Analysis of the Telegram Key Exchange[⋆]

Martin R. Albrecht[1], Lenka Mareková[2], Kenneth G. Paterson[2], Eyal Ronen[3], and Igors Stepanovs[4]

[1] King's College London
martin.albrecht@kcl.ac.uk
[2] Applied Cryptography Group, ETH Zurich
{lenka.marekova,kenny.paterson}@inf.ethz.ch
[3] Tel-Aviv University
eyalronen@tauex.tau.ac.il
[4] Amazon[⋆⋆]
igors.stepanovs@gmail.com

10 March 2025

Abstract. We describe, formally model, and prove the security of Telegram's key exchange protocols for client-server communications. To achieve this, we develop a suitable multi-stage key exchange security model along with pseudocode descriptions of the Telegram protocols that are based on analysis of Telegram's specifications and client source code. We carefully document how our descriptions differ from reality and justify our modelling choices. Our security proofs reduce the security of the protocols to that of their cryptographic building blocks, but the subsequent analysis of those building blocks requires the introduction of a number of novel security assumptions, reflecting many design decisions made by Telegram that are suboptimal from the perspective of formal analysis. Along the way, we provide a proof of IND-CCA security for the variant of RSA-OEAP+ used in Telegram and identify a hypothetical attack exploiting current Telegram server behaviour (which is not captured in our protocol descriptions). Finally, we reflect on the broader lessons about protocol design that can be taken from our work.

[⋆] This is the full version of a work that appeared at EUROCRYPT 2025.
[⋆⋆] This work was conducted outside of the author's employment at Amazon and is unrelated to their position at Amazon.

Table of Contents

1 Introduction

Telegram is a chat protocol, service and application suite. It enables users to exchange one-to-one and group messages, features public broadcast channels, permits voice and video calls, public livestreams and file sharing. As of July 2024, Telegram reportedly had 950 million monthly active users. For chat, Telegram offers two modes: "cloud chats", which provide encryption between clients and the Telegram servers; and "secret chats", offering end-to-end encryption between clients. The latter is optional and not available for group chats. Prior research established that "secret chats" play a minor role, even in heightened-risk settings such as among protesters [ABJM21]. For cloud chats, Telegram uses its own bespoke MTProto 2.0 protocol to secure the connection between the Telegram servers and clients.[5] Note that the decision to use a bespoke protocol instead of a standardised one like TLS (which would be ideally suited to the task of protecting client-server communications) is not unusual in the secure messenger world. For example, WhatsApp uses a Noise protocol for client-server connections, while Threema, too, employs its own bespoke protocol [PST23].

Prior work. The cryptographic core of Telegram's MTProto 2.0 consists of a key exchange protocol and a secure channel protocol. The amount of attention that has been paid to analysing the security of MTProto 2.0 is small in comparison to its significance and scale of deployment. Indeed, to date, no comprehensive formal security analysis of this protocol has been conducted. In [JO16], an attack against the IND-CCA security of MTProto 1.0 was reported, prompting an update to all parts of the protocol except the key exchange. In [SK17] and [Kob18], vulnerabilities were identified in the Android client and Windows Phone client, respectively, due to improper input validation. In [MV21] MTProto 2.0 was proven secure in a symbolic model, but assuming ideal building blocks and abstracting away all implementation and primitive details. Similarly, [CCD$^+$23] provided a symbolic proof of the key exchange. While it allowed for weaknesses in some underlying hash functions, it did not model, for example, the custom SHA-1-then-IGE-mode-AES-encrypt scheme used in the MTProto 2.0 key exchange protocol.

In [AMPS22], formal theorems concerning the security guarantees of the MTProto 2.0 secure channel protocol were proven (this channel roughly equates to the Record Protocol in TLS). These theorems hold in a computational model, and so do not abstract away building blocks. Indeed, the authors [AMPS22] had to rely on unstudied, somewhat ad-hoc assumptions about Telegram's cryptographic building blocks due to how these are composed in the protocol. Another interpretation of this prior work is that it managed to coalesce Telegram's idiosyncratic design decisions for MTProto 2.0 into two self-contained security assumptions about SHACAL-2, the blockcipher underlying SHA-2, suitable for further cryptanalysis. The same work also reported vulnerabilities in MTProto 2.0 at an implementation and at a protocol level, which have since been mitigated. This highlights that for Telegram, symbolic models with perfect building blocks are insufficient to provide strong assurances and a more detailed analysis is required.

Our contributions. We provide the first comprehensive analysis of the security of Telegram's MTProto 2.0 key exchange protocol in a computational security model, establishing formal security guarantees under clearly stated security assumptions concerning its building blocks.

We faced significant obstacles to attaining security guarantees for MTProto 2.0. In short, and in contrast to a protocol like TLS 1.3, the protocol was not defined with formal security analysis in mind. Moreover, the protocol lacks a complete specification of client and server behaviour (meaning that we must infer some intended behaviour from client source code and infer as best we can from direct observation some of the server behaviour). The protocol is also complicated, involving two different but related sub-protocols, which we denote as $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$, both of them multi-stage (in the sense

[5] Telegram's web client, Telegram Web Z, uses TLS because it relies on WebSockets [vAP23].

of [FG14, FG17, DFGS21, DDGJ22]), with key material established in one being carried forward into the other.

On the surface, these two sub-protocols do look quite simple and therefore amenable to analysis: for both, in a first stage, RSA encryption is used to transport a session key from the client to the server and, in a second stage, this session key is used to protect a Diffie-Hellman (DH) key exchange in a prime field; the final session key is made from the shared DH value. $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ has a third stage in which a previously established session key is used to authenticate the client to the server over an MTProto 2.0 secure channel.

However, the protocols rely on non-standard building blocks, such as a custom variant of the RSA-OAEP+ scheme [Sho02] lacking proper domain separation and relying on IGE mode internally, yet whose IND-CCA security we needed to establish for our main result. The protocols also rely on a custom hash-then-IGE mode for its integrity properties, but this mode does not achieve standard AEAD security. In addition, the $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ protocol in its final stage encapsulates MTProto 1.0 messages in an instance of the MTProto 2.0 secure channel for the purpose of client authentication, yet the MTProto 1.0 protocol on its own does not immediately provide sufficient integrity for this purpose [JO16]. This demands yet another bespoke analysis. As further examples of features that induce complexity in our analysis, the protocols use weak (SHA-1) and truncated hashes, they reuse long-term public keys across protocols, they reuse symmetric keys in different algorithms for multiple purposes within a single protocol run, they have short session identifiers, they use a bespoke algorithm for internal key derivation, and they have complex retry handling to ensure unique session identifiers.

Any one of the above features on its own would present a challenge to completing a formal security analysis. In combination, they add up to a significant barrier. Yet, the real-world significance of Telegram and the attendant need to develop an understanding of its security in the service of those who rely on it necessitates such an analysis. We do not get to choose what and how cryptography is used in Telegram. Rather, we have to analyse what lies in front of us.

Informally, we show that $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ do achieve strong key indistinguishability and authentication properties under suitable, new assumptions. In particular, we show that both sub-protocols achieve forward secrecy for session keys (i.e. session keys remain secure even after compromise of server long-term keys).[6] Moreover, both achieve server authentication while $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ also achieves a form of client authentication based on long-term symmetric keys established in a prior run of $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$. Neither sub-protocol attains post-compromise security, but this was likely not a goal of the designers.

To achieve all of this, in Section 3 we first develop a variant of existing multi-stage key exchange (MSKE) security models. Our model is somewhat tailored to the analysis of MTProto 2.0; see the main body for details and rationale. As usual, the model gives the adversary access to various oracles for setting up keys, running sessions with parties, corrupting them to obtain their long-term keys, and revealing their session keys. A test oracle gives the adversary either real or random session keys; the adversary wins if it can successfully distinguish which it is given. Trivial wins are ruled out through carefully constructed predicates operating on session states. Our model is multi-stage, meaning that it caters to protocols that execute in different stages, with potentially different session key security and authentication properties being established at each stage. Our model is rich enough to model authentication via public keys (as in $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$) and via long-term symmetric keys (as in $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$), as well as handling stage session keys which may be non-forward secure (in stage 1 of both sub-protocols) or forward secure (in stage 2 of both sub-protocols).

Using our MSKE model, in Section 4 we give a description of MTProto 2.0's two sub-protocols $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$, derived both from the Telegram specification and from inspecting code of official

[6] Even here we must be circumspect, because session keys can be long-lived, with a client and server typically running the key exchange protocol to establish a new session key only once every 24 hours. This is in contrast to secure messengers like Signal that derive a new forward-secure session key for every single message and brings into question the claim made by Telegram that their protocol offers "Perfect Forward Secrecy", see `https://core.telegram.org/api/pfs`.

Telegram implementations. We strike a balance between capturing all the cryptographically relevant features and making the description too complex to work with. We do not omit any cryptographic algorithms or features involved, but we do abstract away some of the low-level message encoding format. We discuss how the on-the-wire protocol differs from our model, and describe a hypothetical attack against MTProto 2.0 that arose because Telegram servers failed to check the freshness of certain timestamps. After we disclosed it to Telegram, this behaviour was fixed.

After this, in Section 5 we state our theorems on the MSKE security of $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$, with proofs in Appendix H. The theorems are stated in terms of high-level security properties of the involved cryptographic components. Our proof that Telegram's RSA-OAEP+ variant achieves IND-CCA security can be found in Appendix D. In Appendices E to G, we show how the other high-level properties can be reduced to low-level properties of the involved primitives. Because of the above-mentioned features of the protocols, some of the low-level properties we need to rely on for our proofs are unusual or non-standard in themselves. For example, we must make certain "partially known key" security assumptions about SHACAL-1, the block cipher underlying SHA-1. We must also rely on restricted forms of second-preimage resistance for functions built upon SHA-1, and make an assumption that key reuse of these functions is not detectable by an adversary. We have tried to minimise the number of new assumptions we make; they are defined in Appendix C. They may eventually be invalidated by cryptanalysis, but we stress that such progress may not directly yield attacks on Telegram; rather, they are what is needed for our current proofs to go through. We leave it as a challenge for future work to develop proofs under milder assumptions. A similar situation can be observed in the analysis of MTProto 2.0's secure channel in [AMPS22]. We also rely on two results from prior work that involve asymptotic reductions or statements: a reduction from the discrete logarithm with short exponent (DLSE) assumption from [KK04] to the "short exponent" assumption in prime fields, that is used in our main proofs, and a result about the security of the Even-Mansour cipher from [EM97] that we use to prove a high-level integrity property.

Appendix I highlights specific features of MTProto 2.0 that hindered our analysis, and draws broader lessons for protocol designers. Our core message there is that modularity of design and breaking dependencies between components is essential in making the task of analysis tractable for cryptographers. So these features should be primary, not secondary, in any protocol design.

2 Preliminaries

2.1 Notation

Concatenation is denoted by $\|$, the zero byte by 00 and the empty string by ε. Let x be a bit string of length $|x| = \ell$ bits. Then $x[i]$ is the i-th bit of x, where $0 \le i < \ell$, and $x[i:j] := x[i] \| \ldots \| x[j-1]$ is the bit-slice of x from i to j, where $0 \le i < j \le \ell$. If x is a byte string, i.e. $|x| = 8 \cdot n$ for some $n \in \mathbb{N}$, we use $\mathsf{len}(x)$ to refer to the 32-bit big-endian encoding of its byte length n. We use "LSBs" and "MSBs" to refer to least and most significant bits respectively. Constant byte strings are shown in hexadecimal. Let $\mathbb{G}_{i:j} := \{h[i:j] \mid h \in \mathbb{G}\}$ for a group $\mathbb{G}$, where $h[i:j]$ is interpreted as a string. We use the shorthand $(x)^\ell$ to refer to the tuple $(x, \ldots, x)$ of length ℓ. We use $\hat{+}$ as the addition operator over 32-bit words. We use "**find** $x \in \mathcal{L}$ **s.t.** *condition*", which searches the list $\mathcal{L}$ for a unique element x satisfying *condition* and returns x or makes its calling query return $\perp$. We assume that modifying the found element updates $\mathcal{L}$.

The serialisation of an API request `req` with parameters $x, y, \ldots$ using the TL schema [Tel22f] is denoted by $\mathsf{TL}(\mathtt{req}, x, y, \ldots)$. From now on, when we refer to MTProto, we mean MTProto 2.0 unless indicated otherwise. In a two-party protocol execution, we denote one party as the *initiator*, representing the client, and the other party as the *responder*, representing the server.

2.2 Standard definitions

Lemma 1 (Fundamental Lemma of Game Playing [BR06]). *Let* $\Pr[\mathsf{bad}^{\mathrm{G}}]$ *denote the probability of setting flag* bad *in game* G. *For any two games* G_i, G_{i+1} *that are identical until* bad, *we have* $\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}] \le \beta$ *where* $\beta = \Pr[\mathsf{bad}^{\mathrm{G}_i}] = \Pr[\mathsf{bad}^{\mathrm{G}_{i+1}}]$.

Definition 1 (Short exponent problem [KK04]). *Let $\mathbb{G}$ be a group of prime order q with a generator g. Let $n = |q|$ and c be such that $\log(n) < c < n$. Let $\mathsf{D}_0 = \{(g, g^x) \mid x \leftarrow\!\!\$ \{0, ..., q-1\}\}$, $\mathsf{D}_{n-c} = \{(g, g^z) \mid z \leftarrow 2^{n-c} \cdot u, u \leftarrow\!\!\$ \{0,1\}^c\}$ be distributions. Let* S-EXP *be the problem of distinguishing between D_0 and D_{n-c}. The advantage of an adversary $\mathcal{D}$ in breaking* S-EXP *is defined as* $\mathsf{Adv}^{\text{S-EXP}}_{\mathbb{G},q}(\mathcal{D}) := \Pr[\mathcal{D}(g, W) = 1 \mid (g, W) \leftarrow\!\!\$ \mathsf{D}_{n-c}] - \Pr[\mathcal{D}(g, W) = 1 \mid (g, W) \leftarrow\!\!\$ \mathsf{D}_0]$.

In [KK04, Theorem 1], the discrete logarithm with short exponent (DLSE) problem is reduced to the S-EXP problem.

3 Model

Since no model from previous work allows for expressing MTProto's key exchange fully, we define a new model for the protocol. In doing this, we take inspiration from the multi-stage key exchange (MSKE) models of [FG14, FG17, DFGS21, DDGJ22], which are based on the Bellare-Rogaway family of models [BR94, BFWW11]. We follow [DDGJ22] in using code-based definitions and soundness predicates.

The cited models capture some of the properties of MTProto, e.g. the separation of authentication guarantees between stages, the modelling of internal and external session keys, and the usage of contributive identifiers for honest but unauthenticated partners. We omit those properties that are not relevant for MTProto, e.g. the replayability that was of concern for 0-RTT in TLS 1.3. We further augment the model to allow for both asymmetric and symmetric secrets to be used in different stages of the same protocol run. Note that since the session keys in the stages of MTProto key exchange are not key independent, we cannot implicitly rely on the composition guarantees of the existing MSKE models [Gün18]. Finally, since our aim is to give a security analysis that is as close to the complex reality of MTProto as possible, rather than a fully general treatment, we simplify the model by restricting it in several ways, e.g. we allow for responder-only and mutual but not initiator-only authentication, and we consider corruptions of asymmetric secrets only. We do not model the possibility of side-channel attacks.

3.1 Parameters and syntax

Following [Gün18, DFGS21], we use a vector of properties to parametrise the definition of our model. We use the vector $(\mathsf{M}, \mathsf{TYPE}, \mathsf{AUTH}, \mathsf{TS}, \mathsf{USE}, \mathsf{FS}, \mathsf{KD}, \mathsf{DIST})$, where:

- $\mathsf{M} \in \mathbb{N}$ is the number of stages.
- $\mathsf{TYPE} \subseteq \{\texttt{pub}, \texttt{sym}\}$ denotes the supported types of long-term secrets (public-key keypairs, symmetric keys). We assume that TYPE always includes `pub`.
- $\mathsf{AUTH} \in \{\texttt{none}, \texttt{R-only}, \texttt{mutual}\}^{\mathsf{M}}$ defines the expected authentication property of each stage (unauthenticated, responder-only, mutual authentication).
- $\mathsf{TS} \in \{\texttt{testable}, \texttt{auth-only}\}^{\mathsf{M}}$ defines for each stage whether its session keys can be tested directly. We introduce this parameter to enable the handling of stages that provide authentication but do not meet key indistinguishability.
- $\mathsf{USE} \in \{\texttt{internal}, \texttt{external}, \texttt{none}\}^{\mathsf{M}}$ defines for each stage whether its session key is used within the key exchange protocol itself or only outside of it. Internal stages require adjustments to maintain consistency of session keys, to avoid trivial attacks. We use `none` for stages that are `auth-only` in TS.
- $\mathsf{FS} = i$ where $i \in \{1, \ldots, \mathsf{M}\} \cup \{\infty\}$ defines the stage from which forward secrecy should apply with respect to the corruption of the public-key keypairs.[7] We use ∞ to denote no forward secrecy.

[7] We do not model corruption with respect to the symmetric keys as MTProto only uses them for authentication (see Section 3.2).

– $\mathsf{KD} \in \{\texttt{independent}, \texttt{dependent}\}$ defines if the protocol is key dependent, i.e. if at some stage i the session key is derived using the session key of the previous stage $i-1$ (so revealing the session key of stage $i-1$ before the stage i has finished running would lead to trivial attacks, as in e.g. QUIC [Gün18]). Though we will only use $\mathsf{KD} = \texttt{dependent}$, we surface this as an explicit parameter to make it clear that security for MTProto is only possible in this weaker model.

– $\mathsf{DIST} = (\mathcal{K}^1_{\mathsf{test}}, \ldots, \mathcal{K}^{\mathsf{M}}_{\mathsf{test}})$ defines the session key distributions for each stage.

Though the above properties allow for different combinations of authentication types, our model assumes that responder authentication is via public keys and initiator authentication is via symmetric keys. Table 1 in Section 4 shows the properties we use in our analysis of MTProto.

To define the model, we use the following syntax:

– $\mathcal{U}$ is the set of user identities in the protocol.

– $\star$ represents an unknown user identity.

– $\mathcal{U}_{\mathsf{role}}$ is the set of users along with their intended roles. Each element of $\mathcal{U}_{\mathsf{role}}$ is of the form $U = (\mathsf{user}, \mathsf{role})$, where $\mathsf{user} \in \mathcal{U}$ and $\mathsf{role} \in \{\texttt{I}, \texttt{R}\}$.

– $\mathcal{L}_{\mathsf{S}}$ is the list of sessions s, each of the form

$$s = (\mathsf{label}, \mathsf{uid}, \mathsf{vid}, \mathsf{kid}, \mathsf{stage}, \mathsf{st}_{\mathsf{exec}}, \mathsf{sid}, \mathsf{cid}, \mathsf{kcid}, \mathsf{sskey}, \mathsf{st}_{\mathsf{sskey}}, \mathsf{st}_{\mathsf{KE}}, \mathsf{st}_{\mathsf{test}}, \mathsf{st}_{\mathsf{pause}}),$$

where:

- $\mathsf{label} = (U, V, j) \in \mathcal{U} \times (\mathcal{U} \cup \{\star\}) \times \mathbb{N}$ is a unique administrative label to mark that this is the j-th session owned by U with the intended partner V.
- $\mathsf{uid} \in \mathcal{U}_{\mathsf{role}}$ is the owner of the session.
- $\mathsf{vid} \in \mathcal{U}_{\mathsf{role}} \cup \{(\star, \texttt{I})\}$ is the intended partner of the session, allowing for the identity of the initiator to be unspecified.
- $\mathsf{kid} \in \{0,1\}^* \cup \{\bot\}$ identifies the symmetric key used (if any).
- $\mathsf{stage} \in \{0, \ldots, \mathsf{M}\}$ is the latest accepted stage, default value: 0.
- $\mathsf{st}_{\mathsf{exec}} \in \{\texttt{running}_i \mid 0 \leq i \leq \mathsf{M}\} \cup \{\texttt{accepted}_i \mid 0 < i \leq \mathsf{M}\} \cup \{\texttt{rejected}_i \mid 0 < i \leq \mathsf{M}\}$ is the state of the execution, default value: $\texttt{running}_0$.
- $\mathsf{sid} \in (\{0,1\}^* \cup \{\bot\})^{\mathsf{M}}$ defines the session identifier agreed upon acceptance in each stage, default value for each item: $\bot$.
- $\mathsf{cid} \in (\{0,1\}^* \cup \{\bot\})^{\mathsf{M}}$ defines the contributive identifier of each stage, which may be set several times before acceptance, default value for each item: $\bot$.
- $\mathsf{kcid} \in (\{0,1\}^* \cup \{\bot\})^{\mathsf{M}}$ defines the key confirmation identifier that is set before acceptance in each stage, default value for each item: $\bot$.
- $\mathsf{sskey} \in (\{0,1\}^* \cup \{\bot\})^{\mathsf{M}}$ defines the session key output by each stage upon acceptance, default value for each item: $\bot$.
- $\mathsf{st}_{\mathsf{sskey}} \in \{\texttt{fresh}, \texttt{revealed}\}^{\mathsf{M}}$ is the state of the session key in each stage, default value for each item: $\texttt{fresh}$.
- $\mathsf{st}_{\mathsf{KE}}$ is the session state for the session, default value: ε.
- $\mathsf{st}_{\mathsf{test}} \in \{\texttt{false}, \texttt{true}\}^{\mathsf{M}}$ indicates whether the session was tested by the adversary in each stage, default value for each item: $\texttt{false}$.
- $\mathsf{st}_{\mathsf{pause}} \in \{\texttt{false}, \texttt{true}\}$ tracks if the execution of the session was paused after one stage accepted and before moving to the next stage, default value: $\texttt{false}$.

– $\mathcal{L}_{\mathsf{K}}$ is the list of long-term secrets of the form $k = (\mathsf{type}, \mathsf{id}, \mathsf{key}, \mathsf{st}_{\mathsf{key}})$, where:

  - $\mathsf{type} \in \{\mathtt{pub}, \mathtt{sym}\}$ denotes key type (public-key keypair or a symmetric key).
  - If $\mathsf{type} = \mathtt{pub}$, then $\mathsf{id} \in \mathcal{U}_{\mathsf{role}}$ is the owner of the keypair. If $\mathsf{type} = \mathtt{sym}$, then id is a unique identifier of the form (V, kid) where $V \in \mathcal{U}_{\mathsf{role}}$ and $kid \in \{0,1\}^*$.
  - $\mathsf{key} = (pk, sk)$ if $\mathsf{type} = \mathtt{pub}$, and $\mathsf{key} = key$ if $\mathsf{type} = \mathtt{sym}$.
  - $\mathsf{st}_{\mathsf{key}} \in \{\mathtt{honest}, \mathtt{corrupted}\}$ is the state of the key/keypair, default: $\mathtt{honest}$.

– $b_{\mathsf{test}} \in \{0,1\}$ is the random challenge bit generated by the game.

– $lost \in \{\mathtt{false}, \mathtt{true}\}$ indicates whether the adversary has lost the game.

– $\mathcal{C}_{\mathsf{pub}} \subseteq \mathcal{U}_{\mathsf{role}}$ is the set of users whose private keys sk have been corrupted.

For $s \in \mathcal{L}_{\mathsf{S}}$, we use e.g. $s.\mathsf{uid}$ to denote the value of uid in the s tuple, and $uid \leftarrow s.\mathsf{uid}$ to distinguish tuple identifiers from variables. We use e.g. $s.\mathsf{sid}.i$ to refer to the value of sid for the i-th stage. We assume that $\mathcal{U}_{\mathsf{role}}$ is generated externally (formally, the game takes $\mathcal{U}_{\mathsf{role}}$ as input before initialising; see Fig. 3). During game setup, $\mathcal{L}_{\mathsf{S}}$ is initialised as an empty list and $\mathcal{L}_{\mathsf{K}}$ is populated to contain newly-generated public-key keypairs for all $V \in \mathcal{U}_{\mathsf{role}}$ such that $V.\mathsf{role} = \mathtt{R}$ (assuming at most one keypair for each V), while the symmetric keys can be generated during the game by the adversary using the NewSecret query.

Key exchange protocol. We define KE as a triple of algorithms $\mathsf{KE.KGen}$, $\mathsf{KE.Init}$ and $\mathsf{KE.Run}$. More precisely, we use $\mathsf{KE.KGen}_{\mathsf{pub}}$ and $\mathsf{KE.KGen}_{\mathsf{sym}}$ to refer to the long-term key generation algorithms for the public-key keypairs and symmetric keys, respectively. Due to quirks of MTProto, $\mathsf{KE.KGen}_{\mathsf{sym}}$ takes a user identity as input. Let $s \in \mathcal{L}_{\mathsf{S}}$ and $m \in \{0,1\}^*$. $\mathsf{KE.Init}(s)$ initiates a session s such that $s.\mathsf{uid.role} = \mathtt{I}$ and $s.\mathsf{st}_{\mathsf{exec}} = \mathtt{running}_0$ (responder sessions are initiated when they receive the first protocol message). $\mathsf{KE.Run}(s, m)$ then executes the protocol for the session s, processing the received protocol message m. We use $\mathsf{KE.Run}(s)$ to express continuing a paused execution (i.e. one that is still processing some previously received m; we explain this mechanism in more detail in the next subsection). We assume that whenever KE sets $s.\mathsf{st}_{\mathsf{exec}} = \mathtt{accepted}_i$ for some i, it sets $s.\mathsf{sid}.i, s.\mathsf{sskey}.i$ to non-$\perp$ values and $s.\mathsf{stage}$ to i. Note that KE itself is responsible for setting its session's $s.\mathsf{kid}$ to the id of the used symmetric key, as well as for setting its session's $s.\mathsf{vid}$ if it was previously unset.

In addition, we use the shorthand

$$\begin{aligned}\mathsf{Running}(s) &:= \exists i \in \{0,\dots,\mathsf{M}\} \colon s.\mathsf{st}_{\mathsf{exec}} = \mathtt{running}_i\\ \mathsf{Accepted}(s) &:= \exists i \in \{1,\dots,\mathsf{M}\} \colon s.\mathsf{st}_{\mathsf{exec}} = \mathtt{accepted}_i\\ \mathsf{Rejected}(s) &:= \exists i \in \{1,\dots,\mathsf{M}\} \colon s.\mathsf{st}_{\mathsf{exec}} = \mathtt{rejected}_i\end{aligned}$$

to denote that the session $s \in \mathcal{L}_{\mathsf{S}}$ is running/accepted/rejected in some stage.

Partnering. We follow a standard definition for partnered sessions from [BFWW11] that depends upon a jointly-agreed session identifier.

Definition 2 (Partnering). *Sessions $s, s' \in \mathcal{L}_{\mathsf{S}}$ such that $s \neq s'$ are defined to be* partnered *if and only if $s.\mathsf{sid} = s'.\mathsf{sid} \neq \perp$. We use the following abstractions:*

$$\begin{aligned}\exists\mathsf{Partner}(s, i, \mathsf{rule}) &:= \exists s' \in \mathcal{L}_{\mathsf{S}} \colon (s \neq s' \wedge s.\mathsf{sid}.i = s'.\mathsf{sid}.i \neq \perp \wedge \mathsf{rule}(s'))\\ \mathsf{Partners}(s, i, \mathsf{rule}) &:= \{s' \mid s' \in \mathcal{L}_{\mathsf{S}} \wedge s.\mathsf{sid}.i = s'.\mathsf{sid}.i \neq \perp \wedge \mathsf{rule}(s')\}\\ \exists\mathsf{Partners}(\mathsf{rule}) &:= \exists s, s' \in \mathcal{L}_{\mathsf{S}}, i \in \{1,\dots,\mathsf{M}\} \colon\\ &\qquad (s \neq s' \wedge s.\mathsf{sid}.i = s'.\mathsf{sid}.i \neq \perp \wedge \mathsf{rule}(s, s', i)).\end{aligned}$$

where rule *is an additional condition for each specific context. If this is not needed, we define:*

$$\exists\mathsf{Partner}(s,i) := \exists\mathsf{Partner}(s,i,\mathtt{true})$$
$$\mathsf{Partners}(s,i) := \mathsf{Partners}(s,i,\mathtt{true}).$$

Note that $s \in \mathsf{Partners}(s,i)$ *is always true by definition. We can thus define*

$$\nexists\mathsf{Partner}(s,i) := |\mathsf{Partners}(s,i)| = 1.$$

3.2 Adversarial queries

In our games, an adversary $\mathcal{A}$ interacts with the protocol using the oracle queries given in Figs. 1 to 2. Below, we describe the queries in more detail:

```
NEWSECRET($U, V$)    // $U, V \in \mathcal{U}_{\mathsf{role}}, U.\mathsf{role} = \mathtt{I}, V.\mathsf{role} = \mathtt{R}$
if $\mathtt{sym} \in \mathsf{TYPE}$:
    $(kid, key) \leftarrow\$ \mathsf{KE.KGen}_{\mathsf{sym}}(V)$
    append $(\mathtt{sym}, (V, kid), key, \mathtt{honest})$ to $\mathcal{L}_{\mathsf{K}}$
    for each $s \in \mathcal{L}_{\mathsf{S}}$ s.t. $s.\mathsf{uid} = U \wedge s.\mathsf{kid} = \bot$: $s.\mathsf{st}_{\mathsf{KE}} \leftarrow (s.\mathsf{st}_{\mathsf{KE}}, (V, kid, key))$
    for each $s \in \mathcal{L}_{\mathsf{S}}$ s.t. $s.\mathsf{uid} = V \wedge s.\mathsf{kid} = \bot$: $s.\mathsf{st}_{\mathsf{KE}} \leftarrow (s.\mathsf{st}_{\mathsf{KE}}, (U, kid, key))$
    return $kid$
return $\bot$
```

Fig. 1. NEWSECRET query for $\mathcal{A}$.

NEWSECRET(U, V) is only relevant for protocols that use symmetric keys as long-term secrets. It generates a new symmetric key shared between an initiator U and a responder V, and associates it to a globally-unique key identifier. Due to quirks of MTProto, this key identifier consists of two parts, the identity of the responder V and a key identifier *kid* that is only required to be unique with respect to a given V and which is generated using $\mathsf{KE.KGen}_{\mathsf{sym}}$.[8] $\mathcal{A}$ must call NEWSECRET before the first stage that requires the use of symmetric keys.

NEWSESSION(U, V) creates a new session with a unique label that is owned by the user U and has the intended partner V. If the identity of the intended partner is unknown, we set $V = (\star, \mathtt{I})$.[9] The helper function $\mathsf{GetPubKeys}(U, V)$ ensures that the correct keys are included as part of the initial state of the key exchange protocol: a session owned by a responder obtains its own keypair, while a session owned by an initiator gets the public key of its intended partner.

CORRUPT(U) adds the user U to the set of corrupted users and reveals the user's long-term public-key keypair to $\mathcal{A}$. We model forward secrecy but not post-compromise security. To capture the effect of corruption in the key-dependent setting we use the helper function KDReveal, which is also used by the REVEAL query below. Note that the adversary is not allowed to compromise the session state itself or its randomness as, similar to TLS, MTProto is not expected to be secure in that scenario. We do not model corruption of long-term symmetric keys because in MTProto, they are only used for authentication.[10]

REVEAL$(label, i)$ allows $\mathcal{A}$ to reveal the session key for the accepted and testable stage i of the session with label *label*. In the case of key dependence, KDReveal also marks all future stages of the session and its partners as revealed.

[8] This is true to practice: a server is expected to maintain a mapping from key identifiers to user identities and the corresponding keys.

[9] We only allow unknown initiators, as MTProto clients can always identify the servers.

[10] Corrupting such a key before the relevant stage accepts would have to be excluded due to trivial wins, but corrupting it afterwards could not enable the adversary to learn anything about past session keys.

```
$\textsc{NewSession}(U, V)$
$\mathbin{/\!/}\ U \in \mathcal{U}_{\text{role}}, V \in \mathcal{U}_{\text{role}} \cup \{(\star, \mathtt{I})\}$
$pbk \leftarrow \mathsf{GetPubKeys}(U, V)$
$j \leftarrow \max(\{j' \mid j' \in \mathbb{N} \wedge s \in \mathcal{L}_\mathsf{S}$
    $\wedge s.\mathsf{label} = (U.\mathsf{user}, V.\mathsf{user}, j')\})$
$label \leftarrow (U.\mathsf{user}, V.\mathsf{user}, j+1)\ ;\ kid \leftarrow \bot$
$i \leftarrow 0\ ;\ st_{\text{exec}} \leftarrow \mathtt{running}_0\ ;\ st_{\mathsf{KE}} \leftarrow pbk\ ;\ sid \leftarrow (\bot)^{\mathsf{M}}$
$cid \leftarrow (\bot)^{\mathsf{M}}\ ;\ kcid \leftarrow (\bot)^{\mathsf{M}}\ ;\ sskey \leftarrow (\bot)^{\mathsf{M}}$
$st_{\text{sskey}} \leftarrow (\mathtt{fresh})^{\mathsf{M}}\ ;\ st_{\text{test}} \leftarrow (\mathtt{false})^{\mathsf{M}}\ ;\ st_{\text{pause}} \leftarrow \mathtt{false}$
append $(label, U, V, kid, i, st_{\text{exec}}, sid, cid, kcid,$
  $sskey, st_{\text{sskey}}, st_{\mathsf{KE}}, st_{\text{test}}, st_{\text{pause}})$ to $\mathcal{L}_\mathsf{S}$
return $label$
```

```
$\mathsf{GetPubKeys}(U, V)$
if $U.\mathsf{role} = \mathtt{R}$ : $uid \leftarrow U$ else : $uid \leftarrow V$
$k_{\text{pub}} \leftarrow$ find $k \in \mathcal{L}_\mathsf{K}$ s.t. $k.\mathsf{type} = \mathsf{pub} \wedge k.\mathsf{id} = uid$
$(pk, sk) \leftarrow k_{\text{pub}}.\mathsf{key}$
if $U.\mathsf{role} = \mathtt{R}$ : $pbk \leftarrow (pk, sk)$ else : $pbk \leftarrow pk$
return $pbk$
```

```
$\textsc{Corrupt}(U)$
$\mathbin{/\!/}\ U \in \mathcal{U}_{\text{role}}$
$k_{\text{pub}} \leftarrow$ find $k \in \mathcal{L}_\mathsf{K}$ s.t. $k.\mathsf{type} = \mathsf{pub} \wedge k.\mathsf{id} = U$
$k_{\text{pub}}.\mathsf{st}_{\text{key}} \leftarrow \mathtt{corrupted}$
$\mathcal{C}_{\text{pub}} \leftarrow \mathcal{C}_{\text{pub}} \cup \{U\}$
$\mathcal{S} \leftarrow \{s \mid s \in \mathcal{L}_\mathsf{S} \wedge (s.\mathsf{uid} = U \vee s.\mathsf{vid} = U)\}$
for each $s \in \mathcal{S}$ : for each $i \in \{1, \ldots, \mathsf{M}\}$ :
  if $(\mathsf{AUTH}.i = \mathtt{R\text{-}only}) \wedge ((i < \mathsf{FS}) \vee (i > s.\mathsf{stage}))$ :
    $s.\mathsf{st}_{\text{sskey}}.i \leftarrow \mathtt{revealed}$ ; $\mathsf{KDReveal}(s, i)$
return $k_{\text{pub}}.\mathsf{key}$
```

```
$\textsc{Reveal}(label, i)$
$\mathbin{/\!/}\ i \in \{1, \ldots, \mathsf{M}\}$
$s \leftarrow$ find $s \in \mathcal{L}_\mathsf{S}$ s.t. $s.\mathsf{label} = label$
if $\mathsf{TS}.i \neq \mathtt{testable} \vee s.\mathsf{stage} < i$ : return $\bot$
$s.\mathsf{st}_{\text{sskey}}.i \leftarrow \mathtt{revealed}$
$\mathcal{S} \leftarrow \mathsf{Partners}(s, i)$
for each $s' \in \mathcal{S}$ : $\mathsf{KDReveal}(s', i)$
return $s.\mathsf{sskey}.i$
```

```
$\mathsf{KDReveal}(s, i)$
$\mathbin{/\!/}\ s \in \mathcal{L}_\mathsf{S}, i \in \{1, \ldots, \mathsf{M}\}$
if $\mathsf{KD} = \mathtt{dependent} \wedge s.\mathsf{stage} \leq i$ :
  for each $j \in \{i+1, \ldots, \mathsf{M}\}$ : $s.\mathsf{st}_{\text{sskey}}.j \leftarrow \mathtt{revealed}$
```

```
$\textsc{Send}(label, m)$
$s \leftarrow$ find $s \in \mathcal{L}_\mathsf{S}$ s.t. $s.\mathsf{label} = label$
if $m = \mathtt{init} \wedge s.\mathsf{uid}.\mathsf{role} = \mathtt{I} \wedge s.\mathsf{st}_{\text{exec}} = \mathtt{running}_0$ :
  $m_{\text{resp}} \leftarrow \mathsf{KE.Init}(s)$
elseif $\mathsf{Running}(s) \vee \mathsf{Accepted}(s)$ :
  $m_{\text{resp}} \leftarrow \mathsf{Handle}(\mathsf{KE.Run}(s, m))$
else : return $\bot$
return $m_{\text{resp}}, s.\mathsf{st}_{\text{exec}}$
```

```
$\mathsf{Handle}(\mathsf{KE.Run}(s, m))$
if $m = \mathtt{continue} \wedge s.\mathsf{st}_{\text{pause}} = \mathtt{true}$ :
  // continue execution of previous run
  $m_{\text{resp}} \leftarrow \mathsf{KE.Run}(s)$ ; $s.\mathsf{st}_{\text{pause}} \leftarrow \mathtt{false}$
else : run $\mathsf{KE.Run}(s, m)$ until
      $\mathsf{Accepted}(s)$ or $m_{\text{resp}} \leftarrow \mathsf{KE.Run}(s, m)$
  // pause execution when $s$ accepts
  if $\mathsf{Accepted}(s)$ :
    $i \leftarrow s.\mathsf{stage}$
    if $(\mathsf{AUTH}.i = \mathtt{R\text{-}only} \wedge (s.\mathsf{uid} \in \mathcal{C}_{\text{pub}} \vee s.\mathsf{vid} \in \mathcal{C}_{\text{pub}}))$
      $\vee\ \exists \mathsf{Partner}(s, i, \mathsf{st}_{\text{sskey}}.i = \mathtt{revealed})$ :
        $s.\mathsf{st}_{\text{sskey}}.i \leftarrow \mathtt{revealed}$ ; $\mathsf{KDReveal}(s, i)$
    $\mathcal{S} \leftarrow \mathsf{Partners}(s, i, \mathsf{st}_{\text{test}}.i = \mathtt{true})$
    if $\mathcal{S} \neq \emptyset$ :
      $s.\mathsf{st}_{\text{test}}.i \leftarrow \mathtt{true}$
      for each $s' \in \mathcal{S}$ :
        if $\mathsf{USE}.i = \mathtt{internal}$ : $s.\mathsf{sskey}.i \leftarrow s'.\mathsf{sskey}.i$
    $s.\mathsf{st}_{\text{pause}} \leftarrow \mathtt{true}$
    return $\mathtt{paused}$
return $m_{\text{resp}}$
```

```
$\textsc{Test}(label, i)$
$s \leftarrow$ find $s \in \mathcal{L}_\mathsf{S}$ s.t. $s.\mathsf{label} = label$
if $\mathsf{TS}.i \neq \mathtt{testable} \vee s.\mathsf{st}_{\text{exec}} \neq \mathtt{accepted}_i \vee s.\mathsf{st}_{\text{test}}.i = \mathtt{true}$ :
  return $\bot$
if $\mathsf{USE}.i = \mathtt{internal} \wedge \exists \mathsf{Partner}(s, i, \mathsf{st}_{\text{exec}} \neq \mathtt{accepted}_i)$ :
  // after key was used internally by a partner
  $lost \leftarrow \mathtt{true}$
if $\mathsf{AUTH}.i = \mathtt{R\text{-}only} \wedge s.\mathsf{uid}.\mathsf{role} = \mathtt{R}$
     $\wedge\ (\nexists s' \in \mathcal{L}_\mathsf{S} : s' \neq s \wedge s'.\mathsf{cid}.i = s.\mathsf{cid}.i)$ :
  // a responder w/o honest contributive partner
  $lost \leftarrow \mathtt{true}$
$\mathcal{S} \leftarrow \mathsf{Partners}(s, i, \mathsf{st}_{\text{exec}} = \mathtt{accepted}_i)$
for each $s' \in \mathcal{S}$ :
  $s'.\mathsf{st}_{\text{test}}.i \leftarrow \mathtt{true}$ ; $\mathsf{KDReveal}(s', i)$
if $b_{\text{test}} = 0$ :
  $sskey \leftarrow\!\!\$\ \mathcal{K}^i_{\text{test}}$
  if $\mathsf{USE}.i = \mathtt{internal}$ :
    for each $s' \in \mathcal{S}$ : $s'.\mathsf{sskey}.i \leftarrow sskey$
else : $sskey \leftarrow s.\mathsf{sskey}.i$
return $sskey$
```

Fig. 2. NEWSESSION, CORRUPT, REVEAL, SEND and TEST queries for $\mathcal{A}$.

