,1\}^{8 \cdot i}$
$\ell' \leftarrow 128 - \ell \bmod 128$
$bn \leftarrow\!\!\$\ \{2, 3, \cdots, 63\}$
$\mathsf{padding} \leftarrow\!\!\$\ \{0,1\}^{\ell' + bn * 128}$
Return $\mathsf{padding}$
```

(a) ME.Init and ME.Encode.

Figure 67. Construction of MTProto's message encoding scheme ME, where *aux* is a 32-bit timestamp.

```
$\underline{\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)}$
$(N_{\mathsf{sent}}, \mathsf{session\_id}, \mathsf{last\_sent\_msg\_id}, \mathsf{S}, \mathsf{M}) \leftarrow st_{\mathsf{ME},u}$
$\mathsf{server\_salt} \leftarrow p[0:64]$
$\mathsf{session\_id}' \leftarrow p[64:128]$
$\mathsf{msg\_id} \leftarrow p[128:192]$
$\mathsf{msg\_seq\_no} \leftarrow p[192:224]$
$\mathsf{msg\_length} \leftarrow p[224:256]$
$\ell \leftarrow |p| - 256$
If $(u = \mathcal{R}) \wedge (\mathsf{server\_salt} \notin \mathsf{ValidSalts}(\mathsf{S}, aux))$ then
   Return $(st_{\mathsf{ME},u}, \perp)$
If $(u = \mathcal{R}) \wedge (N_{\mathsf{recv}} = 0)$ then
   $\mathsf{session\_id} \leftarrow \mathsf{session\_id}'$
Else if $\mathsf{session\_id}' \neq \mathsf{session\_id}$ then
   Return $(st_{\mathsf{ME},u}, \perp)$
If $\neg(aux - t_p \leq (\mathsf{msg\_id} \gg 32) \leq aux + t_f) \vee$
   $(\mathsf{msg\_id} \in \mathsf{M.IDs}) \vee (\mathsf{msg\_id} < \min(\mathsf{M.IDs}))$ then
      Return $(st_{\mathsf{ME},u}, \perp)$
If $(u = \mathcal{R}) \wedge (\exists (i, s) \in \mathsf{M}:$
   $((\mathsf{msg\_seq\_no} \leq s) \wedge (\mathsf{msg\_id} > i)) \vee$
   $((\mathsf{msg\_seq\_no} \geq s) \wedge (\mathsf{msg\_id} < i)))$ then
      Return $(st_{\mathsf{ME},u}, \perp)$
If $((u = \mathcal{I}) \wedge (\mathsf{msg\_id} \bmod 4 \neq 1)) \vee$
   $((u = \mathcal{R}) \wedge (\mathsf{msg\_id} \bmod 4 \neq 0))$ then
      Return $(st_{\mathsf{ME},u}, \perp)$
$\mathsf{padding\_length} \leftarrow \ell/8 - \mathsf{msg\_length}$
If $\neg(0 < \mathsf{msg\_length} \leq \ell/8) \vee$
   $\neg(12 \leq \mathsf{padding\_length} \leq 1024)$ then
      Return $(st_{\mathsf{ME},u}, \perp)$
$m \leftarrow p[256 : 256 + \mathsf{msg\_length} \cdot 8]$
$\mathsf{M} \leftarrow \mathsf{M.add}(\mathsf{msg\_id}, \mathsf{msg\_seq\_no})$
$st_{\mathsf{ME},u} \leftarrow (N_{\mathsf{sent}}, \mathsf{session\_id}, \mathsf{last\_sent\_msg\_id}, \mathsf{S}, \mathsf{M})$
Return $(st_{\mathsf{ME},u}, m)$
```

(b) ME.Decode.

Figure 67. Construction of MTProto's message encoding scheme ME, where *aux* is a 32-bit timestamp.

the construction of MTP-HASH evaluates SHACAL-1 using uniformly random independent keys, and hence produces random-looking outputs if SHACAL-1 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 *data processing inequality.*

Proposition 5. *Let* $\mathcal{D}_{\mathrm{OTWIND}}$ *be an adversary against the* OTWIND*-security of the function family* MTP-HASH *from Definition 7. Then we can build an adversary* $\mathcal{D}_{\mathrm{OTPRF}}$ *against the* OTPRF*-security of the block cipher* SHACAL-1 *such that*

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

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. For a message of length 2048, we denote the added block of padding by x_p. Define P as the public function $P(H_i) \coloneqq h_{160}(H_i, x_p)$, i.e. the last iteration of SHA-1 over the padding block.

Consider games G_0–G_1 in Fig. 68. Game G_0 expands the code of algorithm MTP-HASH.Ev in game $\mathrm{G}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH},\mathcal{D}_{\mathrm{OTWIND}}}$. The evaluation of function family MTP-HASH (on 2048-bit long inputs) can be expanded into five calls to the compression function h_{160} of SHA-1. The third and fourth calls to the compression function h_{160} would take as input two blocks that are formed from the function key of MTP-HASH, i.e. $hk[32 : 1056]$. Game G_0 rewrites these calls to use two invocations of SHACAL-1.Ev accordingly, using uniformly random and independent keys $hk[32 : 544]$ and $hk[544 : 1056]$. Game G_0 is functionally equivalent to game $\mathrm{G}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH},\mathcal{D}_{\mathrm{OTWIND}}}$, so $\Pr[\mathrm{G}_0] = \Pr[\mathrm{G}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH},\mathcal{D}_{\mathrm{OTWIND}}}]$. We then construct game G_1 in which the outputs of the aforementioned SHACAL-1.Ev calls are replaced with random values. In this game, the adversary $\mathcal{D}_{\mathrm{OTWIND}}$ is given $\mathsf{auth\_key\_id} = P(H_3 \hat{+} r_1)[96 : 160]$ for a uniformly random value r_1 that does not depend on the challenge bit b, so the probability of $\mathcal{D}_{\mathrm{OTWIND}}$ winning in this game is $\Pr[\mathrm{G}_1] = \frac{1}{2}$.

```
Games $\mathrm{G}_0$–$\mathrm{G}_1$
$b \leftarrow\!\$ \{0,1\}$ ; $hk \leftarrow\!\$ \{0,1\}^{\mathsf{MTP\text{-}HASH.kl}}$
$x_0 \leftarrow\!\$ \mathsf{MTP\text{-}HASH.In}$ ; $x_1 \leftarrow\!\$ \mathsf{MTP\text{-}HASH.In}$
$H_0 \leftarrow \mathsf{IV}_{160}$
$H_1 \leftarrow h_{160}(H_0, x_b[0:512])$
$H_2 \leftarrow h_{160}(H_1, x_b[512:672] \,\|\, hk[0:32] \,\|\, x_b[672:992])$
$r_0 \leftarrow \mathsf{SHACAL\text{-}1.Ev}(hk[32:544], H_2)$                    // $\mathrm{G}_0$
$r_0 \leftarrow\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}1.ol}}$                              // $\mathrm{G}_1$
$H_3 \leftarrow H_2 \hat{+} r_0$
$r_1 \leftarrow \mathsf{SHACAL\text{-}1.Ev}(hk[544:1056], H_3)$                  // $\mathrm{G}_0$
$r_1 \leftarrow\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}1.ol}}$                              // $\mathrm{G}_1$
$H_4 \leftarrow H_3 \hat{+} r_1$
$\mathsf{auth\_key\_id} \leftarrow P(H_4)[96:160]$
$b' \leftarrow\!\$ \mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, \mathsf{auth\_key\_id})$
Return $b' = b$
```

Figure 68. Games G_0–G_1 for the proof of Proposition 5. The code added by expanding the algorithm MTP-HASH.Ev in game $\mathrm{G}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH},\mathcal{D}_{\mathrm{OTWIND}}}$ is highlighted in grey; here MTP-HASH.Ev is expressed using the underlying calls to the compression function h_{160} and the block cipher SHACAL-1 of SHA-1.

We construct an adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the OTPRF-security of SHACAL-1 as shown in Fig. 69 such that $\Pr[\,\mathrm{G}_0\,] - \Pr[\,\mathrm{G}_1\,] = \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathrm{OTPRF}})$. Let d be the challenge bit in game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1},\mathcal{D}_{\mathrm{OTPRF}}}$ and d' be the output of the adversary in that game. If $d = 1$ then queries to RoR made by $\mathcal{D}_{\mathrm{OTPRF}}$ return the output of evaluating function SHACAL-1 with random keys. If $d = 0$ then each call to RoR returns a uniformly random value from $\{0,1\}^{\mathsf{SHACAL\text{-}1.ol}}$.

We can write:

$$
\begin{aligned}
&\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathrm{OTPRF}}) \\
&\quad = \Pr\left[\, d' = 1 \,|\, d = 1 \,\right] - \Pr\left[\, d' = 1 \,|\, d = 0 \,\right] \\
&\quad = \Pr\left[\, \mathrm{G}_0 \,\right] - \Pr\left[\, \mathrm{G}_1 \,\right] \\
&\quad = \frac{1}{2} \cdot \left(\mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\mathrm{OTWIND}}) + 1 \right) - \frac{1}{2} \\
&\quad = \frac{1}{2} \cdot \mathsf{Adv}^{\mathsf{otwind}}_{\mathsf{MTP\text{-}HASH}}(\mathcal{D}_{\mathrm{OTWIND}}).
\end{aligned}
$$

The inequality follows. □

E.2 RKPRF of MTP-KDF

We now reduce the related-key PRF security of MTP-KDF to the leakage-resilient, related-key PRF security of SHACAL-2. Recall that MTP-KDF is defined in Definition 9 to return concatenated outputs of two SHA-256 calls, when evaluated on inputs $\mathsf{msg\_key} \,\|\, kk_0$ and $kk_0 \,\|\, \mathsf{msg\_key}$ respectively. The key observation here is that these two strings are both only 416 bits long, so the resulting SHA-padded payloads $sk_0 = \mathsf{SHA\text{-}pad}(\mathsf{msg\_key} \,\|\, kk_0)$ and $sk_1 = \mathsf{SHA\text{-}pad}(kk_0 \,\|\, \mathsf{msg\_key})$ each consist of a single 512-bit block. So for SHA-256 compression function h_{256} and initial state IV_{256} we need to show that $h_{256}(\mathsf{IV}_{256}, sk_0) \,\|\, h_{256}(\mathsf{IV}_{256}, sk_1)$ is indistinguishable from a uniformly random string. When this is expressed through the underlying block cipher SHACAL-2, it is sufficient that $\mathsf{SHACAL\text{-}2.Ev}(sk_0, \mathsf{IV}_{256})$ and $\mathsf{SHACAL\text{-}2.Ev}(sk_1, \mathsf{IV}_{256})$ both look independent and uniformly random (even while an adversary can choose the values of msg_key that are used to build sk_0, sk_1). This requirement is exactly satisfied if SHACAL-2 is assumed to be a related-key PRF for appropriate related-key-deriving functions, which in Section 5.2 was formalised as the notion of LRKPRF-security with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$. We capture this claim in Proposition 6.

Proposition 6. *Let $\mathcal{D}_{\mathrm{RKPRF}}$ be an adversary against the* RKPRF*-security of the function family* MTP-KDF *from Definition 9 with respect to the related-key-deriving function ϕ_{KDF} from Fig. 24. Let $\phi_{\mathsf{SHACAL\text{-}2}}$ be the related-key-deriving function as defined in Fig. 29. Then we can build an adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ against the* LRKPRF*-security of the block cipher* SHACAL-2 *with respect to related-key-deriving functions $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$ (abbrev. with ϕ) such that*

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

Proof. Consider game $\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF}, \phi_{\mathsf{KDF}}, \mathcal{D}_{\mathrm{RKPRF}}}$ (Fig. 26) defining the RKPRF-security experiment in which the adversary $\mathcal{D}_{\mathrm{RKPRF}}$ plays against the function family MTP-KDF with respect to the related-key-deriving function ϕ_{KDF}. We first rewrite the game in a functionally equivalent way as G_0 in Fig. 70

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathrm{OTPRF}}$
$b \leftarrow\$ \{0,1\}$ ; $hk' \leftarrow\$ \{0,1\}^{32}$
$x_0 \leftarrow\$ \mathsf{MTP\text{-}HASH.In}$ ; $x_1 \leftarrow\$ \mathsf{MTP\text{-}HASH.In}$
$H_0 \leftarrow \mathsf{IV}_{160}$
$H_1 \leftarrow h_{160}(H_0, x_b[0:512])$
$H_2 \leftarrow h_{160}(H_1, x_b[512:672] \,\|\, hk' \,\|\, x_b[672:992])$
$r_0 \leftarrow \mathrm{RoR}(H_2)$
$H_3 \leftarrow H_2 \hat{+} r_0$
$r_1 \leftarrow \mathrm{RoR}(H_3)$
$H_4 \leftarrow H_3 \hat{+} r_1$
$\mathsf{auth\_key\_id} \leftarrow P(H_4)[96:160]$
$b' \leftarrow\$ \mathcal{D}_{\mathrm{OTWIND}}(x_0, x_1, \mathsf{auth\_key\_id})$
If $b' = b$ then return 1 else return 0
```

Figure 69. Adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the OTPRF-security of SHACAL-1 for the proof of Proposition 5. Depending on the challenge bit in game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{SHACAL\text{-}1}, \mathcal{D}_{\mathrm{OTPRF}}}$, adversary $\mathcal{D}_{\mathrm{OTPRF}}$ simulates game G_0 or G_1 for adversary $\mathcal{D}_{\mathrm{OTWIND}}$.

using the definition of algorithm MTP-KDF.Ev, expanded to SHA-256 and then expressed through the underlying block cipher SHACAL-2, which is called twice on related keys, each built by appending SHA padding to msg_key $\|$ kk_0 or kk_1 $\|$ msg_key. We have $\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) = 2 \cdot \Pr[\mathrm{G}_0] - 1$. Then G_1 rewrites the derivation of sk_0, sk_1 in G_0 in terms of the related-key-deriving function $\phi_{\mathsf{SHACAL\text{-}2}}$ (Fig. 29). Game G_1 is functionally equivalent to game G_0, so $\Pr[\mathrm{G}_1] = \Pr[\mathrm{G}_0]$. Finally, game G_2 replaces both SHACAL-2 outputs with uniformly random values that are independent of the challenge bit. In this game $\mathcal{D}_{\mathrm{RKPRF}}$ can have no advantage better than simply guessing the challenge bit, so $\Pr[\mathrm{G}_2] = \frac{1}{2}$.

```
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 $b' = b$

$\mathrm{RoR}(u, \mathsf{msg\_key})$   // $u \in \{\mathcal{I}, \mathcal{R}\}$, $|\mathsf{msg\_key}| = 128$
$(kk_0, kk_1) \leftarrow kk_u$                                          // $\mathrm{G}_0$
$sk_0 \leftarrow \mathsf{SHA\text{-}pad}(\mathsf{msg\_key} \| kk_0)$              // $\mathrm{G}_0$
$sk_1 \leftarrow \mathsf{SHA\text{-}pad}(kk_1 \| \mathsf{msg\_key})$              // $\mathrm{G}_0$
$(sk_0, sk_1) \leftarrow \phi_{\mathsf{SHACAL\text{-}2}}(kk_u, \mathsf{msg\_key})$        // $\mathrm{G}_1$–$\mathrm{G}_2$
$r_0 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(sk_0, \mathsf{IV}_{256})$                // $\mathrm{G}_0$–$\mathrm{G}_1$
If $\mathsf{R}[u, 0, \mathsf{msg\_key}] = \bot$ then                        // $\mathrm{G}_2$
    $\mathsf{R}[u, 0, \mathsf{msg\_key}] \leftarrow_\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$        // $\mathrm{G}_2$
$r_0 \leftarrow \mathsf{R}[u, 0, \mathsf{msg\_key}]$                            // $\mathrm{G}_2$
$r_1 \leftarrow \mathsf{SHACAL\text{-}2.Ev}(sk_1, \mathsf{IV}_{256})$                // $\mathrm{G}_0$–$\mathrm{G}_1$
If $\mathsf{R}[u, 1, \mathsf{msg\_key}] = \bot$ then                        // $\mathrm{G}_2$
    $\mathsf{R}[u, 1, \mathsf{msg\_key}] \leftarrow_\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$        // $\mathrm{G}_2$
$r_1 \leftarrow \mathsf{R}[u, 1, \mathsf{msg\_key}]$                            // $\mathrm{G}_2$
$k_1^{(0)} \leftarrow \mathsf{IV}_{256} \hat{+} r_0$ ; $k_1^{(1)} \leftarrow \mathsf{IV}_{256} \hat{+} r_1$
$k_1 \leftarrow k_1^{(0)} \| k_1^{(1)}$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \bot$ then
    $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow_\$ \{0,1\}^{\mathsf{MTP\text{-}KDF.ol}}$
$k_0 \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; Return $k_b$
```

Figure 70. Games G_0–G_2 for the proof of Proposition 6. The code added by expanding the algorithm MTP-KDF.Ev in game $\mathrm{G}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}},\mathcal{D}_{\mathrm{RKPRF}}}$ is highlighted in grey; here MTP-KDF.Ev is expressed using the underlying calls to the block cipher SHACAL-2 of SHA-256.

Here we construct an adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ against the LRKPRF-security of SHACAL-2 with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$ as shown in Fig. 71 such 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. If $d = 1$ then calls to oracle RoR made by $\mathcal{D}_{\mathrm{LRKPRF}}$ are SHACAL-2 invocations with related and partially-chosen keys; we have $\Pr[\,d' = 1 \,|\, d = 1\,] = \Pr[\mathrm{G}_1]$. If $d = 0$ then each call to oracle RoR draws a uniformly random value r_i and so $k_1 = (\mathsf{IV}_{256} \hat{+} r_0) \| (\mathsf{IV}_{256} \hat{+} r_1)$ is a uniformly random string; we have $\Pr[\,d' = 1 \,|\, d = 0\,] = \Pr[\mathrm{G}_2]$.

```
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

$\mathrm{RoRSim}(u, \mathsf{msg\_key})$
$r_0 \leftarrow \mathrm{RoR}(u, 0, \mathsf{msg\_key})$
$r_1 \leftarrow \mathrm{RoR}(u, 1, \mathsf{msg\_key})$
$k_1^{(0)} \leftarrow \mathsf{IV}_{256} \hat{+} r_0$ ; $k_1^{(1)} \leftarrow \mathsf{IV}_{256} \hat{+} r_1$
$k_1 \leftarrow k_1^{(0)} \| k_1^{(1)}$
If $\mathsf{T}[u, \mathsf{msg\_key}] = \bot$ then
    $\mathsf{T}[u, \mathsf{msg\_key}] \leftarrow_\$ \{0,1\}^{\mathsf{MTP\text{-}KDF.ol}}$
$k_0 \leftarrow \mathsf{T}[u, \mathsf{msg\_key}]$ ; Return $k_b$
```

Figure 71. Adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ against the LRKPRF-security of SHACAL-2 with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$ (abbrev. with ϕ) for the proof of Proposition 6. Depending on the challenge bit in game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi,\mathcal{D}_{\mathrm{LRKPRF}}}$, adversary $\mathcal{D}_{\mathrm{LRKPRF}}$ simulates game G_1 or G_2 for adversary $\mathcal{D}_{\mathrm{RKPRF}}$.

We can use the above to write:

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi}(\mathcal{D}_{\mathrm{LRKPRF}}) \\
&\qquad = \Pr[\, d' = 1 \,|\, d = 1 \,] - \Pr[\, d' = 1 \,|\, d = 0 \,] \\
&\qquad = \Pr[\, \mathrm{G}_1 \,] - \Pr[\, \mathrm{G}_2 \,] \\
&\qquad = \Pr[\, \mathrm{G}_0 \,] - \Pr[\, \mathrm{G}_2 \,] \\
&\qquad = \frac{1}{2}\left(\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}) + 1\right) - \frac{1}{2} \\
&\qquad = \frac{1}{2}\mathsf{Adv}^{\mathsf{rkprf}}_{\mathsf{MTP\text{-}KDF},\phi_{\mathsf{KDF}}}(\mathcal{D}_{\mathrm{RKPRF}}).
\end{aligned}$$

The inequality follows. □

E.3 UPRKPRF of MTP-MAC

The UPRKPRF-security (Fig. 28) of MTP-MAC roughly requires that the function $\mathsf{MTP\text{-}MAC.Ev}(mk_u, \cdot)$ is a related-key PRF (simultaneously for both $u \in \{\mathcal{I}, \mathcal{R}\}$) if the adversary is only allowed to evaluate this function on inputs that have distinct 256-bit prefixes.

Recall that $\mathsf{MTP\text{-}MAC.Ev}(mk_u, p)$ is defined to return a truncated output of $\mathsf{SHA\text{-}256}(mk_u \,\|\, p)$ where the key mk_u is 256-bit long for any $u \in \{\mathcal{I}, \mathcal{R}\}$, and the payload p is guaranteed (according to the definition of MTP-ME) to be longer than 256 bits. Furthermore, the construction of MTP-ME ensures that the 256-bit prefix of p will be unique (as long as the number of total produced payloads is upper-bounded by some large constant) because this prefix of p encodes various counters. This enables us to consider the output of the first SHA-256 compression function h_{256} while evaluating $\mathsf{SHA\text{-}256}(mk_u \,\|\, p)$; we can assume that this output is uniformly random by assuming the HRKPRF-security of SHACAL-2 (as defined in Section 5.2). Now it remains to show that every next h_{256} call that is made to evaluate $\mathsf{SHA\text{-}256}(mk_u \,\|\, p)$ will return a uniformly random output as well, which is true when h_{256} is assumed to be a PRF.