$\textsc{Send}(\mathit{label}, m)$ lets $\mathcal{A}$ pass a network message m to the running session with label *label*. If m is the special message `init`, it initiates a run of the key exchange protocol with $\mathsf{KE.Init}$. Otherwise, it continues running the protocol with $\mathsf{KE.Run}$ which gets the session and the message m as input, and outputs a response message m_{resp} which is given to $\mathcal{A}$ along with the updated execution state of the session. The helper function Handle acts as a wrapper on the execution of $\mathsf{KE.Run}$, monitoring any changes to the session's state and pausing the execution when the protocol accepts. This is to allow $\mathcal{A}$ to test an accepted session key before the protocol potentially moves on to another stage: $\mathcal{A}$ is given the special message `paused` as output, and it can resume the execution by calling $\textsc{Send}(\mathit{label}, \texttt{continue})$. Handle performs a number of checks upon acceptance. If it is executing a responder-only authenticated stage and the intended partner is corrupted, or if a partnered session key was previously revealed, the session key of the current session (as well as future sessions in the case of key dependence) is also marked as revealed. Finally, if a partnered session was previously tested, the current session is marked as tested and if in addition the current session key is internal, it is set to match the partnered session key to avoid trivial distinguishing attacks.

$\textsc{Test}(\mathit{label}, i)$ allows $\mathcal{A}$ to test any session which has just accepted stage i, where this stage was not previously marked as tested. Stages not marked as testable in TS cannot be queried, but the adversary may still win if it can break authentication at those stages. There are two conditions which will set the *lost* flag to `true`, thus preventing trivial attacks: first, if the key is internal and the tested session has a partner that has moved on to the next stage, and second, if the tested protocol stage is responder-only authenticated and the tested session is a responder session without an honest contributive partner. $\textsc{Test}$ outputs either a randomly sampled key or the real session key based on the value of the challenge bit b_{test}. If $b_{\mathsf{test}} = 0$ and the session key is internal, the session key of the current session as well as its partnered sessions is replaced with the random key for consistency. Finally, the partnered sessions are also marked as tested. In the case of key dependence, all future stages of the partnered sessions are marked as revealed.[11]

3.3 Security game

Our multi-stage key exchange security game draws inspiration from [BFWW11, DFGS21]. Note that instead of defining a separate game for soundness of session identifiers (referred to as Match-security in other works), we include a soundness predicate within the key indistinguishability game as in [DDGJ22].

Definition 3. *Let $\mathrm{G}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{KE},\mathcal{U}_{\mathsf{role}},\mathcal{A}}$ be as given in Figs. 1 to 4. The advantage of adversary $\mathcal{A}$ in breaking the* $\mathsf{Multi\text{-}Stage}$*-security of the key exchange protocol* KE *with the set of users $\mathcal{U}_{\mathsf{role}}$ is defined as* $\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{KE},\mathcal{U}_{\mathsf{role}}}(\mathcal{A}) := 2 \cdot \Pr[\mathrm{G}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{KE},\mathcal{U}_{\mathsf{role}},\mathcal{A}}] - 1$.

Handling authentication-only stages. Following the terminology of [FGSW16, DFW20], we define predicates that specify when the adversary wins by breaking implicit authentication or key confirmation. We define the predicates as winning conditions rather than specifying the desired security goals. Our definitions differ from those of [FGSW16, DFW20] in three ways. First, we do not include the case where partnered sessions derive a different key as a win, since this is already modelled as part of the soundness predicate. Second, we restrict our implicit authentication predicate to fresh sessions, since MTProto is not secure against key-compromise impersonation attacks.[12] Third, our freshness predicate speaks about both the target session and its partners, but expresses the same idea – the sessions should not have been corrupted prior to acceptance.

[11] This latter step is necessary to prevent trivial attacks where the adversary uses the tested session key to perform a MitM attack on the following stage. The original key-dependent MSKE model of [FG14, Gün18] is missing this step.

[12] An attacker can always impersonate clients to a corrupted server in stage 3, though such a scenario is more contrived than the setting of TLS where malicious client certificates can be installed [HGFS15].

```
$G^{\text{Multi-Stage}}_{\text{KE},\mathcal{U}_{\text{role}},\mathcal{A}}$
$\mathcal{L}_\mathsf{K}, \mathcal{K}_{\text{pub}} \leftarrow \mathsf{Init}(\mathcal{U}_{\text{role}})$
$\mathcal{L}_\mathsf{S} \leftarrow [\,]$ ; $\mathcal{C}_{\text{pub}} \leftarrow \emptyset$ ; $lost \leftarrow \mathtt{false}$
$b_{\text{test}} \leftarrow\$ \{0,1\}$
$b'_{\text{test}} \leftarrow \mathcal{A}^{\text{NewSecret},\ldots,\text{Test}}(\mathcal{K}_{\text{pub}})$
if $\neg\mathsf{Sound}$ : return 1
if $\neg\mathsf{Auth}$ : return 1
if $(\exists s, s' \in \mathcal{L}_\mathsf{S}, i \in \{1,\ldots,\mathsf{M}\}$ :
    $s.\mathsf{sid}.i = s'.\mathsf{sid}.i$
    $\wedge\, s.\mathsf{st}_{\mathsf{sskey}}.i = \mathtt{revealed}$
    $\wedge\, s'.\mathsf{st}_{\mathsf{test}}.i = \mathtt{true})$ :
      $lost \leftarrow \mathtt{true}$
return $b'_{\text{test}} = b_{\text{test}} \wedge lost = \mathtt{false}$
```

```
$\mathsf{Init}(\mathcal{U}_{\text{role}})$
$\mathcal{L}_\mathsf{K} \leftarrow [\,]$ ; $\mathcal{K}_{\text{pub}} \leftarrow \emptyset$
for each $V \in \mathcal{U}_{\text{role}}$ s.t. $V.\mathsf{role} = \mathtt{R}$ :
    $(pk, sk) \leftarrow\$ \mathsf{KE.KGen}_{\text{pub}}()$
    append $(\mathtt{pub}, V, (pk, sk), \mathtt{honest})$ to $\mathcal{L}_\mathsf{K}$
    $\mathcal{K}_{\text{pub}} \leftarrow \mathcal{K}_{\text{pub}} \cup \{(V, pk)\}$
return $\mathcal{L}_\mathsf{K}, \mathcal{K}_{\text{pub}}$
```

Fig. 3. Game for Multi-Stage-security.

The authentication predicate Auth is only used in stages that are not testable.[13] In the following, let $j \in \{1,\ldots,\mathsf{M}\}$ be such that $\mathsf{TS}.j = \mathtt{auth\text{-}only}$; we will have $j = 3$ for the three-stage MTProto protocol. In order to simplify the notation, we assume the predicates below have access to $\mathcal{L}_\mathsf{S}$. Let $\mathcal{L}_\mathsf{S}^{\mathtt{R}} := \{s \in \mathcal{L}_\mathsf{S} \mid s.\mathsf{uid}.\mathsf{role} = \mathtt{R}\}$. We first introduce a freshness predicate that defines what makes a win permissible, avoiding trivial attacks:[14]

$$\mathsf{Fresh}(j,s) := \forall s' \in \mathcal{L}_\mathsf{S} \colon s.\mathsf{sid}.j = s'.\mathsf{sid}.j \neq \bot \implies s'.\mathsf{st}_{\mathsf{sskey}}.j = \mathtt{fresh}$$

Then we define:

$$\begin{aligned} \neg\mathsf{ImplicitAuth}(j) &:= \exists s, s' \in \mathcal{L}_\mathsf{S} \colon (\mathsf{Fresh}(j,s) \wedge s.\mathsf{sskey}.j = s'.\mathsf{sskey}.j \wedge s.\mathsf{vid} \neq s'.\mathsf{uid}) \\ \neg\mathsf{KCAlmost}(j) &:= \exists s \in \mathcal{L}_\mathsf{S}^{\mathtt{R}} \colon (\mathsf{Fresh}(j,s) \wedge s.\mathsf{stage} = j \wedge \nexists s' \in \mathcal{L}_\mathsf{S} \colon \\ &\qquad (s' \neq s \wedge s'.\mathsf{kcid}.j = s.\mathsf{kcid}.j \neq \bot)) \end{aligned}$$

We only define almost-full key confirmation, as this is the strongest possible model for MTProto: as we will see, clients cannot get key confirmation from stage 3 at all despite receiving the last protocol message because the message they get back does not depend on the long-term shared symmetric key.

4 Telegram protocols

In this section, we define two MTProto key exchange protocols, $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$ and $\mathsf{MTP\text{-}KE}_{3\mathsf{st}}$, each of which instantiates KE in the model described in Section 3. In Section 4.1, we give a high-level overview. Then, Section 4.2 introduces some of the custom primitives the protocols use. Sections 4.3 and 4.4 give a formal definition of $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$ and $\mathsf{MTP\text{-}KE}_{3\mathsf{st}}$. Finally, in Section 4.5 we summarise the key differences between our formalisation and Telegram's implementation. For a high-level display of the protocols, readers may consult Fig. 10.

[13] We do not define the predicates for all stages as the two-stage protocol of MTProto, which only authenticates the responder, likely does not provide key confirmation.

[14] The effects of public-key keypair corruptions are captured via checking the session key state, since session keys are marked as revealed if and only if the public-key keypair is corrupted prior to the end of the stage authenticated by it.

```
Sound
if $\exists s, s', s'' \in \mathcal{L}_{\mathsf{S}}, i \in \{1, \ldots, \mathsf{M}\} \colon (|\{s, s', s''\}| = 3 \wedge s.\mathsf{sid}.i = s'.\mathsf{sid}.i = s''.\mathsf{sid}.i \neq \bot)$ :
    return false   // at most two sessions can be partnered
if $\exists \mathsf{Partners}(s.\mathsf{uid}.\mathsf{role} = s'.\mathsf{uid}.\mathsf{role})$ :
    return false   // partners can't have the same role
if $\exists s, s' \in \mathcal{L}_{\mathsf{S}}, i, j \in \{1, \ldots, \mathsf{M}\} \colon (i \neq j \wedge s.\mathsf{sid}.i = s'.\mathsf{sid}.j)$ :
    return false   // session ids of different stages can't match
if $\exists \mathsf{Partners}(s.\mathsf{sskey}.i \neq s'.\mathsf{sskey}.i \wedge s.\mathsf{st}_{\mathsf{exec}}.i = s'.\mathsf{st}_{\mathsf{exec}}.i = \mathtt{accepted}_i)$ :
    return false   // partners can't have different keys
if $\exists \mathsf{Partners}(s.\mathsf{cid}.i \neq s'.\mathsf{cid}.i \vee s.\mathsf{cid}.i = s'.\mathsf{cid}.i = \bot)$ :
    return false   // partners can't have different or unset contrib. ids
if $\mathtt{pub} \in \mathsf{TYPE} \wedge \exists \mathsf{Partners}(\mathsf{AUTH}.i = \mathtt{R\text{-}only} \wedge s.\mathsf{uid}.\mathsf{role} = \mathtt{I} \wedge s'.\mathsf{uid}.\mathsf{role} = \mathtt{R} \wedge s.\mathsf{vid} \neq s'.\mathsf{uid})$ :
    return false   // responder-only auth. with public keys
if $\mathtt{sym} \in \mathsf{TYPE} \wedge \exists \mathsf{Partners}(\mathsf{AUTH}.i = \mathtt{mutual} \wedge s.\mathsf{kid} \neq s'.\mathsf{kid})$ :
    return false   // mutual auth. with symmetric keys
return true

Auth
for each $j \in \{i \mid i \in \{1, \ldots, \mathsf{M}\} \wedge \mathsf{TS}.i = \mathtt{auth\text{-}only}\}$ :
    if $\neg\mathsf{ImplicitAuth}(j) \vee \neg\mathsf{KCAlmost}(j)$ : return false
return true
```

Fig. 4. Soundness and authentication predicates.

4.1 Overview

When a user wants to set up a new Telegram account, the app initiates two independent runs of a key exchange protocol in the background: first, to agree on a long-term shared key *ak*, referred to as *auth key*, and then to agree on a short-term shared key ak_{t}, referred to as *temporary auth key*. Then, ak_{t} is used to establish a secure channel, in which the first message is a special "binding" ciphertext [Tel22b], meant to bind the temporary key ak_{t} to the long-term key *ak*. Each *ak* may only be bound to a single ak_{t} at a time. Afterwards, the recently-established channel is used to transmit user registration messages such as a verification code or a password (if password-based 2FA is enabled [Tel22g]).

The use of the temporary auth key ak_{t} is meant to provide forward secrecy and is enabled by default in official clients, changing not with every new session but on a daily basis (each ak_{t} comes with an expiration timestamp). Once the key expires or shortly before, the client initiates a run of the key exchange protocol to agree on a new temporary key and binds it to their long-term auth key *ak* as described above. We assume that each user runs the Telegram client on a single device, establishing sessions with one or more Telegram servers.[15]

Let MTP-CH denote Telegram's MTProto 2.0 secure channel as formalised in [AMPS22]. Let $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ denote the initial key exchange protocol and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ its version for establishing a temporary key. As we will see, both key exchange protocols naturally lend themselves to a multi-stage model: both start by agreeing on an ephemeral key (stage 1), which is then used to encrypt and authenticate the protocol messages that follow (stage 2). Then, $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ includes an additional exchange of messages intended to provide client authentication (stage 3).

[15] This is, of course, a simplifying assumption. There is a long-term auth key and a temporary auth key for each device-datacenter pair. For instance, the Android client can maintain a connection with up to 5 datacenters, and each will be associated with different keys. Similarly, if a user uses Telegram on multiple devices, each device will establish a different set of auth keys.

4.2 Custom primitives

Here we define those primitives used by the key exchange protocols which are custom-made or diverge from standard practice.[16] For definitions of standard functions used by these primitives, such as AES-256-IGE and SHACAL-1, see [AMPS23, Section 2.2].

TOAEP^+**: custom RSA padding format.** In response to the key exchange attack described in [AMPS22], Telegram developers modified the protocol employed by MTProto 2.0 [Tel22c]. Instead of using textbook RSA with a custom padding scheme, they implemented a custom variant of RSA-OAEP+ [Sho02] (see Appendix D.3), which we will refer to as TOAEP^+. We give pseudocode for this construction in Fig. 5. In Appendix D, we give a tight proof that TOAEP^+ achieves IND-CCA-security under the standard RSA one-wayness assumption.

```
TOAEP+.KGen()
1 : $(N, p, q, e, d) \leftarrow\$ \mathsf{RSA.KGen}()$ ; $pk \leftarrow (N, e)$ ; $sk \leftarrow (N, d)$ ; return $(pk, sk)$

TOAEP+.Enc(pk, m)
1 : $(N, e) \leftarrow pk$ ; $K \leftarrow\$ \{0,1\}^{256}$
2 : $pad \leftarrow\$ \{0,1\}^{1536-|m|}$
3 : $m_{\mathsf{padded}} \leftarrow m \parallel pad$
4 : $h \leftarrow \mathsf{SHA\text{-}256}(K \parallel m_{\mathsf{padded}})$
5 : $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \parallel h$
6 : $c_{\mathsf{ige}} \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Enc}(K, 0^{256}, p_{\mathsf{ige}})$
7 : $r \leftarrow \mathsf{SHA\text{-}256}(c_{\mathsf{ige}}) \oplus K$
8 : $p_{\mathsf{rsa}} \leftarrow r \parallel c_{\mathsf{ige}}$
9 : $z \leftarrow p_{\mathsf{rsa}}$   // Parse $p_{\mathsf{rsa}}$ as an integer.
10 : if $z \notin \mathbb{Z}_N$ : goto line 1
11 : $c_{\mathsf{rsa}} \leftarrow z^e \bmod N$
12 : return $c_{\mathsf{rsa}}$

TOAEP+.Dec(sk, c_rsa)
$(N, d) \leftarrow sk$
$z \leftarrow (c_{\mathsf{rsa}})^d \bmod N$
$p_{\mathsf{rsa}} \leftarrow z$   // Parse $z$ as a 2048-bit string.
$r \leftarrow p_{\mathsf{rsa}}[0 : 256]$
$c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}[256 : 2048]$
$K \leftarrow \mathsf{SHA\text{-}256}(c_{\mathsf{ige}}) \oplus r$
$p_{\mathsf{ige}} \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Dec}(K, 0^{256}, c_{\mathsf{ige}})$
$m_{\mathsf{padded}} \leftarrow \mathsf{reverse}(p_{\mathsf{ige}}[0 : 1536])$
$h \leftarrow p_{\mathsf{ige}}[1536 : 1792]$
if $h \neq \mathsf{SHA\text{-}256}(K \parallel m_{\mathsf{padded}})$ : return $\perp$
$m \leftarrow \mathsf{RemovePadding}(m_{\mathsf{padded}})$
return $m$
```

Fig. 5. TOAEP^+: Telegram's variant of OAEP+. Here $\mathsf{reverse}(x)$ returns x in reverse byte order and RemovePadding removes padding; N is always a 2048-bit integer.

SKDF**: custom key derivation function.** To derive ephemeral keys used as part of the key exchange protocol, MTProto makes use of a custom function based on SHA-1, which we will refer to as SKDF. Formally, $\mathsf{SKDF.Ev} : \{0,1\}^{256} \times \{0,1\}^{128} \to \{0,1\}^{512}$ returns $y \leftarrow \mathsf{SKDF.Ev}(k, x)$ for key k and input x where

$$y = \mathsf{SHA\text{-}1}(k \parallel x) \parallel \mathsf{SHA\text{-}1}(x \parallel k) \parallel \mathsf{SHA\text{-}1}(k \parallel k) \parallel k[0 : 32].$$

Notably, the function directly outputs the first 32 bits of its key, severely curtailing the security properties it can be shown to achieve in general. In this work, we analyse it in conjunction with HtE-SE and NH, defined below.

HtE-SE**: custom Hash-then-Encrypt scheme.** Figure 6 defines the custom symmetric encryption scheme HtE-SE that encrypts some of the key exchange messages in the second stage. Note that SHA-1 is not

[16] Note that the abstractions of function calls into named primitives are largely our own; Telegram documentation and code does not use such abstractions.

computed over the padding, so the length of the padding is not known immediately upon decryption. The Android client gets around this using a trial and error approach [Tel22a]. This is reflected in Fig. 6. Though HtE-SE is used for encryption, the only goal it needs to meet is integrity, as it is used to protect Diffie-Hellman shares. In Appendix F, we show that HtE-SE meets a suitable notion of INT-PTXT security. Note that IND-CCA security (and thus INT-CTXT) is not achievable for HtE-SE.

```
$\mathsf{HtE\text{-}SE.Enc}(k_{se}, m)$    // $|k_{se}| = 512$
$k \leftarrow k_{se}[0:256]$ ; $iv \leftarrow k_{se}[256:512]$
$\ell \leftarrow (128 - (160 + |m|)) \bmod 128$
$r \leftarrow\$ \{0,1\}^{\ell}$    // pad to block length
$p \leftarrow \mathsf{SHA\text{-}1}(m) \parallel m \parallel r$
$c \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Enc}(k, iv, p)$
return $c$
```

```
$\mathsf{HtE\text{-}SE.Dec}(k_{se}, c)$    // $|k_{se}| = 512$
$k \leftarrow k_{se}[0:256]$ ; $iv \leftarrow k_{se}[256:512]$
$p \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Dec}(k, iv, c)$
for $i = 0, \ldots, 15$:    // bytes of padding
  $m \leftarrow p[160 : |p| - i \cdot 8]$
  if $\mathsf{SHA\text{-}1}(m) = p[0:160]$:
    return $m$
return $\bot$
```

Fig. 6. Construction of symmetric encryption scheme HtE-SE.

NH**: custom confirmation hash.** To signal key confirmation or the necessity of a retry in the protocol, MTProto uses a custom stateful hash algorithm based on SHA-1 which we will refer to as NH. For given session state st, $\mathsf{NH}_{st}\colon \{0,1\}^{256} \times \{0,1\}^{64} \rightarrow \{0,1\}^{128}$ returns $h \leftarrow \mathsf{NH}_{st}.\mathsf{Ev}(n_n, ax)$ for key n_n and input ax where

$$h = \mathsf{SHA\text{-}1}(n_n \parallel 0i \parallel ax)[32:160],$$

with $i = 1$ if the state st indicates an accept and $i = 2$ a retry. To ease exposition, we may also surface i as an explicit input to NH.Ev.

MTP-CH**: MTProto 2.0 channel.** MTProto key exchange actually encapsulates some of its messages within the newly established MTProto 2.0 channel MTP-CH. Since these messages have a specific format distinct from normal use of the channel and they only occur directly after completing key exchange, we model it as a separate stage of the key exchange protocol. We use the formalisation of [AMPS22] with small syntactical changes. Recall that MTP-ME is the MTProto message encoding scheme. We modify MTP-CH.Init to take the keys and server_salt, a value computed during key exchange, as input as shown in Fig. 7 (in turn, MTP-ME.Init must take server_salt as input, and MTP-ME.Encode must set salt using the provided state).

```
$\mathsf{MTP\text{-}CH.Init}((kk_t, mk_t), (aid_t, \mathsf{server\_salt}))$
$\boxed{kk \leftarrow kk_t}$; $\boxed{mk \leftarrow mk_t}$; $\boxed{\mathsf{auth\_key\_id} \leftarrow aid_t}$
$(kk_{\mathcal{I}}, kk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MTP\text{-}KDF}}(kk)$; $(mk_{\mathcal{I}}, mk_{\mathcal{R}}) \leftarrow \phi_{\mathsf{MTP\text{-}MAC}}(mk)$
$\mathsf{key}_{\mathcal{I}} \leftarrow (kk_{\mathcal{I}}, mk_{\mathcal{I}})$; $\mathsf{key}_{\mathcal{R}} \leftarrow (kk_{\mathcal{R}}, mk_{\mathcal{R}})$
$\boxed{(st_{\mathsf{ME},\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}}) \leftarrow \mathsf{MTP\text{-}ME.Init}(\mathsf{server\_salt})}$
$st_{\mathcal{I}} \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_{\mathcal{I}}, \mathsf{key}_{\mathcal{R}}, st_{\mathsf{ME},\mathcal{I}})$; $st_{\mathcal{R}} \leftarrow (\mathsf{auth\_key\_id}, \mathsf{key}_{\mathcal{R}}, \mathsf{key}_{\mathcal{I}}, st_{\mathsf{ME},\mathcal{R}})$
return $(st_{\mathcal{I}}, st_{\mathcal{R}})$
```

Fig. 7. Modified channel initialisation of MTP-CH with $\boxed{\text{highlighted}}$ changes.

$\mathsf{CHv1}$**: MTProto 1.0 encryption.** Within MTP-CH, MTProto key exchange also uses the deprecated 1.0 channel protocol to provide client authentication. However, since it is only used to encrypt a single message of a particular type, we do not model it as a full-fledged channel protocol.[17] We refer to this (restricted) encryption scheme as CHv1, shown in Fig. 8 together with its constituent functions Hv1, KDFv1 and SEv1. In this definition, we assume that it is only ever used with a single request, which means that the length as well as the format of the message are fixed; parts of the plaintext itself are also constant, fixed values. In Appendix G, we show the scheme achieves a weak form of plaintext integrity.

The definition of KDFv1 makes a simplifying assumption, similar to the one made in [AMPS22] for the MTProto 2.0 channel, in that it does not interleave smaller bit ranges of the SHA-1 outputs A, B, C, D but uses them in order. This does not impact security.[18] Further, the definition of Decode involves explicit checks on all fixed parts of the plaintext. In practice, some of these are implicit in the way the server parses the request (e.g. a message with a different header would eventually cause an error). The condition $p[192 : 224] \neq 00000000$ may not be checked in practice by the server, but we assume for our analysis that it is (see Section 4.5).

4.3 Parameters

First, for both protocols we define a number of protocol-specific parameters:

- $\mathsf{rid}_{\mathsf{max}} \in \mathbb{N}$ defines the maximum number of retries allowed in each session. A *retry* occurs if the protocol state of a given party is rewound, enabling it to send/receive a protocol message of the same type as it had already sent/received before; both MTProto protocols exhibit this behaviour. Concretely, one could for example set $\mathsf{rid}_{\mathsf{max}} = 32$.[19]
- We fix the group parameters for Diffie-Hellman according to MTProto: $\mathbb{G}$ denotes a cyclic group of prime order q generated by $g \in \mathbb{Z}_p^*$ where p is a prime of the form $p = 2q + 1$.[20] Though these parameters appear to be fixed in practice, they are not fixed in code – it is the server's responsibility to provide the parameters during the protocol, and the client's responsibility to check them, as we describe later.

Second, for each protocol we define in Table 1 the model parameters according to the syntax introduced in Section 3.1. We explain the session key distributions defined in the table as follows. For both protocols, $\mathcal{K}^1_{\mathsf{test}} = \{0,1\}^{256}$. For $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$, $\mathcal{K}^2_{\mathsf{test}} = \mathbb{G}_{0:1024} := \{g^c[0 : 1024] \mid g^c \in \mathbb{G}\}$, i.e. it denotes the set of strings formed by taking the 1024 MSBs of elements of $\mathbb{G}$. For $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$, $\mathcal{K}^2_{\mathsf{test}}$ is almost the same but omits 32 bits compared to $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$. Finally, for $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$, $\mathcal{K}^3_{\mathsf{test}} = \mathcal{K}^2_{\mathsf{test}}$, since the third stage is not testable.

The definition of $\mathcal{K}^2_{\mathsf{test}}$ is clearly nonstandard for two reasons. First, we do not consider a uniform distribution of strings, but rather one that includes (parts of) elements of $\mathbb{G}$. This is because MTProto uses these bits of the agreed DH values directly as key material, and so this definition represents the best possible security that can be achieved by MTProto.[21] Second, we consider a distribution with specific bit ranges of elements of $\mathbb{G}$. Formally, this is a restriction introduced by our model (i.e. MTProto documentation has no such view of session keys), however it reflects how the agreed DH values are used in practice, since not all bits are used as key material in the subsequent channel protocols. Making this restriction allows us to reason about the key exchange while avoiding trivial key confirmation attacks (see Appendix B.2).

[17] Previous attacks on MTProto 1.0 rule out the possibility of a general proof.

[18] In reality, MTProto 1.0 uses $k = A[0 : 64] \parallel B[64 : 160] \parallel C[32 : 128]$ and $iv = A[64 : 160] \parallel B[0 : 64] \parallel C[128 : 160] \parallel D[0 : 64]$.

[19] We justify this in Section 4.5.

[20] The prime p used by MTProto is shown in `https://core.telegram.org/mtproto/auth_key`.

[21] If we defined $\mathcal{K}^2_{\mathsf{test}} := \{0,1\}^{1024}$, there would be a trivial distinguishing attack between random 1024-bit strings and the first half of the MSBs of elements of $\mathbb{G}$. Given the fixed p that MTProto is using, when sampling a random 1024-bit string the probability of hitting a value r such that $p < r \cdot 2^{1024} < 2^{2048}$ is around 0.22.

```
$\mathsf{CHv1.Enc}(ak_{\mathsf{v1}}, (mid, m))$
    // $|ak_{\mathsf{v1}}| = 1024, |mid| = 64, |m| = 288$
    $r \leftarrow\$ \{0,1\}^{128}$
    $msk \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
    $k, iv \leftarrow \mathsf{KDFv1.Ev}(ak_{\mathsf{v1}}, msk)$
    $c \leftarrow \mathsf{SEv1.Enc}(k, iv, (r, mid, m))$
    return $msk, c$

$\mathsf{CHv1.Dec}(ak_{\mathsf{v1}}, (msk, c))$
    // $|ak_{\mathsf{v1}}| = 1024, |msk| = 128, |c| = 640$
    $k, iv \leftarrow \mathsf{KDFv1.Ev}(ak_{\mathsf{v1}}, msk)$
    $out \leftarrow \mathsf{SEv1.Dec}(k, iv, c)$
    if $out = \perp$ : return $\perp$
    $r, mid, m \leftarrow out$
    $msk' \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
    if $msk' \neq msk$ : return $\perp$
    return $mid, m$

$\mathsf{Hv1.Ev}(r, mid, m)$    // $|r| = 128, |mid| = 64, |m| = 288$
    $p \leftarrow \mathsf{Encode}(r, mid, m)$ ; $msk \leftarrow \mathsf{SHA\text{-}1}(p)[32:160]$ ; return $msk$

$\mathsf{KDFv1.Ev}(ak_{\mathsf{v1}}, msk)$    // $|ak_{\mathsf{v1}}| = 1024, |msk| = 128$
    $A \leftarrow \mathsf{SHA\text{-}1}(msk \,\|\, ak_{\mathsf{v1}}[0:256])$
    $B \leftarrow \mathsf{SHA\text{-}1}(ak_{\mathsf{v1}}[256:384] \,\|\, msk \,\|\, ak_{\mathsf{v1}}[384:512])$
    $C \leftarrow \mathsf{SHA\text{-}1}(ak_{\mathsf{v1}}[512:768] \,\|\, msk)$
    $D \leftarrow \mathsf{SHA\text{-}1}(msk \,\|\, ak_{\mathsf{v1}}[768:1024])$
    $k \,\|\, iv \leftarrow A \,\|\, B \,\|\, C[32:160] \,\|\, D[0:64]$    // $|k| = |iv| = 256$
    return $k, iv$

$\mathsf{SEv1.Enc}(k, iv, (r, mid, m))$
    // $|r| = 128, |mid| = 64, |m| = 288$
    $p \leftarrow \mathsf{Encode}(r, mid, m)$
    $r_{\mathsf{pad}} \leftarrow\$ \{0,1\}^{64}$ ; $p_{\mathsf{pad}} \leftarrow p \,\|\, r_{\mathsf{pad}}$
    $c \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Enc}(k, iv, p_{\mathsf{pad}})$
    return $c$

$\mathsf{SEv1.Dec}(k, iv, c)$    // $|c| = 640$
    $p_{\mathsf{pad}} \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Dec}(k, iv, c)$
    $p \leftarrow p_{\mathsf{pad}}[0:576]$ ; $out \leftarrow \mathsf{Decode}(p)$
    if $out = \perp$ : return $\perp$
    $r, mid, m \leftarrow out$
    return $r, mid, m$

$\mathsf{Encode}(r, mid, m)$    // $|r| = 128, |mid| = 64, |m| = 288$
    $m_{\mathsf{TL}} \leftarrow \mathsf{65f7a375} \,\|\, m$    // $m_{\mathsf{TL}} = \mathsf{TL}(\mathtt{req}, m)$
    $p \leftarrow r \,\|\, mid \,\|\, \mathsf{00000000} \,\|\, \mathsf{len}(m_{\mathsf{TL}}) \,\|\, m_{\mathsf{TL}}$ ; return $p$

$\mathsf{Decode}(p)$    // $|p| = 576$
    if $p[192:224] \neq \mathsf{00000000} \lor p[224:256] \neq \mathsf{00000028} \lor p[256:288] \neq \mathsf{65f7a375}$ :
        return $\perp$
    $r \leftarrow p[0:128]$ ; $mid \leftarrow p[128:192]$ ; $m \leftarrow p[288:576]$ ; return $r, mid, m$
```

Fig. 8. MTProto 1.0 channel encryption for a message sent from the client to the server. Note that the scheme is with respect to the particular binding request $\mathtt{req} = \mathtt{bind\_auth\_key\_inner}$, and we have $\mathsf{len}(m_{\mathsf{TL}}) = \mathsf{len}(1^{32+288}) = \mathsf{00000028}$.

Table 1. Parameters for MTProto key exchange protocols, where $\mathbb{G}' = \mathbb{G}_{0:672} \times \mathbb{G}_{704:1024}$.

	$\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$	$\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$
M	2	3
TYPE	$\{\texttt{pub}\}$	$\{\texttt{pub}, \texttt{sym}\}$
AUTH	$(\texttt{R-only}, \texttt{R-only})$	$(\texttt{R-only}, \texttt{R-only}, \texttt{mutual})$
TS	$(\texttt{testable}, \texttt{testable})$	$(\texttt{testable}, \texttt{testable}, \texttt{auth-only})$
USE	$(\texttt{internal}, \texttt{external})$	$(\texttt{internal}, \texttt{internal}, \texttt{none})$
FS	2	2
KD	`dependent`	`dependent`
DIST	$(\{0,1\}^{256}, \mathbb{G}_{0:1024})$	$(\{0,1\}^{256}, \mathbb{G}', \mathbb{G}')$

4.4 Protocol definitions

Key generation for both protocols is shown in Fig. 9. Notice that symmetric key generation is stateful to enable modelling the fact that *aid* values are randomly sampled without replacement for each server.[22]

```
KE.KGen_pub()
    // 2048-bit
    $(pk, sk) \leftarrow\$ \mathsf{RSA.KGen}()$
    return $(pk, sk)$
```

```
MTP-KE_3st.KGen_sym(V) // $V \in \mathcal{U}_{\mathsf{role}} \wedge V.\mathsf{role} = \mathtt{R}$
    if $\mathcal{S}^V_{\mathsf{aid}}$ undefined : $\mathcal{S}^V_{\mathsf{aid}} \leftarrow \emptyset$
    $aid \leftarrow\$ \{0,1\}^{64} \setminus \mathcal{S}^V_{\mathsf{aid}}$ ; $\mathcal{S}^V_{\mathsf{aid}} \leftarrow \mathcal{S}^V_{\mathsf{aid}} \cup \{aid\}$
    $ak_{\mathsf{v1}} \leftarrow\$ \{0,1\}^{1024}$ ; return $(aid, ak_{\mathsf{v1}})$
```

Fig. 9. Key generation for $\mathsf{KE} = \mathsf{MTP\text{-}KE}_{\mathsf{2st}}, \mathsf{MTP\text{-}KE}_{\mathsf{3st}}$.

In the following subsections, we define the behaviour of $\mathsf{KE.Init}$, $\mathsf{KE.Run}$ for $\mathsf{KE} = \mathsf{MTP\text{-}KE}_{\mathsf{2st}}, \mathsf{MTP\text{-}KE}_{\mathsf{3st}}$, where Init is assumed to cover the first message sent by the client. The protocol definitions are based on Telegram documentation [Tel22c] as well as the source code of the official apps for Android and desktop [Tel22d, Tel22e]. Though we do not model this explicitly, packets sent as part of the key exchange are of the form $\mathsf{00000000\ 00000000} \parallel mid \parallel \mathsf{len}(m) \parallel m$, where *mid* is a random 64-bit message identifier and m itself is in plaintext or partially encrypted.

Figure 10 shows a simplified overview of both protocols, while Appendix A provides detailed figures for each stage. Though $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ uses *ak* to denote the stage 2 session key, $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ uses *ak* to denote the long-term shared symmetric key and ak_{t} to refer to the stage 2 session key. In the figures, we omit displaying TL encoding of protocol messages at any level, and we use "$x \ldots y$" as shorthand for $x \parallel z \parallel y$ where z is unimportant; we denote the current state of the protocol session by *st*. Note that TL decoding failures always result in rejection.

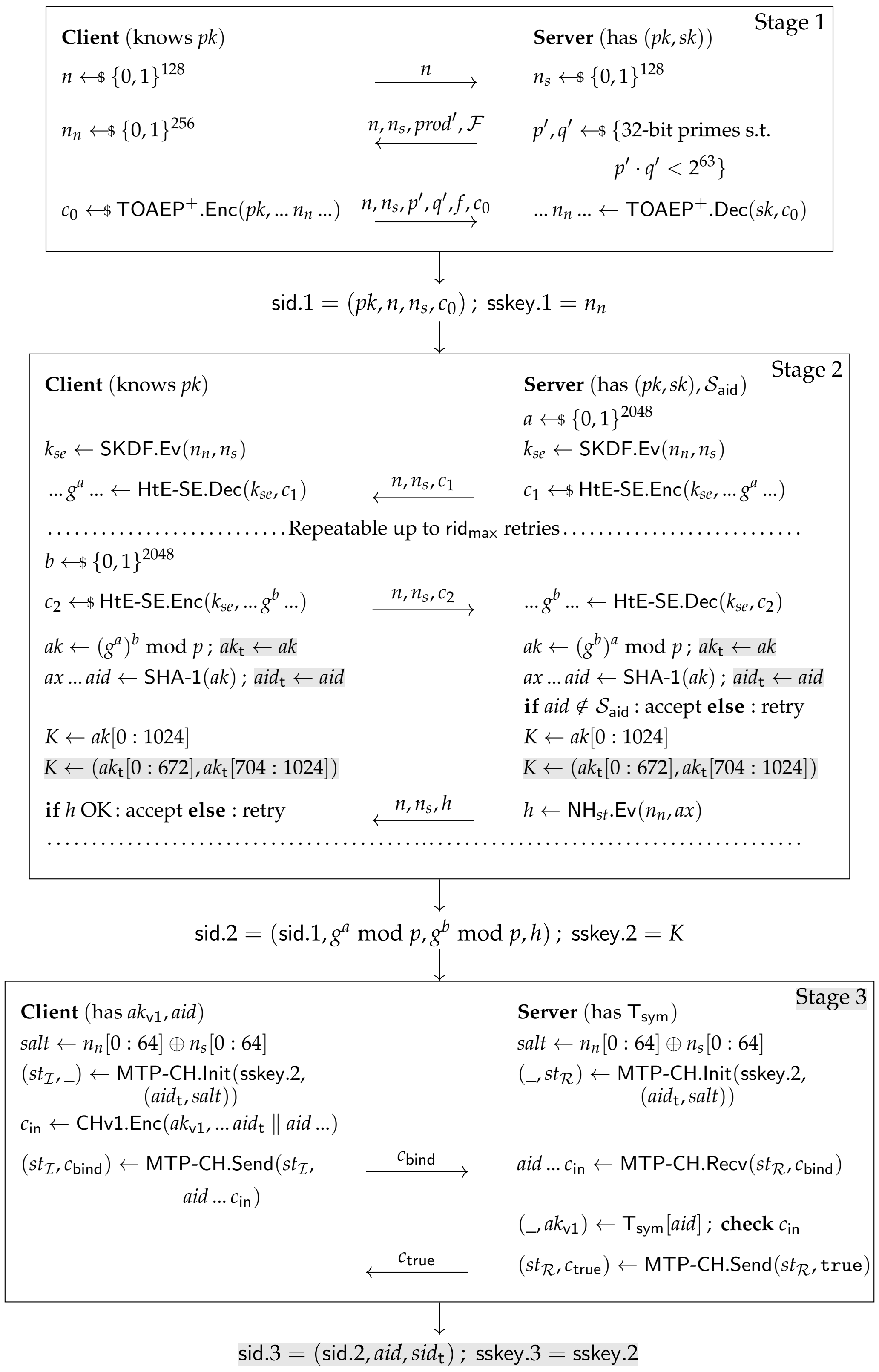
The two-stage protocol $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$.

Stage 1. The first stage (also shown in Fig. 11) proceeds as follows:

1. The client is initialised with the knowledge of the server's RSA public key *pk*,[23] and sets $s.\mathsf{vid}$ to the server's identity. The client samples a nonce $n \leftarrow\$ \{0,1\}^{128}$ and sends $m = \mathsf{TL}(\texttt{req\_pq\_multi}, n)$ to the server.

[22] As we explain in Section 4.5, our description of $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}.\mathsf{KGen}_{\mathsf{sym}}$ represents a simplifying assumption, since the real *ak* from which ak_{v1} and *aid* are derived is actually the output of an instance of $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$. We do not model this dependency between the 2-stage and 3-stage protocols in this work.

[23] Telegram clients are shipped with the hardcoded key, though its value may differ between platforms.



Stage 1

Client (knows pk)		**Server** (has (pk, sk))
$n \leftarrow\$ \{0,1\}^{128}$	$\xrightarrow{n}$	$n_s \leftarrow\$ \{0,1\}^{128}$
$n_n \leftarrow\$ \{0,1\}^{256}$	$\xleftarrow{n, n_s, prod', \mathcal{F}}$	$p', q' \leftarrow\$ \{$32-bit primes s.t. $p' \cdot q' < 2^{63}\}$
$c_0 \leftarrow\$ \mathsf{TOAEP}^+.\mathsf{Enc}(pk, \ldots n_n \ldots)$	$\xrightarrow{n, n_s, p', q', f, c_0}$	$\ldots n_n \ldots \leftarrow \mathsf{TOAEP}^+.\mathsf{Dec}(sk, c_0)$

$$\mathsf{sid.1} = (pk, n, n_s, c_0)\ ;\ \mathsf{sskey.1} = n_n$$

Stage 2

Client (knows pk)		**Server** (has $(pk, sk), \mathcal{S}_{\mathsf{aid}}$)
		$a \leftarrow\$ \{0,1\}^{2048}$
$k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$		$k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
$\ldots g^a \ldots \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_1)$	$\xleftarrow{n, n_s, c_1}$	$c_1 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k_{se}, \ldots g^a \ldots)$
........................Repeatable up to $\mathsf{rid}_{\mathsf{max}}$ retries		....
$b \leftarrow\$ \{0,1\}^{2048}$		
$c_2 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k_{se}, \ldots g^b \ldots)$	$\xrightarrow{n, n_s, c_2}$	$\ldots g^b \ldots \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_2)$
$ak \leftarrow (g^a)^b \bmod p$; $ak_{\mathsf{t}} \leftarrow ak$		$ak \leftarrow (g^b)^a \bmod p$; $ak_{\mathsf{t}} \leftarrow ak$
$ax \ldots aid \leftarrow \mathsf{SHA\text{-}1}(ak)$; $aid_{\mathsf{t}} \leftarrow aid$		$ax \ldots aid \leftarrow \mathsf{SHA\text{-}1}(ak)$; $aid_{\mathsf{t}} \leftarrow aid$
		if $aid \notin \mathcal{S}_{\mathsf{aid}}$: accept **else** : retry
$K \leftarrow ak[0:1024]$		$K \leftarrow ak[0:1024]$
$K \leftarrow (ak_{\mathsf{t}}[0:672], ak_{\mathsf{t}}[704:1024])$		$K \leftarrow (ak_{\mathsf{t}}[0:672], ak_{\mathsf{t}}[704:1024])$
if h OK : accept **else** : retry	$\xleftarrow{n, n_s, h}$	$h \leftarrow \mathsf{NH}_{st}.\mathsf{Ev}(n_n, ax)$
....		....

$$\mathsf{sid.2} = (\mathsf{sid.1}, g^a \bmod p, g^b \bmod p, h)\ ;\ \mathsf{sskey.2} = K$$

Stage 3

Client (has ak_{v1}, aid)		**Server** (has $\mathsf{T}_{\mathsf{sym}}$)
$salt \leftarrow n_n[0:64] \oplus n_s[0:64]$		$salt \leftarrow n_n[0:64] \oplus n_s[0:64]$
$(st_{\mathcal{I}}, \_) \leftarrow \mathsf{MTP\text{-}CH.Init}(\mathsf{sskey.2}, (aid_{\mathsf{t}}, salt))$		$(\_, st_{\mathcal{R}}) \leftarrow \mathsf{MTP\text{-}CH.Init}(\mathsf{sskey.2}, (aid_{\mathsf{t}}, salt))$
$c_{\mathsf{in}} \leftarrow \mathsf{CHv1.Enc}(ak_{\mathsf{v1}}, \ldots aid_{\mathsf{t}} \parallel aid \ldots)$		
$(st_{\mathcal{I}}, c_{\mathsf{bind}}) \leftarrow \mathsf{MTP\text{-}CH.Send}(st_{\mathcal{I}}, aid \ldots c_{\mathsf{in}})$	$\xrightarrow{c_{\mathsf{bind}}}$	$aid \ldots c_{\mathsf{in}} \leftarrow \mathsf{MTP\text{-}CH.Recv}(st_{\mathcal{R}}, c_{\mathsf{bind}})$
		$(\_, ak_{\mathsf{v1}}) \leftarrow \mathsf{T}_{\mathsf{sym}}[aid]$; **check** c_{in}
	$\xleftarrow{c_{\mathsf{true}}}$	$(st_{\mathcal{R}}, c_{\mathsf{true}}) \leftarrow \mathsf{MTP\text{-}CH.Send}(st_{\mathcal{R}}, \mathtt{true})$

$$\mathsf{sid.3} = (\mathsf{sid.2}, aid, sid_{\mathsf{t}})\ ;\ \mathsf{sskey.3} = \mathsf{sskey.2}$$

Fig. 10. Simplified overview of the protocols $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ (parts only present in $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ are shown in grey).

2. The server samples a "server nonce" $n_s \leftarrow\$ \{0,1\}^{128}$, picks two primes p', q' such that their product $prod' = p' \cdot q' \leq 2^{63} - 1$, and puts together a set $\mathcal{F} = \{\mathsf{SHA\text{-}1}(pk)[96 : 160]\}$ (for a suitable serialisation of pk) that contains the "fingerprint" of its RSA public key.[24] It sends $m = \mathsf{TL}(\texttt{resPQ}, n, n_s, prod', \mathcal{F})$ to the client.

3. The client factors the product $prod'$ to recover p', q', samples a "new nonce" $n_n \leftarrow\$ \{0,1\}^{256}$, checks that the fingerprint $f = \mathsf{SHA\text{-}1}(pk)[96 : 160]$ is in the set $\mathcal{F}$, and creates the ciphertext $c_0 \leftarrow\$ \mathsf{TOAEP}^{+}.\mathsf{Enc}(pk, m_0)$ where

$$m_0 \leftarrow \mathsf{TL}(\texttt{p\_q\_inner\_data\_dc}, prod', p', q', n, n_s, n_n, dc),$$

   for some $dc \in \{0,1\}^{32}$.[25] It sends $m = \mathsf{TL}(\texttt{req\_DH\_params}, n, n_s, p', q', f, c_0)$ to the server.

4. The server decrypts c_0 with its private key sk. If decryption succeeds and m_0 is valid and contains the expected values n, n_s, the server accepts $\mathsf{sid}.1 \leftarrow (pk, n, n_s, c_0)$ and $\mathsf{sskey}.1 \leftarrow n_n$ and proceeds with stage 2. Else, it rejects.

Stage 2. The second stage (also shown in Fig. 12) proceeds as follows:

1. The server, continuing its computation, proceeds to sample $a \leftarrow\$ \{0,1\}^{2048}$ and to create the ciphertext $c_1 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k \parallel iv, m_1)$ where

$$\begin{aligned} k \parallel iv &\leftarrow \mathsf{SKDF.Ev}(n_n, n_s) \\ m_1 &\leftarrow \mathsf{TL}(\texttt{server\_DH\_inner\_data}, n, n_s, g, p, g^a \bmod p, servertime), \end{aligned}$$

   for the fixed DH parameters g, p and a 32-bit timestamp *servertime*.
   It sends $m = \mathsf{TL}(\texttt{server\_DH\_params\_ok}, n, n_s, c_1)$ to the client.

2. The client recomputes $k \parallel iv \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$, decrypts c_1, parses the decrypted m_1, verifies that the received n, n_s match the expected values and checks p, g, g^a.[26] Then, it samples $b \leftarrow\$ \{0,1\}^{2048}$ and creates the ciphertext $c_2 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k \parallel iv, m_2)$ where

$$m_2 \leftarrow \mathsf{TL}(\texttt{client\_DH\_inner\_data}, n, n_s, rid, g^b \bmod p),$$

   for "retry id" $rid \leftarrow 0$ if this is the first run of this step, else $rid \leftarrow rid + 1$.[27] It sends $m = \mathsf{TL}(\texttt{set\_client\_DH\_params}, n, n_s, c_2)$ to the server.

3. The server decrypts c_2, parses the decrypted m_2 and verifies that the received n, n_s match the expected values. The server computes the auth key $ak \leftarrow (g^b)^a \bmod p$ as well as the corresponding "auth key aux hash" $ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0 : 64]$ and the "auth key id" $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96 : 160]$. Then it proceeds in one of the following ways:

   (a) If aid is unique with respect to $\mathcal{S}_{\mathsf{aid}}$, the set of all accepted aid's of this server, it computes $h_1 \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$ and sends $m = \mathsf{TL}(\texttt{dh\_gen\_ok}, n, n_s, h_1)$ to the client. It accepts $\mathsf{sid}.2 \leftarrow (\mathsf{sid}.1, g^a \bmod p, g^b \bmod p, h)$ and $\mathsf{sskey}.2 \leftarrow ak[0 : 1024]$.

   (b) If aid is not unique, i.e. $aid \in \mathcal{S}_{\mathsf{aid}}$, it sets $h_2 \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$ and sends $m = \mathsf{TL}(\texttt{dh\_gen\_retry}, n, n_s, h_2)$ to the client. The server restarts from Step 3, expecting a fresh g^b and rid.

[24] The protocol is designed to work with a number of public keys, but in our model each server only holds a single pk, and so $|\mathcal{F}| = 1$. We discuss this more in Section 4.5.

[25] The value dc identifies the Telegram data centre, however it is not relevant for our analysis of the protocol.

[26] In more detail, this includes checking if p is a safe prime, $2 < g < 7$, the order of g is $(p-1)/2$, and various checks on bit sizes. Since in practice fixed parameters are used, we assume these checks always pass; see Section 4.5 for more discussion.

[27] Here, we diverge from current implementation, which sets $rid \leftarrow \mathsf{SHA\text{-}1}(ak)[0 : 64]$ for ak from the previous attempt. See Section 4.5 for justification for this change.

4. The client computes $ak \leftarrow (g^a)^b \bmod p$ and $ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$, $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$. Then, depending on the TL type of the received m:

   (a) If $m = \mathsf{TL}(\texttt{dh\_gen\_ok}, n, n_s, h)$ and $h = \mathsf{NH.Ev}(n_n, ax, 1)$, it accepts $\mathsf{sid}.2 \leftarrow (\mathsf{sid}.1, g^a \bmod p, g^b \bmod p, h)$, $\mathsf{sskey}.2 \leftarrow ak[0:1024]$.

   (b) If $m = \mathsf{TL}(\texttt{dh\_gen\_retry}, n, n_s, h)$ and $h = \mathsf{NH.Ev}(n_n, ax, 2)$, it restarts from Step 2.[28]

Both the client and the server reject if the number of total retries reaches $\mathsf{rid}_{\mathsf{max}}$.

The three-stage protocol $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$**.**

Stages 1 and 2. The first and the second stage are identical with that of $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$, except for the following. In stage 1, the message m_0 computed in Step 3 has a different header and contains one more message field:

$$m_0 \leftarrow \mathsf{TL}(\texttt{p\_q\_inner\_data\_temp\_dc}, prod', p', q', n, n_s, n_n, dc, exp),$$

where $exp \leftarrow 24 \cdot 60 \cdot 60$ is the "expiration" time of the temporary key in seconds. In stage 2, the output session key uses 32 fewer bits of the agreed DH value, in particular $\mathsf{sskey}.2 \leftarrow (ak_{\mathsf{t}}[0:672], ak_{\mathsf{t}}[704:1024])$. Figures 11 and 12 in Appendix A highlight these differences.

Stage 3. Both the client and the server are expected to have been initialised with at least one tuple (W, kid, key) where W is the identity of the expected partner, $kid = aid$, and $key = ak_{\mathsf{v1}}$. The server collates all such tuples it receives in a table $\mathsf{T}_{\mathsf{sym}}$ so that $\mathsf{T}_{\mathsf{sym}}[aid] = (W, ak_{\mathsf{v1}})$. The client sets its $s.\mathsf{kid}$ to the first value of (W, aid) it receives.

The third stage (also shown in Fig. 13) proceeds as follows:

1. The client computes a "server salt" value $salt \leftarrow n_n[0:64] \oplus n_s[0:64]$, and uses it with aid_{t} to initialise the MTProto 2.0 channel under the stage 2 session key $(ak_{\mathsf{t}}[0:672], ak_{\mathsf{t}}[704:1024])$. It then sends a special message m_{bind} within this channel, which itself contains another ciphertext encrypted under the MTProto 1.0 channel using the shared symmetric key:

$$\begin{aligned} m_{\mathsf{in}} &\leftarrow \mathsf{TL}(\texttt{bind\_auth\_key\_inner}, n_{\mathsf{b}}, aid_{\mathsf{t}}, aid, sid_{\mathsf{t}}, exp) \\ c_{\mathsf{in}} &\leftarrow \mathsf{CHv1.Enc}(ak_{\mathsf{v1}}, (mid, m_{\mathsf{in}})) \\ m_{\mathsf{bind}} &\leftarrow \mathsf{TL}(\texttt{auth.bindTempAuthKey}, aid, n_{\mathsf{b}}, exp, c_{\mathsf{in}}) \end{aligned}$$

   where $n_{\mathsf{b}} \leftarrow\!\!\$ \{0,1\}^{64}$, sid_{t} is the session id from the outer MTProto 2.0 channel, and exp is a 32-bit expiration timestamp set to 24 hours from current time. The mid field that serves as input to the inner MTProto 1.0 encryption algorithm (Fig. 8) is the same value used in the outer $\mathsf{MTP\text{-}CH}$ ciphertext.[29]

2. The server likewise initiates the MTProto 2.0 channel and uses it to process the client's message m_{bind}. If the value of exp within m_{bind} is in the past, it rejects. It uses the table $\mathsf{T}_{\mathsf{sym}}[aid]$ with aid taken from the client's message to find the claimed identity W as well as the key ak_{v1}. It sets $s.\mathsf{kid} \leftarrow (W, aid)$. It retrieves mid and sid_{t} from the state of the channel. Then, it decrypts the inner ciphertext c_{in} with $\mathsf{CHv1.Dec}$ under ak_{v1} to get (mid', m'_{in}).

   It checks that $mid' = mid$, and then checks the contents of m'_{in}, which must contain n_{b} and exp from m_{bind} as well as aid_{t}, aid and sid_{t}. If all checks pass, it sends the message $m_{\mathsf{true}} \leftarrow \mathsf{TL}(\texttt{boolTrue})$ using the channel, sets $s.\mathsf{vid} \leftarrow W$ and accepts $\mathsf{sid}.3 = (\mathsf{sid}.2, aid, sid_{\mathsf{t}})$ and $\mathsf{sskey}.3 = \mathsf{sskey}.2$, i.e. this stage has no "new" session key output.[30]

[28] In the code, the client also checks if $m = \mathsf{TL}(\texttt{dh\_gen\_fail}, n, n_s, h)$ where $h = \mathsf{NH.Ev}(n_n, ax, 3)$, in which case it rejects; however, this case never seems to arise in practice, and so we omit it from our model of the protocol.

[29] The value of mid is normally generated as part of $\mathsf{MTP\text{-}CH.Send}$, but here it must be picked first in order to compute c_{in}. Note that in the formalisation of $\mathsf{MTP\text{-}CH}$ in [AMPS22], the field mid is subsumed into a counter value N_{sent} alongside the sequence number, but we explicitly surface it here to stay closer to the implementation.

[30] This does not introduce issues because stage 3 is not testable.

3. The client processes the server's message using its existing MTProto 2.0 channel. It accepts if and only if it gets the message $m_{\mathsf{true}} = \mathsf{TL}(\mathtt{boolTrue})$.

Session identifiers. For both protocols, we define the accepting outputs as:

$$\begin{aligned}
sid.1 &:= (pk, n, n_s, c_0)\\
sskey.1 &:= n_n\\
sid.2 &:= ((pk, n, n_s, c_0), g^a \bmod p, g^b \bmod p, h)\\
sskey.2 &:= ak[0:1024] \text{ in } \mathsf{MTP\text{-}KE_{2st}}\,;\ (ak_{\mathsf{t}}[0:672], ak_{\mathsf{t}}[704:1024]) \text{ in } \mathsf{MTP\text{-}KE_{3st}}\\
sid.3 &:= (((pk, n, n_s, c_0), g^a \bmod p, g^b \bmod p, h), aid, sid_{\mathsf{t}})\\
sskey.3 &:= (ak_{\mathsf{t}}[0:672], ak_{\mathsf{t}}[704:1024]).
\end{aligned}$$

In Appendix B.2, we justify this choice of definition. In both protocols, the contributive identifiers for the first stage are set as $cid.1 = pk$ at the beginning, $cid.1 = (pk, n)$ after the first message, $cid.1 = (pk, n, n_s)$ after the second message and $cid.1 = (pk, n, n_s, c_0) = sid.1$ after the third message. The second stage sets $cid.2 = (pk, n, n_s, c_0)$ at the beginning, $cid.2 = (pk, n, n_s, c_0, g^a \bmod p)$ after the first message and $cid.2 = (pk, n, n_s, c_0, g^a \bmod p, g^b \bmod p)$ after the second message. Since the third stage is mutually authenticated, $cid.3 = sid.3$ is set after the first message. The protocol also sets $kcid.3 = cid.3$ at the same time.

4.5 Differences and the scope of the model

Here we list the main differences between our model and Telegram implementations. The remaining differences can be found in Appendix B.1. For justification of some of the modelling choices we make which do not represent a change from the implementation, see Appendix B.2.

Independence of protocols. We do not model that $\mathsf{MTP\text{-}KE_{3st}}$ and $\mathsf{MTP\text{-}KE_{2st}}$ are run composed in practice, i.e. we do not capture that the long-term symmetric key used in the binding request in $\mathsf{MTP\text{-}KE_{3st}}$ comes from a previous run of $\mathsf{MTP\text{-}KE_{2st}}$, though both are denoted as *ak*. That is, in our model we introduce a function $\mathsf{MTP\text{-}KE_{3st}.KGen_{sym}}$ that we allow the adversary to call in NEWSECRET. This approach allows us to focus on the main task of analysing the security of each protocol in isolation. We leave the task of proving that composing $\mathsf{MTP\text{-}KE_{2st}}$ with $\mathsf{MTP\text{-}KE_{3st}}$ does not introduce additional issues as future work.

Our model also does not allow for generic composition of our theorems about $\mathsf{MTP\text{-}KE_{3st}}$ and the existing results about the channel MTP-CH. Obtaining a specific composition result would first require bridging the gap between the nonstandard session key distribution and the uniform distribution assumed by the proofs in [AMPS22]. Proving composition would however still be difficult due to other features of the protocol such as key dependence and internal use of session keys, but also due to the way we define session identifiers.

One public key per server. We assume that each server has a single associated public key rather than multiple; at the time of writing, this is true from the perspective of the official clients, it has however not always been so. While we could make the model more general, we could not capture that clients may update their store of server public keys out-of-band in response to corruption. As a result, there would be a trivial downgrade attack in such a model, and to avoid it we would have to restrict the CORRUPT query to corrupt all keys of a given server at the same time in order to argue about its security. Thus we model a single key per server, which is equivalent in behaviour but allows for a simpler presentation.

Fixed number of retries. Our model of the protocol makes two modifications to the retry mechanism in stage 2. First, we define *rid* as a simple counter instead of setting it to $\mathsf{SHA\text{-}1}(ak)[0:64]$ of the previously agreed *ak*. In theory, this could enable replay attacks due to collisions in *rid*, however such a replayed message would never lead to the server accepting the replayed key: by definition, the server must have proceeded with a retry for the original message because $aid \in \mathcal{S}_{\mathsf{aid}}$, which will likely still be true at the time of the replay.[31] Similarly, if a particular retry attempt collided on *rid*, and the attacker dropped the message preceding the one with the collision, this would not be noticed by the server. However, this is a highly contrived scenario, since none of the information necessary to make a correct choice on which message to drop would be available to the attacker.

Second, the implementation allows a large number of retries. In practice, because of a time limit of 10 minutes for each run of the key exchange, the number of possible retries is actually bounded. But the precise bound would be dependent on network conditions. Moreover, a server's probability of requesting another retry is directly related to the size of $\mathcal{S}_{\mathsf{aid}}$, i.e. the number of other sessions in which it has completed an exchange. We have modified the protocol to upper-bound the number of possible retries by a parameter $\mathsf{rid}_{\mathsf{max}}$. This enables us to achieve a proof with meaningful security bounds.[32] We argue that a value as low as $\mathsf{rid}_{\mathsf{max}} = 32$ would suffice for smooth operation of the protocol and would be easy for Telegram to implement.

A hypothetical replay attack. If the adversary could replay past binding messages in stage 3 of $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$, it could break client authentication. There is an attack in the following scenario: the adversary has compromised a past ak_{t} (which is now expired and deleted by the server), and captured its binding message c_{bind}. This means the attacker can decrypt the channel layer to learn m_{bind} as well as all of its $\mathsf{MTP\text{-}CH}$ headers, including sid_{t} and mid. Then the attacker does the following:

1. Run the first two stages of $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ with the server to establish a new ak^*_{t} such that $aid^*_{\mathsf{t}} = aid_{\mathsf{t}}$ from the compromised session. This requires finding a second-preimage for truncated $\mathsf{SHA\text{-}1}$, i.e. setting b in $ak^*_{\mathsf{t}} = (g^a)^b \bmod p$ for server-chosen g^a and p such that $\mathsf{SHA\text{-}1}(ak^*_{\mathsf{t}})[96:160] = \mathsf{SHA\text{-}1}(ak_{\mathsf{t}})[96:160]$. This would require on the order of 2^{64} hash computations, conceivably within reach for a well-resourced attacker.
2. Use kk^*, mk^* derived from ak^*_{t}, the recovered sid_{t} and mid, and $\mathsf{server\_salt}^*$ computed from the first two stages to re-encrypt m_{bind}. Then, send the new ciphertext c^*_{bind}.

The server will decrypt c^*_{bind}, which will parse successfully as a message binding ak^*_{t} to ak only if the ak^*_{t} is not considered expired. The server used to not check the actual value of exp as long as it matched the one inside the $\mathsf{CHv1}$-encrypted ciphertext (i.e. a valid key used with a timestamp in the past went through, while an expired key used with a current timestamp did not). We disclosed this to Telegram developers, who addressed the issue in response, i.e. the validity of the outer timestamp is checked now. In our model, we include this check explicitly.

5 Theorem statements

In this section, we state our results about the two MTProto key exchange protocols. Since the two protocols largely overlap, the main result on $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ lies in Theorem 1, with proof in Appendix H.1. Then Theorem 2, whose proof is in Appendix H.2, expresses how the third stage of $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ additionally achieves client authentication. Our proofs rely on a number of high-level properties, which have their own separate proofs in Appendices D, F and G.

Theorem 1. *Let $\mathcal{U}_{\mathsf{role}}$ be any set of users with roles such that no user holds more than one role. Let $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ be as defined in Section 4, and $\mathcal{A}$ be an adversary against $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ in the $\mathsf{Multi\text{-}Stage}$ game (Definition 3) parametrised by $\mathcal{U}_{\mathsf{role}}$ with at most $\mathsf{n_S}$ sessions and $\mathsf{n_{pk}}$ different parties in the role of a responder. Let $\mathsf{n_R}$ be the maximum number of retries performed by any session, so $\mathsf{n_R} \leq \mathsf{rid}_{\mathsf{max}}$. Then there exist adversaries $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ against the* IND-CCA*-security of TOAEP^+ (Fig. 21), $\mathcal{A}_{\mathsf{INT\text{-}PTXT}}$ against the* INT-PTXT*-security of $\mathsf{HtE\text{-}SE}$ with SKDF and $\mathsf{SAMP}[\cdot,\cdot,g,p]$ (Definition 13), $\mathcal{D}_{\mathsf{IND\text{-}KEY}}$ against the* IND-KEY*-security of SKDF and NH*

[31] It would not be true in the edge case that the given *aid* had just expired and been therefore removed from $\mathcal{S}_{\mathsf{aid}}$ in the time between the original message and the replay.

[32] See Appendix I for further discussion related to this point.

(Definition 9), $\mathcal{D}_{\mathsf{S\text{-}EXP}}$ against the S-EXP *problem in the fixed group $\mathbb{G}$ of order q (Definition 1), $\mathcal{D}_{\mathsf{DDH}}$ against the DDH assumption in $\mathbb{G}$, and $\mathcal{D}_{\mathsf{OTPRF}}$ against the* OTPRF*-security of* SHACAL-1 *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MTP\text{-}KE}_{2st},\mathcal{U}_{\mathsf{role}}}(\mathcal{A}) \leq \frac{\mathsf{n}_\mathsf{S}^2}{2^{384}} + \frac{\mathsf{n}_{\mathsf{pk}}\,\mathsf{n}_\mathsf{S}^2}{2^{497}} + \frac{16\,\mathsf{n}_{\mathsf{pk}}\,\mathsf{n}_\mathsf{S}^3}{q} \\ + 4\,\mathsf{n}_{\mathsf{pk}}\,\mathsf{n}_\mathsf{S}\,(\mathsf{n}_\mathsf{S}+1)\cdot\Big(2\,\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{TOAEP}^+}(\mathcal{A}_{\mathsf{IND\text{-}CCA}}) \\ + \mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p}(\mathcal{A}_{\mathsf{INT\text{-}PTXT}})\Big) \\ + 4\,\mathsf{n}_{\mathsf{pk}}\,\mathsf{n}_\mathsf{S}^2\cdot\Big(\mathsf{Adv}^{\mathsf{IND\text{-}KEY}}_{\mathsf{SKDF},\mathsf{NH}}(\mathcal{D}_{\mathsf{IND\text{-}KEY}}) + \mathsf{Adv}^{\mathsf{S\text{-}EXP}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{S\text{-}EXP}}) \\ + \mathsf{Adv}^{\mathsf{DDH}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{DDH}}) + \mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{OTPRF}})\Big)\,.\end{aligned}$$

Theorem 2. *Let $\mathcal{U}_{\mathsf{role}}$ and* $\mathsf{MTP\text{-}KE}_{2st}$ *be as in Theorem 1. Let* $\mathsf{MTP\text{-}KE}_{3st}$ *be as defined in Section 4, and $\mathcal{A}$ be an adversary against* $\mathsf{MTP\text{-}KE}_{3st}$ *in the* Multi-Stage *game (Definition 3) parametrised by $\mathcal{U}_{\mathsf{role}}$. Then there exist adversaries $\mathcal{A}_{\mathsf{EUF\text{-}CMA}}$ against the* EUF-CMA*-security of* CHv1 *(Definition 17), and $\mathcal{A}_{\mathsf{KE}}$ against the* Multi-Stage*-security of* $\mathsf{MTP\text{-}KE}_{2st}$ *such that*

$$\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MTP\text{-}KE}_{3st},\mathcal{U}_{\mathsf{role}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MTP\text{-}KE}_{2st},\mathcal{U}_{\mathsf{role}}}(\mathcal{A}_{\mathsf{KE}}) + 2\,\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1}}(\mathcal{A}_{\mathsf{EUF\text{-}CMA}}).$$

Acknowledgements

We thank Felix Günther and Joseph Jaeger for discussions and insights. The research of Albrecht was supported by EPSRC grant EP/X017524/1. The research of Mareková was supported by ZISC. The research of Ronen was supported in part by an ISF grant no. 1807/23.