We start with the latter step, showing that the Merkle-Damgård construction is a secure PRF as long as the underlying compression function is a secure PRF. This claim about the Merkle-Damgård transform is analogous to the basic cascade PRF security proved in [BCK96], except that we only prove *one-time* security and hence we do not require prefix-free inputs.

Lemma 1. *Consider the compression function h_{256} of* SHA-256. *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 of the function family* $\mathsf{MD}[h_{256}]$ *from Section 2.2 that makes queries of length at most T blocks (i.e. at most $T \cdot 512$ bits). Then we can build an adversary $\mathcal{D}_{\mathsf{compr}}$ against the* OTPRF*-security of* H *such that*

$$\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}]}(\mathcal{D}_{\mathsf{md}}) \le T \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{H}}(\mathcal{D}_{\mathsf{compr}}).$$

Proof. This proof uses games G_0–G_T in Fig. 72. The oracle RoR on input x returns $\mathsf{MD}[h_{256}].\mathsf{Ev}(H_0, x)$ for a uniformly random key $H_0 \in \mathsf{H.Keys}$ in game G_0, and it returns a uniformly random value from $\{0,1\}^{\mathsf{H.ol}}$ in game G_T.

```
Game $\mathrm{G}_j$   // $0 \le j \le T$
$b' \leftarrow\!\$\ \mathcal{D}_{\mathsf{md}}^{\mathrm{RoR}}$
Return $b' = 1$

$\mathrm{RoR}(x_1 \,\|\, \ldots \,\|\, x_t)$   // $|x_i| = 512$, $t \le T$
$H_0 \leftarrow\!\$\ \mathsf{H.Keys}$
For $i = 1, \ldots, t$ do
    If $i \le j$ then $H_i \leftarrow\!\$\ \{0,1\}^{\mathsf{H.ol}}$
    If $i > j$ then $H_i \leftarrow \mathsf{H.Ev}(H_{i-1}, x_i)$
Return $H_t$
```

Figure 72. Games G_0–G_T for the proof of Lemma 1.

Let b be the challenge bit in game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}],\mathcal{D}_{\mathsf{md}}}$, and let b' be the output of $\mathcal{D}_{\mathsf{md}}$ in that game. Then we have

$$\begin{aligned}\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}]}(\mathcal{D}_{\mathsf{md}}) &= \Pr\left[\, b' = 1 \,|\, b = 1 \,\right] - \Pr\left[\, b' = 1 \,|\, b = 0 \,\right] \\ &= \Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_T] \\ &= \sum_{q=1}^{T} \left(\Pr[\mathrm{G}_{q-1}] - \Pr[\mathrm{G}_q]\right). \end{aligned} \tag{3}$$

Consider adversary $\mathcal{D}_{\mathsf{compr}}$ in Fig. 73. Let h be the value sampled in the first step of $\mathcal{D}_{\mathsf{compr}}$. For any choice of $h \in \{1, \ldots, T\}$, adversary $\mathcal{D}_{\mathsf{compr}}$ (playing in game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{H}}$) perfectly simulates the view of $\mathcal{D}_{\mathsf{md}}$ in either G_{h-1} or G_h, depending on whether $\mathcal{D}_{\mathsf{compr}}$'s oracle RoR is returning real evaluations of $\mathsf{H.Ev}$ or uniformly random values from $\{0,1\}^{\mathsf{H.ol}}$.

```
Adversary $\mathcal{D}^{\text{RoR}}_{\mathsf{compr}}$
$j \leftarrow\!\!\$ \{1, \ldots, T\}$
$b' \leftarrow\!\!\$ \mathcal{D}^{\text{RoRSim}}_{\mathsf{md}}$
Return $b'$

RoRSim($x_1 \,\|\, \ldots \,\|\, x_t$)    // $|x_i| = 512,\ t \le T$
$H_0 \leftarrow\!\!\$ \mathsf{H.Keys}$
For $i = 1, \ldots, t$ do
    If $i < j$ then $H_i \leftarrow\!\!\$ \{0,1\}^{\mathsf{H.ol}}$
    If $i = j$ then $H_i \leftarrow\!\!\$ \text{RoR}(x_i)$
    If $i > j$ then $H_i \leftarrow \mathsf{H.Ev}(H_{i-1}, x_i)$
Return $H_t$
```

Figure 73. Adversary $\mathcal{D}_{\mathsf{compr}}$ against the OTPRF-security of H for the proof of Lemma 1.

Let d be the challenge bit in game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{H},\mathcal{D}_{\mathsf{compr}}}$, and let d' be the output of $\mathcal{D}_{\mathsf{compr}}$ in that game. It follows that for any $j \in \{1, \ldots, T\}$ we have

$$\begin{aligned}\Pr[\mathrm{G}_{j-1}] &= \Pr\left[\, d' = 1 \,|\, d = 1, h = j \,\right], \\ \Pr[\mathrm{G}_{j}] &= \Pr\left[\, d' = 1 \,|\, d = 0, h = j \,\right]. \end{aligned}$$

Let us express $\Pr\left[\, d' = 1 \,|\, d = 1 \,\right]$ and $\Pr\left[\, d' = 1 \,|\, d = 0 \,\right]$ using the above:

$$\begin{aligned}\Pr\left[\, d' = 1 \,|\, d = 1 \,\right] &= \sum_{q=1}^{T} \Pr[h = j] \cdot \Pr\left[\, d' = 1 \,|\, d = 1, h = j \,\right] \\ &= \frac{1}{T} \sum_{q=1}^{T} \Pr\left[\, d' = 1 \,|\, d = 1, h = j \,\right]. \\ \Pr\left[\, d' = 1 \,|\, d = 0 \,\right] &= \sum_{q=1}^{T} \Pr[h = j] \cdot \Pr\left[\, d' = 1 \,|\, d = 0, h = j \,\right] \\ &= \frac{1}{T} \sum_{q=1}^{T} \Pr\left[\, d' = 1 \,|\, d = 0, h = j \,\right]. \end{aligned}$$

We can now rewrite Eq. (3) as follows:

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}]}(\mathcal{D}_{\mathsf{md}}) \\ &= \sum_{q=1}^{T} \left(\Pr\left[\, d' = 1 \,|\, d = 1, h = j \,\right] - \Pr\left[\, d' = 1 \,|\, d = 0, h = j \,\right]\right) \\ &= T \cdot \left(\Pr\left[\, d' = 1 \,|\, d = 1 \,\right] - \Pr\left[\, d' = 1 \,|\, d = 0 \,\right]\right) \\ &= T \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{H}}(\mathcal{D}_{\mathsf{compr}}). \end{aligned}$$

This concludes the proof. □

Security reduction for MTP-MAC. We are ready to state the main result about the security of MTP-MAC, which we reduce to two assumptions in Proposition 7: (a) that $\mathsf{SHACAL\text{-}2.Ev}(k, m)$ is a PRF under known fixed m, partially known k and related-key-deriving function ϕ_{MAC} and (b) that $h_{256}(k, \cdot)$ is a one-time PRF. Concretely, $h_{256}(a, b) := a \hat{+} \mathsf{SHACAL\text{-}2.Ev}(b, a)$ and thus we require both assumptions to hold for SHACAL-2.[51] The former assumption is captured by the HRKPRF-security of SHACAL-2, whereas the latter was used in Lemma 1 in order to show that the MD-transform inherits the PRF-security of its underlying compression function (given that the initial state of the MD-transform is already uniformly random).

Proposition 7. *Let $\mathcal{D}_{\mathrm{UPRKPRF}}$ be an adversary against the* UPRKPRF*-security of* MTP-MAC *from Definition 8 under the related-key-deriving function ϕ_{MAC} from Fig. 24, 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 *with respect to ϕ_{MAC}, and an adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the* OTPRF*-security of the Merkle–Damgård transform of* SHA-256*, captured as function family $\mathsf{MD}[h_{256}]$ in Section 2.2, such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}}) \leq\ & 2 \cdot \mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}}) \\ & + 2 \cdot \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}]}(\mathcal{D}_{\mathrm{OTPRF}}).\end{aligned}$$

Proof. Consider game $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$ (Fig. 28).

```
Games $G_0$–$G_3$
$b \leftarrow\$ \{0,1\}$ ; $mk \leftarrow\$ \{0,1\}^{320}$ ; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MAC}}(mk)$
$X_{\mathcal{I}} \leftarrow X_{\mathcal{R}} \leftarrow \emptyset$ ; $b' \leftarrow\$ \mathcal{D}^{\mathrm{RoR}}_{\mathrm{UPRKPRF}}$ ; Return $b' = b$

$\mathrm{RoR}(u, p)$   // $u \in \{\mathcal{I}, \mathcal{R}\}$, $p \in \{0,1\}^*$
If $|p| < 256$ then return $\perp$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then return $\perp$
$X_u \leftarrow X_u \cup \{p_0\}$
$p \leftarrow \mathsf{SHA\text{-}pad}(0^{|mk_u|} \,\|\, p)$ ; $p_1 \leftarrow p[512 : |p|]$
$r \leftarrow \mathsf{SHACAL\text{-}2.Ev}(mk_u \,\|\, p_0, \mathsf{IV}_{256})$          // $G_0$
$r \leftarrow\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$                          // $G_1$
$H_1 \leftarrow \mathsf{IV}_{256} \hat{+} r$                                   // $G_0$–$G_1$
$H_1 \leftarrow\$ \{0,1\}^{256}$                                    // $G_2$
If $|p_1| > 0$ then
    $z \leftarrow \mathsf{MD}[h_{256}].\mathsf{Ev}(H_1, p_1)$                      // $G_0$–$G_2$
    $z \leftarrow\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$                      // $G_3$
Else $z \leftarrow H_1$
$\mathsf{msg\_key}_1 \leftarrow z[64 : 192]$
$\mathsf{msg\_key}_0 \leftarrow\$ \{0,1\}^{\mathsf{MTP\text{-}MAC.ol}}$
Return $\mathsf{msg\_key}_b$
```

Figure 74. Games G_0–G_3 for the proof of Proposition 7. The code added by expanding the algorithm MTP-MAC.Ev in game $\mathrm{G}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{UPRKPRF}}}$ is highlighted in grey; here MTP-HASH.Ev is expressed by recursively expanding the underlying MD-transform, where only the first call to the SHA-256 block cipher SHACAL-2 is singled out.

Recall that

$$\begin{aligned}\mathsf{MTP\text{-}MAC.Ev}(mk_u, p) &= \mathsf{SHA\text{-}256}(mk_u \,\|\, p)[64 : 192] \\ &= \mathsf{MD}[h_{256}].\mathsf{Ev}(\mathsf{IV}_{256}, \mathsf{SHA\text{-}pad}(mk_u \,\|\, p))[64 : 192].\end{aligned}$$

[51] 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 $k \hat{+} \mathsf{SHACAL\text{-}2.Ev}(m, k)$ being a PRF.

We first rewrite the game in a functionally equivalent way as G_0 in Fig. 74, splitting the $\mathsf{MD}[h_{256}].\mathsf{Ev}$ 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 the output of the first SHACAL-2 call should be indistinguishable from random if SHACAL-2 is a leakage-resilient PRF under related keys, and G_2 extends it to the output of the first compression function call h_{256}; games G_1 and G_2 are functionally equivalent so we have $\Pr[\mathrm{G}_1] = \Pr[\mathrm{G}_2]$. Then G_3 replaces the MD-transform call on the remaining input (if there is any) with a uniformly random value. This is the final reduction game, and it returns a random value regardless of the challenge bit, so $\mathcal{D}_{\mathrm{UPRKPRF}}$ cannot have a better than guessing advantage to win, i.e. $\Pr[\mathrm{G}_3] = \frac{1}{2}$.

```
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 $\perp$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then return $\perp$
$X_u \leftarrow X_u \cup \{p_0\}$
$p \leftarrow \mathsf{SHA\text{-}pad}(0^{256} \,\|\, p)$
$p_1 \leftarrow p[512:|p|]$
$r \leftarrow \mathrm{RoR}(u, p_0)$
$H_1 \leftarrow \mathsf{IV}_{256} \hat{+} r$
If $|p_1| > 0$ then
    $z \leftarrow \mathsf{MD}[h_{256}].\mathsf{Ev}(H_1, p_1)$
Else $z \leftarrow H_1$
$\mathsf{msg\_key}_1 \leftarrow z[64:192]$
$\mathsf{msg\_key}_0 \leftarrow\$ \{0,1\}^{\mathsf{MTP\text{-}MAC.ol}}$
Return $\mathsf{msg\_key}_b$
```

Figure 75. Adversary $\mathcal{D}_{\mathrm{HRKPRF}}$ against the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} for the proof of Proposition 7. Depending on the challenge bit in game $\mathrm{G}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}_{\mathrm{HRKPRF}}}$, adversary $\mathcal{D}_{\mathrm{HRKPRF}}$ simulates game G_0 or G_1 for adversary $\mathcal{D}_{\mathrm{UPRKPRF}}$.

```
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 $\perp$
$p_0 \leftarrow p[0:256]$
If $p_0 \in X_u$ then return $\perp$
$X_u \leftarrow X_u \cup \{p_0\}$
$p \leftarrow \mathsf{SHA\text{-}pad}(0^{256} \,\|\, p)$
$p_1 \leftarrow p[512:|p|]$
$H_1 \leftarrow\$ \{0,1\}^{256}$
If $|p_1| > 0$ then
    $z \leftarrow \mathrm{RoR}(p_1)$
Else $z \leftarrow H_1$
$\mathsf{msg\_key}_1 \leftarrow z[64:192]$
$\mathsf{msg\_key}_0 \leftarrow\$ \{0,1\}^{\mathsf{MTP\text{-}MAC.ol}}$
Return $\mathsf{msg\_key}_b$
```

Figure 76. Adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the OTPRF-security of $\mathsf{MD}[h_{256}]$ for the proof of Proposition 7. Depending on the challenge bit in game $\mathrm{G}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}],\mathcal{D}_{\mathrm{OTPRF}}}$, adversary $\mathcal{D}_{\mathrm{OTPRF}}$ simulates game G_2 or G_3 for adversary $\mathcal{D}_{\mathrm{UPRKPRF}}$.

We first build an adversary $\mathcal{D}_{\mathrm{HRKPRF}}$ against the HRKPRF-security of SHACAL-2 with respect to ϕ_{MAC} as shown in Fig. 75, such that we obtain $\Pr[\,\mathrm{G}_0\,] - \Pr[\,\mathrm{G}_1\,] = \mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}})$. Next, we build an adversary $\mathcal{D}_{\mathrm{OTPRF}}$ against the OTPRF-security of $\mathsf{MD}[h_{256}]$ as shown in Fig. 76, such that $\Pr[\,\mathrm{G}_1\,] - \Pr[\,\mathrm{G}_2\,] = \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}]}(\mathcal{D}_{\mathrm{OTPRF}})$. Note that $\mathcal{D}_{\mathrm{OTPRF}}$ calls its oracle RoR only if $\mathcal{D}_{\mathrm{UPRKPRF}}$ calls RoRSim on large enough inputs. However, adversary $\mathcal{D}_{\mathrm{UPRKPRF}}$ does not benefit

from calling its own RoR oracle on smaller inputs because at this point in the security reduction we already swapped out the output of the first call to compression function h_{256} with a uniformly random value.
We have the following:

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{uprkprf}}_{\mathsf{MTP\text{-}MAC},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{UPRKPRF}})\\ &= 2 \cdot \Pr[\mathrm{G}_0] - 1\\ &= 2 \cdot \left(\sum_{i=1}^{3} (\Pr[\mathrm{G}_{i-1}] - \Pr[\mathrm{G}_i]) + \Pr[\mathrm{G}_3] \right) - 1\\ &= 2 \cdot (\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}_{\mathrm{HRKPRF}}) + \mathsf{Adv}^{\mathsf{otprf}}_{\mathsf{MD}[h_{256}]}(\mathcal{D}_{\mathrm{OTPRF}})).\end{aligned}$$

The inequality follows. □

E.4 OTIND\$ of IGE

Recall that the deterministic symmetric encryption scheme MTP-SE is defined in Definition 10 as the IGE block cipher mode of operation, parametrised with the block cipher E = AES-256. We now show that IGE mode with any block cipher E is one-time indistinguishable (i.e. OTIND\$-secure; defined in Fig. 3) if the CBC mode, based on the same E, is one-time indistinguishable. This follows from an observation that the IGE encryption algorithm IGE[E].Enc, for any block cipher E, can be expressed in terms of the CBC encryption algorithm CBC[E].Enc as shown in Fig. 77.

```
IGE[E].Enc($K \| c_0 \| m_0, m_1 \| \ldots \| m_t$)   // $|K| = \mathsf{E.kl}$, $|c_0| = |m_i| = \mathsf{E.ol}$
$m_{-1} \leftarrow 0^{\mathsf{E.ol}}$
For $i = 1, \ldots, t$ do $m'_i \leftarrow m_i \oplus m_{i-2}$
$c'_1 \| \ldots \| c'_t \leftarrow$ CBC[E].Enc($K \| c_0, m'_1 \| \ldots \| m'_t$)
For $i = 1, \ldots, t$ do $c_i \leftarrow c'_i \oplus m_{i-1}$
Return $c_1 \| \ldots \| c_t$
```

Figure 77. Construction of algorithm IGE[E].Enc from algorithm CBC[E].Enc, where E is any block cipher.

Proposition 8. *Let* E *be a block cipher. Consider the deterministic symmetric encryption schemes* $\mathsf{SE}_{\mathsf{IGE}} = \mathsf{IGE[E]}$ *and* $\mathsf{SE}_{\mathsf{CBC}} = \mathsf{CBC[E]}$ *as defined in Fig. 4. 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*

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

Proof. Consider adversary $\mathcal{D}_{\mathsf{CBC}}$ in Fig. 78. We now show that when this adversary plays in game $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{CBC}},\mathcal{D}_{\mathsf{CBC}}}$ for any challenge bit $b \in \{0,1\}$, it simulates game $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{IGE}},\mathcal{D}_{\mathsf{IGE}}}$ for adversary $\mathcal{D}_{\mathsf{IGE}}$ with respect to the same challenge bit.

```
Adversary $\mathcal{D}^{\mathrm{RoR}}_{\mathsf{CBC}}$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{RoRSim}}_{\mathsf{IGE}}$
Return $b'$

RoRSim($m_1 \| \ldots \| m_t$)   // $|m_i| = \mathsf{E.ol}$
$m_{-1} \leftarrow 0^{\mathsf{E.ol}}$ ; $m_0 \leftarrow\$ \{0,1\}^{\mathsf{E.ol}}$
For $i = 1, \ldots, t$ do
    $m'_i \leftarrow m_i \oplus m_{i-2}$
$c' \leftarrow$ RoR($m'_1 \| \ldots \| m'_t$)
For $i = 1, \ldots, t$ do
    $c_i \leftarrow c'_i \oplus m_{i-1}$
Return $c_1 \| \ldots \| c_t$
```

Figure 78. Adversary $\mathcal{D}_{\mathsf{CBC}}$ against the OTIND\$-security of CBC[E] for the proof of Proposition 8.

If $b = 0$ in $\mathrm{G}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{CBC}},\mathcal{D}_{\mathsf{CBC}}}$, then RoR$(m')$ returns a uniformly random value as c', which is preserved under XOR. If $b = 1$, we get $c' = \mathsf{SE}_{\mathsf{CBC}}.\mathsf{Enc}(k, m')$ for a uniformly random $\mathsf{SE}_{\mathsf{CBC}}$ challenge key $k = K \| c'_0$.

Here $c'_i = \mathsf{E.Ev}(K, m_i \oplus m_{i-2} \oplus c'_{i-1})$. Since $c_i = c'_i \oplus m_{i-1}$, we get $c_i = \mathsf{E.Ev}(K, m_i \oplus c_{i-1}) \oplus m_{i-1}$ and so $c = \mathsf{SE}_{\mathsf{IGE}}.\mathsf{Enc}(k \,\|\, m_0, m)$. In both cases adversary $\mathcal{D}_{\mathsf{CBC}}$ perfectly simulates the RoR oracle for adversary $\mathcal{D}_{\mathsf{IGE}}$, so $\mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{CBC}}}(\mathcal{D}_{\mathsf{CBC}}) = \mathsf{Adv}^{\mathsf{otind\$}}_{\mathsf{SE}_{\mathsf{IGE}}}(\mathcal{D}_{\mathsf{IGE}})$. □

E.5 EINT of MTP-ME with respect to SUPP

In this section we prove that the message encoding scheme MTP-ME provides encoding integrity with respect to the support function SUPP for adversaries that request at most 2^{96} encoded payloads. As discussed in Section 3.5, this means that MTP-ME manages to prevent an attacker from silently replaying, reordering or dropping payloads in a channel that otherwise provides integrity (i.e. ensures that each received payload was at some point honestly produced by the opposite user). We note that if an adversary requests a single user to encode more than 2^{96} payloads, then this user's MTP-ME counter N_{sent} wraps modulo 2^{96}, allowing a trivial attack; below we will define an EINT-security adversary that in such case wins with advantage 1.