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. 3

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. 3, 5, 13, 14, 15, 16, 21, 22, 31, 84, 106

AMPS23. Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for telegram. Cryptology ePrint Archive, Report 2023/469, 2023. 14, 81

BCJ$^+$24. Chris Brzuska, Cas Cremers, Håkon Jacobsen, Douglas Stebila, and Bogdan Warinschi. Falsifiability, composability, and comparability of game-based security models for key exchange protocols. Cryptology ePrint Archive, Paper 2024/1215, 2024. 31

BDPR98. Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45. Springer, Berlin, Heidelberg, August 1998. 40

BFS$^+$12. C. Brzuska, M. Fischlin, N. P. Smart, B. Warinschi, and S. Williams. Less is more: Relaxed yet composable security notions for key exchange. Cryptology ePrint Archive, Report 2012/242, 2012. 31

BFWW11. Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 51–62. ACM Press, October 2011. 6, 8, 11, 31

BR94. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Berlin, Heidelberg, August 1994. 6

BR95. Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 92–111. Springer, Berlin, Heidelberg, May 1995. 38, 40

BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006. 5

CCD$^+$23. Vincent Cheval, Cas Cremers, Alexander Dax, Lucca Hirschi, Charlie Jacomme, and Steve Kremer. Hash gone bad: Automated discovery of protocol attacks that exploit hash function weaknesses. In Calandrino and Troncoso [CT23], pages 5899–5916. 3

CT23. Joseph A. Calandrino and Carmela Troncoso, editors. *USENIX Security 2023*. USENIX Association, August 2023. 25, 26

DDGJ22. Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. On the concrete security of TLS 1.3 PSK mode. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 876–906. Springer, Cham, May / June 2022. 4, 6, 11

DFGS21. Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 34(4):37, October 2021. 4, 6, 11, 90

DFW20. Cyprien Delpech de Saint Guilhem, Marc Fischlin, and Bogdan Warinschi. Authentication in key-exchange: Definitions, relations and composition. In Limin Jia and Ralf Küsters, editors, *CSF 2020 Computer Security Foundations Symposium*, pages 288–303. IEEE Computer Society Press, 2020. 11

EM97. Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of Cryptology*, 10(3):151–162, June 1997. 5, 69, 70

FG14. Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1193–1204. ACM Press, November 2014. 4, 6, 11

FG17. Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In *2017 IEEE European Symposium on Security and Privacy*, pages 60–75. IEEE Computer Society Press, April 2017. 4, 6

FGSW16. Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *2016 IEEE Symposium on Security and Privacy*, pages 452–469. IEEE Computer Society Press, May 2016. 11

FOPS01. Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 260–274. Springer, Berlin, Heidelberg, August 2001. 40, 42

FPSZ06. Pierre-Alain Fouque, David Pointcheval, Jacques Stern, and Sébastien Zimmer. Hardness of distinguishing the MSB or LSB of secret keys in Diffie-Hellman schemes. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP 2006, Part II*, volume 4052 of *LNCS*, pages 240–251. Springer, Berlin, Heidelberg, July 2006. 102

Gün18. Felix Günther. Modeling advanced security aspects of key exchange and secure channel protocols. https://www.felixguenther.info/publications/phdthesis-felix-guenther.pdf, 2018. 6, 7, 11, 106

HGFS15. Clemens Hlauschek, Markus Gruber, Florian Fankhauser, and Christian Schanes. Prying open pandora's box: KCI attacks against TLS. In Aurélien Francillon and Thomas Ptacek, editors, *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*. USENIX Association, 2015. 11

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. 3, 4, 105

Jut00. Charanjit Jutla. Attack on free-mac, sci.crypt. https://groups.google.com/forum/#!topic/sci.crypt/4bkzm_n7UGA, Sep 2000. 43

KK04. Takeshi Koshiba and Kaoru Kurosawa. Short exponent Diffie-Hellman problems. In Feng Bao, Robert Deng, and Jianying Zhou, editors, *PKC 2004*, volume 2947 of *LNCS*, pages 173–186. Springer, Berlin, Heidelberg, March 2004. 5, 6

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. 3

LP20. Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 1839–1856. USENIX Association, August 2020. 105

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. 3

PST23. Kenneth G. Paterson, Matteo Scarlata, and Kien T. Truong. Three lessons from threema: Analysis of a secure messenger. In Calandrino and Troncoso [CT23], pages 1289–1306. 3

RSA78. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978. 40, 42

SBK$^+$17. Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 570–596. Springer, Cham, August 2017. 105

Sho02. Victor Shoup. OAEP reconsidered. *Journal of Cryptology*, 15(4):223–249, September 2002. 4, 14, 38, 40, 41, 43, 44

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. 3

Tel22a. Telegram. Android – Handshake.cpp, lines 604-611. https://github.com/DrKLO/Telegram/blob/e9a35cea54c06277c69d41b8e25d94b5d7ede065/TMessagesProj/jni/tgnet/Handshake.cpp#L604-L611, 2022. 15

Tel22b. Telegram. auth.bindTempAuthKey. https://web.archive.org/web/20221027143231/https://core.telegram.org/method/auth.bindTempAuthKey, Oct 2022. 13

Tel22c. Telegram. Creating an authorization key. https://web.archive.org/web/20230913145441/https://core.telegram.org/mtproto/auth_key, Oct 2022. 14, 18

Tel22d. Telegram. Official android client. https://github.com/DrKLO/Telegram/, 2022. 18

Tel22e. Telegram. Official desktop client. https://github.com/telegramdesktop/tdesktop/, 2022. 18

Tel22f. Telegram. Schema. https://web.archive.org/web/20221027161143/https://core.telegram.org/schema, Oct 2022. 5

Tel22g. Telegram. Two-factor authentication. https://web.archive.org/web/20221027152821/https://core.telegram.org/api/srp, Oct 2022. 13

vAP23. Theo von Arx and Kenneth G. Paterson. On the cryptographic fragility of the telegram ecosystem. In Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik, editors, *ASIACCS 23*, pages 328–341. ACM Press, July 2023. 3

A Detailed protocol figures

```
Stage 1 of $\mathsf{MTP\text{-}KE}_{2st}$ / $\boxed{\mathsf{MTP\text{-}KE}_{3st}}$

Client (knows $pk$)                                              Server (has $(pk, sk)$)

$n \leftarrow\$ \{0,1\}^{128}$             $\xrightarrow{n}$                  $n_s \leftarrow\$ \{0,1\}^{128}$
                                                                 $p', q' \leftarrow\$ \{p', q' \mid p', q'$ prime
                                                                         $\wedge\, p' \cdot q' < 2^{63}\}$
                                                                 $prod' \leftarrow p' \cdot q'$
$p', q' \leftarrow$ factor $prod'$          $\xleftarrow{n, n_s, prod', \mathcal{F}}$   $\mathcal{F} \leftarrow \{\mathsf{SHA\text{-}1}(pk)[96:160]\}$
$f \leftarrow \mathsf{SHA\text{-}1}(pk)[96:160]$
if $f \notin \mathcal{F}$ : reject
$n_n \leftarrow\$ \{0,1\}^{256}$
$\boxed{exp \leftarrow 24 \cdot 60 \cdot 60}$
$m_0 \leftarrow (prod', p', q', n,$
    $n_s, n_n, \ldots, \boxed{exp})$
$c_0 \leftarrow\$ \mathsf{TOAEP}^+.\mathsf{Enc}(pk, m_0)$   $\xrightarrow{n, n_s, p', q', f, c_0}$   $m'_0 \leftarrow \mathsf{TOAEP}^+.\mathsf{Dec}(sk, c_0)$
                                                                 if $m'_0 = \bot$ : reject
                                                                 $(prod'', p'', q'', n', n'_s, n_n,$
                                                                     $\ldots, \boxed{exp}) \leftarrow m'_0$
                                                                 $valid \leftarrow p'' \cdot q'' = prod'$
                                                                     $\wedge\, (n', n'_s) = (n, n_s)$
                                                                 if $\neg valid$ : reject
                                    $\downarrow$
                    $\mathsf{sid}.1 = (pk, n, n_s, c_0)$ ; $\mathsf{sskey}.1 = n_n$
```

Fig. 11. Stage 1 of $\mathsf{MTP\text{-}KE}_{2st}$ and $\mathsf{MTP\text{-}KE}_{3st}$ in pseudocode. $\boxed{\text{Boxed}}$ code is only relevant for $\mathsf{MTP\text{-}KE}_{3st}$. We omit displaying the TL encoding of protocol messages. Fields not relevant for the cryptographic protocol are denoted only as "...".

```
Stage 2 of $\mathsf{MTP\text{-}KE_{2st}}$ / $\boxed{\mathsf{MTP\text{-}KE_{3st}}}$

Client (knows $pk$)                                              Server (has $(pk, sk), \mathcal{S}_{\mathsf{aid}}$)
                                                                 $a \leftarrow\$ \{0,1\}^{2048}$
                                                                 $m_1 \leftarrow (n, n_s, g, p, g^a \bmod p, ...)$
                                                                 $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
$k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$            $\xleftarrow{n, n_s, c_1}$            $c_1 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_1)$
$m_1 \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_1)$
if $m_1 = \bot$ : reject
$(n', n'_s, g, p, g^a \bmod p, ...) \leftarrow m_1$
if $(n', n'_s) \neq (n, n_s)$ : reject
$rid \leftarrow 0$                                               $rid \leftarrow 0$
........................ Repeatable while $rid \leq \mathsf{rid_{max}}$ ........................
$b \leftarrow\$ \{0,1\}^{2048}$
$m_2 \leftarrow (n, n_s, rid, g^b \bmod p)$
$c_2 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_2)$            $\xrightarrow{n, n_s, c_2}$            $m_2 \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_2)$
                                                                 if $m_2 = \bot$ : reject
                                                                 $(n', n'_s, rid', g^b \bmod p) \leftarrow m_2$
                                                                 if $(n', n'_s, rid') \neq (n, n_s, rid)$ : reject
$ak \leftarrow (g^a)^b \bmod p$ ; $\boxed{ak_{\mathsf{t}} \leftarrow ak}$                  $ak \leftarrow (g^b)^a \bmod p$ ; $\boxed{ak_{\mathsf{t}} \leftarrow ak}$
$ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$                       $ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$
$aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$                    $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$
$\boxed{aid_{\mathsf{t}} \leftarrow aid}$                                   $\boxed{aid_{\mathsf{t}} \leftarrow aid}$
                                                                 if $aid \notin \mathcal{S}_{\mathsf{aid}}$ :
$h_1 \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$                                  $h \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$
                                                                   accept
                                                                 else :
$h_2 \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$                                  $h \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$
                                                                   $rid \leftarrow rid + 1$
                                                                   retry
if $h = h_1$ : accept                      $\xleftarrow{n, n_s, h}$
elseif $h = h_2$ :
  $rid \leftarrow rid + 1$
  retry
else : reject
.............................................................................................
```

$\downarrow$

$\mathsf{sid.2} = (\mathsf{sid.1}, g^a \bmod p, g^b \bmod p, h)$; $\mathsf{sskey.2} = ak[0:1024]$ $\boxed{(ak_{\mathsf{t}}[0{:}672], ak_{\mathsf{t}}[704{:}1024])}$

Fig. 12. Stage 2 of $\mathsf{MTP\text{-}KE_{2st}}$ and $\mathsf{MTP\text{-}KE_{3st}}$ in pseudocode. $\boxed{\text{Boxed}}$ code is only relevant for $\mathsf{MTP\text{-}KE_{3st}}$. We omit displaying the TL encoding of protocol messages.

Stage 3 of MTP-KE$_{3st}$

```
Client (has $ak_{v1}, aid$)
$salt \leftarrow n_n[0:64] \oplus n_s[0:64]$
$(st_{\mathcal{I}}, \_) \leftarrow$ MTP-CH.Init(sskey.2, $(aid_t, salt)$)
$(\_, \_, \_, st_{ME,\mathcal{I}}) \leftarrow st_{\mathcal{I}}$ ; $(sid_t, \_, \_) \leftarrow st_{ME,\mathcal{I}}$
$n_b \leftarrow\$ \{0,1\}^{64}$
$exp \leftarrow$ CurrentTime() $+ 24 \cdot 60 \cdot 60$
$m_{in} \leftarrow (n_b, aid_t, aid, sid_t, exp)$
$mid \leftarrow\$ \{0,1\}^{64}$
$c_{in} \leftarrow$ CHv1.Enc($ak_{v1}$, $(mid, m_{in})$)
$m_{bind} \leftarrow (aid, n_b, exp, c_{in})$
$st_{\mathcal{I}} \leftarrow$ SetNextMsgId($st_{\mathcal{I}}, mid$)
$(st_{\mathcal{I}}, c_{bind}) \leftarrow$ MTP-CH.Send($st_{\mathcal{I}}, m_{bind}, \varepsilon$)

                $\xrightarrow{c_{bind}}$

Server (has $T_{sym}$)
$salt \leftarrow n_n[0:64] \oplus n_s[0:64]$
$(\_, st_{\mathcal{R}}) \leftarrow$ MTP-CH.Init(sskey.2, $(aid_t, salt)$)
$(st_{\mathcal{R}}, m_{bind}) \leftarrow$ MTP-CH.Recv($st_{\mathcal{R}}, c_{bind}, \varepsilon$)
$(\_, \_, \_, st_{ME,\mathcal{R}}) \leftarrow st_{\mathcal{R}}$ ; $(sid_t, \_, \_) \leftarrow st_{ME,\mathcal{R}}$
if $m_{bind} = \perp$ : reject
$mid \leftarrow$ GetLastMsgId($st_{\mathcal{R}}$)
$(aid, n_b, exp, c_{in}) \leftarrow m_{bind}$
if $exp \geq$ CurrentTime() : reject
if $T_{sym}[aid] = \perp$ : reject
$(\_, ak_{v1}) \leftarrow T_{sym}[aid]$
$(mid', m_{in}) \leftarrow$ CHv1.Dec($ak_{v1}, c_{in}$)
if $m_{in} = \perp$ : reject
if $mid' \neq mid$ : reject
$(n'_b, aid'_t, aid', sid'_t, exp') \leftarrow m_{in}$
if $n'_b \neq n_b \vee aid'_t \neq aid_t \vee aid' \neq aid$
    $\vee\, sid'_t \neq sid_t \vee exp' \neq exp$ : reject
$m_{true} \leftarrow$ true
$(st_{\mathcal{R}}, c_{true}) \leftarrow$ MTP-CH.Send($st_{\mathcal{R}}, m_{true}, \varepsilon$)

                $\xleftarrow{c_{true}}$

Client
$(st_{\mathcal{I}}, m_{true}) \leftarrow$ MTP-CH.Recv($st_{\mathcal{I}}, c_{true}, \varepsilon$)
if $m_{true} \neq$ true :
    reject
```

$\downarrow$

sid.3 = (sid.2, aid, sid_t) ; sskey.3 = sskey.2

Fig. 13. Stage 3 of MTP-KE$_{3st}$ in pseudocode. We omit displaying the TL encoding of protocol messages. Note that SetNextMsgId abstracts away updating the channel state so that the next invocation of MTP-CH.Send uses the given message id, and GetLastMsgId extracts it from the receiver's state.

B Model details

B.1 Further differences

Following from Section 4.5, here we describe several additional differences of our model compared to the Telegram implementations.

Fixed Diffie-Hellman parameters. We assume that the server always chooses the same group $\mathbb{G}$ for DDH. To the best of our knowledge, this is true to practice, however it is possible the server could use the flexibility of the protocol to give different parameters to different clients. However, the impact of this should be limited as long as the clients check the parameters as prescribed.

Keeping past sessions in memory. In our model, values are never removed from the set $\mathcal{S}_{\mathsf{aid}}$. This is likely not true in practice since temporary keys normally expire after 24 hours. However, we do not have visibility into how Telegram servers manage their stores of current or past sessions, and modelling that sessions can be forgotten after arbitrary conditions would add undue complexity.

Time synchronisation. We implicitly assume that client and server time is synchronised. In practice, Telegram client implementations use the *servertime* timestamp received from the server in Step 1 of $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ to adjust their value of $\mathsf{CurrentTime}()$ computed in stage 3.

Changing of session ids and server salts. Telegram "sessions" do not map neatly to runs of the key exchange protocol. Our model does not capture this. In a normal execution, a run of $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$ would be composed with the channel $\mathsf{MTP\text{-}CH}$ using a fixed "session id" (a field set by the client in each message[33]) and a fixed server salt. We do capture this behaviour, however we do not model that during the execution of the channel, the session identifier and the server salt may change value without a new run of $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$. Correctly capturing these changes would require modifications to the model of the channel itself.

B.2 Modelling choices

Here we expand on our reasoning for some of the choices made in defining the model for MTProto.

Session identifiers. To ensure consistency of the computed session keys (to satisfy the soundness predicate), the session identifiers must depend on n_n (through c_0) as well as $g^a \bmod p, g^b \bmod p$. Further, the session identifiers must be such that a reduction from an adversary in the $\mathsf{Multi\text{-}Stage}$ game that makes multiple TEST queries to an adversary that makes a single TEST query (which is the first step in our proof) can compute which sessions are partnered at the end of each stage from the observed queries of the original multi-TEST adversary. This means that n_n cannot be included in *sid*.1 in plaintext form, because the constructed single-TEST adversary cannot compute this value (since it is also the stage-1 session key) but without it there is not enough information in *sid*.1 to compute which sessions are partnered (i.e. using only the values (pk, n, n_s) is not enough).

For the second stage, we face a different problem. The shares $g^a, g^b \bmod p$ are transmitted in encrypted form, but they cannot be fully part of the session identifier in ciphertext form because the encryption scheme $\mathsf{HtE\text{-}SE}$ is not IND-CCA secure, permitting the adversary to re-randomise the ciphertexts (either by appending a random block at the end, or by changing the last block to brute-force the eight or four bytes of plaintext in it).[34] This is an issue because the adversary could cause sessions which derive the same key to not be considered partnered, which would allow it to call REVEAL on one session and TEST

[33] Not to be confused with the session identifier used to define the notion of partnering.

[34] Note that it could be possible to use a prefix of the ciphertext (without the last block) as a public session identifier, as it is only the last block that can be re-randomised, and the prefix would be bound to a large portion of g^a and g^b; however, this would result in a more complex proof.

on the other without setting the *lost* flag, thus winning the Multi-Stage game. Thus, $g^a, g^b \bmod p$ are part of the session identifier in plaintext form.[35]

Session keys. As we hint in Section 4.3, the question of which part of the agreed key material should be modelled as the session key for each stage does not have a clear answer in MTProto. This has multiple causes. First, consider defining the stage 2 session key to be ak for $\mathsf{MTP\text{-}KE_{2st}}$ (or ak_t for $\mathsf{MTP\text{-}KE_{3st}}$). An adversary that corrupts the server's RSA keypair and tests a completed session at stage 2 (which is admissible, since we target forward secrecy) could obtain n_n, trivially recompute the value of h and check the stage 2 messages for a match. We can avoid this key confirmation attack by defining the stage 2 session key to be only the part of ak that is used later, i.e. $ak[0:1024]$ for $\mathsf{MTP\text{-}KE_{2st}}$ (similarly for $\mathsf{MTP\text{-}KE_{3st}}$).

Similarly, consider what should be the session key output of stage 3 in $\mathsf{MTP\text{-}KE_{3st}}$. The same channel which has already been instantiated using the session key output of stage 2 will continue running after the key exchange. However, there is no key refresh step, so stage 3 effectively outputs the same session key as stage 2. This causes another trivial key confirmation attack, as the adversary can simply check whether the session key it is given had been used to instantiate the channel. Prior work has tackled the issue by excluding such steps from analysis, adding artificial key refresh steps or performing analysis in a monolithic model which includes composition with the channel (see [BFS$^+$12]). Neither of these appear to be satisfactory approaches for our use case. Since the only purpose of stage 3 in $\mathsf{MTP\text{-}KE_{3st}}$ is to provide client authentication, we forbid the adversary from testing the stage 3 session key directly but enable it to win the game if it can break authentication. This is, of course, a weakening of the model, but it appears that stronger security is not achievable without changes to the protocol.

C New security assumptions

In this section, we define a total of five new security games and related assumptions which are needed in our main proofs for MTProto. These all stem from MTProto's nonstandard use of certain primitives, in particular of SHA-1. In each case we give a formal game definition and define the adversary's advantage. Since our final security bounds for MTProto are stated concretely in terms of these advantages, we omit formal statements of security assumptions relating to these games.

Our belief is that all five of these advantages are small for all adversaries consuming "reasonable" resources. In each case, we have given some informal reasoning for holding this view. Our belief could be invalidated by cryptanalysis, and our notions certainly warrant further study. We have tried to minimise the number and complexity of new notions, but we emphasise that we do need all these notions for our main proofs to go through. Reducing the number or relating them to more standard assumptions is a challenge for future work. We also stress that invalidating one or more of the five assumptions may not directly lead to an attack on MTProto, since it may still be possible to prove these protocols secure under weaker assumptions. That is, our assumptions are collectively sufficient, but may not all be necessary, to obtain a proof.

C.1 4PRF: SHACAL-1 as a "four-way" PRF with leakage

The assumption below entails PRF-like security for SHACAL-1 in a setting where the adversary has some control and knowledge of the 512-bit key: specifically, it can control 128 bits in various different positions and it knows the last 128 bits which are a fixed value; the remaining 256 bits are chosen at random. However, in this game SHACAL-1 is only evaluated on a fixed plaintext $\mathtt{IV}_{160}$, the IV value used in SHA-1. This assumption is similar to the SHACAL-2 assumptions used in the proof of the MTProto 2.0 channel [AMPS22].

[35] The use of plaintexts in *sid*.2 combined with the re-randomisation property normally prevents the existence of a public session matching algorithm, which is needed to make use of general composition results [BFWW11]. However, recent work [BCJ$^+$24] suggests that an alternative proof strategy may circumvent this requirement.

Definition 4. *Consider the game* $G^{4PRF}_{\mathsf{SHACAL\text{-}1},\mathcal{D}}$ *in Fig. 14 with the block cipher* SHACAL-1*, and an adversary* $\mathcal{D}$*. The advantage of* $\mathcal{D}$ *in breaking the* 4PRF*-security of* SHACAL-1 *is defined as* $\mathsf{Adv}^{4PRF}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}) := 2 \cdot \Pr\left[G^{4PRF}_{\mathsf{SHACAL\text{-}1},\mathcal{D}}\right] - 1$.

```
Game $G^{4PRF}_{\mathsf{SHACAL\text{-}1},\mathcal{D}}$
$b \leftarrow\$ \{0,1\}$
$\mathsf{K} \leftarrow []$
$ak_{\mathsf{v1}} \leftarrow\$ \{0,1\}^{1024}$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{ROR}}()$
return $b' = b$

$\mathrm{ROR}(msk, type)$    // $|msk| = 128, type \in \{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}\}$
if $type = \mathtt{A}$:
    $k \leftarrow msk \,\|\, ak_{\mathsf{v1}}[0:256]$
if $type = \mathtt{B}$:
    $k \leftarrow ak_{\mathsf{v1}}[256:384] \,\|\, msk \,\|\, ak_{\mathsf{v1}}[384:512]$
if $type = \mathtt{C}$:
    $k \leftarrow ak_{\mathsf{v1}}[512:768] \,\|\, msk$
if $type = \mathtt{D}$:
    $k \leftarrow msk \,\|\, ak_{\mathsf{v1}}[768:1024]$
$y_1 \leftarrow \mathsf{SHACAL\text{-}1.Ev}(k \,\|\, \mathtt{pad}, \mathtt{IV}_{160})$
if $\mathsf{K}[msk, type] = \perp$:
    $\mathsf{K}[msk, type] \leftarrow\$ \{0,1\}^{160}$
$y_0 \leftarrow \mathsf{K}[msk, type]$
return $y_b$
```

Fig. 14. Pseudorandomness of SHACAL-1 with leakage in four modes, where pad is fixed SHA-1 padding for a message of length 384 and $\mathtt{IV}_{160}$ is the initial state for SHA-1.

C.2 3TPRF: SHACAL-1 as a "three-time" PRF with leakage

The assumption defined here appears similar to the 4PRF notion from the previous section, in that it again speaks to pseudorandomness of SHACAL-1 with leakage, however it is more restricted. The game below is more akin to a "one-time" notion, because the adversary only gets to make a single ROR query per random key, but this query makes three SHACAL-1 invocations for different arrangements of the input x and the key k (in the 4PRF game, this type of behaviour would be captured by ROR queries for different *msk* but the same *type*). Finally, since this notion will be used in a proof for SKDF, the ROR oracle also directly leaks the first 32 bits of the key k – this implies that any notion allowing multiple ROR queries for the same key k would be trivially broken.

Definition 5. *Consider the game* $G^{3TPRF}_{\mathsf{SHACAL\text{-}1},\mathcal{D}}$ *in Fig. 15 with the block cipher* SHACAL-1*, and an adversary* $\mathcal{D}$*. The advantage of* $\mathcal{D}$ *in breaking the* 3TPRF*-security of* SHACAL-1 *is defined as* $\mathsf{Adv}^{3TPRF}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}) := 2 \cdot \Pr\left[G^{3TPRF}_{\mathsf{SHACAL\text{-}1},\mathcal{D}}\right] - 1$.

C.3 SPR: Sampler-based second-preimage resistance of SHA-1

Message sampler. Before defining the next assumption, we first define a stateful *message sampling* function. We assume that the initial state of the message sampler is always $st = \varepsilon$. The message sampler in Fig. 16 models the fact that a number of retries could happen in the key exchange protocol. A server can send a single message to the client first, and then the client can send an unlimited number of messages to the server in response (though in practice this would be constrained by $\mathsf{rid}_{\mathsf{max}} + 1$). The sampler will

```
Game $G^{3TPRF}_{SHACAL-1,\mathcal{D}}$
    $b \leftarrow\$ \{0,1\}$
    $b' \leftarrow\$ \mathcal{D}^{ROR}()$
    return $b' = b$

ROR($x$)    // $|x| = 128$
    $k \leftarrow\$ \{0,1\}^{256}$
    $r_0 \leftarrow \mathsf{SHACAL\text{-}1.Ev}(k \| x \| \mathtt{pad}, \mathtt{IV}_{160})$
    $r_1 \leftarrow \mathsf{SHACAL\text{-}1.Ev}(x \| k \| \mathtt{pad}, \mathtt{IV}_{160})$
    $r_2 \leftarrow \mathsf{SHACAL\text{-}1.Ev}(k \| k, \mathtt{IV}_{160})$
    $r'_0 \leftarrow\$ \{0,1\}^{160}$ ; $r'_1 \leftarrow\$ \{0,1\}^{160}$ ; $r'_2 \leftarrow\$ \{0,1\}^{160}$
    $y_1 \leftarrow (r_0, r_1, r_2)$
    $y_0 \leftarrow (r'_0, r'_1, r'_2)$
    return $k[0:32], y_b$
```

Fig. 15. Three-time pseudorandomness of SHACAL-1 with leakage, where pad is fixed SHA-1 padding for a message of length 384 and $\mathtt{IV}_{160}$ is the initial state for SHA-1.

thus later determine the possible inputs to an encryption oracle in a notion of plaintext integrity (see Appendix F.1). Note that the sampler generates Diffie-Hellman shares, but outputs the secret alongside the message containing the share; this is because at this point we are not concerned with safeguarding these secrets but rather ensuring their integrity.

Definition 6. *A message sampler* Samp $=$ SAMP$[n, n_s, g, p]$ *is a stateful algorithm defined in Fig. 16, parametrised by the nonces n, n_s and the Diffie-Hellman parameters g, p.* Samp *takes $aux \in \{0,1\}^*$ as input and outputs the generated messages m, x.*

```
Samp($st, aux$)    // $|aux| = 32$ if $st = \varepsilon$, else $|aux| = 64$
    if $st = \varepsilon$ : $st \leftarrow$ "server"
    $x \leftarrow\$ \{0,1\}^{2048}$
    if $st =$ "server" :
        $servertime \leftarrow aux$
        $m \leftarrow \mathsf{TL}(\mathtt{server\_DH\_inner\_data}, n, n_s, g, p, g^x \bmod p, servertime)$
        $st \leftarrow$ "client"
    else :    // $st=$ "client"
        $rid \leftarrow aux$
        $m \leftarrow \mathsf{TL}(\mathtt{client\_DH\_inner\_data}, n, n_s, rid, g^x \bmod p)$
    return $(st, m, x)$
```

Fig. 16. Message sampler Samp $=$ SAMP$[n, n_s, g, p]$ for nonces n, n_s, and for DH parameters g, p.

Definition of SPR. In the following assumption, for some message m sampled by the message sampler Samp we require $\mathcal{A}$ to find another message m^* such that the SHA-1 hashes of these two messages match in the first 128 bits. This definition appears similar to second-preimage resistance, however it does not match perfectly for several reasons. First, the messages are not sampled randomly, but according to Samp – the messages have a common, fixed prefix[36] but always contain a large, randomly-derived value of the form $g^x \bmod p$. Second, the adversary is given a small amount of control over the message in the form of

[36] Though the first, "server", message has a different header than any following, "client", messages.

an *aux* value, which is either 32 bits or 64 bits in size. Finally, the adversary can make an arbitrary number of queries to the oracle NEWMSG, however as used in the key exchange protocol, the number of queries would be upper-bounded by $\mathsf{rid}_{\mathsf{max}} + 1$.

We argue informally that despite these differences, breaking SPR would require breaking second-preimage resistance of SHA-1, as there is sufficient randomness in each message and the degree of freedom the adversary has over the input is very small (especially when compared to what is normally required to produce SHA-1 collisions). The loss would be dependent on the size of $\mathcal{M}$, however the value $\mathsf{rid}_{\mathsf{max}} = 32$ suggested for the key exchange protocol would provide a reasonable limit. Note that we do not constrain the adversary on the format of the message m^*, though in practice this would be necessary to make use of breaking the assumption in the context of the key exchange protocol.

Definition 7. *Consider the game* $\mathrm{G}^{\mathsf{SPR}}_{\mathsf{SHA\text{-}1},\mathsf{Samp},\mathcal{A}}$ *in Fig. 17 with the hash function* SHA-1*, the message sampler* Samp *(Fig. 16), and an adversary* $\mathcal{A}$*. The advantage of* $\mathcal{A}$ *in breaking the* SPR*-security of* SHA-1 *with respect to* Samp *is defined as* $\mathsf{Adv}^{\mathsf{SPR}}_{\mathsf{SHA\text{-}1},\mathsf{Samp}}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{SPR}}_{\mathsf{SHA\text{-}1},\mathsf{Samp},\mathcal{A}}\right]$.

```
Game G^SPR_{SHA-1,Samp,A}
  $st \leftarrow \varepsilon$ ; $\mathcal{M} \leftarrow \emptyset$ ; $(m, m^*) \leftarrow\$ \mathcal{A}^{\text{NEWMSG}}$
  $p_1 \leftarrow \mathsf{SHA\text{-}1}(m)[0:128]$
  $p_1^* \leftarrow \mathsf{SHA\text{-}1}(m^*)[0:128]$
  return $(m \in \mathcal{M}) \wedge (m \neq m^*) \wedge (p_1 = p_1^*)$

NEWMSG(aux)    // $|aux| = 32$ or $64$
  $(st, m, x) \leftarrow\$ \mathsf{Samp}(st, aux)$
  if $m = \bot$ : return $\bot$
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
  return $(m, x)$
```

Fig. 17. Samp-based second-preimage resistance of truncated SHA-1.

C.4 UPCR: Unpredictable-prefix collision resistance of SHA-1

The assumption below reflects the way SHA-1 is used in CHv1 to compute *msk*, which we have abstracted as a function Hv1 that first encodes the given r, mid, m for the use by the MTProto 1.0 channel before calling SHA-1. Since we cannot rely on plain collision resistance, the UPCR notion narrows the setting to something more akin to second-preimage resistance, which is still believed to be hard for SHA-1. In more detail, the assumption requires the adversary to first commit to a set of messages, each of which is assigned a random value r that is then revealed to the adversary.[37] Only then can it submit a colliding pair, which must be such that m^* is new and msk^* was the result of one of the previous evaluations.

Breaking this notion would not necessarily lead to breaking the unforgeability of CHv1 (see Appendix G). To make use of a UPCR colliding pair, an adversary would be much more restricted than in the game shown in Fig. 18. This is because in CHv1, the value r corresponding to each EVAL call is never directly revealed to the adversary; though, since it is encapsulated in a ciphertext block, an adversary could in theory produce a colliding pair without knowledge of r simply by reusing the first ciphertext block. For CHv1 decryption to succeed, there would be a remaining obstacle in the form of decoding checks. Since SEv1 is built on AES-256-IGE, it can in principle be malleable. However, the adversary would need to construct four ciphertext blocks such that they would decrypt to the values mid^*, m^* from the colliding pair (in addition to all of the fixed plaintext fields). Hence, in an attack control over the colliding pair would exist only in a trade-off with control over the ciphertext blocks.

Definition 8. *Consider the game* $\mathrm{G}^{\mathsf{UPCR}}_{\mathsf{Hv1},\mathcal{A}}$ *in Fig. 18 with the hash function* Hv1 *defined in Fig. 8, and an adversary* $\mathcal{A}$*. The advantage of* $\mathcal{A}$ *in breaking the* UPCR*-security of* Hv1 *is defined as* $\mathsf{Adv}^{\mathsf{UPCR}}_{\mathsf{Hv1}}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{UPCR}}_{\mathsf{Hv1},\mathcal{A}}\right]$.

[37] If the random value r was instead part of input to EVAL, and (r, mid, m) was tracked by $\mathcal{M}$ instead of just m, we would recover a notion close to the standard definition of collision resistance.

```
Game $G^{\mathsf{UPCR}}_{\mathsf{Hv1},\mathcal{A}}$
    $\mathcal{M} \leftarrow \emptyset$ ; $\mathcal{H} \leftarrow \emptyset$
    $r^*, mid^*, m^* \leftarrow\$ \mathcal{A}^{\mathrm{EVAL}}()$
    $msk^* \leftarrow \mathsf{Hv1.Ev}(r^*, mid^*, m^*)$
    return $m^* \notin \mathcal{M} \wedge msk^* \in \mathcal{H}$

EVAL$(mid, m)$    // $|mid| = 64, |m| = 288$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
    $r \leftarrow\$ \{0,1\}^{128}$
    $msk \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{msk\}$
    return $r, msk$
```

Fig. 18. Unpredictable-prefix collision resistance of Hv1.

C.5 IND-KEY: Indistinguishability of key reuse between SKDF and NH

Below, we define a notion to capture the requirement put on SKDF with respect to key reuse in $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$. This is because n_n, the key used by SKDF, is later used to "key" possibly several additional NH calls on partially adversarially-controlled input, and both functions are built using SHA-1. In the game, the adversary is thus given a task to distinguish whether key reuse has taken place, given the output of SKDF as well as access to an evaluation oracle for NH.[38]

Note that in the IND-KEY game, the adversary is directly given the first 32 bits of n_n, which it can learn from the value y. However, that still leaves 224 bits that are unknown, which could be viewed to "key" the NH calls. Though both SKDF and NH output SHA-1 values, there is implicit domain separation in that the length of the input is different, and so different (but fixed) SHA-1 padding is applied. This means that the adversary cannot force a collision in the inputs. However, it is possible that it could use a feature of SHACAL-1 to distinguish the two cases, since it knows more than half of the bits that form the "real" SHACAL-1 keys.

Definition 9. *Consider the game $G^{\mathsf{IND\text{-}KEY}}_{\mathsf{SKDF},\mathsf{NH},\mathcal{D}}$ in Fig. 19 with the function families* SKDF *and* NH *defined in Section 4.2, and an adversary $\mathcal{D} = (\mathcal{D}_1, \mathcal{D}_2)$. The advantage of $\mathcal{D}$ in breaking the* IND-KEY*-security of* SKDF *and* NH *is defined as* $\mathsf{Adv}^{\mathsf{IND\text{-}KEY}}_{\mathsf{SKDF},\mathsf{NH}}(\mathcal{D}) := 2 \cdot \Pr\left[G^{\mathsf{IND\text{-}KEY}}_{\mathsf{SKDF},\mathsf{NH},\mathcal{D}}\right] - 1$.

```
Game $G^{\mathsf{IND\text{-}KEY}}_{\mathsf{SKDF},\mathsf{NH},\mathcal{D}}$
    $b \leftarrow\$ \{0,1\}$
    $n_n \leftarrow\$ \{0,1\}^{256}$ ; $n'_n \leftarrow\$ \{0,1\}^{256}$
    $(n_s, st_{\mathcal{D}}) \leftarrow\$ \mathcal{D}_1()$
    $y \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
    $b' \leftarrow \mathcal{D}_2^{\mathrm{EVAL}}(st_{\mathcal{D}}, y)$
    return $b' = b$

EVAL$(x, i)$    // $i \in \{1,2\}, |x| = 64$
    $h_1 \leftarrow \mathsf{NH.Ev}(n_n, x, i)$
    $h_0 \leftarrow \mathsf{NH.Ev}(n'_n, x, i)$
    return $h_b$
```

Fig. 19. Indistinguishability of key reuse between SKDF and NH.

D Analysis of Telegram-OAEP+ public-key encryption scheme

In this section, we use "**require** *condition*" as shorthand for "**if** $\neg$*condition* : **return** $\perp$".

[38] Note that this equips the adversary with more power than it would necessarily have in $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$, so a break of this property does not immediately translate to an attack on the protocol.

D.1 Birthday bound

```
Game $G^{\text{birthday-experiment}}_{p,a,i,N}$
$S \leftarrow \emptyset$
PopulateSet$(p)$
for $\tau = 0, \ldots, a-1$:
    AttemptACollision$()$
    PopulateSet$(i)$
return false

Function AttemptACollision$()$
$e \leftarrow\$ \{0, 1, \ldots, N-1\}$
if $e \in S$ : abort(true)
$S \leftarrow S \cup \{e\}$

Function PopulateSet(count)
for $i = 0, \ldots, \mathsf{count} - 1$:
    $e \leftarrow\$ \{0, 1, \ldots, N-1\} \setminus S$
    $S \leftarrow S \cup \{e\}$
```

Fig. 20. Extended birthday experiment.

Consider game $G^{\text{birthday-experiment}}_{p,a,i,N}$ of Fig. 20, defined for integers $p, a, i, N \geq 0$. If $p = i = 0$, then this game captures the standard birthday experiment, in which "a" elements are sampled one-by-one, uniformly at random, from a set of size N. The sampled elements are stored in set S. The game returns `true` iff a collision occurs. If $p > 0$, then the game populates the set S with p unique elements prior to starting the standard birthday experiment. If $i > 0$, then every time a new element is sampled and added to the set S within the scope of the standard birthday experiment, the game subsequently expands S with i more unique elements (that were not yet in S). We define function

$$\mathsf{\underline{b}irthday\text{-}\underline{b}ound(\underline{p}rior\text{-}population, \underline{a}ttempts, \underline{i}ncrement\text{-}per\text{-}attempt, output\text{-}space\ \underline{N})}$$

and abbreviate it with $\mathsf{bb}(p, a, i, N)$. Let $\mathsf{bb}(p, a, i, N) = \Pr[G^{\text{birthday-experiment}}_{p,a,i,N}]$ be the probability of obtaining a collision in our birthday game. We upper bound it as follows:

$$\mathsf{bb}(p, a, i, N) \leq \sum_{x=1}^{a} \frac{p + (x-1) \cdot (i+1)}{N} = \frac{a \cdot (2 \cdot p + (a-1) \cdot (i+1))}{2N}.$$

The standard birthday bound can be recovered as $\mathsf{bb}(0, a, 0, N) \leq \frac{a \cdot (a-1)}{2N}$. Note that function PopulateSet in game $G^{\text{birthday-experiment}}_{p,a,i,N}$ chooses unique elements by sampling them at random. The randomness here is not necessary; any method of choosing unique elements could have been used.

We rely on our birthday bound even in cases where an adversary can replace the implementation of PopulateSet(count) with an arbitrary method of adding count elements to the set S (e.g. adding random elements that are not necessarily unique). Our birthday bound is still applicable in such cases, because choosing unique elements that are not in S (as done in $G^{\text{birthday-experiment}}_{p,a,i,N}$) yields the best possible upper bound.

D.2 Standard definitions

Public-key encryption schemes. A public-key encryption scheme PKE specifies algorithms PKE.KGen, PKE.Enc and PKE.Dec, where PKE.Dec is deterministic. Associated to PKE is a message space PKE.MS. The key generation algorithm PKE.KGen returns a key pair (pk, sk), where pk is a public key and sk is a secret key. The encryption algorithm PKE.Enc takes pk and a message $m \in$ PKE.MS to return a ciphertext c. The decryption algorithm PKE.Dec takes sk, c to return $m \in \mathsf{PKE.MS} \cup \{\perp\}$, where $\perp$ denotes incorrect decryption. Decryption correctness requires that $\mathsf{PKE.Dec}(sk, c) = m$ for all $(pk, sk) \in [\mathsf{PKE.KGen}()]$, all $m \in$ PKE.MS, and all $c \in [\mathsf{PKE.Enc}(pk, m)]$.