Proposition 9. *Let session_id* $\in \{0,1\}^{64}$ *and* $pb, \mathsf{bl} \in \mathbb{N}$. *Denote by* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathit{session\_id}, \mathit{pb}, \mathsf{bl}]$ *the message encoding scheme defined in Definition 6. Let* $\mathsf{supp} = \mathsf{SUPP}$ *be the support function defined in Fig. 32. Let* $\mathcal{F}$ *be any adversary against the* EINT*-security of* ME *with respect to* supp *making* $q \le 2^{96}$ *queries to its oracle* Send. *Then*

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

Proof. Consider game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$ (Fig. 15). For any receiver $u \in \{\mathcal{I}, \mathcal{R}\}$, the security game only allows oracle Recv queries on inputs u, p, aux such that the payload p was previously honestly produced by the opposite user $\overline{u}$ (i.e. p was produced in response to a prior oracle call $\textsc{Send}(\overline{u}, m', aux', r')$ for some values m', aux', r'). Thus it is sufficient to consider the following two cases, and show that the win flag cannot be set true in either of them: *a*) the payload p was successfully decoded by a prior call to $\textsc{Recv}(u, p, \cdot)$ (i.e. for an arbitrary auxiliary information value), and *b*) the payload p was not successfully decoded by a prior call to $\textsc{Recv}(u, p, \cdot)$.

In both cases, we will rely on the fact that the first $q = 2^{96}$ calls to oracle $\textsc{Send}(\overline{u}, \cdot, \cdot, \cdot)$ produce distinct payloads p. This is true because the algorithm ME.Encode ensures that every payload p returned by $\textsc{Send}(\overline{u}, \cdot, \cdot, \cdot)$ includes a 96-bit counter seq_no (in a fixed position of p) that starts at 0^{96} and is incremented modulo 2^{96} after each time a message is encoded.

We now consider the two cases listed above. Let

$$p = \mathsf{salt} \,\|\, \mathsf{session\_id} \,\|\, \mathsf{seq\_no} \,\|\, \mathsf{length} \,\|\, m' \,\|\, \mathsf{padding}.$$

Let $st_{\mathsf{ME},u} = (\mathsf{session\_id}, \cdot, N_{\mathsf{recv},u})$ be the ME state of the user u at the beginning of the current call to $\textsc{Recv}(u, p, aux)$, where aux is an arbitrary auxiliary information string.

Payload p is reused. There was a prior call to oracle $\textsc{Recv}(u, p, aux'')$ that successfully decoded p, meaning the transcript tr_u now contains $(\mathsf{recv}, m', p, aux'')$ for $m' \ne \bot$. We know that the condition $\mathsf{seq\_no} = N_{\mathsf{recv}}$ evaluated to true inside algorithm ME.Decode during the prior call (where seq_no was parsed from $p[128\colon 224]$, and $N_{\mathsf{recv}} < N_{\mathsf{recv},u}$ is a prerequisite to the prior decoding having succeeded). This means that the condition $\mathsf{seq\_no} = N_{\mathsf{recv},u}$ will evaluate to false during the current call, and the decoding will fail (i.e. return $\bot$). But the support function $\mathsf{supp}(u, \mathsf{tr}_u, \mathsf{tr}_{\overline{u}}, p, aux)$ likewise returns $m^* = \bot$, because $\mathsf{find}(\mathsf{recv}, \mathsf{tr}_u, p)$ iterates over all recv-type entries in tr_u and finds a match for p that corresponds to the decoded message $m' \ne \bot$. We are guaranteed that $m = m^*$, and hence $\mathcal{F}$ cannot set the win flag in this case.

Payload p is fresh. Either there was no $\textsc{Recv}(u, p, aux'')$ call in the past for any aux'', or each entry $(\mathsf{recv}, m, p, \cdot)$ in the transcript tr_u has $m = \bot$. 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 returns $(n_u, \bot)$ where n_u is the number of entries of tr_u of the form $(\mathsf{recv}, m, p', \cdot)$ for $m \ne \bot$ and $p' \ne p$. Next, it calls $\mathsf{find}(\mathsf{sent}, \mathsf{tr}_{\overline{u}}, p)$ which returns $(n_{\overline{u}}, m')$ 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 the target entry. Then the support function checks whether $n_{\overline{u}} = n_u + 1$.

Let us consider both $n_{\overline{u}}$ and n_u. Whenever an entry $(\mathsf{recv}, m, p', \cdot)$ for $m \ne \bot$ 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 $\mathsf{ME.Encode}$ was called, saving the prior number of sent messages $N_{\mathsf{sent},\overline{u}}$ in the sequence number field $\mathsf{seq\_no}$, then incrementing it by one and including it in the updated state of ME. It follows that $n_{\overline{u}} = \mathsf{seq\_no} + 1$ as long as $n_{\overline{u}} \leq 2^{96}$, which we assumed at the beginning. Then the support function and the algorithm $\mathsf{ME.Decode}(st_{\mathsf{ME},u}, p, aux)$ both evaluate the same condition, checking whether $\mathsf{seq\_no} + 1 = N_{\mathsf{recv},u} + 1$. Hence the support function returns m' if and only if $\mathsf{ME.Decode}$ does, and $\mathcal{F}$ cannot win in this case either. This concludes the proof. □

Counter overflows. For completeness, let us now deal with the case of an overflow (modulo 2^{96}) happening in the N_{sent} and N_{recv} counters of $\mathsf{MTP\text{-}ME}$. In this case, we show that there exists an adversary that can trivially win with advantage 1.

```
Adversary $\mathcal{F}^{\textsc{Send},\textsc{Recv}}(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
// Let $aux = \varepsilon$. Choose any $m \in \mathsf{ME.MS}$ and $r \in \mathsf{ME.EncRS}$.
For $i = 1, \ldots, 2^{96} + 1$ do
  $p_i \leftarrow \textsc{Send}(\mathcal{I}, m, aux, r)$
  $\textsc{Recv}(\mathcal{R}, p_i, aux)$
```

Figure 79. Adversary $\mathcal{F}$ against the EINT-security of $\mathsf{MTP\text{-}ME}$ with respect to SUPP for the proof of Proposition 10.

Proposition 10. *Let* $\mathsf{session\_id} \in \{0,1\}^{64}$ *and* $\mathsf{pb}, \mathsf{bl} \in \mathbb{N}$. *Denote by* $\mathsf{ME} = \mathsf{MTP\text{-}ME}[\mathsf{session\_id}, \mathsf{pb}, \mathsf{bl}]$ *the message encoding scheme defined in Definition 6. Let* $\mathsf{supp} = \mathsf{SUPP}$ *be the support function defined in Fig. 32. Let* $\mathcal{F}$ *be an adversary against the* EINT*-security of* ME *with respect to* supp *as defined in Fig. 79, making* $q = 2^{96} + 1$ *queries to its oracle* $\textsc{Send}$. *Then*

$$\mathsf{Adv}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp}}(\mathcal{F}) = 1.$$

Proof. Adversary $\mathcal{F}$ repeatedly queries its oracles $\textsc{Send}$ and $\textsc{Recv}$ in order to exhaust all possible values of the 96-bit field $\mathsf{seq\_no}$. When user $\mathcal{I}$ sends the 2^{96}-th payload, its counter overflows (modulo 2^{96}) to become $N_{\mathsf{sent}} = 0$; after user $\mathcal{R}$ accepts this payload, its counter likewise overflows to become $N_{\mathsf{recv}} = 0$. It follows that the next payload will be equal to the first payload, i.e. $p_0 = p_{2^{96}+1}$.

This causes a mismatch: in $\mathsf{ME.Decode}$ the $\mathsf{seq\_no}$ check passes because the counter wrapped around, and so it returns m. But the corresponding evaluation of supp in game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$ determines that the label $p_{2^{96}+1} = p_0$ was already received before (i.e. $\mathsf{find}(\mathsf{recv}, \mathsf{tr}_{\mathcal{R}}, p_0) \to m \neq \perp$) so the support function returns $\perp$. This triggers the win flag in game $\mathrm{G}^{\mathsf{eint}}_{\mathsf{ME},\mathsf{supp},\mathcal{F}}$. □

E.6 UNPRED of MTP-SE and MTP-ME

We now prove unpredictability of the deterministic symmetric encryption scheme $\mathsf{SE} = \mathsf{MTP\text{-}SE}$ (Definition 10) with respect to the message encoding scheme $\mathsf{ME} = \mathsf{MTP\text{-}ME}$ (Definition 6). In our proof, we show that it is hard for any adversary $\mathcal{F}$ to find an SE ciphertext c_{se} such that its decryption under a uniformly random key $k \in \{0,1\}^{\mathsf{SE.kl}}$ begins with $p_1 = \mathsf{salt} \| \mathsf{session\_id}$, where $\mathsf{session\_id}$ is a value chosen by the adversary via st_{ME} and salt is arbitrary.

Recall that Definition 10 specifies $\mathsf{MTP\text{-}SE} = \mathsf{IGE}[\mathsf{AES\text{-}256}]$. We state and prove our result for a more general case of $\mathsf{SE} = \mathsf{IGE}[\mathsf{E}]$, where E is an arbitrary block cipher with block length $\mathsf{E.ol} = 128$ (that matches the output block length bl of ME).

Note that our proof is not tight, i.e. the advantage could potentially be lower if we also considered the $\mathsf{seq\_no}$ 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 $\mathsf{MTP\text{-}ME}$ upon decoding. The bound could also be improved if $\mathsf{MTP\text{-}ME}$ 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 11. *Let* $\textsf{session\_id} \in \{0,1\}^{64}$*,* $\textsf{pb} \in \mathbb{N}$ *and* $\textsf{bl} = 128$*. Denote by* $\textsf{ME} = \textsf{MTP-ME}[\textsf{session\_id}, \textsf{pb}, \textsf{bl}]$ *the message encoding scheme defined in Definition 6. Let* $\textsf{E}$ *be a block cipher with block length* $\textsf{E.ol} = 128$*. Let* $\textsf{SE} = \textsf{IGE}[\textsf{E}]$ *be the deterministic symmetric encryption scheme defined in Section 2.2. Let* $\mathcal{F}$ *be any adversary against the* UNPRED*-security of* $\textsf{SE}, \textsf{ME}$ *making* q_{CH} *queries to its oracle* CH*. Then*