IND-CCA security of PKE. Consider game $\mathrm{G}^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{D}_{\mathsf{IND\text{-}CCA}}}$ of Fig. 21, defined for a public-key encryption scheme PKE and an adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$. The advantage of $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ in breaking the IND-CCA security of PKE is defined as $\mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{PKE}}(\mathcal{D}_{\mathsf{IND\text{-}CCA}}) = 2 \cdot \Pr[\mathrm{G}^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{D}_{\mathsf{IND\text{-}CCA}}}] - 1$. Note that the game uses variable c^*_{rsa} to ensure that $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ makes at most a single query to its encryption oracle ENC, and that the decryption oracle DEC does not subsequently accept the challenge ciphertext as input.

```
Game $\mathrm{G}^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{D}_{\mathsf{IND\text{-}CCA}}}$
$b \leftarrow\$ \{0,1\}$ ; $c^*_{\mathsf{rsa}} \leftarrow \bot$
$(pk, sk) \leftarrow\$ \mathsf{PKE.KGen}()$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{ENC},\mathrm{DEC}}_{\mathsf{IND\text{-}CCA}}(pk)$
return $b = b'$

Oracle ENC$(m_0, m_1)$
require $(c^*_{\mathsf{rsa}} = \bot) \wedge (|m_0| = |m_1|)$
require $m_0, m_1 \in \mathsf{PKE.MS}$
$c \leftarrow\$ \mathsf{PKE.Enc}(pk, m_b)$
$c^*_{\mathsf{rsa}} \leftarrow c$
return $c$

Oracle DEC$(c)$
require $c \neq c^*_{\mathsf{rsa}}$
$m \leftarrow \mathsf{PKE.Dec}(sk, c)$
return $m$
```

Fig. 21. IND-CCA security of public-key encryption scheme PKE.

Trapdoor function families. A trapdoor function family TDF specifies algorithms TDF.KGen, TDF.Ev and TDF.Inv, where TDF.Ev and TDF.Inv are deterministic. Associated to TDF is a domain TDF.Dom. The key generation algorithm TDF.KGen returns (fk, tk), where fk is a function key and tk is a trapdoor key. The evaluation algorithm TDF.Ev takes fk and an input $x \in \mathsf{TDF.Dom}$ to return an output $y \in \mathsf{TDF.Dom}$. The inversion algorithm TDF.Inv takes tk, y to return $x \in \mathsf{TDF.Dom} \cup \{\bot\}$, where $\bot$ denotes incorrect inversion. Inversion correctness requires that $\mathsf{TDF.Inv}(tk, \mathsf{TDF.Ev}(fk, x)) = x$ for all $(fk, tk) \in [\mathsf{TDF.KGen}()]$ and all $x \in \mathsf{TDF.Dom}$.

One-wayness of TDF. Consider game $\mathrm{G}^{\text{OW}}_{\mathsf{TDF},\mathcal{F}_{\mathsf{OW}}}$ of Fig. 22, defined for a trapdoor function family TDF and an adversary $\mathcal{F}_{\mathsf{OW}}$. The advantage of $\mathcal{F}_{\mathsf{OW}}$ against the one-wayness of TDF is defined as $\mathsf{Adv}^{\text{OW}}_{\mathsf{TDF}}(\mathcal{F}_{\mathsf{OW}}) = \Pr[\mathrm{G}^{\text{OW}}_{\mathsf{TDF},\mathcal{F}_{\mathsf{OW}}}]$.

```
Game $\mathrm{G}^{\text{OW}}_{\mathsf{TDF},\mathcal{F}_{\mathsf{OW}}}$
$(fk, tk) \leftarrow\$ \mathsf{TDF.KGen}()$
$x \leftarrow\$ \mathsf{TDF.Dom}$
$y \leftarrow \mathsf{TDF.Ev}(fk, x)$
$x' \leftarrow\$ \mathcal{F}_{\mathsf{OW}}(fk, y)$
return $x = x'$
```

Fig. 22. One-wayness of trapdoor function family TDF.

RSA key generators. An RSA key generator RSA.KGen is a randomised algorithm that returns integers N, p, q, e, d. Here, p, q are primes, $N = p \cdot q$ is an RSA modulus, and $e, d \in \mathbb{Z}^*_{\phi(N)}$ are encryption and decryption exponents such that $e \cdot d \bmod \phi(N) = 1$. The outputs of an RSA key generator can be used to define the RSA function $f(x) := x^e \bmod N$ and its inverse $g(y) := y^d \bmod N$, where $x, y \in \mathbb{Z}^*_N$. Note that we could have modelled this pair of functions as a trapdoor function family (modifying our definition of TDFs to allow fk-parameterised domains). We will use RSA only to define and analyse Telegram's OAEP+ encryption scheme. Our treatment of Telegram's scheme is meant to be descriptive rather than prescriptive, so we prefer to avoid introducing an extra layer of abstraction.

One-wayness of RSA. Consider game $G^{\text{OW-RSA}}_{\mathsf{RSA.KGen},\mathcal{F}_{\mathsf{OWRSA}}}$ of Fig. 23, defined for an RSA key generator $\mathsf{RSA.KGen}$ and an adversary $\mathcal{F}_{\mathsf{OWRSA}}$. The advantage of $\mathcal{F}_{\mathsf{OWRSA}}$ against the one-wayness of RSA is defined as $\mathsf{Adv}^{\text{OW-RSA}}_{\mathsf{RSA.KGen}}(\mathcal{F}_{\mathsf{OWRSA}}) = \Pr[G^{\text{OW-RSA}}_{\mathsf{RSA.KGen},\mathcal{F}_{\mathsf{OWRSA}}}]$. More precisely, this game defines one-wayness of the RSA function that is defined based on the outputs of the RSA key generator $\mathsf{RSA.KGen}$. We require one-wayness with respect to inputs from $\mathbb{Z}_N$ rather than $\mathbb{Z}_N^*$; this is a common requirement.

```
Game $G^{\text{OW-RSA}}_{\mathsf{RSA.KGen},\mathcal{F}_{\mathsf{OWRSA}}}$
$(N, p, q, e, d) \leftarrow\$ \mathsf{RSA.KGen}()$
$pk \leftarrow (N, e)$
$z \leftarrow\$ \{0, \ldots, N-1\}$
$c_{\mathsf{rsa}} \leftarrow z^e \bmod N$
$z' \leftarrow\$ \mathcal{F}_{\mathsf{OWRSA}}(pk, c_{\mathsf{rsa}})$
**return** $z = z'$
```

Fig. 23. One-wayness of RSA.

D.3 Shoup's public-key encryption scheme OAEP+

In Appendix D.4 we define and analyse the public-key encryption scheme that is used in Telegram's MTProto. Telegram's public-key encryption scheme can be seen (and is reported by Telegram) as a variant of Shoup's OAEP+ scheme [Sho02], so we call it $\mathsf{TELEGRAM\text{-}OAEP}^+$. The OAEP+ scheme builds an IND-CCA secure PKE from a one-way trapdoor function in the random oracle model. In this section, we define the OAEP+ scheme, and provide a high-level intuition behind its security proof. Our security analysis of the $\mathsf{TELEGRAM\text{-}OAEP}^+$ scheme in Appendix D.4 will rely on similar high-level intuition, although a lot of details will differ.

PKE scheme $\mathsf{PKE\text{-}BLUEPRINT}$. As a warmup, consider the public-key encryption scheme that could be defined based on the encryption algorithm $\mathsf{PKE\text{-}BLUEPRINT.Enc}$ in Fig. 24. Given a one-way function f with the corresponding trapdoor function g, such a PKE scheme would use f as its public key and g as its secret key. It would employ a symmetric encryption scheme SE (with key length $\mathsf{SE.kl}$), and a hash function H. We can instantiate $\mathsf{PKE\text{-}BLUEPRINT}$ with different choices of (f, g), SE, H to recover Bellare-Rogaway's OAEP scheme [BR95], Shoup's OAEP+ scheme [Sho02], or Telegram's $\mathsf{TELEGRAM\text{-}OAEP}^+$ scheme. However, some high-level security goals can be established even at the current level of abstraction.

Algorithm $\mathsf{PKE\text{-}BLUEPRINT.Enc}$ uses a one-time key k_{se} for SE, i.e. it samples a fresh uniformly random key for every encryption operation. It creates an SE ciphertext c_{se} that encrypts x. It then creates a string $\tilde{\kappa}$ that attempts to mask the one-time key k_{se} by XOR-ing it with the ciphertext's hash $\mathsf{H}(c_{\mathsf{se}})$. The scheme returns the result of applying the one-way function f to the concatenation $c_{\mathsf{se}} \parallel \tilde{\kappa}$. In order to benefit from the one-wayness of f, both c_{se} and $\tilde{\kappa}$ need to look uniformly random. This means that the ciphertexts of the symmetric encryption scheme SE (with one-time keys) should look uniformly random, and the hash function H is modelled as a random oracle. In the literature, the former requirement is often captured using the security notion called $\mathsf{IND\$}$ or $\mathsf{IND\$\text{-}CPA}$.

If SE ciphertexts look uniformly random and H is modelled as a random oracle, then intuitively $\mathsf{PKE\text{-}BLUEPRINT}$ should be at least IND-CPA secure based only on the one-wayness of f.[39] However, a black-box security reduction does not necessarily exist. Intuitively, in order to learn any information about the encrypted x, an IND-CPA adversary would need first to recover $c_{\mathsf{se}} \parallel \tilde{\kappa}$ from y (inverting the

[39] We discuss IND-CCA below.

```
$\mathsf{PKE\text{-}BLUEPRINT.Enc}(f, x)$
$k_{\mathsf{se}} \leftarrow\!\!\$ \{0,1\}^{\mathsf{SE.kl}}$
$\boxed{c_{\mathsf{se}} \leftarrow\!\!\$ \mathsf{SE.Enc}(k_{\mathsf{se}}, x)}$
$\tilde{\kappa} \leftarrow \mathsf{H}(c_{\mathsf{se}}) \oplus k_{\mathsf{se}}$
$y \leftarrow f(c_{\mathsf{se}} \parallel \tilde{\kappa})$
**return** $y$
```

```
$\mathsf{OAEP\text{-}BLUEPRINT.Enc}(f, x)$
$k_{\mathsf{se}} \leftarrow\!\!\$ \{0,1\}^{\mathsf{SE.kl}}$
$\boxed{c_{\mathsf{se}} \leftarrow \mathsf{G}(k_{\mathsf{se}}) \oplus (x \parallel \langle \textit{const. padding} \rangle)}$
$\tilde{\kappa} \leftarrow \mathsf{H}(c_{\mathsf{se}}) \oplus k_{\mathsf{se}}$
$y \leftarrow f(c_{\mathsf{se}} \parallel \tilde{\kappa})$
**return** $y$
```

```
$\mathsf{OAEP^{+}\text{-}BLUEPRINT.Enc}(f, x)$
$k_{\mathsf{se}} \leftarrow\!\!\$ \{0,1\}^{\mathsf{SE.kl}}$
$\boxed{\begin{array}{l} c_{\mathsf{base}} \leftarrow \mathsf{G}(k_{\mathsf{se}}) \oplus x \\ \textit{tag} \leftarrow \mathsf{H'}(k_{\mathsf{se}} \parallel x) \\ c_{\mathsf{se}} \leftarrow c_{\mathsf{base}} \parallel \textit{tag} \end{array}}$
$\tilde{\kappa} \leftarrow \mathsf{H}(c_{\mathsf{se}}) \oplus k_{\mathsf{se}}$
$y \leftarrow f(c_{\mathsf{se}} \parallel \tilde{\kappa})$
**return** $y$
```

Fig. 24. Encryption algorithms for public-key encryption schemes PKE-BLUEPRINT, OAEP-BLUEPRINT, and OAEP^{+}-BLUEPRINT. Each of them uses a one-way function f as a public key (for which the corresponding trapdoor function g exists, to be used as the secret decryption key).

one-way function), and then to compute $k_{\mathsf{se}} \leftarrow \tilde{\kappa} \oplus \mathsf{H}(c_{\mathsf{se}})$. But a black-box security reduction, in the random oracle model and for a generic choice of SE, is only guaranteed to obtain the value of c_{se} (and not of $\tilde{\kappa}$) from the adversary's queries to its random oracle. To avoid this issue, the scheme PKE-BLUEPRINT could be instantiated with a symmetric encryption algorithm SE.Enc that computes a hash of its key, i.e. $\mathsf{G}(k_{\mathsf{se}})$ for some hash function G. Ideally, the hash function G should be distinct from H, providing domain separation. In an IND-CPA security proof these functions would then be modelled as two distinct random oracles. Once the IND-CPA adversary queries both c_{se} and k_{se} to the corresponding random oracles, the value of $\tilde{\kappa}$ can be computed as $\tilde{\kappa} \leftarrow \mathsf{H}(c_{\mathsf{se}}) \oplus k_{\mathsf{se}}$.

The necessity of non-malleable SE. In order to prove that an instantiation of PKE-BLUEPRINT is IND-CCA secure, with a black-box reduction to the assumed one-wayness of an arbitrary (one-way) function f, the PKE scheme must be built from SE that is non-malleable (i.e. roughly NM-CPA secure). Assume for a contradiction that PKE-BLUEPRINT is instantiated with some f that is not one-way and SE that is malleable. Then there exists an attack against the IND-CCA security of PKE-BLUEPRINT that relies on a one-wayness inverter for f, yet a security reduction cannot use such an attack in a black-box way to build a standalone one-wayness inverter for f. In particular, we assume the following as a prerequisite:

– There exists an adversary against one-wayness of f, recovering $c_{\mathsf{se}} \parallel \tilde{\kappa}$ from $y = f(c_{\mathsf{se}} \parallel \tilde{\kappa})$ when $c_{\mathsf{se}} \parallel \tilde{\kappa}$ is from a distribution that is indistinguishable from uniformly random.

– There exists an adversary against non-malleability of SE, i.e. given a ciphertext c_{se} returned by $\mathsf{SE.Enc}(k_{\mathsf{se}}, x)$, it is able to create another ciphertext $c'_{\mathsf{se}} \neq c_{\mathsf{se}}$ such that $\mathsf{SE.Dec}(k_{\mathsf{se}}, c'_{\mathsf{se}}) = \tau(x)$ for some known non-constant function τ.

In IND-CCA security game against PKE-BLUEPRINT, consider a challenge ciphertext y returned by oracle ENC. It is computed as follows from some plaintext x:

$$y \leftarrow f(c_{\mathsf{se}} \parallel \tilde{\kappa}) \text{ for } c_{\mathsf{se}} \leftarrow\!\!\$ \mathsf{SE.Enc}(k_{\mathsf{se}}, x) \text{ and } \tilde{\kappa} \leftarrow \mathsf{H}(c_{\mathsf{se}}) \oplus k_{\mathsf{se}}.$$

The IND-CCA adversary could recover $c_{\mathsf{se}} \parallel \tilde{\kappa}$ from y by using the one-wayness inverter for f, and then compute some $c'_{\mathsf{se}} \neq c_{\mathsf{se}}$ from c_{se} by using the adversary against non-malleability of SE. The IND-CCA adversary could then forge a new PKE-BLUEPRINT ciphertext as follows:

$$y' \leftarrow f(c'_{\mathsf{se}} \parallel \tilde{\kappa}') \text{ for } \tilde{\kappa}' \leftarrow \tilde{\kappa} \oplus \mathsf{H}(c_{\mathsf{se}}) \oplus \mathsf{H}(c'_{\mathsf{se}}).$$

If the resulting ciphertext y' is queried to oracle DEC in the IND-CCA security game, the oracle would return $\tau(x)$ as output. This would allow the adversary to win in the IND-CCA security game. Crucially, this IND-CCA adversary only reveals c_{se} by calling $\mathsf{H}(c_{\mathsf{se}})$, i.e. it does not need to query any oracle with $\tilde{\kappa}$ as input. This means that, in general, no black-box reduction can use this adversary to break the one-wayness of f. (Implicit in this argument is an assumption that the adversary against non-malleability of SE does *not* recover its secret key k_{se} and subsequently compute its hash $\mathsf{G}(k_{\mathsf{se}})$ in the random oracle model. Otherwise $\tilde{\kappa}$ could still be recovered by computing $\tilde{\kappa} \leftarrow \mathsf{H}(c_{\mathsf{se}}) \oplus k_{\mathsf{se}}$.)

PKE scheme $\mathsf{OAEP\text{-}BLUEPRINT}$**.** Consider the public-key encryption scheme $\mathsf{OAEP\text{-}BLUEPRINT}$ that could be defined based on the encryption algorithm $\mathsf{OAEP\text{-}BLUEPRINT.Enc}$ in Fig. 24. It instantiates $\mathsf{PKE\text{-}BLUEPRINT}$ with

$$\mathsf{SE.Enc}(k_{\mathsf{se}}, x) := \mathsf{G}(k_{\mathsf{se}}) \oplus (x \parallel \langle \textit{const. padding} \rangle),$$

where G is a hash function that is distinct from H. This roughly captures Bellare-Rogaway's OAEP scheme [BR95], which was designed to provide IND-CCA security. However, note that the underlying SE scheme is trivially malleable. So the above argument can be applied to show that there does not exist a black-box reduction to the one-wayness of f, highlighting a gap in the initial OAEP security proof by [BR95]. Shoup [Sho02] pointed this out and proposed the OAEP+ scheme as an improved, IND-CCA secure variant of OAEP. Subsequent work [FOPS01] proved the IND-CCA security of OAEP in the random oracle model assuming *partial-domain* one-wayness of function f. Whereas a classic result shows [RSA78] that the partial-domain one-wayness assumption for $f = \mathsf{RSA}$ is equivalent to the (general) one-wayness assumption for RSA. Together this demonstrates that the initial claim — that OAEP is IND-CCA secure in the random oracle model based on the one-wayness of f — is true for some specific choices of f, in spite of the above argument that highlights a gap in proving this for a generic one-way function f.

PKE scheme $\mathsf{OAEP^{+}\text{-}BLUEPRINT}$**.** Consider the public-key encryption scheme $\mathsf{OAEP^{+}\text{-}BLUEPRINT}$ that could be defined based on the encryption algorithm $\mathsf{OAEP^{+}\text{-}BLUEPRINT.Enc}$ in Fig. 24. Here, G and H' are hash functions that are distinct from H. This roughly captures Shoup's OAEP+ scheme [Sho02]. Shoup proved that OAEP+ is IND-CCA secure in the random oracle model based on the one-wayness of an arbitrary underlying (one-way) function f. At a high level, the proof establishes the following claims:

– OAEP+ is plaintext-aware. The plaintext awareness [BR95] requires that an adversary cannot construct any ciphertext unless it "knows" the underlying plaintext. Prior work shows that an IND-CPA secure public-key encryption scheme is also IND-CCA secure if it satisfies an appropriate variant of a plaintext awareness notion [BDPR98]. The overall proof strategy for OAEP+ can be thought of as implicitly replicating such a result.

  Shoup builds an adversary $\mathcal{F}$ against the one-wayness of f. It simulates the IND-CCA game for some adversary $\mathcal{D}$ attacking $\mathsf{OAEP^{+}\text{-}BLUEPRINT}$. In particular, $\mathcal{F}$ simulates the random oracles for $\mathcal{D}$. The information $\mathcal{F}$ obtains from the random oracle queries allows it to perfectly simulate the decryption oracle DEC for $\mathcal{D}$ as follows. Any $\mathsf{OAEP^{+}\text{-}BLUEPRINT}$ ciphertext contains an integrity tag $tag = \mathsf{H}'(k_{\mathsf{se}} \parallel x)$ that is verified during decryption. In order to construct a valid ciphertext, adversary $\mathcal{D}$ must first query its oracle H' on the corresponding input $k_{\mathsf{se}} \parallel x$. There is an injective mapping from any string $k_{\mathsf{se}} \parallel x$ to the corresponding ciphertext. So on the one hand, $\mathcal{D}$ is not able to reuse the same integrity tag across different ciphertexts. And on the other hand, adversary $\mathcal{F}$ can iterate through all $\mathcal{D}$'s prior queries to the simulated oracle H' in order to determine whether any ciphertext y (queried by $\mathcal{D}$ into its simulated DEC oracle) is valid. If $\mathcal{F}$ finds some $k_{\mathsf{se}} \parallel x$ that can be used to reconstruct the queried ciphertext y, then it simply returns x as its decryption. Otherwise, $\mathcal{F}$ returns $\perp$.

– OAEP+ is IND-CPA secure. In the first phase of the proof, we established that the one-wayness adversary $\mathcal{F}$ can perfectly simulate the decryption oracle for the IND-CCA adversary $\mathcal{D}$. In the second phase of the proof, we need to show that $\mathcal{F}$ can break the one-wayness of f whenever $\mathcal{D}$ breaks the IND-CPA security of $\mathsf{OAEP^{+}\text{-}BLUEPRINT}$. The IND-CPA challenge ciphertext $y \leftarrow f(c_{\mathsf{se}} \parallel \tilde{\kappa})$ is computed from some input string $c_{\mathsf{se}} \parallel \tilde{\kappa}$ that, informally, "looks" uniformly random (in ROM) unless $\mathcal{D}$ can break the one-wayness of f. This allows $\mathcal{F}$ to replace y with its own one-wayness challenge because it, too, is computed by applying f on a uniformly random input. The only way $\mathcal{D}$ might be able to distinguish between having received such a one-wayness challenge vs. an IND-CPA challenge ciphertext is by doing the following: (1) invert y to obtain c_{se} and $\tilde{\kappa}$, (2) compute the one-time key $k_{\mathsf{se}} \leftarrow \mathsf{H}(c_{\mathsf{se}}) \oplus \tilde{\kappa}$ by querying its random oracle H on c_{se}, and (3) compute one of the c_{base} or tag

either by querying its random oracle G on k_{se} or by querying its random oracle H' on $k_{\mathsf{se}} \parallel x$ for some candidate plaintext x. Adversary $\mathcal{F}$ would be able to recover the preimage of y based on the values of c_{se} and k_{se} that it obtains in steps (2) and (3).

Shoup's OAEP+ scheme. We reproduce the construction of OAEP+ in Definition 10 and Fig. 25, using the original variable and function names from [Sho02]. We reproduce the accompanying security theorem in Theorem 3.

Definition 10 (OAEP+ [Sho02]). *Let $n, k_0, k_1 \geq 1$ be integers. Let $k = n + k_0 + k_1$. Let TDF be a trapdoor function family with $\mathsf{TDF.Dom} = \{0,1\}^k$. Let G, H', H be any functions such that $\mathsf{G}\colon \{0,1\}^{k_0} \to \{0,1\}^n$, $\mathsf{H}'\colon \{0,1\}^{n+k_0} \to \{0,1\}^{k_1}$, and $\mathsf{H}\colon \{0,1\}^{n+k_1} \to \{0,1\}^{k_0}$. Then $\mathsf{PKE} = \mathsf{OAEP}^{+}[\mathsf{TDF}, \mathsf{G}, \mathsf{H}', \mathsf{H}, n, k_0, k_1]$ is the public-key encryption scheme as defined in Fig. 25, with $\mathsf{PKE.MS} = \{0,1\}^n$.*

```
Algorithm PKE.KGen()
$(fk, tk) \leftarrow\$ \mathsf{TDF.KGen}()$
$f(\cdot) \leftarrow \mathsf{TDF.Ev}(fk, \cdot)$   // Define $f$ as a single-argument function.
$g(\cdot) \leftarrow \mathsf{TDF.Inv}(tk, \cdot)$   // Function $g$ is the inverse of function $f$.
return $(f, g)$
```

```
Algorithm PKE.Enc$(f, x)$
require $x \in \{0,1\}^n$
$r \leftarrow\$ \{0,1\}^{k_0}$
$c_{\mathsf{base}} \leftarrow \mathsf{G}(r) \oplus x$
$tag \leftarrow \mathsf{H}'(r \parallel x)$
$s \leftarrow c_{\mathsf{base}} \parallel tag$
$t \leftarrow \mathsf{H}(s) \oplus r$
$w \leftarrow s \parallel t$
$y \leftarrow f(w)$
return $y$
```

```
Algorithm PKE.Dec$(g, y)$
require $y \in \{0,1\}^k$
$w \leftarrow g(y)$
$c_{\mathsf{base}} \leftarrow w[0 : n]$
$tag \leftarrow w[n : n + k_1]$
$t \leftarrow w[n + k_1 : n + k_1 + k_0]$
$s \leftarrow c_{\mathsf{base}} \parallel tag$
$r \leftarrow \mathsf{H}(s) \oplus t$
$x \leftarrow \mathsf{G}(r) \oplus c_{\mathsf{base}}$
$c \leftarrow tag$   // For consistency with [Sho02].
if $c \neq \mathsf{H}'(r \parallel x)$ : return $\bot$
return $x$
```

Fig. 25. Public-key encryption scheme $\mathsf{PKE} = \mathsf{OAEP}^{+}[\mathsf{TDF}, \mathsf{G}, \mathsf{H}', \mathsf{H}, n, k_0, k_1]$.

Theorem 3 (OAEP+ [Sho02]). *Let $\mathsf{TDF}, \mathsf{G}, \mathsf{H}', \mathsf{H}, n, k_0, k_1$ be any entities that meet the requirements stated in Definition 10 of OAEP^{+}. Let $\mathsf{PKE} = \mathsf{OAEP}^{+}[\mathsf{TDF}, \mathsf{G}, \mathsf{H}', \mathsf{H}, n, k_0, k_1]$, in which $\mathsf{G}, \mathsf{H}', \mathsf{H}$ are modelled as independent random oracles. Let $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ be any adversary against the IND-CCA security of PKE, making at most the following number of oracle queries: $\mathsf{n}_{\mathrm{DEC}}$ to oracle DEC, q_{G} to random oracle for function G, and $q_{\mathsf{H}'}$ to random oracle for function H'. Then we can build an adversary $\mathcal{F}_{\mathsf{OW}}$ against the one-wayness of TDF such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}}(\mathcal{D}_{\mathsf{IND\text{-}CCA}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{OW}}_{\mathsf{TDF}}(\mathcal{F}_{\mathsf{OW}}) + 2 \cdot \left(\frac{q_{\mathsf{H}'} + \mathsf{n}_{\mathrm{DEC}}}{2^{k_1}} + \frac{(\mathsf{n}_{\mathrm{DEC}} + 1) \cdot q_{\mathsf{G}}}{2^{k_0}} \right).$$

D.4 Public-key encryption scheme Telegram-OAEP+

To aid intuition, we first define and discuss a stylised and simplified variant of Telegram's OAEP+ scheme. The full scheme is given and proven below.

PKE scheme TELEGRAM-OAEP$^+$-BLUEPRINT**.** Consider the encryption algorithm, for a PKE scheme, defined in Fig. 26. It captures the key components of Telegram's OAEP+ scheme at a high level. Let TELEGRAM-OAEP$^+$-BLUEPRINT be the public-key encryption scheme that could be formalised by defining the corresponding key generation and decryption algorithms. We now discuss it based on the informal concepts introduced, and the intuition developed, in Appendix D.3.

```
TELEGRAM-OAEP+-BLUEPRINT.Enc($f, x$)
require $|x| = 1536$
$k_{se} \leftarrow\$ \{0,1\}^{256}$
$\boxed{tag \leftarrow \mathsf{SHA\text{-}256}(k_{se} \| x)}$
$\boxed{c_{se} \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Enc}(k_{se}, iv = 0^{256}, x \| tag)}$
$\tilde{\kappa} \leftarrow \mathsf{SHA\text{-}256}(c_{se}) \oplus k_{se}$
$y \leftarrow f(\tilde{\kappa} \| c_{se})$
return $y$
```

Fig. 26. The encryption algorithm for public-key encryption scheme TELEGRAM-OAEP$^+$-BLUEPRINT. It uses a one-way function f as a public key (the corresponding trapdoor function should be used as the secret key).

TELEGRAM-OAEP$^+$-BLUEPRINT can be thought of as an instantiation of the PKE-BLUEPRINT scheme from Fig. 24. It uses a one-way function f as its public key. Telegram instantiates f with the RSA function. Our IND-CCA security proof for Telegram's OAEP+ scheme will not rely on any properties that are specific to RSA, such as the equivalence between the one-wayness and partial-domain one-wayness notions for RSA [RSA78, FOPS01] (which relies on the self-reducibility of RSA). So in the high-level discussion below, we temporarily treat f as a generic one-way function.

TELEGRAM-OAEP$^+$-BLUEPRINT encrypts 1536-bit messages. It calls SHA-256 twice, always on 1792-bit long inputs. There is no domain separation between these two calls. The two boxed lines together instantiate the algorithm SE.Enc of PKE-BLUEPRINT, meaning that

$$\mathsf{SE.Enc}(k_{se}, x) := \mathsf{AES\text{-}256\text{-}IGE.Enc}(k_{se}, iv = 0^{256}, x \| \mathsf{SHA\text{-}256}(k_{se} \| x)).$$

Based on our informal analysis of the PKE-BLUEPRINT scheme (and the requirements towards the underlying SE) in Appendix D.3, we now highlight the main observations that pertain to proving the IND-CCA security of the TELEGRAM-OAEP$^+$-BLUEPRINT scheme. We emphasise that this discussion is purely intuitive, provided to gain insight into why an IND-CCA security proof should be possible, and what it would entail. The intuition that we discuss below is not explicitly formalised and treated as a part of our IND-CCA proof. It is resolved implicitly at different points throughout the sequence of games that we define for our IND-CCA proof.

– AES-256 must be modelled as an ideal cipher. Consider a hypothetical attacker that might be able to recover $\tilde{\kappa}$ and c_{se} by inverting f. It could then reconstruct the one-time key $k_{se} \leftarrow \mathsf{SHA\text{-}256}(c_{se}) \oplus \tilde{\kappa}$ and use it to decrypt c_{se}, obtaining $x \| tag$. This would allow the attacker to break the IND-CPA security of TELEGRAM-OAEP$^+$-BLUEPRINT. Note that the attacker does not need to verify that $tag = \mathsf{SHA\text{-}256}(k_{se} \| x)$ is true, precluding us from using this SHA-256 call (when SHA-256 is modelled as a random oracle) to construct an adversary against the one-wayness of f in a black-box way. In order to enable the security reduction, we need to model AES-256 as an ideal cipher so that the attacker is forced to reveal the value of k_{se} when it decrypts c_{se}.
– The distribution of function f's inputs $\tilde{\kappa} \| c_{se}$. In order to use the assumed one-wayness of f, at some point in the proof we need to be able to show that its input $\tilde{\kappa} \| c_{se}$ comes from a distribution that is computationally indistinguishable from the uniformly random distribution. In our IND-CCA security

proof of Telegram's full scheme, we will model SHA-256 as a random oracle and AES-256 as an ideal cipher. Observe that the one-time key k_{se} that is sampled in the encryption algorithm is unlikely to collide with any previously used key. This means that the values of *tag* and c_{se} (and subsequently also the value of $\tilde{\kappa}$) will be sampled uniformly at random (by the corresponding idealised oracles) during the encryption operation. Similarly to Shoup's proof for OAEP+ [Sho02], our proof will show that the IND-CCA adversary is unlikely to query its idealised oracles on inputs that were used to produce the challenge ciphertext. This will allow us to eventually treat the string $\tilde{\kappa} \parallel c_{\mathsf{se}}$ used for the challenge ciphertext as being uniformly random, sampled independently from adversary's view of the game.

– Plaintext-awareness of TELEGRAM-OAEP$^+$-BLUEPRINT. Scheme TELEGRAM-OAEP$^+$-BLUEPRINT is plaintext-aware in the sense that any ciphertext has to contain a valid tag that is computed as a hash over the input $k_{\mathsf{se}} \parallel x$. And each pair k_{se}, x uniquely corresponds to a single valid ciphertext. Similarly to Shoup's proof for OAEP+ [Sho02], this will allow the one-wayness adversary in our proof to perfectly simulate the decryption oracle for the IND-CCA adversary. (But some care needs to be taken in order to appropriately resolve the details that arise due to the lack of domain separation in Telegram's scheme.)

– Non-malleability of SE. In Appendix D.3 we roughly argued that a black-box reduction from IND-CCA security of PKE-BLUEPRINT to the one-wayness of f could exist only if SE is non-malleable. More precisely, we pointed out that a black-box reduction could exist if an adversary against non-malleability of SE could only succeed after computing a hash of SE's secret key k_{se} (i.e. querying the corresponding random oracle on k_{se}). We now argue that the SE scheme used in Telegram, as defined above, satisfies this property. This is true in spite of the IGE block cipher mode of operation on its own being trivially malleable [Jut00]. In particular, it is hard to forge a new SE ciphertext $c_{\mathsf{se}} = \mathsf{AES\text{-}256\text{-}IGE.Enc}(k_{\mathsf{se}}, 0^{256}, x \parallel tag)$ for $tag = \mathsf{SHA\text{-}256}(k_{\mathsf{se}} \parallel x)$ without explicitly calling SHA-256 to compute *tag* first. The string $k_{\mathsf{se}} \parallel x$ uniquely determines the value of c_{se}, so an attacker cannot reuse the entirety of $tag = \mathsf{SHA\text{-}256}(k_{\mathsf{se}} \parallel x)$ from a prior ciphertext. Instead the attacker would have to find some values $(tag, k_{\mathsf{se}}, x)$ such that $tag = \mathsf{SHA\text{-}256}(k_{\mathsf{se}} \parallel x)$ holds, without explicitly computing $\mathsf{SHA\text{-}256}(k_{\mathsf{se}} \parallel x)$. That is implausible.

Detailed design of Telegram-OAEP+. We formally define Telegram's OAEP+ in Definition 11.

Definition 11. *Let* RSA.KGen *be an RSA key generator that always returns N such that* $2^{2047} < N < 2^{2048}$. *Let* $\mathcal{M} \subseteq \bigcup_{\ell \leq 1152} \{0,1\}^{\ell}$. *Let* $\mathsf{RemovePadding}\colon \{0,1\}^{1536} \to \mathcal{M}$. *Let* $\mathsf{max\text{-}attempts} \geq 1$ *be an integer. Then* $\mathsf{PKE} = \mathsf{TELEGRAM\text{-}OAEP}^+[\mathsf{RSA.KGen}, \mathcal{M}, \mathsf{RemovePadding}, \mathsf{max\text{-}attempts}]$ *is the public-key encryption scheme as defined in Fig. 27, with* $\mathsf{PKE.MS} = \mathcal{M}$.

Remark 1. In our definition, we do not require correctness for all choices of pairs RemovePadding and $\mathcal{M}$, i.e. RemovePadding might not interact correctly with $\mathcal{M}$. Telegram instantiates Definition 11 with a correct pair and our definition is not intended to capture more than this instantiation.

Remark 2. Our introduction of max-attempts is a simplification of the actual Telegram scheme which will continue to resample forever. Picking an appropriate value of max-attempts gives a simplification that has a negligible effect on correctness and security.