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

Proof. We rewrite game $\text{G}^{\textsf{unpred}}_{\textsf{SE},\textsf{ME},\mathcal{F}}$ (Fig. 35) as game G in Fig. 80 by expanding algorithms $\textsf{SE.Dec}$ and $\textsf{ME.Decode}$ with the following relaxations. Algorithm $\textsf{SE.Dec}$ is partially expanded to only decrypt the first block of ciphertext c_{se} into a 128-bit long payload block p_1. Algorithm $\textsf{ME.Decode}$ is partially expanded to only surface the sanity-check of p_1, which (as per Fig. 20) should consist of two concatenated 64-bit values $\textsf{salt} \,\|\, \textsf{session\_id}$, where $\textsf{session\_id}$ should match the fixed constant that is stored inside the ME's state $st_{\textsf{ME}}$. Since game G does not implement all of the checks from algorithm $\textsf{ME.Decode}$, adversary $\mathcal{F}$ is more likely to win in G than in the original game $\text{G}^{\textsf{unpred}}_{\textsf{SE},\textsf{ME},\mathcal{F}}$, but $\mathcal{F}$ is not able to detect these changes because CH always returns $\perp$. We have

$$\textsf{Adv}^{\textsf{unpred}}_{\textsf{SE},\textsf{ME}}(\mathcal{F}) \leq \Pr[\,\text{G}\,].$$

```
Game G
win ← false
$\mathcal{F}^{\text{Expose},\text{Ch}}$
Return win

Expose($u$, msg_key)
// msg_key $\in \{0,1\}^*$
S[$u$, msg_key] ← true
Return T[$u$, msg_key]
```

```
Ch($u$, msg_key, $c_{se}$, $st_{\textsf{ME}}$, $aux$)
// msg_key $\in \{0,1\}^*$
If $\neg$S[$u$, msg_key] then
    If T[$u$, msg_key] = $\perp$ then
        T[$u$, msg_key] $\leftarrow_\$ \{0,1\}^{\textsf{SE.kl}}$
    $k$ ← T[$u$, msg_key]
    $K \,\|\, c_0 \,\|\, p_0 \leftarrow k$   // s.t. $|K| = \textsf{E.kl}$, $|c_0| = |p_0| = 128$
    $c_1 \leftarrow c_{se}[0:128]$
    $p_1 \leftarrow \textsf{E.Inv}(K, c_1 \oplus p_0) \oplus c_0$
    (session_id, $N_{\textsf{sent}}$, $N_{\textsf{recv}}$) ← $st_{\textsf{ME}}$
    session_id' ← $p_1[64:128]$
    If session_id' = session_id then
        win ← true
Return $\perp$
```

Figure 80. Game G for the proof of Proposition 11. Game G is built from game $\text{G}^{\textsf{unpred}}_{\textsf{SE},\textsf{ME},\mathcal{F}}$ by partially expanding the algorithms $\textsf{SE.Dec}$ and $\textsf{ME.Decode}$ to recover the first 128-bit block of payload p and then verify the 64-bit value of $\textsf{session\_id}$ inside this block. The expanded code is highlighted in grey.

The adversary $\mathcal{F}$ can only win in game G if $p_1[64:128] = \textsf{session\_id}$ for some p_1 that is defined by the equation $p_1 = \textsf{E.Inv}(K, c_1 \oplus p_0) \oplus c_0$. We can rewrite this winning condition as

$$\textsf{E.Inv}(K, c_1 \oplus p_0)[64:128] \oplus \textsf{session\_id} = c_0[64:128].$$

Here $c_0[64:128]$ is a bit string that is sampled uniformly at random for each pair $(u, \textsf{msg\_key})$ and that is unknown to the adversary.

Consider for a moment a particular pair $(u, \textsf{msg\_key})$; suppose that $\mathcal{F}$ makes $q_{u,\textsf{msg\_key}}$ queries to CH relating to this pair. These queries result in some specific set of values $X_{u,\textsf{msg\_key}}$ for $\textsf{E.Inv}(K, c_1 \oplus p_0)[64:128] \oplus \textsf{session\_id}$ arising in the game. Moreover, $\mathcal{F}$ wins for one of these queries if and only if some element of the set $X_{u,\textsf{msg\_key}}$ 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 $\perp$). Combining these facts, we see that $\mathcal{F}$'s winning probability for this set of $q_{u,\textsf{msg\_key}}$ queries is no larger than $q_{u,\textsf{msg\_key}}/2^{64}$ (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 $(u, \textsf{msg\_key})$ 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 $(u, \textsf{msg\_key})$ involved in $\mathcal{F}$'s queries and using the union bound, we obtain that

$$\Pr[\,\text{G}\,] \leq q_{\text{CH}} \cdot 2^{-64}.$$

The inequality follows. □

F Concrete security of the novel SHACAL-2 assumptions in the ICM

In Section 5.2 we defined the LRKPRF-security and the HRKPRF-security of the block cipher $\mathsf{SHACAL\text{-}2}$ (with respect to some related-key-deriving functions). Both assumptions roughly require $\mathsf{SHACAL\text{-}2}$ to be related-key PRF-secure when evaluated on the fixed input IV_{256}. As discussed in Section 5.2, these assumptions construct the $\mathsf{SHACAL\text{-}2}$ challenge keys in significantly different ways, but both of them allow the attacker to directly choose certain bits of the challenge keys.

The notions of LRKPRF and HRKPRF security are novel, and hence further analysis is needed to determine whether they hold for $\mathsf{SHACAL\text{-}2}$ in the standard model. We leave this task as an open problem. Here we justify both assumptions in the ideal cipher model [Sha49], where a block cipher is modelled as a random (and independently chosen) permutation for every key in its key space. More formally, our analysis will assume that $\mathsf{SHACAL\text{-}2.Ev}(sk, \cdot)$ is a random permutation for every choice of $sk \in \{0,1\}^{\mathsf{SHACAL\text{-}2.kl}}$. This will allow us to derive an upper bound for any adversary attacking either of the two assumptions.

In this section, for any $\ell \in \mathbb{N}$ we use $\mathcal{P}(\ell)$ to denote the set of all bit-string permutations with domain and range $\{0,1\}^\ell$. For any permutation $\pi \in \mathcal{P}(\ell)$ and any $x, y \in \{0,1\}^\ell$ we write $\pi(x)$ to denote the result of evaluating π on x, and we write $\pi^{-1}(y)$ to denote the result of evaluating the inverse of π on y. A basic correctness condition stipulates that $\pi^{-1}(\pi(x)) = x$ for all $x \in \{0,1\}^\ell$.

Proposition 12. *Let ϕ_{KDF} be the related-key-deriving function as defined in Fig. 24. Let $\phi_{\mathsf{SHACAL\text{-}2}}$ be the related-key-deriving function as defined in Fig. 29. Let the block cipher $\mathsf{SHACAL\text{-}2}$ from Section 2.2 be modelled as the ideal cipher with key length $\mathsf{SHACAL\text{-}2.kl}$ and block length $\mathsf{SHACAL\text{-}2.ol}$. Let $\mathcal{D}$ be any adversary against the LRKPRF-security of $\mathsf{SHACAL\text{-}2}$ with respect to $\phi_{\mathsf{KDF}}, \phi_{\mathsf{SHACAL\text{-}2}}$. Assume that $\mathcal{D}$ makes a total of q queries to its ideal cipher oracles. Then the advantage of $\mathcal{D}$ is upper-bounded as follows:*

$$\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) < 2^{-156} + q \cdot 2^{-285}.$$

Proof. This proof uses games G_0–G_3 in Fig. 81, and games G_4–G_5 in Fig. 82. Game G_0 is designed to be equivalent to game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ in the ideal cipher model. In particular, game G_0 gives its adversary $\mathcal{D}$ access to oracles IC and IC^{-1} that evaluate the direct and inverse calls to the ideal cipher respectively. The evaluation of $\mathsf{SHACAL\text{-}2.Ev}(sk_i, \mathsf{IV}_{256})$ inside oracle RoR of game $\mathrm{G}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}},\mathcal{D}}$ is replaced with a call to $\mathsf{IC}(sk_i, \mathsf{IV}_{256})$ in game G_0. Oracles IC and IC^{-1} assign a random permutation $\pi\colon \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}} \to \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$ to any block cipher key $sk \in \{0,1\}^{\mathsf{SHACAL\text{-}2.kl}}$ that is seen for the first time, and store it in the table entry $\mathsf{P}[sk]$. On input (sk, x) oracle IC evaluates the permutation $\pi \leftarrow \mathsf{P}[sk]$ on input x and returns the result $\pi(x)$; on input (sk, y) oracle IC evaluates the inverse of the permutation $\pi \leftarrow \mathsf{P}[sk]$ on input y and returns the result $\pi^{-1}(y)$. Game G_0 also expands the code of the related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$. We have

$$\mathsf{Adv}^{\mathsf{lrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{KDF}},\phi_{\mathsf{SHACAL\text{-}2}}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}_0] - 1.$$

Game G_0 adds some bookkeeping code highlighted in green to its oracle RoR. This code does not affect the input-output behaviour of RoR.

Throughout transitions from G_0 to G_3, the code highlighted in green is used to gradually eliminate the possibility that the adversary $\mathcal{D}$ calls its oracle RoR on two distinct input tuples $(u', i', \mathsf{msg\_key}')$ and $(u, i, \mathsf{msg\_key})$ that both lead to the same block cipher key sk_i. If this was not true, then $\mathcal{D}$ could trivially win the game by comparing the equality of outputs returned by $\mathrm{RoR}(u', i', \mathsf{msg\_key}')$ and $\mathrm{RoR}(u, i, \mathsf{msg\_key})$. Depending on the values of $i', i \in \{0,1\}$ used in $(u', i', \mathsf{msg\_key}') \neq (u, i, \mathsf{msg\_key})$, the block cipher keys produced across the two corresponding calls to RoR can be the equal if the intersection of sets $\{\mathsf{msg\_key}' \,\|\, kk_{u',0}, kk_{u',1} \,\|\, \mathsf{msg\_key}'\}$ and $\{\mathsf{msg\_key} \,\|\, kk_{u,0}, kk_{u,1} \,\|\, \mathsf{msg\_key}\}$ is not empty. We now show that it will be empty with high probability.

Let $i \in \{0,1,2\}$. Games G_i and G_{i+1} are identical until bad_i is set. We have

$$\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}] \leq \Pr[\mathsf{bad}_i^{\mathrm{G}_i}].$$

Note that whenever the bad_i flag is set in game G_{i+1}, we use the **abort**(false) instruction (as introduced in Section 2.1) to immediately halt the game with output false, meaning $\mathcal{D}$ loses the game right after setting the flag. In order for it to be possible to set each of the flags, certain bit segments in kk need to be equal; this is a necessary, but not a sufficient condition. We use that to upper bound the

```
Games $\mathrm{G}_0$–$\mathrm{G}_3$
$b \leftarrow\$ \{0,1\}$ ; $kk \leftarrow\$ \{0,1\}^{672}$
$kk_{\mathcal{I},0} \leftarrow kk[0:288]$ ; $kk_{\mathcal{R},0} \leftarrow kk[64:352]$
$kk_{\mathcal{I},1} \leftarrow kk[320:608]$ ; $kk_{\mathcal{R},1} \leftarrow kk[384:672]$
$kk_{\mathcal{I}} \leftarrow (kk_{\mathcal{I},0}, kk_{\mathcal{I},1})$ ; $kk_{\mathcal{R}} \leftarrow (kk_{\mathcal{R},0}, kk_{\mathcal{R},1})$
$b' \leftarrow\$ \mathcal{D}^{\text{RoR},\mathsf{IC},\mathsf{IC}^{-1}}$ ; Return $b' = b$

$\text{RoR}(u, i, \mathsf{msg\_key})$   // $u \in \{\mathcal{I},\mathcal{R}\}$, $i \in \{0,1\}$, $|\mathsf{msg\_key}| = 128$
$(kk_{u,0}, kk_{u,1}) \leftarrow kk_u$
$sk_0 \leftarrow \mathsf{SHA\text{-}pad}(\mathsf{msg\_key} \,\|\, kk_{u,0})$
$sk_1 \leftarrow \mathsf{SHA\text{-}pad}(kk_{u,1} \,\|\, \mathsf{msg\_key})$
If $(\mathsf{K}[sk_i] \neq \bot) \wedge (\mathsf{K}[sk_i] \neq (u, i, \mathsf{msg\_key}))$ then
    $(u', i', \mathsf{msg\_key}') \leftarrow \mathsf{K}[sk_i]$
    If $(i' = 0) \wedge (i = 0)$ then
        // $\mathsf{msg\_key}' \,\|\, kk_{u',0} = \mathsf{msg\_key} \,\|\, kk_{u,0}$
        // i.e. $kk_{\mathcal{I},0} = kk_{\mathcal{R},0}$
        $\mathsf{bad}_0 \leftarrow \mathsf{true}$
        $\mathbf{abort}(\mathsf{false})$                    // $\mathrm{G}_1$–$\mathrm{G}_3$
    Else if $(i' = 1) \wedge (i = 1)$ then
        // $kk_{u',1} \,\|\, \mathsf{msg\_key}' = kk_{u,1} \,\|\, \mathsf{msg\_key}$
        // i.e. $kk_{\mathcal{I},1} = kk_{\mathcal{R},1}$
        $\mathsf{bad}_1 \leftarrow \mathsf{true}$
        $\mathbf{abort}(\mathsf{false})$                    // $\mathrm{G}_2$–$\mathrm{G}_3$
    Else   // $i' \neq i$
        // $\mathsf{msg\_key}' \,\|\, kk_{u',0} = kk_{u,1} \,\|\, \mathsf{msg\_key}$ if $(i' = 0) \wedge (i = 1)$
        // $kk_{u',1} \,\|\, \mathsf{msg\_key}' = \mathsf{msg\_key} \,\|\, kk_{u,0}$ if $(i' = 1) \wedge (i = 0)$
        // i.e. $kk_{a,0}[0:160] = kk_{b,1}[128:288]$ for $a, b \in \{\mathcal{I},\mathcal{R}\}$
        $\mathsf{bad}_2 \leftarrow \mathsf{true}$
        $\mathbf{abort}(\mathsf{false})$                    // $\mathrm{G}_3$
$\mathsf{K}[sk_i] \leftarrow (u, i, \mathsf{msg\_key})$
$y_1 \leftarrow \mathsf{IC}(sk_i, \mathsf{IV}_{256})$
If $\mathsf{T}[u, i, \mathsf{msg\_key}] = \bot$ then
    $\mathsf{T}[u, i, \mathsf{msg\_key}] \leftarrow\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
$y_0 \leftarrow \mathsf{T}[u, i, \mathsf{msg\_key}]$ ; Return $y_b$

$\mathsf{IC}(sk, x)$   // $|sk| = \mathsf{SHACAL\text{-}2.kl}$, $|x| = \mathsf{SHACAL\text{-}2.ol}$
If $\mathsf{P}[sk] = \bot$ then $\mathsf{P}[sk] \leftarrow\$ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$ ; Return $\pi(x)$

$\mathsf{IC}^{-1}(sk, y)$   // $|sk| = \mathsf{SHACAL\text{-}2.kl}$, $|y| = \mathsf{SHACAL\text{-}2.ol}$
If $\mathsf{P}[sk] = \bot$ then $\mathsf{P}[sk] \leftarrow\$ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$ ; Return $\pi^{-1}(y)$
```

Figure 81. Games G_0–G_3 for the proof of Proposition 12. The code added by expanding the related-key-deriving functions ϕ_{KDF} and $\phi_{\mathsf{SHACAL\text{-}2}}$ in game G_0 is highlighted in grey.

corresponding probabilities as follows, when measured over the randomness of sampling $kk \leftarrow\!\!\$\ \{0,1\}^{672}$ (here kk is implicitly parsed into $kk_{\mathcal{I},0}$, $kk_{\mathcal{I},1}$, $kk_{\mathcal{R},0}$, $kk_{\mathcal{R},1}$ as specified by the related-key-deriving function ϕ_{KDF}):

$$\begin{aligned}
\Pr[\mathsf{bad}_0^{\mathrm{G}_0}] &\le \Pr[kk_{\mathcal{I},0} = kk_{\mathcal{R},0}] = 2^{-288}.\\
\Pr[\mathsf{bad}_1^{\mathrm{G}_1}] &\le \Pr[kk_{\mathcal{I},1} = kk_{\mathcal{R},1}] = 2^{-288}.\\
\Pr[\mathsf{bad}_2^{\mathrm{G}_2}] &\le \Pr[\exists a, b \in \{\mathcal{I}, \mathcal{R}\} \colon kk_{a,0}[0:160] = kk_{b,1}[128:288]]\\
&\le \sum_{a,b\in\{\mathcal{I},\mathcal{R}\}} \Pr[kk_{a,0}[0:160] = kk_{b,1}[128:288]]\\
&= \Pr[kk[0:160] = kk[448:608]]\\
&\quad + \Pr[kk[0:160] = kk[512:672]]\\
&\quad + \Pr[kk[64:224] = kk[448:608]]\\
&\quad + \Pr[kk[64:224] = kk[512:672]]\\
&= 4 \cdot 2^{-160}\\
&= 2^{-158}.
\end{aligned}$$

We used the union bound in order to upper-bound $\Pr[\mathsf{bad}_2^{\mathrm{G}_2}]$. The upper bound on $\Pr[\mathsf{bad}_2^{\mathrm{G}_2}]$ could be significantly lowered by capturing the idea that the adversary $\mathcal{D}$ also needs to match both $\mathsf{msg\_key}$ and $\mathsf{msg\_key}'$ to the corresponding bits of kk. Adversary $\mathcal{D}$ cannot efficiently learn kk based on the responses from its oracles; the best it could do in an attempt to set bad_2 is to repeatedly try guessing the bits of kk by supplying different values of $\mathsf{msg\_key}, \mathsf{msg\_key}' \in \{0,1\}^{128}$.[52] The upper bound would then depend on the number of calls that $\mathcal{D}$ makes to its oracle RoR. We omit this analysis, and settle for a less precise lower-bound.

Game G_4 in Fig. 82 rewrites game G_3 in an equivalent way. The calls to $\mathsf{SHA\text{-}pad}$ are expanded according to its definition in Fig. 6. The three conditional statements that inevitably lead to **abort**(false) are replaced with an immediate call to **abort**(false). The code of the IC oracle is expanded in place of the single call to IC from within oracle RoR. These changes are marked in grey. We have

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

Game G_4 also adds some code highlighted in green to its ideal cipher oracles IC and IC^{-1}; this does not affect the input-output behaviour of the oracles.

By now, we have determined that in game G_4 each query to the RoR oracle uses a distinct block cipher key sk (except in the trivial case when RoR is queried twice with the same input tuple). In the ideal cipher model, this key is mapped to a random permutation, which is then stored in $\mathsf{P}[sk]$. We want to show that the adversary $\mathcal{D}$ cannot distinguish between this permutation evaluated on input IV_{256} and a uniformly random value from $\{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$. The only way it could distinguish between the two cases is if $\mathcal{D}$ managed to guess sk and query one of its ideal cipher oracles with sk as input. This requires $\mathcal{D}$ to guess the corresponding 288-bit segment of kk that is used to build sk inside oracle RoR: either $sk[128:416]$ should be equal to one of $\{kk_{\mathcal{I},0}, kk_{\mathcal{R},0}\}$, or $sk[0:288]$ should be equal to one of $\{kk_{\mathcal{I},1}, kk_{\mathcal{R},1}\}$. We show that this is hard to achieve.

Formally, games G_4 and G_5 are identical until bad_3 is set. We have

$$\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] \le \Pr[\mathsf{bad}_3^{\mathrm{G}_5}].$$

Note that kk can take a total of 2^{672} different values. Each query to an ideal cipher oracle IC or IC^{-1} either sets bad_3, or silently rejects *at most* $4 \cdot 2^{384}$ candidate kk values. In particular, if bad_3 was not set, then kk cannot contain $sk[128:416]$ in one of the positions that correspond to $kk_{\mathcal{I},0}$ or $kk_{\mathcal{R},0}$, and it cannot contain $sk[0:288]$ in one of the positions that correspond to $kk_{\mathcal{I},1}$ or $kk_{\mathcal{R},1}$. Here we use the fact that for any fixed 288-bit string, there are $2^{672-288} = 2^{384}$ different ways to choose the remaining bits of kk. Beyond eliminating some candidate keys as per above, the ideal cipher oracles

[52] Note that the 256 total bits of $\mathsf{msg\_key}, \mathsf{msg\_key}' \in \{0,1\}^{128}$ should not always be independent. For example, when trying to match $\mathsf{msg\_key}' \parallel kk_{\mathcal{R},0} = kk_{\mathcal{I},1} \parallel \mathsf{msg\_key}$, the 32-bit long bit-string $kk[320:352]$ should appear both in the prefix of $\mathsf{msg\_key}'$ and in the suffix of $\mathsf{msg\_key}$.

do not return any useful information about the contents of kk. So we can upper-bound the probability of setting bad_3 in game G_5 after making q queries to oracles IC and IC^{-1} as follows:

$$\Pr[\mathsf{bad}_3^{\mathrm{G}_5}] \leq q \cdot \frac{4 \cdot 2^{384}}{2^{672}} = q \cdot 2^{-286}.$$

Finally, in game G_5 the ideal cipher oracles can no longer help $\mathcal{D}$ learn any information about the bits of kk, or about the corresponding random permutations. So we have

$$\Pr[\mathrm{G}_5] = \frac{1}{2}.$$

Combining all of the above, we get

$$\begin{aligned}\Pr[\mathrm{G}_0] &= \sum_{0 \leq i \leq 4} (\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}]) + \Pr[\mathrm{G}_5] \\ &= (2^{-288} + 2^{-288} + 2^{-158} + 0 + q \cdot 2^{-286}) + \frac{1}{2} \\ &< 2^{-157} + q \cdot 2^{-286} + \frac{1}{2}.\end{aligned}$$