Theorem 4. *Let* RSA.KGen, $\mathcal{M}$, RemovePadding, max-attempts *be any entities that meet the requirements stated in Definition 11 of* TELEGRAM-OAEP$^+$. *Let* $\mathsf{PKE} = \mathsf{TELEGRAM\text{-}OAEP}^+[\mathsf{RSA.KGen}, \mathcal{M}, \mathsf{RemovePadding}, \mathsf{max\text{-}attempts}]$, *in which* SHA-256 *is modelled as a random oracle and* AES-256 *is modelled as an ideal cipher. Let* $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ *be any adversary against the IND-CCA security of* PKE, *making at most the following number of oracle queries:* $\mathsf{n}_{\mathrm{ENC}}$ *to oracle* ENC, $\mathsf{n}_{\mathrm{DEC}}$ *to oracle* DEC, n_{H} *to random oracle* H, *and* n_{IC} *jointly to ideal-cipher oracles* IC *and* IC^{-1}. *Assume that*[40] $\mathsf{max\text{-}attempts} = 121$, *and* $\mathsf{n}_{\mathrm{ENC}}, \mathsf{n}_{\mathrm{DEC}}, \mathsf{n}_{\mathrm{H}}, \mathsf{n}_{\mathsf{IC}} \leq 2^{126}$ *such that* $\mathsf{n}_{\mathrm{DEC}} \cdot \mathsf{n}_{\mathrm{H}} \leq 2^{134}$,

[40] We only use these constraints to derive a sample numeric bound below. The proof allows to choose arbitrary constraints.

```
PKE.Enc(pk, m)
1 :  require $m \in \mathcal{M}$
2 :  $(N, e) \leftarrow pk$ ; attempt $\leftarrow 1$
3 :  $K \leftarrow\!\$ \{0,1\}^{256}$
4 :  $pad \leftarrow\!\$ \{0,1\}^{1536-|m|}$
5 :  $m_{\mathsf{padded}} \leftarrow m \parallel pad$
6 :  $h \leftarrow \mathsf{SHA\text{-}256}(K \parallel m_{\mathsf{padded}})$
7 :  $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \parallel h$
8 :  $c_{\mathsf{ige}} \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Enc}(K, 0^{256}, p_{\mathsf{ige}})$
9 :  $r \leftarrow \mathsf{SHA\text{-}256}(c_{\mathsf{ige}}) \oplus K$
10 : $p_{\mathsf{rsa}} \leftarrow r \parallel c_{\mathsf{ige}}$
11 : $z \leftarrow p_{\mathsf{rsa}}$   // Parse $p_{\mathsf{rsa}}$ as an integer.
12 : if $z \notin \mathbb{Z}_N$ :
13 :   attempt $\leftarrow$ attempt $+ 1$
14 :   if attempt $>$ max-attempts :
15 :     return $(\lightning, m)$
16 :   goto line 3
17 : $c_{\mathsf{rsa}} \leftarrow z^e \bmod N$
18 : return $c_{\mathsf{rsa}}$
```

```
PKE.KGen()
1 :  $(N, p, q, e, d) \leftarrow\!\$ \mathsf{RSA.KGen}()$
2 :  $pk \leftarrow (N, e)$ ; $sk \leftarrow (N, d)$
3 :  return $(pk, sk)$
```

```
PKE.Dec(sk, c_rsa)
1 :  $(N, d) \leftarrow sk$
2 :  if $c_{\mathsf{rsa}} \notin \mathbb{Z}_N$ : return $\perp$
3 :  $z \leftarrow (c_{\mathsf{rsa}})^d \bmod N$
4 :  $p_{\mathsf{rsa}} \leftarrow z$   // Parse $z$ as a 2048-bit string.
5 :  $r \leftarrow p_{\mathsf{rsa}}[0 : 256]$
6 :  $c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}[256 : 2048]$
7 :  $K \leftarrow \mathsf{SHA\text{-}256}(c_{\mathsf{ige}}) \oplus r$
8 :  $p_{\mathsf{ige}} \leftarrow \mathsf{AES\text{-}256\text{-}IGE.Dec}(K, 0^{256}, c_{\mathsf{ige}})$
9 :  $m_{\mathsf{padded}} \leftarrow \mathsf{reverse}(p_{\mathsf{ige}}[0 : 1536])$
10 : $h \leftarrow p_{\mathsf{ige}}[1536 : 1792]$
11 : if $h \neq \mathsf{SHA\text{-}256}(K \parallel m_{\mathsf{padded}})$ : return $\perp$
12 : $m \leftarrow \mathsf{RemovePadding}(m_{\mathsf{padded}})$
13 : return $m$
```

Fig. 27. Public-key encryption scheme $\mathsf{PKE} = \mathsf{TELEGRAM\text{-}OAEP}^{+}[\mathsf{RSA.KGen}, \mathcal{M}, \mathsf{RemovePadding}, \mathsf{max\text{-}attempts}]$.

and that RSA.KGen *returns* $2^{2047} < N < 2^{2048} - 2^{256}$*. Then we can build an adversary* $\mathcal{F}_{\mathsf{OWRSA}}$ *against the one-wayness of RSA such that*

$$\mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{PKE}}(\mathcal{D}_{\mathsf{IND\text{-}CCA}}) \leq 2 \cdot \mathsf{Adv}^{\text{OW-RSA}}_{\mathsf{RSA.KGen}}(\mathcal{F}_{\mathsf{OWRSA}}) + 2^{-116}.$$

Our proof first rewrites the encryption oracle ENC to return a ciphertext that is obtained by applying the RSA function on a uniformly random integer from $\mathbb{Z}_N$. In particular, we manage to get rid of the rejection sampling iterations along the way. In order to maintain equivalence between our games (i.e. ensure the consistency of adversary's view between these games), we temporarily program the idealised oracles in a way that they map the output of the encryption oracle to an appropriate challenge message m_b. This process is similar to the random oracle programming that is done in games 4–5 of Shoup's proof for OAEP+ [Sho02]. Our proof requires a lot more steps because of the necessity to appropriately resolve the rejection sampling and the lack of domain separation between different random oracle calls.

Having simplified the encryption oracle, we then use the plaintext-awareness of Telegram's OAEP+ scheme in order to rewrite the decryption oracle DEC in a way that a one-wayness adversary can perfectly simulate it for the IND-CCA adversary.

Proof. This proof uses games G_0 through G_{31}, split across the following figures:

Games	**Figure**
G_0	Fig. 29
G_1–G_{11}	Fig. 30
G_{12}–G_{13}	Fig. 31
G_{14}–G_{18}	Fig. 32
G_{19}–G_{20}	Fig. 33
G_{21}–G_{26}	Fig. 34
G_{27}–G_{29}	Fig. 35
G_{30}–G_{31}	Fig. 36

In a game, the code highlighted in gray is generally equivalent to the code from the previous previous. It often expands the definition of some algorithm or oracle, or rewrites something in an alternative way. The code highlighted in green is generally added for the transitions between future games. We build adversary $\mathcal{F}_{\mathsf{OWRSA}}$ in Fig. 37. In this adversary, the code highlighted in orange modifies the simulated game G_{31}.

Below we will show that $\Pr[G_0] = \Pr[G^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{D}_{\mathsf{IND\text{-}CCA}}}]$. This allows to express the advantage of adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ as follows:

$$\mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{PKE}}(\mathcal{D}_{\mathsf{IND\text{-}CCA}}) = 2 \cdot \Pr[G_0] - 1 = 2 \cdot \left(\sum_{i=0}^{30} (\Pr[G_i] - \Pr[G_{i+1}]) + \Pr[G_{31}] \right) - 1.$$

The probability upper bounds for all transitions between games, as well as $\Pr[G_{31}]$, are summarised in Fig. 28. We will justify each of these probabilities as we move through the proof steps. For the constraints specified in Theorem 4 (i.e. $\mathsf{max\text{-}attempts} = 121$, and $\mathsf{n}_{\text{ENC}}, \mathsf{n}_{\text{DEC}}, \mathsf{n}_{\text{H}}, \mathsf{n}_{\text{IC}} \leq 2^{126}$ such that $\mathsf{n}_{\text{DEC}} \cdot \mathsf{n}_{\text{H}} \leq 2^{134}$, and $\mathsf{RSA.KGen}$ returns $2^{2047} < N < 2^{2048} - 2^{256}$), each non-zero probability term for transitions in Fig. 28 can be trivially upper-bounded by 2^{-121}. There are a total of 10 transitions like that, so we have

$$\begin{aligned}\mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{PKE}}(\mathcal{D}_{\mathsf{IND\text{-}CCA}}) &\leq 2 \cdot (\mathsf{Adv}^{\text{OW-RSA}}_{\mathsf{RSA.KGen}}(\mathcal{F}_{\mathsf{OWRSA}}) + 10 \cdot 2^{-121}) \\ &< 2 \cdot \mathsf{Adv}^{\text{OW-RSA}}_{\mathsf{RSA.KGen}}(\mathcal{F}_{\mathsf{OWRSA}}) + 2^{-116}.\end{aligned}$$

This justifies the bound in the theorem statement. We now explain each step of the proof.

Analysis of G_0. This game, given in Fig. 29, expands the definition of Telegram's PKE scheme into the IND-CCA game in the RO and the ICM models. We are also simultaneously expanding the code of the (ideal cipher based) IGE encryption algorithm inside oracle ENC. The RO and the ICM oracles are implemented via lazy sampling. The object $\mathsf{IGE}^{\text{IC},\text{IC}^{-1}}$ in oracle DEC denotes the IGE block cipher mode of operation that uses the ideal cipher oracle IC as the underlying block cipher and IC^{-1} as its inverse. In the decryption oracle DEC, we call $\mathsf{IGE}^{\text{IC},\text{IC}^{-1}}.\mathsf{Dec}(K, 0^{256}, c_{\mathsf{ige}})$ to evaluate the IGE decryption algorithm. Game G_0 is equivalent to game $G^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{D}_{\mathsf{IND\text{-}CCA}}}$ by design, so

$$\Pr[G_0] = \Pr[G^{\text{IND-CCA}}_{\mathsf{PKE},\mathcal{D}_{\mathsf{IND\text{-}CCA}}}].$$

Note that upon exceeding $\mathsf{max\text{-}attempts}$ of rejection sampling iterations, oracle ENC returns $(\lightning, m_b)$. As discussed above, we introduced the $\mathsf{max\text{-}attempts}$ constant as a simplification of the actual Telegram scheme (in which the rejection sampling runs forever, until success). We chose to unambiguously leak the challenge message to the adversary upon exceeding $\mathsf{max\text{-}attempts}$. As a result, an adversary can "break" the security of the scheme for free whenever its correctness fails. So in our analysis, and in particular – for our final security bound, it is sufficient to choose the smallest value of $\mathsf{max\text{-}attempts}$ that provides a desirable security bound. On the flip side, picking the value of $\mathsf{max\text{-}attempts}$ that is too large, would inherently worsen the security bound because an excessive number of *estimated* rejection sampling iterations would lead to an increased chance of collisions between various random oracle and ideal cipher

$$\Pr[G_0] = \Pr[G^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE},\mathcal{D}_{\mathsf{IND\text{-}CCA}}}]$$
$$\Pr[G_0] - \Pr[G_1] \leq 0.$$
$$\Pr[G_1] - \Pr[G_2] \leq \Pr[\mathsf{bad}_0^{G_2}] < \frac{\mathsf{max\text{-}attempts} \cdot (\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}} + \mathsf{max\text{-}attempts})}{2^{256}}.$$
$$\Pr[G_2] - \Pr[G_3] \leq 0.$$
$$\Pr[G_3] - \Pr[G_4] \leq \Pr[\mathsf{bad}_1^{G_4}] < 2^{-121}.$$
$$\Pr[G_4] - \Pr[G_5] \leq \Pr[\mathsf{bad}_2^{G_5}] < 2^{-121}.$$
$$\Pr[G_5] - \Pr[G_6] \leq \Pr[\mathsf{bad}_3^{G_6}] < \frac{\mathsf{max\text{-}attempts} \cdot (\mathsf{n}_{\mathrm{H}} + \mathsf{n}_{\mathrm{DEC}} + \mathsf{max\text{-}attempts})}{2^{1791}}.$$
$$\Pr[G_6] - \Pr[G_7] \leq 0.$$
$$\Pr[G_7] - \Pr[G_8] \leq 0.$$
$$\Pr[G_8] - \Pr[G_9] \leq \Pr[\mathsf{bad}_4^{G_9}] < 2^{-\mathsf{max\text{-}attempts}}.$$
$$\Pr[G_9] - \Pr[G_{10}] \leq \Pr[\mathsf{bad}_5^{G_{10}}] \leq \frac{\mathsf{max\text{-}attempts}}{2^{384}}.$$
$$\Pr[G_{10}] - \Pr[G_{11}] \leq 0.$$
$$\Pr[G_{11}] - \Pr[G_{12}] \leq 0.$$
$$\Pr[G_{12}] - \Pr[G_{13}] \leq 0.$$
$$\Pr[G_{13}] - \Pr[G_{14}] \leq 0.$$
$$\Pr[G_{14}] - \Pr[G_{15}] \leq \Pr[\mathsf{bad}_6^{G_{15}}] < \frac{\mathsf{max\text{-}attempts} \cdot (\mathsf{n}_{\mathrm{H}} + \mathsf{n}_{\mathrm{DEC}} + \mathsf{max\text{-}attempts})}{\mathbf{min}(2^{1790}, 2^{2046} - N)}.$$
$$\Pr[G_{15}] - \Pr[G_{16}] \leq \Pr[\mathsf{bad}_7^{G_{16}}] < \frac{\mathsf{max\text{-}attempts} \cdot (\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}})}{2^{255}}.$$
$$\Pr[G_{16}] - \Pr[G_{17}] \leq \Pr[\mathsf{bad}_8^{G_{17}}] \leq 0.$$
$$\Pr[G_{17}] - \Pr[G_{18}] \leq 0.$$
$$\Pr[G_{18}] - \Pr[G_{19}] \leq 0.$$
$$\Pr[G_{19}] - \Pr[G_{20}] \leq 0.$$
$$\Pr[G_{20}] - \Pr[G_{21}] \leq 0.$$
$$\Pr[G_{21}] - \Pr[G_{22}] \leq 0.$$
$$\Pr[G_{22}] - \Pr[G_{23}] \leq 0.$$
$$\Pr[G_{23}] - \Pr[G_{24}] \leq 0.$$
$$\Pr[G_{24}] - \Pr[G_{25}] \leq 0.$$
$$\Pr[G_{25}] - \Pr[G_{26}] \leq 0.$$
$$\Pr[G_{26}] - \Pr[G_{27}] \leq 0.$$
$$\Pr[G_{27}] - \Pr[G_{28}] \leq \Pr[\mathsf{bad}_9^{G_{28}}] \leq \frac{\mathsf{n}_{\mathrm{DEC}} \cdot (\mathsf{n}_{\mathrm{H}} + 1)}{2^{256}}.$$
$$\Pr[G_{28}] - \Pr[G_{29}] \leq \frac{\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}}}{2^{256}}.$$
$$\Pr[G_{29}] - \Pr[G_{30}] \leq 0.$$
$$\Pr[G_{30}] - \Pr[G_{31}] \leq \Pr[\mathsf{bad}_{10}^{G_{31}}] \leq \mathsf{Adv}^{\mathsf{OW\text{-}RSA}}_{\mathsf{RSA.KGen}}(\mathcal{F}_{\mathsf{OWRSA}}).$$
$$\Pr[G_{31}] = \frac{1}{2}.$$

Fig. 28. The probability upper bounds for transitions between games G_0 through G_{31} for the proof of Theorem 4.

outputs. However, we emphasise that this is only an issue – and an artificial one at that – if the proof uses an excessive number of expected rejection sampling iterations, i.e. many more than would have happened in practice. We argue that our approach to modelling $\mathsf{max\text{-}attempts}$, as described above, allows to choose a somewhat accurate expected number of iterations, and hence to arrive at a reasonable overall security upper bound.

Analysis of $\mathrm{G}_0 \to \mathrm{G}_1$**.** This game only changes the encryption oracle $\textsc{Enc}$, the other oracles and the main game remain the same. The changes to $\textsc{Enc}$ are given in G_1. The games G_0 and G_1 are functionally equivalent. The latter adds some code, highlighted in green, that is not yet used for anything and does not affect the functionality of the oracle. It also rewrites one call to the ideal cipher (highlighted in gray) into a conditional statement, where both branches contain the same call. This again means there is no change of functionality of the oracle. We have

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

Analysis of $\mathrm{G}_1 \to \mathrm{G}_2$**.** Consider game G_2. Adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ makes at most a single query to oracle $\textsc{Enc}$. Prior to calling $\textsc{Enc}$, the adversary can make at most $\mathsf{n}_{\mathrm{DEC}}$ queries to oracle $\textsc{Dec}$ and at most n_{IC} queries to oracles IC, IC^{-1}. Each query to $\textsc{Dec}$, IC, or IC^{-1} can add at most a single key $K \in \{0,1\}^{256}$ to the set S_{IC}. So when $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ queries oracle $\textsc{Enc}$, the set S_{IC} might contain at most $\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}}$ distinct keys. Oracle $\textsc{Enc}$ iteratively samples at most $\mathsf{max\text{-}attempts}$ new keys, checking each key for a collision prior to adding it to S_{IC}. The flag $\mathsf{bad}_0^{\mathrm{G}_2}$ is set if a collision occurs. The probability of obtaining a collision is $\leq \mathsf{bb}(\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}}, \mathsf{max\text{-}attempts}, 0, 2^{256})$ and we have

$$\begin{aligned}\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \Pr[\mathsf{bad}_0^{\mathrm{G}_2}] &\leq \mathsf{bb}(\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}}, \mathsf{max\text{-}attempts}, 0, 2^{256}) \\ &\leq \frac{\mathsf{max\text{-}attempts} \cdot (2 \cdot \mathsf{n}_{\mathrm{DEC}} + 2 \cdot \mathsf{n}_{\mathsf{IC}} + \mathsf{max\text{-}attempts} - 1)}{2^{257}} \\ &< \frac{\mathsf{max\text{-}attempts} \cdot (\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}} + \mathsf{max\text{-}attempts})}{2^{256}}.\end{aligned}$$

Analysis of $\mathrm{G}_2 \to \mathrm{G}_3$**.** Line 16 of oracle $\textsc{Enc}$ in game G_2 evaluates the instruction

$$y_i \leftarrow \mathsf{IC}(K, u_i) \oplus x_{i-1}$$

in the conditional branch where $\mathsf{A}_K[u_i] = \perp$ was established to be true. Under this condition, the code underlying the call to $\mathsf{IC}(K, u_i)$ can be expanded to rewrite line 16 in an equivalent way:

$$v_i \leftarrow\!\!\$\; \{0,1\}^{128} \setminus \mathcal{R}_K\,;\; y_i \leftarrow v_i \oplus x_{i-1}\,;\; \mathsf{AddRelationToIC}(K, u_i, v_i).$$

Game G_3 extends this to first run

$$y_i \leftarrow\!\!\$\; \{0,1\}^{128}\,;\; v_i \leftarrow y_i \oplus x_{i-1}.$$

If $v_i \notin \mathcal{R}_K$, then $\mathsf{AddRelationToIC}(K, u_i, v_i)$ is called next. Otherwise, the above (originally expanded) code is used as the fallback. Games G_2 and G_3 are functionally equivalent. In particular, the pair (y_i, v_i) is sampled from the same distribution in these games. We have

$$\Pr[\mathrm{G}_2] = \Pr[\mathrm{G}_3].$$

Analysis of $\mathrm{G}_3 \to \mathrm{G}_4$**.** Consider game G_4. Adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ makes at most a single query to oracle $\textsc{Enc}$. Oracle $\textsc{Enc}$ runs for at most $\mathsf{max\text{-}attempts}$ iterations. It guarantees that in each iteration the ideal cipher is used with a distinct, fresh key K. For a single key, at most 14 values $v_1, v_2, \ldots, v_{14}$ are sampled

Game G_0

```
1 :  $b \leftarrow\$ \{0,1\}$ ; $c^*_{\mathsf{rsa}} \leftarrow \bot$
2 :  $S_{\mathsf{IC}} \leftarrow \emptyset$ ; $\mathcal{D} \leftarrow \emptyset$ ; $\mathcal{R} \leftarrow \emptyset$
3 :  $(N, p, q, e, d) \leftarrow\$ \mathsf{RSA.KGen}()$
4 :  $pk \leftarrow (N, e)$
5 :  $b' \leftarrow\$ \mathcal{D}^{\text{ENC},\text{DEC},\mathrm{H},\mathrm{IC},\mathrm{IC}^{-1}}_{\mathsf{IND\text{-}CCA}}(pk)$
6 :  return $b = b'$
```

Oracle $\text{ENC}(m_0, m_1)$

```
1 :  require $(c^*_{\mathsf{rsa}} = \bot) \wedge (|m_0| = |m_1|)$
2 :  require $m_0, m_1 \in \mathcal{M}$
3 :  $\mathsf{attempt} \leftarrow 1$
4 :  $K \leftarrow\$ \{0,1\}^{256}$
5 :  $pad \leftarrow\$ \{0,1\}^{1536-|m_b|}$
6 :  $m_{\mathsf{padded}} \leftarrow m_b \,\|\, pad$
7 :  $h \leftarrow \mathrm{H}(K \,\|\, m_{\mathsf{padded}})$
8 :  $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \,\|\, h$
9 :  // Parse into 128-bit blocks.
10 : $x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
11 : $x_1 \,\|\, \ldots \,\|\, x_{14} \leftarrow p_{\mathsf{ige}}$
12 : for $i = 1, \ldots, 14$ :
13 :   $u_i \leftarrow x_i \oplus y_{i-1}$
14 :   $y_i \leftarrow \mathsf{IC}(K, u_i) \oplus x_{i-1}$
15 : $c_{\mathsf{ige}} \leftarrow y_1 \,\|\, \ldots \,\|\, y_{14}$
16 : $r \leftarrow \mathrm{H}(c_{\mathsf{ige}}) \oplus K$
17 : $p_{\mathsf{rsa}} \leftarrow r \,\|\, c_{\mathsf{ige}}$
18 : $z \leftarrow p_{\mathsf{rsa}}$    // Parse $p_{\mathsf{rsa}}$ as an integer.
19 : if $z \notin \mathbb{Z}_N$ :
20 :   $\mathsf{attempt} \leftarrow \mathsf{attempt} + 1$
21 :   if $\mathsf{attempt} > \mathsf{max\text{-}attempts}$ :
22 :     $c^*_{\mathsf{rsa}} \leftarrow (\lightning, m_b)$
23 :     return $c^*_{\mathsf{rsa}}$
24 :   goto line 4
25 : $c^*_{\mathsf{rsa}} \leftarrow z^e \bmod N$
26 : return $c^*_{\mathsf{rsa}}$
```

Oracle $\text{DEC}(c_{\mathsf{rsa}})$

```
1 :  require $(c_{\mathsf{rsa}} \neq c^*_{\mathsf{rsa}}) \wedge (c_{\mathsf{rsa}} \in \mathbb{Z}_N)$
2 :  $z \leftarrow (c_{\mathsf{rsa}})^d \bmod N$
3 :  $p_{\mathsf{rsa}} \leftarrow z$    // Parse $z$ as a 2048-bit string.
4 :  $r \,\|\, c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}$    // s.t. $|r| = 256$, $|c_{\mathsf{ige}}| = 1792$.
5 :  $K \leftarrow \mathrm{H}(c_{\mathsf{ige}}) \oplus r$
6 :  $p_{\mathsf{ige}} \leftarrow \mathsf{IGE}^{\mathrm{IC},\mathrm{IC}^{-1}}.\mathsf{Dec}(K, 0^{256}, c_{\mathsf{ige}})$
7 :  $m_{\mathsf{padded}} \leftarrow \mathsf{reverse}(p_{\mathsf{ige}}[0 : 1536])$
8 :  $h \leftarrow p_{\mathsf{ige}}[1536 : 1792]$
9 :  if $h \neq \mathrm{H}(K \,\|\, m_{\mathsf{padded}})$ : return $\bot$
10 : $m \leftarrow \mathsf{RemovePadding}(m_{\mathsf{padded}})$
11 : return $m$
```

Random oracle $\mathrm{H}(a)$ for $a \in \{0,1\}^{1792}$

```
1 :  if $\mathsf{T}[a] = \bot$ : $\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256}$
2 :  return $\mathsf{T}[a]$
```

Ideal cipher $\mathrm{IC}(K, u)$ for $K \in \{0,1\}^{256}$, $u \in \{0,1\}^{128}$

```
1 :  if $\mathsf{A}_K[u] = \bot$ :
2 :    $v \leftarrow\$ \{0,1\}^{128} \setminus \mathcal{R}_K$
3 :    $\mathsf{AddRelationToIC}(K, u, v)$
4 :  return $\mathsf{A}_K[u]$
```

Ideal cipher $\mathrm{IC}^{-1}(K, v)$ for $K \in \{0,1\}^{256}$, $v \in \{0,1\}^{128}$

```
1 :  if $\mathsf{B}_K[v] = \bot$ :
2 :    $u \leftarrow\$ \{0,1\}^{128} \setminus \mathcal{D}_K$
3 :    $\mathsf{AddRelationToIC}(K, u, v)$
4 :  return $\mathsf{B}_K[v]$
```

Function $\mathsf{AddRelationToIC}(K, u, v)$

```
1 :  $S_{\mathsf{IC}} \leftarrow S_{\mathsf{IC}} \cup \{K\}$
2 :  $\mathsf{A}_K[u] \leftarrow v$ ; $\mathsf{B}_K[v] \leftarrow u$
3 :  $\mathcal{D}_K \leftarrow \mathcal{D}_K \cup \{u\}$
4 :  $\mathcal{R}_K \leftarrow \mathcal{R}_K \cup \{v\}$
```

Fig. 29. Game G_0 for the proof of Theorem 4.

```
Games $G_1$–$G_7$ : Oracle $\textsc{Enc}(m_0, m_1)$
1 :  require $(c^*_{\mathsf{rsa}} = \bot) \land (|m_0| = |m_1|)$
2 :  require $m_0, m_1 \in \mathcal{M}$
3 :  attempt $\leftarrow 1$ ; $K \leftarrow\!\!\$\ \{0,1\}^{256}$
4 :  if $K \in S_{\mathsf{IC}}$ :
5 :    $\mathsf{bad}_0 \leftarrow \mathtt{true}$
6 :    abort$(\mathtt{false})$    // $G_2$–$G_7$
7 :  $pad \leftarrow\!\!\$\ \{0,1\}^{1536-|m_b|}$
8 :  $m_{\mathsf{padded}} \leftarrow m_b \parallel pad$
9 :  $h \leftarrow \mathrm{H}(K \parallel m_{\mathsf{padded}})$
10 : $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \parallel h$
11 : $x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
12 : $x_1 \parallel \ldots \parallel x_{14} \leftarrow p_{\mathsf{ige}}$    // Parse into 128-bit blocks.
13 : for $i = 1, \ldots, 14$ :
14 :   $u_i \leftarrow x_i \oplus y_{i-1}$
15 :   if $\mathsf{A}_K[u_i] = \bot$ :
16 :     $y_i \leftarrow \mathsf{IC}(K, u_i) \oplus x_{i-1}$    // $G_1$-$G_2$
17 :     $y_i \leftarrow\!\!\$\ \{0,1\}^{128}$ ; $v_i \leftarrow y_i \oplus x_{i-1}$    // $G_3$-$G_7$
18 :     if $v_i \in \mathcal{R}_K$ :    // $G_3$-$G_7$
19 :       $\mathsf{bad}_1 \leftarrow \mathtt{true}$    // $G_3$-$G_7$
20 :       $v_i \leftarrow\!\!\$\ \{0,1\}^{128} \setminus \mathcal{R}_K$ ; $y_i \leftarrow v_i \oplus x_{i-1}$    // $G_3$
21 :     $\mathsf{AddRelationToIC}(K, u_i, v_i)$    // $G_3$-$G_7$
22 :   else :    // $\mathsf{A}_K[u_i] \neq \bot$
23 :     $\mathsf{bad}_2 \leftarrow \mathtt{true}$
24 :     $y_i \leftarrow \mathsf{IC}(K, u_i) \oplus x_{i-1}$    // $G_1$-$G_4$
25 :     $y_i \leftarrow\!\!\$\ \{0,1\}^{128}$ ; $v_i \leftarrow y_i \oplus x_{i-1}$    // $G_5$-$G_7$
26 :     $\mathsf{AddRelationToIC}(K, u_i, v_i)$    // $G_5$-$G_7$
27 : $c_{\mathsf{ige}} \leftarrow y_1 \parallel \ldots \parallel y_{14}$
28 : if $\mathsf{T}[c_{\mathsf{ige}}] \neq \bot$ :
29 :   $\mathsf{bad}_3 \leftarrow \mathtt{true}$
30 :   abort$(\mathtt{false})$    // $G_6$-$G_7$
31 : $r \leftarrow \mathrm{H}(c_{\mathsf{ige}}) \oplus K$    // $G_1$-$G_6$
32 : $r \leftarrow\!\!\$\ \{0,1\}^{256}$ ; $\mathsf{T}[c_{\mathsf{ige}}] \leftarrow r \oplus K$    // $G_7$
33 : $p_{\mathsf{rsa}} \leftarrow r \parallel c_{\mathsf{ige}}$
34 : $z \leftarrow p_{\mathsf{rsa}}$    // Parse $p_{\mathsf{rsa}}$ as an integer.
35 : if $z \notin \mathbb{Z}_N$ :
36 :   attempt $\leftarrow$ attempt $+ 1$
37 :   if attempt $>$ max-attempts :
38 :     $c^*_{\mathsf{rsa}} \leftarrow (\lightning, m_b)$ ; return $c^*_{\mathsf{rsa}}$
39 :   goto line 3
40 : $c^*_{\mathsf{rsa}} \leftarrow z^e \bmod N$
41 : return $c^*_{\mathsf{rsa}}$
```

```
Games $G_8$–$G_{11}$ : Oracle $\textsc{Enc}(m_0, m_1)$
1 :  require $(c^*_{\mathsf{rsa}} = \bot) \land (|m_0| = |m_1|)$
2 :  require $m_0, m_1 \in \mathcal{M}$
3 :  attempt $\leftarrow 1$
4 :  $p_{\mathsf{rsa}} \leftarrow\!\!\$\ \{0,1\}^{2048}$
5 :  $z \leftarrow p_{\mathsf{rsa}}$    // Parse $p_{\mathsf{rsa}}$ as an integer.
6 :  $r \parallel c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}$    // s.t. $|r| = 256$, $|c_{\mathsf{ige}}| = 1792$.
7 :  // Parse into 128-bit blocks.
8 :  $y_1 \parallel \ldots \parallel y_{14} \leftarrow c_{\mathsf{ige}}$
9 :  if $\mathsf{T}[c_{\mathsf{ige}}] \neq \bot$ : abort$(\mathtt{false})$    // $G_{11}$
10 : $K \leftarrow\!\!\$\ \{0,1\}^{256}$
11 : $\mathsf{T}[c_{\mathsf{ige}}] \leftarrow r \oplus K$    // $G_{11}$
12 : if $K \in S_{\mathsf{IC}}$ : abort$(\mathtt{false})$
13 : $pad \leftarrow\!\!\$\ \{0,1\}^{1536-|m_b|}$
14 : $m_{\mathsf{padded}} \leftarrow m_b \parallel pad$
15 : if $c_{\mathsf{ige}} = K \parallel m_{\mathsf{padded}}$ :
16 :   $\mathsf{bad}_5 \leftarrow \mathtt{true}$
17 :   abort$(\mathtt{false})$    // $G_{10}$–$G_{11}$
18 : $h \leftarrow \mathrm{H}(K \parallel m_{\mathsf{padded}})$
19 : $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \parallel h$
20 : $x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
21 : // Parse into 128-bit blocks.
22 : $x_1 \parallel \ldots \parallel x_{14} \leftarrow p_{\mathsf{ige}}$
23 : for $i = 1, \ldots, 14$ :
24 :   $u_i \leftarrow x_i \oplus y_{i-1}$
25 :   $v_i \leftarrow y_i \oplus x_{i-1}$
26 :   $\mathsf{AddRelationToIC}(K, u_i, v_i)$
27 : if $\mathsf{T}[c_{\mathsf{ige}}] \neq \bot$ : abort$(\mathtt{false})$    // $G_8$–$G_{10}$
28 : $\mathsf{T}[c_{\mathsf{ige}}] \leftarrow r \oplus K$    // $G_8$–$G_{10}$
29 : if $z \notin \mathbb{Z}_N$ :
30 :   attempt $\leftarrow$ attempt $+ 1$
31 :   if attempt $>$ max-attempts :
32 :     $\mathsf{bad}_4 \leftarrow \mathtt{true}$
33 :     $c^*_{\mathsf{rsa}} \leftarrow (\lightning, m_b)$    // $G_8$
34 :     return $c^*_{\mathsf{rsa}}$    // $G_8$
35 :     abort$(\mathtt{false})$    // $G_9$–$G_{11}$
36 :   goto line 4
37 : $c^*_{\mathsf{rsa}} \leftarrow z^e \bmod N$
38 : return $c^*_{\mathsf{rsa}}$
```

Fig. 30. Games G_1–G_{11} for the proof of Theorem 4.

uniformly at random from the set $\{0,1\}^{128}$, in line 17 of oracle ENC in game G_4. Each value is assigned as an output of the ideal cipher (for distinct inputs $u_1, u_2, \ldots, u_{14}$). The flag $\mathsf{bad}_1^{G_4}$ is set if a collision is obtained between the sampled v_i values, in any of the max-attempts iterations (each constituting an independent experiment). We have

$$\Pr[G_3] - \Pr[G_4] \leq \Pr[\mathsf{bad}_1^{G_4}] \leq \mathsf{max\text{-}attempts} \cdot \mathsf{bb}(0, 14, 0, 2^{128}) \leq \frac{91}{2^{128}} < 2^{-121}.$$

Analysis of $G_4 \to G_5$. Consider game G_5. Adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ makes at most a single query to oracle ENC. Oracle ENC runs for at most max-attempts iterations. It guarantees that in each iteration the ideal cipher is used with a distinct, fresh key K. For a single key, consider the 14 values $u_1, u_2, \ldots, u_{14}$ that are calculated in line 14 of oracle ENC in game G_5 as follows:

$$u_i \leftarrow x_i \oplus y_{i-1}.$$

Note that $y_1, y_2, \ldots, y_{13}$ are sampled uniformly at random, and independently of the corresponding x_i. So each of $u_2, u_3, \ldots, u_{14}$ can likewise be seen as independent, uniformly random value from $\{0,1\}^{128}$. The flag $\mathsf{bad}_2^{G_5}$ is set if a collision is obtained between $u_1, u_2, \ldots, u_{14}$, in any of the max-attempts iterations (each constituting an independent experiment). We have

$$\Pr[G_4] - \Pr[G_5] \leq \Pr[\mathsf{bad}_2^{G_5}] \leq \mathsf{max\text{-}attempts} \cdot \mathsf{bb}(0, 14, 0, 2^{128}) \leq \frac{91}{2^{128}} < 2^{-121}.$$

Analysis of $G_5 \to G_6$. Consider game G_6. Adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ makes at most a single query to oracle ENC. Prior to calling ENC, the adversary can make at most n_H queries to oracle H and at most n_DEC queries to oracle DEC. Each query to H can populate at most 1 empty entry of the random oracle table T, and each query to DEC can populate at most 2 empty entries (via underlying queries to H). So when $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ queries oracle ENC, the table T might contain at most $\mathsf{n}_\mathrm{H} + 2 \cdot \mathsf{n}_\mathrm{DEC}$ non-empty entries. Oracle ENC runs for at most max-attempts iterations. In each iteration, ENC first queries $\mathrm{H}(K \parallel m_{\mathsf{padded}})$, then samples a uniformly random $c_{\mathsf{ige}} \in \{0,1\}^{1792}$ (i.e. by concatenating the 128-bit strings $y_1, y_2, \ldots, y_{14}$ that were sampled independently, and uniformly at random), checks that the table entry $\mathsf{T}[c_{\mathsf{ige}}]$ is empty, and finally queries $\mathrm{H}(c_{\mathsf{ige}})$. The flag $\mathsf{bad}_3^{G_6}$ is set if a collision occurs, i.e. if some $\mathsf{T}[c_{\mathsf{ige}}]$ was already initialised. We have

$$\begin{aligned}\Pr[G_5] - \Pr[G_6] \leq \Pr[\mathsf{bad}_3^{G_6}] &\leq \mathsf{bb}(\mathsf{n}_\mathrm{H} + 2 \cdot \mathsf{n}_\mathrm{DEC} + 1, \mathsf{max\text{-}attempts}, 1, 2^{1792}) \\ &\leq \frac{\mathsf{max\text{-}attempts} \cdot (\mathsf{n}_\mathrm{H} + 2 \cdot \mathsf{n}_\mathrm{DEC} + 1 + \mathsf{max\text{-}attempts} - 1)}{2^{1792}} \\ &< \frac{\mathsf{max\text{-}attempts} \cdot (\mathsf{n}_\mathrm{H} + \mathsf{n}_\mathrm{DEC} + \mathsf{max\text{-}attempts})}{2^{1791}}.\end{aligned}$$

Analysis of $G_6 \to G_7$. This game simply inlines the random oracle call into ENC which does not alter behaviour. We have

$$\Pr[G_6] = \Pr[G_7].$$

Analysis of $G_7 \to G_8$. Observe that the following code is evaluated throughout game G_7:

```
for $i = 1, \ldots, 14$: $y_i \leftarrow\$ \{0,1\}^{128}$
$c_{\mathsf{ige}} \leftarrow y_1 \parallel \ldots \parallel y_{14}$
$r \leftarrow\$ \{0,1\}^{256}$
$p_{\mathsf{rsa}} \leftarrow r \parallel c_{\mathsf{ige}}$
$z \leftarrow p_{\mathsf{rsa}}$    // Parse $p_{\mathsf{rsa}}$ as an integer.
```

We rewrite this code in a functionally equivalent way in lines 4 to 8 of game G_8:

$$\begin{aligned}
&p_{\mathsf{rsa}} \leftarrow\!\!\$\ \{0,1\}^{2048} \\
&z \leftarrow p_{\mathsf{rsa}} \quad \text{// Parse } p_{\mathsf{rsa}} \text{ as an integer.} \\
&r \parallel c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}} \quad \text{// s.t. } |r| = 256, |c_{\mathsf{ige}}| = 1792. \\
&y_1 \parallel \ldots \parallel y_{14} \leftarrow c_{\mathsf{ige}} \quad \text{// Parse into 128-bit blocks.}
\end{aligned}$$