The inequality follows. □

```
Games $\mathrm{G}_4$–$\mathrm{G}_5$
$b \leftarrow\!\!\$ \{0,1\}$ ; $kk \leftarrow\!\!\$ \{0,1\}^{672}$
$kk_{\mathcal{I},0} \leftarrow kk[0:288]$ ; $kk_{\mathcal{R},0} \leftarrow kk[64:352]$
$kk_{\mathcal{I},1} \leftarrow kk[320:608]$ ; $kk_{\mathcal{R},1} \leftarrow kk[384:672]$
$kk_{\mathcal{I}} \leftarrow (kk_{\mathcal{I},0}, kk_{\mathcal{I},1})$ ; $kk_{\mathcal{R}} \leftarrow (kk_{\mathcal{R},0}, kk_{\mathcal{R},1})$
$b' \leftarrow\!\!\$ \mathcal{D}^{\mathrm{RoR},\mathsf{IC},\mathsf{IC}^{-1}}$ ; Return $b' = b$

$\mathrm{RoR}(u, i, \mathsf{msg\_key})$   // $u \in \{\mathcal{I},\mathcal{R}\}$, $i \in \{0,1\}$, $|\mathsf{msg\_key}| = 128$
$(kk_{u,0}, kk_{u,1}) \leftarrow kk_u$
$sk_0 \leftarrow \mathsf{msg\_key} \,\|\, kk_{u,0} \,\|\, 1 \,\|\, 0^{31} \,\|\, \langle|416|\rangle_{64}$
$sk_1 \leftarrow kk_{u,1} \,\|\, \mathsf{msg\_key} \,\|\, 1 \,\|\, 0^{31} \,\|\, \langle|416|\rangle_{64}$
If $(\mathsf{K}[sk_i] \neq \bot) \wedge (\mathsf{K}[sk_i] \neq (u, i, \mathsf{msg\_key}))$ then $\mathbf{abort}(\mathsf{false})$
$\mathsf{K}[sk_i] \leftarrow (u, i, \mathsf{msg\_key})$
If $\mathsf{P}[sk_i] = \bot$ then $\mathsf{P}[sk_i] \leftarrow\!\!\$ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk_i]$ ; $y_1 \leftarrow \pi(\mathsf{IV}_{256})$
If $\mathsf{T}[u, i, \mathsf{msg\_key}] = \bot$ then
    $\mathsf{T}[u, i, \mathsf{msg\_key}] \leftarrow\!\!\$ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
$y_0 \leftarrow \mathsf{T}[u, i, \mathsf{msg\_key}]$ ; Return $y_b$

$\mathsf{IC}(sk, x)$   // $|sk| = \mathsf{SHACAL\text{-}2.kl}$, $|x| = \mathsf{SHACAL\text{-}2.ol}$
If $(sk[128:416] \in \{kk_{\mathcal{I},0}, kk_{\mathcal{R},0}\}) \vee$
    $(sk[0:288] \in \{kk_{\mathcal{I},1}, kk_{\mathcal{R},1}\})$ then
        $\mathsf{bad}_3 \leftarrow \mathsf{true}$
        $\mathbf{abort}(\mathsf{false})$                              // $\mathrm{G}_5$
If $\mathsf{P}[sk] = \bot$ then $\mathsf{P}[sk] \leftarrow\!\!\$ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$ ; Return $\pi(x)$

$\mathsf{IC}^{-1}(sk, y)$   // $|sk| = \mathsf{SHACAL\text{-}2.kl}$, $|y| = \mathsf{SHACAL\text{-}2.ol}$
If $(sk[128:416] \in \{kk_{\mathcal{I},0}, kk_{\mathcal{R},0}\}) \vee$
    $(sk[0:288] \in \{kk_{\mathcal{I},1}, kk_{\mathcal{R},1}\})$ then
        $\mathsf{bad}_3 \leftarrow \mathsf{true}$
        $\mathbf{abort}(\mathsf{false})$                              // $\mathrm{G}_5$
If $\mathsf{P}[sk] = \bot$ then $\mathsf{P}[sk] \leftarrow\!\!\$ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$ ; Return $\pi^{-1}(y)$
```

Figure 82. Games G_4–G_5 for the proof of Proposition 12. The code highlighted in grey is functionally equivalent to the corresponding code in G_3.

Proposition 13. *Let ϕ_{MAC} be the related-key-deriving function as defined in Fig. 24. Let the block cipher* SHACAL-2 *from Section 2.2 be modelled as the ideal cipher with key length* SHACAL-2.kl *and block length* SHACAL-2.ol*. Let $\mathcal{D}$ be any adversary against the* HRKPRF*-security of* SHACAL-2 *with respect to ϕ_{MAC}. Assume that $\mathcal{D}$ makes a total of q queries to its ideal cipher oracles. Then the advantage of $\mathcal{D}$ is upper-bounded as follows:*

$$\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) \le 2^{-255} + q \cdot 2^{-254}.$$

Proof. This proof presents a very similar (but simpler) argument compared to the one used for the proof of Proposition 12. So we provide the games and only the core analysis here, with a minimal amount of justification for each of the steps.

```
Games $G_0$–$G_3$
$b \leftarrow\!\!\$\ \{0,1\}$ ; $mk \leftarrow\!\!\$\ \{0,1\}^{320}$
$mk_{\mathcal{I}} \leftarrow mk[0:256]$
$mk_{\mathcal{R}} \leftarrow mk[64:320]$
If $mk_{\mathcal{I}} = mk_{\mathcal{R}}$ then
    $\mathsf{bad}_0 \leftarrow \mathsf{true}$
    Return false                                              // $G_1$–$G_3$
$b' \leftarrow\!\!\$\ \mathcal{D}^{\mathrm{RoR},\mathsf{IC},\mathsf{IC}^{-1}}$ ; Return $b' = b$

$\mathrm{RoR}(u,p)$    // $u \in \{\mathcal{I},\mathcal{R}\}$, $|p| = 256$
$sk \leftarrow mk_u \,\|\, p$
$y_1 \leftarrow \mathsf{IC}(sk, \mathsf{IV}_{256})$                           // $G_0$–$G_1$
If $\mathsf{P}[sk] = \bot$ then $\mathsf{P}[sk] \leftarrow\!\!\$\ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$      // $G_2$–$G_3$
$\pi \leftarrow \mathsf{P}[sk]$ ; $y_1 \leftarrow \pi(\mathsf{IV}_{256})$                      // $G_2$–$G_3$
If $\mathsf{T}[u,p] = \bot$ then $\mathsf{T}[u,p] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{SHACAL\text{-}2.ol}}$
$y_0 \leftarrow \mathsf{T}[u,p]$ ; Return $y_b$

$\mathsf{IC}(sk,x)$   // $|sk| = \mathsf{SHACAL\text{-}2.kl}$, $|x| = \mathsf{SHACAL\text{-}2.ol}$
If $sk[0:256] \in \{mk_{\mathcal{I}}, mk_{\mathcal{R}}\}$ then
    $\mathsf{bad}_1 \leftarrow \mathsf{true}$
    Return $\bot$                                               // $G_3$
If $\mathsf{P}[sk] = \bot$ then $\mathsf{P}[sk] \leftarrow\!\!\$\ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$ ; Return $\pi(x)$

$\mathsf{IC}^{-1}(sk,y)$   // $|sk| = \mathsf{SHACAL\text{-}2.kl}$, $|y| = \mathsf{SHACAL\text{-}2.ol}$
If $sk[0:256] \in \{mk_{\mathcal{I}}, mk_{\mathcal{R}}\}$ then
    $\mathsf{bad}_1 \leftarrow \mathsf{true}$
    Return $\bot$                                               // $G_3$
If $\mathsf{P}[sk] = \bot$ then $\mathsf{P}[sk] \leftarrow\!\!\$\ \mathcal{P}(\mathsf{SHACAL\text{-}2.ol})$
$\pi \leftarrow \mathsf{P}[sk]$ ; Return $\pi^{-1}(y)$
```

Figure 83. Games G_0–G_3 for the proof of Proposition 13. The code added by expanding the related-key-deriving function ϕ_{MAC} in game G_0 is highlighted in grey. The code added for the transitions between games is highlighted in green.

This proof uses games G_0–G_3 in Fig. 83. Game G_0 is equivalent to game $G^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}},\mathcal{D}}$ in the ideal cipher model, so

$$\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr[G_0] - 1.$$

For the transition from G_0 to G_1 we upper-bound the probability of $mk_{\mathcal{I}} = mk_{\mathcal{R}}$ as follows:

$$\Pr[G_0] - \Pr[G_1] \le \Pr[\mathsf{bad}_0^{G_0}] \le \Pr[mk_{\mathcal{I}} = mk_{\mathcal{R}}] = 2^{-256}.$$

Game G_2 differs from game G_1 by expanding the code of the oracle IC in place of the corresponding call to $\mathsf{IC}(sk, \mathsf{IV}_{256})$ inside the RoR oracle, so both games are equivalent:

$$\Pr[G_1] - \Pr[G_2] = 0.$$

For the transition from G_2 to G_3 we upper-bound the probability that adversary $\mathcal{D}$ calls one of its ideal cipher oracles IC or IC^{-1} with a block cipher key sk that contains $mk_{\mathcal{I}}$ or $mk_{\mathcal{R}}$ as its prefix:

$$\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \leq \Pr[\mathsf{bad}_1^{\mathrm{G}_2}] \leq q \cdot \frac{2 \cdot 2^{256}}{2^{512}} = q \cdot 2^{-255}.$$

In game G_3 the ideal cipher oracles IC and IC^{-1} no longer work with any keys that might be used inside oracle RoR. So the adversary $\mathcal{D}$ cannot distinguish between an evaluation of a random permutation on input IV_{256} and a uniformly random output value from the range of such permutation. We have

$$\Pr[\mathrm{G}_3] = \frac{1}{2}.$$

We now combine all of the above steps:

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{hrkprf}}_{\mathsf{SHACAL\text{-}2},\phi_{\mathsf{MAC}}}(\mathcal{D}) = 2 \cdot \Pr[\mathrm{G}_0] - 1 \\
&\quad = 2 \cdot \left(\sum_{0 \leq i \leq 2} (\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}]) + \Pr[\mathrm{G}_3] \right) - 1 \\
&\quad \leq 2 \cdot (2^{-256} + 0 + q \cdot 2^{-255} + \frac{1}{2}) - 1.
\end{aligned}$$

The inequality follows. □

G Implementation

G.1 Code for the attack in Section 6

Assume Telegram desktop version 2.4.11.[53] 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`[54] 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>
#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
```

[53] https://github.com/telegramdesktop/tdesktop/tree/v2.4.11

[54] https://www.ecrypt.eu.org/ebats/cpucycles.html

```
    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();
}
```

G.2 Code for the attack in Section 7

```
#!/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:
```