The rest of game G_7's code is rewritten unchanged in game G_8 (i.e. the order of the instructions is kept intact, not just the instructions themselves). It follows that games G_7 and G_8 are functionally equivalent, and

$$\Pr[G_7] = \Pr[G_8].$$

Analysis of $G_8 \to G_9$**.** Consider game G_9. Adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ makes at most a single query to oracle ENC. The oracle ENC repeatedly samples $p_{\mathsf{rsa}} \leftarrow\!\!\$\ \{0,1\}^{2048}$ until its integer representation z is in $\mathbb{Z}_N$, or until $\mathsf{max\text{-}attempts}$ consecutive p_{rsa} values were sampled and discarded. If the latter occurred, then the flag $\mathsf{bad}_4^{G_9}$ is set and the game aborts, returning `false`. The probability of this happening is $\left(\frac{2^{2048}-N}{2^{2048}}\right)^{\mathsf{max\text{-}attempts}}$ and we have

$$\Pr[G_8] - \Pr[G_9] \le \Pr[\mathsf{bad}_4^{G_9}] \le \left(\frac{2^{2048}-N}{2^{2048}}\right)^{\mathsf{max\text{-}attempts}} < \left(\frac{2^{2047}}{2^{2048}}\right)^{\mathsf{max\text{-}attempts}} = 2^{-\mathsf{max\text{-}attempts}}.$$

Analysis of $G_9 \to G_{10}$**.** Consider game G_{10}. Adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ makes at most a single query to oracle ENC. Oracle ENC runs for at most $\mathsf{max\text{-}attempts}$ iterations. In each iteration a random padding is sampled and appended to the challenge message as follows:

$$\begin{aligned}
&pad \leftarrow\!\!\$\ \{0,1\}^{1536-|m_b|} \\
&m_{\mathsf{padded}} \leftarrow m_b \parallel pad
\end{aligned}$$

The flag $\mathsf{bad}_5^{G_{10}}$ is set if the equality $c_{\mathsf{ige}} = K \parallel m_{\mathsf{padded}}$ is satisfied for the previously chosen values c_{ige} and K. The message space of PKE contains strings of length at most 1152 bits. It follows that

$$\Pr[G_9] - \Pr[G_{10}] \le \Pr[\mathsf{bad}_5^{G_{10}}] \le \frac{\mathsf{max\text{-}attempts}}{2^{384}}.$$

Analysis of $G_{10} \to G_{11}$**.** Game G_{11} is obtained from game G_{10} by moving the adjacent oracle DEC instructions

$$\begin{aligned}
&\textbf{if } \mathsf{T}[c_{\mathsf{ige}}] \ne \bot : \textbf{abort}(\texttt{false}) \\
&\mathsf{T}[c_{\mathsf{ige}}] \leftarrow r \oplus K
\end{aligned}$$

from lines 27 to 28, up to line 9 and line 11 respectively. A single new entry of the random oracle table T might get initialised between these lines when the instruction $\mathsf{H}(K \parallel m_{\mathsf{padded}})$ is evaluated in line 18 of oracle DEC. However, in both games this random oracle call can be reached only if $K \parallel m_{\mathsf{padded}}$ is distinct from c_{ige}. It follows that the input-output behaviour of oracle ENC is the same across games G_{10} and G_{11}, and

$$\Pr[G_{10}] = \Pr[G_{11}].$$

Analysis of $G_{11} \to G_{12}$**.** Game G_{12} rewrites game G_{11} to add bookkeeping tables $\mathsf{ige\text{-}ciphertext\text{-}to\text{-}key\text{-}map}$ and $\mathsf{ige\text{-}key\text{-}to\text{-}data\text{-}map}$. In game G_{12}, we set $\mathsf{ige\text{-}ciphertext\text{-}to\text{-}key\text{-}map}[c_{\mathsf{ige}}] \leftarrow (r, K)$ whenever oracle ENC sets $\mathsf{T}[c_{\mathsf{ige}}] \leftarrow r \oplus K$. And we set $\mathsf{ige\text{-}key\text{-}to\text{-}data\text{-}map}[K] \leftarrow (p_{\mathsf{ige}}, c_{\mathsf{ige}})$ whenever oracle ENC programs the ideal cipher to be consistent with $c_{\mathsf{ige}} = \mathsf{IGE}^{\mathrm{IC},\mathrm{IC}^{-1}}.\mathsf{Enc}(K, 0^{256}, p_{\mathsf{ige}})$. The code that programs the ideal cipher is now moved out into a separate function EmbedChallengeMessage. This function takes the key K as input, reads the corresponding plaintext-ciphertext pair $(p_{\mathsf{ige}}, c_{\mathsf{ige}})$ from ige-key-to-data-map$[K]$, and then programs the random oracle in the same way as previously done in G_{11}. Note that within a single Enc query, the table entry ige-ciphertext-to-key-map$[c_{\mathsf{ige}}]$ is initialised iff $\mathsf{T}[c_{\mathsf{ige}}]$ is initialised during the same query. Similarly, within a single Enc query, the table entry ige-key-to-data-map$[K]$ is initialised iff K gets added to the set S_{IC} during the same query. Game G_{12} contains conditions checking whether the entries ige-ciphertext-to-key-map$[c_{\mathsf{ige}}]$ and ige-key-to-data-map$[K]$ are initialised. These conditions do not affect the functionality of the game because they are always checked right after the equivalent conditions regarding $\mathsf{T}[c_{\mathsf{ige}}]$ being initialised, or a key belonging to the set S_{IC}. Games G_{11} and G_{12} are functionally equivalent, so

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

Analysis of $\mathrm{G}_{12} \to \mathrm{G}_{13}$**.** Game G_{13} rewrites game G_{12} to no longer perform the random-oracle and ideal-cipher programming during calls to its oracle Enc. In particular, the oracle Enc instructions $\mathsf{T}[c_{\mathsf{ige}}] \leftarrow r \oplus K$ in line 12 and EmbedChallengeMessage(K) in line 22 of game G_{12} – are not kept in game G_{13}. Instead game G_{13} programs its idealised oracles lazily. It is done when the corresponding oracle is queried on an input that is not yet initialised, but for which there exists a table entry (i.e. ige-ciphertext-to-key-map or ige-key-to-data-map) with instructions on how it should be initialised. The lazy sampling can be done because in game G_{12}, except for the RO and IC oracles themselves, only oracle Enc directly accesses the RO table T and the IC tables A, B. Oracle Enc only writes into these tables, not reads from them. And it only writes into them if the corresponding entry is not yet initialised. Games G_{12} and G_{13} are functionally equivalent, and

$$\Pr[\mathrm{G}_{12}] = \Pr[\mathrm{G}_{13}].$$

Analysis of $\mathrm{G}_{13} \to \mathrm{G}_{14}$**.** Games G_{13} and G_{14} are equivalent. In particular, game G_{14} rewrites G_{13} to add conditional statements inside oracle H and function EmbedChallengeMessage. All branches of the conditional statement in H contain a single instruction $\mathsf{T}[a] \leftarrow r \oplus K$, which is consistent with game G_{13}. Both branches of the conditional statement in EmbedChallengeMessage do nothing. We have

$$\Pr[\mathrm{G}_{13}] = \Pr[\mathrm{G}_{14}].$$

Analysis of $\mathrm{G}_{14} \to \mathrm{G}_{15}$**.** Consider game G_{15}. The flag $\mathsf{bad}_6^{\mathrm{G}_{15}}$ is set only when both of the following conditions are true: (1) oracle H was called on some value $a \neq c^*_{\mathsf{ige}}$ for which $\mathsf{T}[a] = \perp$ and ige-ciphertext-to-key-map$[a] \neq \perp$, and (2) the table entry ige-ciphertext-to-key-map$[a] = (r, K)$ contains $K \notin S_{\mathsf{IC}}$. The first condition means that H was queried on the suffix $a = c_{\mathsf{ige}}$ of some p_{rsa} that was previously sampled and discarded in oracle Enc, due to the integer representation of p_{rsa} being outside $\mathbb{Z}_N$. The second condition means that $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ did not simply obtain this value from the corresponding table entry ige-ciphertext-to-key-map that contains the discarded c_{ige}. Together it effectively means that $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ simply guessed such c_{ige} (and queried it to H directly, or triggered such a call to H from another oracle). Note that we do not need to analyse the possibility of Enc having sampled distinct p_{rsa} values that have the same suffix. We avoid this case by keeping the following conditional statement in Enc:

$$\textbf{if } \mathsf{T}[c_{\mathsf{ige}}] \neq \perp : \textbf{abort}(\texttt{false}).$$

There are $2^{2048} - N$ distinct values $p_{\mathsf{rsa}} \in \{0,1\}^{2048}$ whose integer representation is not in $\mathbb{Z}_N$. Amongst all such values, there are $\textbf{min}(2^{1792}, 2^{2048} - N)$ distinct 1792-bit suffixes. The most frequent suffix appears $\left\lceil \frac{2^{2048}-N}{\textbf{min}(2^{1792}, 2^{2048}-N)} \right\rceil$ times. So the probability of guessing a single a priori chosen suffix is at most

$$\frac{\left\lceil \frac{2^{2048}-N}{\psi} \right\rceil}{2^{2048} - N} \leq \frac{\frac{2^{2048}-N}{\psi} + 1}{2^{2048} - N} = \frac{1}{\psi} + \frac{1}{2^{2048} - N} \leq \frac{2}{\psi}.$$

```
Games G_12–G_13
1 :  $b \leftarrow\!\$ \{0,1\}$ ; $c^*_{rsa} \leftarrow \bot$ ; $S_{IC} \leftarrow \emptyset$
2 :  $(N,p,q,e,d) \leftarrow\!\$ \mathsf{RSA.KGen}()$ ; $pk \leftarrow (N,e)$
3 :  $b' \leftarrow\!\$ \mathcal{D}^{\textsc{Enc},\textsc{Dec},\mathrm{H},\mathrm{IC},\mathrm{IC}^{-1}}_{\mathsf{IND\text{-}CCA}}(pk)$
4 :  return $b = b'$

Oracle Enc$(m_0, m_1)$
1 :  require $(c^*_{rsa} = \bot) \wedge (|m_0| = |m_1|)$
2 :  require $m_0, m_1 \in \mathcal{M}$
3 :  attempt $\leftarrow 1$
4 :  $p_{rsa} \leftarrow\!\$ \{0,1\}^{2048}$
5 :  $z \leftarrow p_{rsa}$   // Parse $p_{rsa}$ as an integer.
6 :  $r \,\|\, c_{ige} \leftarrow p_{rsa}$   // s.t. $|r| = 256$, $|c_{ige}| = 1792$.
7 :  if $\mathsf{T}[c_{ige}] \neq \bot$ : abort(false)
8 :  if ige-ciphertext-to-key-map$[c_{ige}] \neq \bot$ :
9 :      abort(false)
10 : $K \leftarrow\!\$ \{0,1\}^{256}$
11 : ige-ciphertext-to-key-map$[c_{ige}] \leftarrow (r, K)$
12 : $\mathsf{T}[c_{ige}] \leftarrow r \oplus K$   // G_12
13 : if $K \in S_{IC}$ : abort(false)
14 : if ige-key-to-data-map$[K] \neq \bot$ :
15 :     abort(false)
16 : $pad \leftarrow\!\$ \{0,1\}^{1536-|m_b|}$
17 : $m_{padded} \leftarrow m_b \,\|\, pad$
18 : if $c_{ige} = K \,\|\, m_{padded}$ : abort(false)
19 : $h \leftarrow \mathrm{H}(K \,\|\, m_{padded})$
20 : $p_{ige} \leftarrow$ reverse$(m_{padded}) \,\|\, h$
21 : ige-key-to-data-map$[K] \leftarrow (p_{ige}, c_{ige})$
22 : EmbedChallengeMessage$(K)$   // G_12
23 : if $z \notin \mathbb{Z}_N$ :
24 :     attempt $\leftarrow$ attempt + 1
25 :     if attempt > max-attempts :
26 :         abort(false)
27 :     goto line 4
28 : $c^*_{rsa} \leftarrow z^e \bmod N$
29 : return $c^*_{rsa}$

Oracle Dec$(c_{rsa})$
// This oracle is identical to the
// corresponding oracle in game G_0 of Fig. 29.

Random oracle H$(a)$ for $a \in \{0,1\}^{1792}$
1 :  if $\mathsf{T}[a] = \bot$ :
2 :      if ige-ciphertext-to-key-map$[a] \neq \bot$ :
3 :          $\mathsf{T}[a] \leftarrow\!\$ \{0,1\}^{256}$   // G_12
4 :          $(r, K) \leftarrow$ ige-ciphertext-to-key-map$[a]$   // G_13
5 :          $\mathsf{T}[a] \leftarrow r \oplus K$   // G_13
6 :      else :   // ige-ciphertext-to-key-map$[a] = \bot$
7 :          $\mathsf{T}[a] \leftarrow\!\$ \{0,1\}^{256}$
8 :  return $\mathsf{T}[a]$

Ideal cipher IC$(K, u)$ for $K \in \{0,1\}^{256}$, $u \in \{0,1\}^{128}$
1 :  EmbedChallengeMessage$(K)$   // G_13
2 :  if $\mathsf{A}_K[u] = \bot$ :
3 :      $v \leftarrow\!\$ \{0,1\}^{128} \setminus \mathcal{R}_K$
4 :      AddRelationToIC$(K, u, v)$
5 :  return $\mathsf{A}_K[u]$

Ideal cipher IC$^{-1}(K, v)$ for $K \in \{0,1\}^{256}$, $v \in \{0,1\}^{128}$
1 :  EmbedChallengeMessage$(K)$   // G_13
2 :  if $\mathsf{B}_K[v] = \bot$ :
3 :      $u \leftarrow\!\$ \{0,1\}^{128} \setminus \mathcal{D}_K$
4 :      AddRelationToIC$(K, u, v)$
5 :  return $\mathsf{B}_K[v]$

Function EmbedChallengeMessage$(K)$
1 :  if $K \in S_{IC}$ : return $\bot$
2 :  if ige-key-to-data-map$[K] = \bot$ : return $\bot$
3 :  $(p_{ige}, c_{ige}) \leftarrow$ ige-key-to-data-map$[K]$
4 :  $x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
5 :  $x_1 \,\|\, \ldots \,\|\, x_{14} \leftarrow p_{ige}$   // Parse into 128-bit blocks.
6 :  $y_1 \,\|\, \ldots \,\|\, y_{14} \leftarrow c_{ige}$   // Parse into 128-bit blocks.
7 :  for $i = 1, \ldots, 14$ :
8 :      $u_i \leftarrow x_i \oplus y_{i-1}$
9 :      $v_i \leftarrow y_i \oplus x_{i-1}$
10 :     AddRelationToIC$(K, u_i, v_i)$

Function AddRelationToIC$(K, u, v)$
// This function is identical to the
// corresponding function in game G_0 of Fig. 29.
```

Fig. 31. Games G_{12}–G_{13} for the proof of Theorem 4.

Games G_{14}–G_{18}

```
1 : $b \leftarrow\$ \{0,1\}$ ; $c^*_{\mathsf{rsa}} \leftarrow \bot$ ; $S_{\mathsf{IC}} \leftarrow \emptyset$
2 : $c^*_{\mathsf{ige}} \leftarrow \bot$ ; $K^* \leftarrow \bot$
3 : $(N, p, q, e, d) \leftarrow\$ \mathsf{RSA.KGen}()$ ; $pk \leftarrow (N, e)$
4 : $b' \leftarrow\$ \mathcal{D}^{\mathrm{Enc,Dec,H,IC,IC}^{-1}}_{\mathsf{IND\text{-}CCA}}(pk)$
5 : return $b = b'$
```

Oracle $\mathrm{Enc}(m_0, m_1)$

```
1 : require $(c^*_{\mathsf{rsa}} = \bot) \wedge (|m_0| = |m_1|)$
2 : require $m_0, m_1 \in \mathcal{M}$
3 : $\mathsf{attempt} \leftarrow 1$
4 : $p_{\mathsf{rsa}} \leftarrow\$ \{0,1\}^{2048}$
5 : $z \leftarrow p_{\mathsf{rsa}}$   // Parse $p_{\mathsf{rsa}}$ as an integer.
6 : $r \,\|\, c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}$   // s.t. $|r| = 256$, $|c_{\mathsf{ige}}| = 1792$.
7 : if $\mathsf{T}[c_{\mathsf{ige}}] \neq \bot$ : abort(false)
8 : if $\mathsf{ige\text{-}ciphertext\text{-}to\text{-}key\text{-}map}[c_{\mathsf{ige}}] \neq \bot$ :
9 :     abort(false)
10 : $K \leftarrow\$ \{0,1\}^{256}$
11 : $\mathsf{ige\text{-}ciphertext\text{-}to\text{-}key\text{-}map}[c_{\mathsf{ige}}] \leftarrow (r, K)$
12 : if $K \in S_{\mathsf{IC}}$ : abort(false)
13 : if $\mathsf{ige\text{-}key\text{-}to\text{-}data\text{-}map}[K] \neq \bot$ :
14 :     abort(false)
15 : $pad \leftarrow\$ \{0,1\}^{1536-|m_b|}$
16 : $m_{\mathsf{padded}} \leftarrow m_b \,\|\, pad$
17 : if $c_{\mathsf{ige}} = K \,\|\, m_{\mathsf{padded}}$ : abort(false)
18 : $h \leftarrow \mathrm{H}(K \,\|\, m_{\mathsf{padded}})$
19 : $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \,\|\, h$
20 : $\mathsf{ige\text{-}key\text{-}to\text{-}data\text{-}map}[K] \leftarrow (p_{\mathsf{ige}}, c_{\mathsf{ige}})$
21 : if $z \notin \mathbb{Z}_N$ :
22 :     $\mathsf{attempt} \leftarrow \mathsf{attempt} + 1$
23 :     if $\mathsf{attempt} > \mathsf{max\text{-}attempts}$ :
24 :         abort(false)
25 :     goto line 4
26 : $c^*_{\mathsf{rsa}} \leftarrow z^e \bmod N$
27 : $c^*_{\mathsf{ige}} \leftarrow c_{\mathsf{ige}}$ ; $K^* \leftarrow K$
28 : return $c^*_{\mathsf{rsa}}$
```

Oracle $\mathrm{Dec}(c_{\mathsf{rsa}})$

```
// This oracle is identical to the
// corresponding oracle in game $\mathrm{G}_0$ of Fig. 29.
```

Function $\mathsf{AddRelationToIC}(K, u, v)$

```
// This function is identical to the
// corresponding function in game $\mathrm{G}_0$ of Fig. 29.
```

Random oracle $\mathrm{H}(a)$ for $a \in \{0,1\}^{1792}$

```
1 : if $\mathsf{T}[a] = \bot$ :
2 :     if $\mathsf{ige\text{-}ciphertext\text{-}to\text{-}key\text{-}map}[a] \neq \bot$ :
3 :         $(r, K) \leftarrow \mathsf{ige\text{-}ciphertext\text{-}to\text{-}key\text{-}map}[a]$
4 :         if $a = c^*_{\mathsf{ige}}$ :   // Might have inverted RSA.
5 :             $\mathsf{T}[a] \leftarrow r \oplus K$
6 :         else :   // $a \neq c^*_{\mathsf{ige}}$
7 :             if $K \notin S_{\mathsf{IC}}$ :
8 :                 $\mathsf{bad}_6 \leftarrow \mathtt{true}$   // Directly guessed $c_{\mathsf{ige}} \neq c_{\mathsf{ige}}$.
9 :                 $\mathsf{T}[a] \leftarrow r \oplus K$   // $\mathrm{G}_{14}$
10 :                abort(false)   // $\mathrm{G}_{15}$–$\mathrm{G}_{17}$
11 :                $\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256}$   // $\mathrm{G}_{18}$
12 :            else :   // $K \in S_{\mathsf{IC}}$
13 :                $\mathsf{bad}_8 \leftarrow \mathtt{true}$   // Recovered $c_{\mathsf{ige}}$ via $K$.
14 :                $\mathsf{T}[a] \leftarrow r \oplus K$   // $\mathrm{G}_{14}$–$\mathrm{G}_{16}$
15 :                $\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256}$   // $\mathrm{G}_{17}$–$\mathrm{G}_{18}$
16 :    else :   // $\mathsf{ige\text{-}ciphertext\text{-}to\text{-}key\text{-}map}[a] = \bot$
17 :        $\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256}$
18 : return $\mathsf{T}[a]$
```

Ideal cipher $\mathrm{IC}(K, u)$ for $K \in \{0,1\}^{256}$, $u \in \{0,1\}^{128}$

Ideal cipher $\mathrm{IC}^{-1}(K, v)$ for $K \in \{0,1\}^{256}$, $v \in \{0,1\}^{128}$

```
// These oracles are identical to the
// corresponding oracles in game $\mathrm{G}_{13}$ of Fig. 31.
```

Function $\mathsf{EmbedChallengeMessage}(K)$

```
1 : if $K \in S_{\mathsf{IC}}$ : return $\bot$
2 : if $\mathsf{ige\text{-}key\text{-}to\text{-}data\text{-}map}[K] = \bot$ : return $\bot$
3 : $(p_{\mathsf{ige}}, c_{\mathsf{ige}}) \leftarrow \mathsf{ige\text{-}key\text{-}to\text{-}data\text{-}map}[K]$
4 : if $\mathsf{T}[c_{\mathsf{ige}}] = \bot$ :
5 :     $\mathsf{bad}_7 \leftarrow \mathtt{true}$   // Directly guessed $K$.
6 :     abort(false)   // $\mathrm{G}_{16}$–$\mathrm{G}_{17}$
7 :     return $\bot$   // $\mathrm{G}_{18}$
8 : else :   // $\mathsf{T}[c_{\mathsf{ige}}] \neq \bot$
9 :     if $c_{\mathsf{ige}} \neq c^*_{\mathsf{ige}}$ :
10 :        $\mathsf{bad}_8 \leftarrow \mathtt{true}$   // Recovered $K$ via $c_{\mathsf{ige}}$.
11 :        return $\bot$   // $\mathrm{G}_{17}$–$\mathrm{G}_{18}$
12 : // This line is reached only if $(\mathsf{T}[c_{\mathsf{ige}}] \neq \bot) \wedge (c_{\mathsf{ige}} = c^*_{\mathsf{ige}})$.
13 : $x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
14 : $x_1 \,\|\, \ldots \,\|\, x_{14} \leftarrow p_{\mathsf{ige}}$   // Parse into 128-bit blocks.
15 : $y_1 \,\|\, \ldots \,\|\, y_{14} \leftarrow c_{\mathsf{ige}}$   // Parse into 128-bit blocks.
16 : for $i = 1, \ldots, 14$ :
17 :     $u_i \leftarrow x_i \oplus y_{i-1}$
18 :     $v_i \leftarrow y_i \oplus x_{i-1}$
19 :     $\mathsf{AddRelationToIC}(K, u_i, v_i)$
```

Fig. 32. Games G_{14}–G_{18} for the proof of Theorem 4.

per attempt, where $\psi = \mathbf{min}(2^{1792}, 2^{2048} - N)$.

Adversary $\mathcal{D}_{\textsf{IND-CCA}}$ makes at most a single query to oracle ENC. Oracle ENC runs for at most max-attempts iterations, discarding at most max-attempts $- 1$ values of $p_{\textsf{rsa}}$. Each of the discarded $p_{\textsf{rsa}}$ might have contained a distinct 1792-bit suffix.

Oracle H is called at most $\mathsf{n_H}$ times directly by adversary $\mathcal{D}_{\textsf{IND-CCA}}$, at most $2 \cdot \mathsf{n_{DEC}}$ times from the queries $\mathcal{D}_{\textsf{IND-CCA}}$ makes to DEC, and at most max-attempts times during the single query $\mathcal{D}_{\textsf{IND-CCA}}$ makes to ENC.

It follows that $\Pr[G_{14}] - \Pr[G_{15}] \leq$

$$\begin{aligned}\Pr[\mathsf{bad}_6^{G_{15}}] &\leq (\mathsf{n_H} + 2 \cdot \mathsf{n_{DEC}} + \textsf{max-attempts}) \cdot (\textsf{max-attempts} - 1) \cdot \frac{2}{\mathbf{min}(2^{1792}, 2^{2048} - N)} \\ &< \frac{\textsf{max-attempts} \cdot (\mathsf{n_H} + \mathsf{n_{DEC}} + \textsf{max-attempts})}{\mathbf{min}(2^{1790}, 2^{2046} - N)}.\end{aligned}$$

Analysis of $G_{15} \to G_{16}$**.** Consider game G_{16}. The flag $\mathsf{bad}_7^{G_{16}}$ is set when EmbedChallengeMessage is called with a key $K \notin S_{\textsf{IC}}$ as input for which the table entry ige-key-to-data-map$[K]$ contains a ciphertext $c_{\textsf{ige}}$ that was not previously queried to the random oracle H. But in oracle ENC every key K is sampled independently of the corresponding ciphertext $c_{\textsf{ige}}$. The only condition oracle ENC applies to the sampled keys is "**if** $c_{\textsf{ige}} = K \parallel m_{\textsf{padded}}$: **abort**(`false`)". This implies that, even if the adversary obtained some $c_{\textsf{ige}}$ but did not query it into H, then $c_{\textsf{ige}}$ could have been matched to any of the $2^{256} - 1$ keys that are not the prefix of $c_{\textsf{ige}}$, each with the same probability. Note that even if we hypothetically allowed $\mathcal{D}_{\textsf{IND-CCA}}$ to query its oracle DEC on the challenge ciphertext $c^*_{\textsf{rsa}}$ as input, the decryption oracle would still have to query the random oracle H on $c^*_{\textsf{ige}}$ in order to learn the challenge key K^*. This implies that our analysis of this transition would remain valid.

Adversary $\mathcal{D}_{\textsf{IND-CCA}}$ makes at most a single query to oracle ENC. Oracle ENC runs for at most max-attempts iterations, sampling at most max-attempts distinct keys in total, and creating an entry in the table ige-key-to-data-map for each of them. The function EmbedChallengeMessage is called with a single key as input per each query to IC, IC^{-1}, and DEC (i.e. a single query to DEC could trigger many calls to EmbedChallengeMessage, but with the same key as input). We have

$$\Pr[\mathsf{bad}_7^{G_{16}}] \leq \frac{\textsf{max-attempts} \cdot (\mathsf{n_{DEC}} + \mathsf{n_{IC}})}{2^{256} - 1} < \frac{\textsf{max-attempts} \cdot (\mathsf{n_{DEC}} + \mathsf{n_{IC}})}{2^{255}}.$$

Analysis of $G_{16} \to G_{17}$**.** Consider game G_{17}. The flag $\mathsf{bad}_8^{G_{17}}$ can be set in two different places in this game, each capturing a distinct case:

- The flag $\mathsf{bad}_8^{G_{17}}$ is set in oracle H when it is queried on an input $c_{\textsf{ige}} \neq c^*_{\textsf{rsa}}$ such that $\mathsf{T}[c_{\textsf{ige}}] = \perp$, and ige-ciphertext-to-key-map$[c_{\textsf{ige}}] = (r, K)$ for some key $K \in S_{\textsf{IC}}$. This could only occur if the adversary previously guessed K. We ruled out this case in the transition $G_{15} \to G_{16}$, making the game G_{17} abort before it could have reached this case.
- The flag $\mathsf{bad}_8^{G_{17}}$ is set in function EmbedChallengeMessage when it is called on an input K such that $K \notin S_{\textsf{IC}}$, and $(p_{\textsf{ige}}, c_{\textsf{ige}}) \leftarrow$ ige-key-to-data-map$[K]$ for a ciphertext $c_{\textsf{ige}} \neq c^*_{\textsf{rsa}}$ satisfying $\mathsf{T}[c_{\textsf{ige}}] \neq \perp$. This could only occur if the adversary previously guessed $c_{\textsf{ige}}$ without querying its ideal cipher on the key K first. We ruled out this case in the transition $G_{14} \to G_{15}$, making the game G_{17} abort before it could have reached this case.

It follows that

$$\Pr[G_{16}] - \Pr[G_{17}] \leq \Pr[\mathsf{bad}_8^{G_{17}}] = 0.$$

Analysis of $\mathrm{G}_{17} \to \mathrm{G}_{18}$**.** Consider game G_{18}. Game G_{18} is obtained from game G_{17} by replacing the **abort**(`false`) calls in line 10 in oracle H and line 6 in function EmbedChallengeMessage with some code in line 11 and line 7 respectively. An **abort**(`false`) call causes the adversary to immediately lose the game. The advantage of an adversary could only increase by replacing it with anything else. We have

$$\Pr[\mathrm{G}_{17}] \leq \Pr[\mathrm{G}_{18}].$$

Analysis of $\mathrm{G}_{18} \to \mathrm{G}_{19}$**.** Game of G_{19} is obtained from game G_{18} by changing the following code. Oracle H of G_{19} simply removes the dead code from the corresponding oracle from G_{18}. The function EmbedChallengeMessage in G_{18} immediately returns $\perp$ unless it is queried on a key K whose table entry ige-key-to-data-map$[K]$ contains c^*_{ige}, and unless $\mathsf{T}[c^*_{\mathsf{ige}}] \neq \perp$. Note that oracle ENC in game G_{18} contains the conditional statement "**if** $\mathsf{T}[c_{\mathsf{ige}}] \neq \perp$: **abort**(`false`)". This means that only the challenge key K^* could contain c^*_{ige} in its entry for table ige-key-to-data-map. The function EmbedChallengeMessage in game G_{19} modifies the corresponding function from G_{18} accordingly. It uses the global of K^* and c^*_{ige} directly, without having to first check the contents of ige-key-to-data-map$[K]$. Games G_{18} and G_{19} are functionally equivalent, so

$$\Pr[\mathrm{G}_{18}] = \Pr[\mathrm{G}_{19}].$$

Analysis of $\mathrm{G}_{19} \to \mathrm{G}_{20}$**.** Consider game G_{20}. Game G_{20} is obtained from game G_{19} by replacing the **abort**(`false`) call in line 24 of oracle ENC with some pseudocode in lines 25 to 27, and simply removing multiple other **abort**(`false`) calls from ENC. An **abort**(`false`) call causes the adversary to immediately lose the game. The advantage of an adversary could only increase by replacing such a call with anything else, or by removing it. We have

$$\Pr[\mathrm{G}_{19}] \leq \Pr[\mathrm{G}_{20}].$$

Analysis of $\mathrm{G}_{20} \to \mathrm{G}_{21}$**.** Game G_{21} simply rewrites game G_{20}, removing the dead code from the ENC oracle. In particular, note that we are no longer using ige-ciphertext-to-key-map anywhere and can thus stop writing to it. We have

$$\Pr[\mathrm{G}_{20}] = \Pr[\mathrm{G}_{21}].$$

Analysis of $\mathrm{G}_{21} \to \mathrm{G}_{22}$**.** Game G_{22} is obtained from game G_{21} by moving the following code from oracle ENC to function EmbedChallengeMessage:

$$h \leftarrow \mathrm{H}(K \parallel m_{\mathsf{padded}})$$
$$p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \parallel h.$$

In this transition, we rely on the two **abort**(`false`) calls that still remain in the ENC oracle. In particular:

- The conditional statement "**if** $\mathsf{T}[c_{\mathsf{ige}}] \neq \perp$: **abort**(`false`)" guarantees that c^*_{ige} will not be the same as one of the $K \parallel m_{\mathsf{padded}}$ values from the prior rejection sampling iterations. This means that the functionality of the game does not change if the random oracle H is no longer called on the discarded $K \parallel m_{\mathsf{padded}}$.
- The conditional statement "**if** $c_{\mathsf{ige}} = K \parallel m_{\mathsf{padded}}$: **abort**(`false`)" guarantees that the behaviour of $\mathrm{H}(c^*_{\mathsf{ige}})$ does not change even if the game does not immediately call $\mathrm{H}(K^* \parallel m^*_{\mathsf{padded}})$ inside the ENC oracle (instead delaying this query until function EmbedChallengeMessage is called).

It follows that

$$\Pr[\mathrm{G}_{21}] = \Pr[\mathrm{G}_{22}].$$

```
Games $G_{19}$–$G_{20}$
1 : $b \leftarrow\$ \{0,1\}$ ; $c^*_{\text{rsa}} \leftarrow \bot$ ; $S_{\text{IC}} \leftarrow \emptyset$
2 : $c^*_{\text{ige}} \leftarrow \bot$ ; $K^* \leftarrow \bot$ ; $r^* \leftarrow \bot$ ; $p^*_{\text{ige}} \leftarrow \bot$
3 : $(N,p,q,e,d) \leftarrow\$ \text{RSA.KGen}()$ ; $pk \leftarrow (N,e)$
4 : $b' \leftarrow\$ \mathcal{D}^{\text{Enc,Dec,H,IC,IC}^{-1}}_{\text{IND-CCA}}(pk)$
5 : return $b = b'$

Oracle $\text{Enc}(m_0, m_1)$
1 : require $(c^*_{\text{rsa}} = \bot) \wedge (|m_0| = |m_1|)$
2 : require $m_0, m_1 \in \mathcal{M}$
3 : $\text{attempt} \leftarrow 1$
4 : $p_{\text{rsa}} \leftarrow\$ \{0,1\}^{2048}$
5 : $z \leftarrow p_{\text{rsa}}$   // Parse $p_{\text{rsa}}$ as an integer.
6 : $r \| c_{\text{ige}} \leftarrow p_{\text{rsa}}$   // s.t. $|r| = 256, |c_{\text{ige}}| = 1792$.
7 : if $\mathsf{T}[c_{\text{ige}}] \neq \bot$ : abort(false)
8 : if ige-ciphertext-to-key-map$[c_{\text{ige}}] \neq \bot$ :   // $G_{19}$
9 :   abort(false)   // $G_{19}$
10 : $K \leftarrow\$ \{0,1\}^{256}$
11 : ige-ciphertext-to-key-map$[c_{\text{ige}}] \leftarrow (r, K)$
12 : if $K \in S_{\text{IC}}$ : abort(false)   // $G_{19}$
13 : if ige-key-to-data-map$[K] \neq \bot$ :   // $G_{19}$
14 :   abort(false)   // $G_{19}$
15 : $pad \leftarrow\$ \{0,1\}^{1536-|m_b|}$
16 : $m_{\text{padded}} \leftarrow m_b \| pad$
17 : if $c_{\text{ige}} = K \| m_{\text{padded}}$ : abort(false)
18 : $h \leftarrow \text{H}(K \| m_{\text{padded}})$
19 : $p_{\text{ige}} \leftarrow \text{reverse}(m_{\text{padded}}) \| h$
20 : ige-key-to-data-map$[K] \leftarrow (p_{\text{ige}}, c_{\text{ige}})$
21 : if $z \notin \mathbb{Z}_N$ :
22 :   $\text{attempt} \leftarrow \text{attempt} + 1$
23 :   if $\text{attempt} > \text{max-attempts}$ :
24 :     abort(false)   // $G_{19}$
25 :     $z \leftarrow\$ \{0, \ldots, N-1\}$   // $G_{20}$
26 :     $p_{\text{rsa}} \leftarrow z$   // Parse $z$ as a 2048-bit string. // $G_{20}$
27 :     goto line 6   // $G_{20}$
28 :   goto line 4
29 : $c^*_{\text{rsa}} \leftarrow z^e \bmod N$
30 : $c^*_{\text{ige}} \leftarrow c_{\text{ige}}$ ; $K^* \leftarrow K$ ; $r^* \leftarrow r$ ; $p^*_{\text{ige}} \leftarrow p_{\text{ige}}$
31 : return $c^*_{\text{rsa}}$

Oracle $\text{Dec}(c_{\text{rsa}})$
// This oracle is identical to the
// corresponding oracle in game $G_0$ of Fig. 29.

Random oracle $\text{H}(a)$ for $a \in \{0,1\}^{1792}$
1 : if $\mathsf{T}[a] = \bot$ :
2 :   if $a = c^*_{\text{ige}}$ :
3 :     $\mathsf{T}[a] \leftarrow r^* \oplus K^*$
4 :   else :   // $a \neq c^*_{\text{ige}}$
5 :     $\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256}$
6 : return $\mathsf{T}[a]$

Ideal cipher $\text{IC}(K, u)$
Ideal cipher $\text{IC}^{-1}(K, v)$
// These oracles are identical to the
// corresponding oracles in game $G_{13}$ of Fig. 31.

Function EmbedChallengeMessage$(K)$
1 : if $K \in S_{\text{IC}}$ : return $\bot$
2 : if $K \neq K^*$ : return $\bot$
3 : if $\mathsf{T}[c^*_{\text{ige}}] = \bot$ : return $\bot$
4 : $x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
5 : $x_1 \| \ldots \| x_{14} \leftarrow p^*_{\text{ige}}$   // Parse into 128-bit blocks.
6 : $y_1 \| \ldots \| y_{14} \leftarrow c^*_{\text{ige}}$   // Parse into 128-bit blocks.
7 : for $i = 1, \ldots, 14$ :
8 :   $u_i \leftarrow x_i \oplus y_{i-1}$
9 :   $v_i \leftarrow y_i \oplus x_{i-1}$
10 :   AddRelationToIC$(K, u_i, v_i)$

Function AddRelationToIC$(K, u, v)$
// This function is identical to the
// corresponding function in game $G_0$ of Fig. 29.
```

Fig. 33. Games G_{19}–G_{20} for the proof of Theorem 4.

Games G_{21}–G_{26}

```
$b \leftarrow\!\!\$ \{0,1\}$ ; $c^*_{rsa} \leftarrow \perp$ ; $S_{IC} \leftarrow \emptyset$
$c^*_{ige} \leftarrow \perp$ ; $K^* \leftarrow \perp$ ; $r^* \leftarrow \perp$ ; $p^*_{ige} \leftarrow \perp$
$m^*_0 \leftarrow \perp$ ; $m^*_1 \leftarrow \perp$ ; $m^*_{padded} \leftarrow \perp$
$(N, p, q, e, d) \leftarrow\!\!\$ \mathsf{RSA.KGen}()$ ; $pk \leftarrow (N, e)$
$b' \leftarrow\!\!\$ \mathcal{D}^{\text{ENC,DEC,H,IC,IC}^{-1}}_{\text{IND-CCA}}(pk)$
return $b = b'$
```

Oracle $\text{ENC}(m_0, m_1)$

```
require $(c^*_{rsa} = \perp) \wedge (|m_0| = |m_1|)$
require $m_0, m_1 \in \mathcal{M}$
$\mathsf{attempt} \leftarrow 1$
$p_{rsa} \leftarrow\!\!\$ \{0,1\}^{2048}$
$z \leftarrow p_{rsa}$   // Parse $p_{rsa}$ as an integer.
$r \parallel c_{ige} \leftarrow p_{rsa}$   // s.t. $|r| = 256$, $|c_{ige}| = 1792$.
if $\mathsf{T}[c_{ige}] \neq \perp$ : abort(false)   // $G_{21}$–$G_{24}$
$K \leftarrow\!\!\$ \{0,1\}^{256}$   // $G_{21}$–$G_{23}$
$pad \leftarrow\!\!\$ \{0,1\}^{1536-|m_b|}$   // $G_{21}$–$G_{23}$
$m_{padded} \leftarrow m_b \parallel pad$   // $G_{21}$–$G_{23}$
if $c_{ige} = K \parallel m_{padded}$ :   // $G_{21}$–$G_{22}$
    abort(false)   // $G_{21}$–$G_{22}$
$h \leftarrow \mathrm{H}(K \parallel m_{padded})$   // $G_{21}$
$p_{ige} \leftarrow \mathsf{reverse}(m_{padded}) \parallel h$   // $G_{21}$
if $z \notin \mathbb{Z}_N$ :
    $\mathsf{attempt} \leftarrow \mathsf{attempt} + 1$
    if $\mathsf{attempt} > \mathsf{max\text{-}attempts}$ :
        $z \leftarrow\!\!\$ \{0, \ldots, N-1\}$
        $p_{rsa} \leftarrow z$   // Parse $z$ as a 2048-bit string.
        goto line 6
    goto line 4
$c^*_{rsa} \leftarrow z^e \bmod N$
$c^*_{ige} \leftarrow c_{ige}$ ; $r^* \leftarrow r$ ; $m^*_0 \leftarrow m_0$ ; $m^*_1 \leftarrow m_1$
$p^*_{ige} \leftarrow p_{ige}$   // $G_{21}$
$K^* \leftarrow K$ ; $m^*_{padded} \leftarrow m_{padded}$   // $G_{21}$–$G_{23}$
return $c^*_{rsa}$
```

Oracle $\text{DEC}(c_{rsa})$

```
// This oracle is identical to the
// corresponding oracle in game $G_0$ of Fig. 29.
```

Random oracle $\mathrm{H}(a)$ for $a \in \{0,1\}^{1792}$

```
if $\mathsf{T}[a] = \perp$ :
    if $a = c^*_{ige}$ :
        $\mathsf{T}[a] \leftarrow r^* \oplus K^*$   // $G_{21}$–$G_{23}$
        $K^* \leftarrow\!\!\$ \{0,1\}^{256}$ ; $\mathsf{T}[a] \leftarrow r^* \oplus K^*$   // $G_{24}$–$G_{25}$
        $\mathsf{T}[a] \leftarrow\!\!\$ \{0,1\}^{256}$ ; $K^* \leftarrow \mathsf{T}[a] \oplus r^*$   // $G_{26}$
    else :   // $a \neq c^*_{ige}$
        $\mathsf{T}[a] \leftarrow\!\!\$ \{0,1\}^{256}$
return $\mathsf{T}[a]$
```

Ideal cipher $\mathrm{IC}(K, u)$

Ideal cipher $\mathrm{IC}^{-1}(K, v)$

```
// These oracles are identical to the
// corresponding oracles in game $G_{13}$ of Fig. 31.
```

Function $\mathsf{EmbedChallengeMessage}(K)$

```
if $K \in S_{IC}$ : return $\perp$
if $K \neq K^*$ : return $\perp$
if $\mathsf{T}[c^*_{ige}] = \perp$ : return $\perp$   // $G_{21}$–$G_{24}$
$pad \leftarrow\!\!\$ \{0,1\}^{1536-|m^*_b|}$   // $G_{24}$–$G_{26}$
$m^*_{padded} \leftarrow m^*_b \parallel pad$   // $G_{24}$–$G_{26}$
$h \leftarrow \mathrm{H}(K^* \parallel m^*_{padded})$   // $G_{22}$–$G_{26}$
$p_{ige} \leftarrow \mathsf{reverse}(m^*_{padded}) \parallel h$   // $G_{22}$–$G_{26}$
$x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
// Parse into 128-bit blocks.
$x_1 \parallel \ldots \parallel x_{14} \leftarrow p^*_{ige}$   // $G_{21}$
$x_1 \parallel \ldots \parallel x_{14} \leftarrow p_{ige}$   // $G_{22}$–$G_{26}$
$y_1 \parallel \ldots \parallel y_{14} \leftarrow c^*_{ige}$
for $i = 1, \ldots, 14$ :
    $u_i \leftarrow x_i \oplus y_{i-1}$
    $v_i \leftarrow y_i \oplus x_{i-1}$
    $\mathsf{AddRelationToIC}(K, u_i, v_i)$
```

Function $\mathsf{AddRelationToIC}(K, u, v)$

```
// This function is identical to the
// corresponding function in game $G_0$ of Fig. 29.
```

Fig. 34. Games G_{21}–G_{26} for the proof of Theorem 4.

Analysis of $G_{22} \to G_{23}$**.** Consider game G_{23}. Game G_{23} is obtained from game G_{22} by removing a conditional statement in the ENC oracle that resulted in calling **abort**(`false`). An **abort**(`false`) call causes the adversary to immediately lose the game. The advantage of an adversary could only increase by removing it.

$$\Pr[G_{22}] \leq \Pr[G_{23}].$$

Analysis of $G_{23} \to G_{24}$**.** Game G_{24} is obtained from game G_{23} by moving the following code from oracle ENC to either oracle H (i.e. the first line) or function EmbedChallengeMessage (i.e. the later two lines):

$$K \leftarrow\$ \{0,1\}^{256}$$
$$pad \leftarrow\$ \{0,1\}^{1536-|m_b|}$$
$$m_{\mathsf{padded}} \leftarrow m_b \parallel pad.$$

The conditional statement "**if** $\mathsf{T}[c_{\mathsf{ige}}] \neq \perp$: **abort**(`false`)" in the ENC oracle guarantees that the random oracle H will necessarily be called before the function EmbedChallengeMessage can be used to program the ideal cipher tables. This allows us to sample K^* at a later time, without changing the functionality of the game. We have

$$\Pr[G_{23}] = \Pr[G_{24}].$$

Analysis of $G_{24} \to G_{25}$**.** Consider game G_{25}. Game G_{25} is obtained from game G_{24} by removing a conditional statement in the ENC oracle that resulted in calling **abort**(`false`), and also by removing the "**if** $\mathsf{T}[c^*_{\mathsf{ige}}] = \perp$: **return** $\perp$" conditional statement from function EmbedChallengeMessage. An **abort**(`false`) call causes the adversary to immediately lose the game; the advantage of an adversary could only increase by removing it. Whereas the condition "$\mathsf{T}[c^*_{\mathsf{ige}}] = \perp$" inside EmbedChallengeMessage can never be true because we now sample K^* inside oracle H, and an earlier conditional statement in EmbedChallengeMessage ensures that $K^* \neq \perp$.

$$\Pr[G_{24}] \leq \Pr[G_{25}].$$

Analysis of $G_{25} \to G_{26}$**.** Game G_{26} is obtained from game G_{25} by replacing oracle H's line 4 containing

$$K^* \leftarrow\$ \{0,1\}^{256} \,;\ \mathsf{T}[a] \leftarrow r^* \oplus K^*$$

with line 5 containing

$$\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256} \,;\ K^* \leftarrow \mathsf{T}[a] \oplus r^*.$$

Games G_{25} and G_{26} are functionally equivalent, so

$$\Pr[G_{25}] = \Pr[G_{26}].$$

Analysis of $G_{26} \to G_{27}$**.** Game G_{27} removes dead code from game G_{26}, and rewrites small snippets of code inside oracles ENC and H. In particular, note that the rejection sampling in game G_{26} does not change the internal state of the game, and always succeeds in sampling a uniformly random element from $\mathbb{Z}_N$. So game G_{27} samples such an element right away, directly from $\mathbb{Z}_N$. And line 5 of oracle H in game G_{27} simply rewrites the corresponding code of G_{26}.

Game G_{27} also rewrites the code of oracle DEC in the following way. First, it defines a helper function PreSampleH that is used only by oracle DEC. The function PreSampleH is only called on inputs that were not previously queried to the random oracle, i.e. PreSampleH(a) for some input "a" is only called if $\mathsf{T}[a] = \perp$. On such an input, function PreSampleH mimics the functionality of the random oracle, sampling a new random-oracle output if needed, and saving it into its own table R. Table R therefore contains the random-oracle outputs that were added by oracle DEC. These values are lazily propagated from R into T when oracle H is called on an input "a" for which $\mathsf{T}[a] = \perp$ and $\mathsf{R}[a] \neq \perp$.

The decryption oracle $\text{D}\textsc{ec}$ contains a conditional statement checking the condition $\mathsf{T}[c_{\mathsf{ige}}] = \bot$, and both of its conditional branches compute the values $K, p_{\mathsf{ige}}, m_{\mathsf{padded}}, h$. When $\mathsf{T}[c_{\mathsf{ige}}] = \bot$ is true, then K is computed via $\mathsf{PreSampleH}(c_{\mathsf{ige}}) \oplus r$ instead of $\mathrm{H}(c_{\mathsf{ige}}) \oplus r$. Both instructions are equivalent because the aforementioned lazy propagation provides consistency between PreSampleH and H. The first instruction after the conditional statement is

$$\textbf{if } h \neq \mathrm{H}(K \parallel m_{\mathsf{padded}}) : \textbf{return } \bot.$$

Both branches of the conditional statement effectively duplicate this instruction when $\mathsf{T}[K \parallel m_{\mathsf{padded}}] = \bot$ is true, expressing the duplicate condition in terms of PreSampleH instead of H as follows:

$$\textbf{if } h \neq \mathsf{PreSampleH}(K \parallel m_{\mathsf{padded}}) : \textbf{return } \bot.$$

(The first conditional branch also requires $(K \parallel m_{\mathsf{padded}} \neq c_{\mathsf{ige}})$ in order to run the duplicate instruction; we discuss it in the next transition.) Games G_{26} and G_{27} are functionally equivalent. We have

$$\Pr[\mathrm{G}_{26}] = \Pr[\mathrm{G}_{27}].$$

Analysis of $\mathrm{G}_{27} \rightarrow \mathrm{G}_{28}$**.** The flag bad_9 can be set from 3 distinct lines of oracle $\text{D}\textsc{ec}$ in game G_{28}: line 12, line 18 and line 27. In two cases (line 12 and line 27), this happens when $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ succeeds in querying its $\text{D}\textsc{ec}$ oracle on a ciphertext for which $\mathsf{T}[K \parallel m_{\mathsf{padded}}] = \bot$ and $h = \mathsf{R}[K \parallel m_{\mathsf{padded}}]$. The former means that $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ has not yet explicitly queried its random oracle on $K \parallel m_{\mathsf{padded}}$. If the remaining case did not exist (i.e. disregarding line 18, purely hypothetically), then $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ could have at best attempted to guess the 256-bit value $\mathsf{R}[K \parallel m_{\mathsf{padded}}]$, one guess per decryption query. However, further analysis is required due to the possibility of setting the flag bad_9 in line 18.

The flag bad_9 is set in line 18 if the instruction $K \leftarrow \mathsf{PreSampleH}(c_{\mathsf{ige}}) \oplus r$ in line 6 of oracle $\text{D}\textsc{ec}$ is used to obtain some $K \parallel m_{\mathsf{padded}}$ for which either of the following statements is true: (a) $K \parallel m_{\mathsf{padded}} = c_{\mathsf{ige}}$, or (b) $\mathsf{T}[K \parallel m_{\mathsf{padded}}] \neq \bot$. Besides guessing $\mathsf{R}[K \parallel m_{\mathsf{padded}}]$ one query at a time (as per above), adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ could learn more information about $\mathsf{R}[K \parallel m_{\mathsf{padded}}]$ by querying ciphertexts to $\mathsf{n}_{\text{D}\textsc{ec}}$ for which (a) or (b) would be true. This exhausts $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$'s options, i.e. the adversary cannot benefit from the instruction $K \leftarrow \mathsf{PreSampleH}(c_{\mathsf{ige}}) \oplus r$ in any other way.

Consider the set $A \leftarrow S_{\mathsf{RO}} \cup \{c_{\mathsf{ige}}\}$ containing all prior inputs to H, along with the value c_{ige}. The above statements (a) or (b) are true iff $\big(\mathsf{PreSampleH}(c_{\mathsf{ige}}) \oplus r\big) \in A$. To see this, we expand

$$h = \mathrm{H}(K \| m_{\mathsf{padded}}) = \mathrm{H}(\mathrm{H}(c_{\mathsf{ige}}) \oplus r \| m_{\mathsf{padded}})$$

and note that the output of $\mathrm{H}(c_{\mathsf{ige}})$ is fed as an input to H after XOR-ing it with r. The probability of setting the flag bad_9 in line 18 can be informally stated $\Pr[\big(\mathsf{PreSampleH}(c_{\mathsf{ige}}) \oplus r\big) \in A]$. In order to obtain a specific upper-bound, consider the set $Q = \{\eta[0:256] \oplus r\}_{\eta \in A}$ containing all 256-bit prefixes of the set A, each XOR-ed with r. Then $\Pr[\big(\mathsf{PreSampleH}(c_{\mathsf{ige}}) \oplus r\big) \in A] \leq \Pr[\mathsf{PreSampleH}(c_{\mathsf{ige}}) \in Q]$ can be used as an upper bound. This means that adversary can at best attempt to guess the 256-bit value $\mathsf{PreSampleH}(c_{\mathsf{ige}})$, but now it gets up to $|Q| = |A| = \mathsf{n}_{\mathrm{H}} + 1$ guesses per decryption query. Each query to oracle $\text{D}\textsc{ec}$ can reach only one of the three lines where the flag bad_9 is set, and line 18 eliminates the biggest number of possible $\mathsf{PreSampleH}(\cdot)$ values per query. We have

$$\Pr[\mathrm{G}_{27}] - \Pr[\mathrm{G}_{28}] \leq \Pr[\mathsf{bad}_9^{\mathrm{G}_{28}}] \leq \frac{\mathsf{n}_{\text{D}\textsc{ec}} \cdot (\mathsf{n}_{\mathrm{H}} + 1)}{2^{256}}.$$

Analysis of $\mathrm{G}_{28} \rightarrow \mathrm{G}_{29}$**.** In game G_{29} we remove the instruction $\textbf{if } a = c^*_{\mathsf{ige}} : K^* \leftarrow \mathsf{R}[a] \oplus r^*$ from the function PreSampleH. The value of K^* is only used inside function EmbedChallengeMessage, and only if adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ triggers a call to EmbedChallengeMessage on input K^*. The adversary does not get any information about K^* until it either queries H on c^*_{ige}, or triggers the call $\mathsf{EmbedChallengeMessage}(K^*)$. By removing the aforementioned line, we changed the behaviour of the latter calls (but only until $\mathrm{H}(c^*_{\mathsf{ige}})$ gets queried). In game G_{28}, the adversary was able to use this behaviour only by exhaustively triggering $\mathsf{EmbedChallengeMessage}$ calls on different values of K until it managed to guess $K = K^*$. In game G_{29} this functionality is removed. A call to $\mathsf{EmbedChallengeMessage}$ is made when adversary either queries its ideal cipher oracles, or when it calls its DEC oracle. The value of K^* in oracle DEC is effectively sampled uniformly at random from $\{0,1\}^{256}$. We have

$$\Pr[\mathrm{G}_{28}] - \Pr[\mathrm{G}_{29}] \leq \frac{\mathsf{n}_{\mathrm{DEC}} + \mathsf{n}_{\mathsf{IC}}}{2^{256}}.$$

Analysis of $\mathrm{G}_{29} \rightarrow \mathrm{G}_{30}$**.** In game G_{29} the temporary random oracle table R is no longer used in oracle DEC. The previous transition removed the $K^* \leftarrow \mathsf{R}[a] \oplus r^*$ instruction. And all conditional branches of oracle DEC that call function $\mathsf{PreSampleH}$ in game G_{29} – always return $\perp$ regardless of the value returned by $\mathsf{PreSampleH}$. So we remove function $\mathsf{PreSampleH}$ along with its table R in game G_{29}. Game G_{30} rewrites the conditional branches of DEC accordingly, i.e. to always return $\perp$. Beyond that, game G_{30} expands the code of all oracles. The games are functionally equivalent, so

$$\Pr[\mathrm{G}_{29}] = \Pr[\mathrm{G}_{30}].$$

Analysis of $\mathrm{G}_{30} \rightarrow \mathrm{G}_{31}$**.** We build an adversary $\mathcal{F}_{\mathsf{OWRSA}}$ in Fig. 37 that can invert the one-wayness of RSA whenever the flag bad_{10} is set in game G_{31}. Note that the encryption and decryption oracles in game G_{31} never call the random oracle H on any inputs that were not previously queried by adversary $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$ directly. This allows the one-wayness adversary $\mathcal{F}_{\mathsf{OWRSA}}$ to simulate the decryption oracle for $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$, by iterating over all prior $\mathcal{D}_{\mathsf{IND\text{-}CCA}}$'s calls to its random oracle H in an attempt to reconstruct every ciphertext that it queries to its decryption oracle DEC. It follows that

$$\Pr[\mathrm{G}_{30}] - \Pr[\mathrm{G}_{31}] \leq \Pr[\mathsf{bad}_{10}^{\mathrm{G}_{31}}] \leq \mathsf{Adv}^{\mathrm{OW\text{-}RSA}}_{\mathsf{RSA.KGen}}(\mathcal{F}_{\mathsf{OWRSA}}).$$

Note that the adversary here does not need to simulate the parts of the game inside oracle H and function $\mathsf{EmbedChallengeMessage}$ that are doing the programming of the idealised oracles.

Analysis of G_{31}**.** Game G_{31} no longer performs the programming of the ideal cipher tables in function $\mathsf{EmbedChallengeMessage}$. In particular, this game no longer uses the challenge bit. The adversary can, at best, try to guess it, so

$$\Pr[\mathrm{G}_{31}] = \frac{1}{2}.$$

This concludes the proof.

Games G_{27}–G_{29}

```
$b \leftarrow\$ \{0,1\}$ ; $c^*_{\mathsf{rsa}} \leftarrow \bot$ ; $S_{\mathsf{IC}} \leftarrow \emptyset$ ; $S_{\mathsf{RO}} \leftarrow \emptyset$
$c^*_{\mathsf{ige}} \leftarrow \bot$ ; $K^* \leftarrow \bot$ ; $r^* \leftarrow \bot$
$m^*_0 \leftarrow \bot$ ; $m^*_1 \leftarrow \bot$
$(N, p, q, e, d) \leftarrow\$ \mathsf{RSA.KGen}()$ ; $pk \leftarrow (N, e)$
$b' \leftarrow\$ \mathcal{D}^{\mathrm{ENC,DEC,H,IC,IC}^{-1}}_{\mathsf{IND\text{-}CCA}}(pk)$
return $b = b'$
```

Oracle $\mathrm{ENC}(m_0, m_1)$

```
require $(c^*_{\mathsf{rsa}} = \bot) \wedge (|m_0| = |m_1|)$
require $m_0, m_1 \in \mathcal{M}$
$z \leftarrow\$ \{0, \ldots, N-1\}$
$p_{\mathsf{rsa}} \leftarrow z$    // Parse $z$ as a 2048-bit string.
$r^* \| c^*_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}$    // s.t. $|r^*| = 256, |c^*_{\mathsf{ige}}| = 1792$.
$m^*_0 \leftarrow m_0$ ; $m^*_1 \leftarrow m_1$ ; $c^*_{\mathsf{rsa}} \leftarrow z^e \bmod N$
return $c^*_{\mathsf{rsa}}$
```

Random oracle $\mathrm{H}(a)$ for $a \in \{0,1\}^{1792}$

```
$S_{\mathsf{RO}} \leftarrow S_{\mathsf{RO}} \cup \{a\}$
if $\mathsf{T}[a] = \bot$ :
  $\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256}$
  if $\mathsf{R}[a] \neq \bot$ : $\mathsf{T}[a] \leftarrow \mathsf{R}[a]$
  if $a = c^*_{\mathsf{ige}}$ : $K^* \leftarrow \mathsf{T}[a] \oplus r^*$
return $\mathsf{T}[a]$
```

Function $\mathsf{EmbedChallengeMessage}(K)$

```
if $K \in S_{\mathsf{IC}}$ : return $\bot$
if $K \neq K^*$ : return $\bot$
$pad \leftarrow\$ \{0,1\}^{1536-|m_b|}$
$m_{\mathsf{padded}} \leftarrow m_b \| pad$
$h \leftarrow \mathrm{H}(K^* \| m_{\mathsf{padded}})$
$p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \| h$
$x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
$x_1 \| \ldots \| x_{14} \leftarrow p_{\mathsf{ige}}$    // Parse into 128-bit blocks.
$y_1 \| \ldots \| y_{14} \leftarrow c^*_{\mathsf{ige}}$    // Parse into 128-bit blocks.
for $i = 1, \ldots, 14$ :
  $u_i \leftarrow x_i \oplus y_{i-1}$
  $v_i \leftarrow y_i \oplus x_{i-1}$
  $\mathsf{AddRelationToIC}(K, u_i, v_i)$
```

Function $\mathsf{AddRelationToIC}(K, u, v)$

```
// This function is identical to the
// corresponding function in game $G_0$ of Fig. 29.
```

Oracle $\mathrm{DEC}(c_{\mathsf{rsa}})$

```
require $(c_{\mathsf{rsa}} \neq c^*_{\mathsf{rsa}}) \wedge (c_{\mathsf{rsa}} \in \mathbb{Z}_N)$
$z \leftarrow (c_{\mathsf{rsa}})^d \bmod N$
$p_{\mathsf{rsa}} \leftarrow z$    // Parse $z$ as a 2048-bit string.
$r \| c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}$    // s.t. $|r| = 256, |c_{\mathsf{ige}}| = 1792$.
if $\mathsf{T}[c_{\mathsf{ige}}] = \bot$ :
  $K \leftarrow \mathsf{PreSampleH}(c_{\mathsf{ige}}) \oplus r$
  $p_{\mathsf{ige}} \leftarrow \mathsf{IGE}^{\mathrm{IC,IC}^{-1}}.\mathsf{Dec}(K, 0^{256}, c_{\mathsf{ige}})$
  $m_{\mathsf{padded}} \leftarrow \mathsf{reverse}(p_{\mathsf{ige}}[0:1536])$
  $h \leftarrow p_{\mathsf{ige}}[1536:1792]$
  if $(\mathsf{T}[K \| m_{\mathsf{padded}}] = \bot) \wedge (K \| m_{\mathsf{padded}} \neq c_{\mathsf{ige}})$ :
    if $h \neq \mathsf{PreSampleH}(K \| m_{\mathsf{padded}})$ : return $\bot$
    $\mathsf{bad}_9 \leftarrow \mathtt{true}$
    return $\bot$    // $G_{28}$–$G_{30}$
  else :    // $(\mathsf{T}[K \| m_{\mathsf{padded}}] \neq \bot) \vee (K \| m_{\mathsf{padded}} = c_{\mathsf{ige}})$
    // $Q \leftarrow \emptyset$ ; $A \leftarrow S_{\mathsf{RO}} \cup \{c_{\mathsf{ige}}\}$
    // for each $\eta \in A$ : $Q \leftarrow Q \cup \{\eta[0:256] \oplus r\}$
    // $\mathsf{PreSampleH}(c_{\mathsf{ige}}) \in Q$
    $\mathsf{bad}_9 \leftarrow \mathtt{true}$
    return $\bot$    // $G_{28}$–$G_{30}$
else :    // $\mathsf{T}[c_{\mathsf{ige}}] \neq \bot$
  $K \leftarrow \mathrm{H}(c_{\mathsf{ige}}) \oplus r$
  $p_{\mathsf{ige}} \leftarrow \mathsf{IGE}^{\mathrm{IC,IC}^{-1}}.\mathsf{Dec}(K, 0^{256}, c_{\mathsf{ige}})$
  $m_{\mathsf{padded}} \leftarrow \mathsf{reverse}(p_{\mathsf{ige}}[0:1536])$
  $h \leftarrow p_{\mathsf{ige}}[1536:1792]$
  if $\mathsf{T}[K \| m_{\mathsf{padded}}] = \bot$ :
    if $h \neq \mathsf{PreSampleH}(K \| m_{\mathsf{padded}})$ : return $\bot$
    $\mathsf{bad}_9 \leftarrow \mathtt{true}$
    return $\bot$    // $G_{28}$–$G_{30}$
if $h \neq \mathrm{H}(K \| m_{\mathsf{padded}})$ : return $\bot$
$m \leftarrow \mathsf{RemovePadding}(m_{\mathsf{padded}})$
return $m$
```

Function $\mathsf{PreSampleH}(a)$ for $a \in \{0,1\}^{1792}$

```
if $\mathsf{R}[a] = \bot$ :
  $\mathsf{R}[a] \leftarrow\$ \{0,1\}^{256}$
  if $a = c^*_{\mathsf{ige}}$ : $K^* \leftarrow \mathsf{R}[a] \oplus r^*$    // $G_{27}$–$G_{28}$
return $\mathsf{R}[a]$
```

Ideal cipher $\mathrm{IC}(K, u)$

Ideal cipher $\mathrm{IC}^{-1}(K, v)$

```
// These oracles are identical to the
// corresponding oracles in game $G_{13}$ of Fig. 31.
```

Fig. 35. Games G_{27}–G_{29} for the proof of Theorem 4.

Games G_{30}–G_{31}

```
1: $b \leftarrow\$ \{0,1\}$ ; $c^*_{\mathsf{rsa}} \leftarrow \bot$ ; $S_{\mathsf{IC}} \leftarrow \emptyset$
2: $c^*_{\mathsf{ige}} \leftarrow \bot$ ; $K^* \leftarrow \bot$ ; $r^* \leftarrow \bot$
3: $m^*_0 \leftarrow \bot$ ; $m^*_1 \leftarrow \bot$
4: $(N, p, q, e, d) \leftarrow\$ \mathsf{RSA.KGen}()$ ; $pk \leftarrow (N, e)$
5: $b' \leftarrow\$ \mathcal{D}^{\textsc{Enc},\textsc{Dec},\mathrm{H},\mathrm{IC},\mathrm{IC}^{-1}}_{\mathsf{IND\text{-}CCA}}(pk)$
6: return $b = b'$
```

Oracle $\textsc{Enc}(m_0, m_1)$

```
1: require $(c^*_{\mathsf{rsa}} = \bot) \wedge (|m_0| = |m_1|)$
2: require $m_0, m_1 \in \mathcal{M}$
3: $z \leftarrow\$ \{0, \ldots, N-1\}$
4: $p_{\mathsf{rsa}} \leftarrow z$   // Parse $z$ as a 2048-bit string.
5: $r^* \| c^*_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}$   // s.t. $|r^*| = 256, |c^*_{\mathsf{ige}}| = 1792$.
6: $m^*_0 \leftarrow m_0$ ; $m^*_1 \leftarrow m_1$
7: $c^*_{\mathsf{rsa}} \leftarrow z^e \bmod N$
8: return $c^*_{\mathsf{rsa}}$
```

Random oracle $\mathrm{H}(a)$ for $a \in \{0,1\}^{1792}$

```
1: if $\mathsf{T}[a] = \bot$ :
2:     $\mathsf{T}[a] \leftarrow\$ \{0,1\}^{256}$
3:     if $a = c^*_{\mathsf{ige}}$ : $K^* \leftarrow \mathsf{T}[a] \oplus r^*$
4: return $\mathsf{T}[a]$
```

Function $\mathsf{EmbedChallengeMessage}(K)$

```
1: if $K \in S_{\mathsf{IC}}$ : return $\bot$
2: if $K \neq K^*$ : return $\bot$
3: $\mathsf{bad}_{10} \leftarrow \mathtt{true}$
4: return $\bot$   // $G_{31}$
5: $pad \leftarrow\$ \{0,1\}^{1536-|m_b|}$
6: $m_{\mathsf{padded}} \leftarrow m_b \| pad$
7: $h \leftarrow \mathrm{H}(K^* \| m_{\mathsf{padded}})$
8: $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \| h$
9: $x_0 \leftarrow 0^{128}$ ; $y_0 \leftarrow 0^{128}$
10: $x_1 \| \ldots \| x_{14} \leftarrow p_{\mathsf{ige}}$   // Parse into 128-bit blocks.
11: $y_1 \| \ldots \| y_{14} \leftarrow c^*_{\mathsf{ige}}$   // Parse into 128-bit blocks.
12: for $i = 1, \ldots, 14$ :
13:     $u_i \leftarrow x_i \oplus y_{i-1}$
14:     $v_i \leftarrow y_i \oplus x_{i-1}$
15:     $\mathsf{AddRelationToIC}(K, u_i, v_i)$
```

Oracle $\textsc{Dec}(c_{\mathsf{rsa}})$

```
1: require $(c_{\mathsf{rsa}} \neq c^*_{\mathsf{rsa}}) \wedge (c_{\mathsf{rsa}} \in \mathbb{Z}_N)$
2: $z \leftarrow (c_{\mathsf{rsa}})^d \bmod N$
3: $p_{\mathsf{rsa}} \leftarrow z$   // Parse $z$ as a 2048-bit string.
4: $r \| c_{\mathsf{ige}} \leftarrow p_{\mathsf{rsa}}$   // s.t. $|r| = 256, |c_{\mathsf{ige}}| = 1792$.
5: if $\mathsf{T}[c_{\mathsf{ige}}] = \bot$ : return $\bot$
6: $K \leftarrow \mathrm{H}(c_{\mathsf{ige}}) \oplus r$
7: $p_{\mathsf{ige}} \leftarrow \mathsf{IGE}^{\mathrm{IC},\mathrm{IC}^{-1}}.\mathsf{Dec}(K, 0^{256}, c_{\mathsf{ige}})$
8: $m_{\mathsf{padded}} \leftarrow \mathsf{reverse}(p_{\mathsf{ige}}[0:1536])$
9: $h \leftarrow p_{\mathsf{ige}}[1536:1792]$
10: if $\mathsf{T}[K \| m_{\mathsf{padded}}] = \bot$ : return $\bot$
11: if $h \neq \mathrm{H}(K \| m_{\mathsf{padded}})$ : return $\bot$
12: $m \leftarrow \mathsf{RemovePadding}(m_{\mathsf{padded}})$
13: return $m$
```

Ideal cipher $\mathrm{IC}(K, u)$

```
1: $\mathsf{EmbedChallengeMessage}(K)$
2: if $\mathsf{A}_K[u] = \bot$ :
3:     $v \leftarrow\$ \{0,1\}^{128} \setminus \mathcal{R}_K$
4:     $\mathsf{AddRelationToIC}(K, u, v)$
5: return $\mathsf{A}_K[u]$
```

Ideal cipher $\mathrm{IC}^{-1}(K, v)$

```
1: $\mathsf{EmbedChallengeMessage}(K)$
2: if $\mathsf{B}_K[v] = \bot$ :
3:     $u \leftarrow\$ \{0,1\}^{128} \setminus \mathcal{D}_K$
4:     $\mathsf{AddRelationToIC}(K, u, v)$
5: return $\mathsf{B}_K[v]$
```

Function $\mathsf{AddRelationToIC}(K, u, v)$

```
1: $S_{\mathsf{IC}} \leftarrow S_{\mathsf{IC}} \cup \{K\}$
2: $\mathsf{A}_K[u] \leftarrow v$ ; $\mathsf{B}_K[v] \leftarrow u$
3: $\mathcal{D}_K \leftarrow \mathcal{D}_K \cup \{u\}$
4: $\mathcal{R}_K \leftarrow \mathcal{R}_K \cup \{v\}$
```

Fig. 36. Games G_{30}–G_{31} for the proof of Theorem 4.

```
Adversary $\mathcal{F}_{\mathsf{OWRSA}}(pk, c_{\mathsf{rsa}})$
1 : $b \leftarrow\!\$ \{0,1\}$ ; $c^*_{\mathsf{rsa}} \leftarrow \bot$ ; $S_{\mathsf{RO}} \leftarrow \emptyset$ ; $S_{\mathsf{IC}} \leftarrow \emptyset$
2 : $(N, e) \leftarrow pk$
3 : $b' \leftarrow\!\$ \mathcal{D}^{\textsc{SimEnc},\textsc{SimDec},\textsc{SimH},\textsc{SimIC},\textsc{SimIC}^{-1}}_{\mathsf{IND\text{-}CCA}}(pk)$

Oracle $\textsc{SimEnc}(m_0, m_1)$
1 : require $(c^*_{\mathsf{rsa}} = \bot) \wedge (|m_0| = |m_1|)$
2 : require $m_0, m_1 \in \mathcal{M}$
3 : $c^*_{\mathsf{rsa}} \leftarrow c_{\mathsf{rsa}}$
4 : return $c^*_{\mathsf{rsa}}$

Random oracle $\textsc{SimH}(a)$ for $a \in \{0,1\}^{1792}$
1 : $S_{\mathsf{RO}} \leftarrow S_{\mathsf{RO}} \cup \{a\}$
2 : if $\mathsf{T}[a] = \bot$ :
3 :     $\mathsf{T}[a] \leftarrow\!\$ \{0,1\}^{256}$
4 : return $\mathsf{T}[a]$

Function $\mathsf{EmbedChallengeMessage}(K)$
1 : if $K \in S_{\mathsf{IC}}$ : return $\bot$
2 : // if $K \neq K^*$ : return $\bot$
3 : for each $c_{\mathsf{ige}} \in S_{\mathsf{RO}}$ :
4 :     $r \leftarrow \mathsf{T}[c_{\mathsf{ige}}] \oplus K$
5 :     $p_{\mathsf{rsa}} \leftarrow r \,\|\, c_{\mathsf{ige}}$
6 :     $z \leftarrow p_{\mathsf{rsa}}$    // Parse $p_{\mathsf{rsa}}$ as an integer.
7 :     if $z \notin \mathbb{Z}_N$ : continue
8 :     $c'_{\mathsf{rsa}} \leftarrow z^e \bmod N$
9 :     if $c^*_{\mathsf{rsa}} = c'_{\mathsf{rsa}}$ : abort$(z)$

Oracle $\textsc{SimDec}(c_{\mathsf{rsa}})$
1 : require $(c_{\mathsf{rsa}} \neq c^*_{\mathsf{rsa}}) \wedge (c_{\mathsf{rsa}} \in \mathbb{Z}_N)$
2 : for each $a \in S_{\mathsf{RO}}$ :
3 :     $K \,\|\, m_{\mathsf{padded}} \leftarrow a$    // s.t. $|K| = 256, |m_{\mathsf{padded}}| = 1536$.
4 :     $h \leftarrow \mathsf{T}[K \,\|\, m_{\mathsf{padded}}]$
5 :     $p_{\mathsf{ige}} \leftarrow \mathsf{reverse}(m_{\mathsf{padded}}) \,\|\, h$
6 :     $c_{\mathsf{ige}} \leftarrow \mathsf{IGE}^{\textsc{SimIC},\textsc{SimIC}^{-1}}.\mathsf{Enc}(K, 0^{256}, p_{\mathsf{ige}})$
7 :     if $\mathsf{T}[c_{\mathsf{ige}}] = \bot$ : continue
8 :     $r \leftarrow \mathsf{T}[c_{\mathsf{ige}}] \oplus K$
9 :     $p_{\mathsf{rsa}} \leftarrow r \,\|\, c_{\mathsf{ige}}$
10 :    $z \leftarrow p_{\mathsf{rsa}}$    // Parse $p_{\mathsf{rsa}}$ as an integer.
11 :    if $z \notin \mathbb{Z}_N$ : continue
12 :    $c'_{\mathsf{rsa}} \leftarrow z^e \bmod N$
13 :    if $c_{\mathsf{rsa}} \neq c'_{\mathsf{rsa}}$ : continue
14 :    $m \leftarrow \mathsf{RemovePadding}(m_{\mathsf{padded}})$
15 :    return $m$
16 : return $\bot$

Ideal cipher $\textsc{SimIC}(K, u)$
1 : $\mathsf{EmbedChallengeMessage}(K)$
2 : if $\mathsf{A}_K[u] = \bot$ :
3 :     $v \leftarrow\!\$ \{0,1\}^{128} \setminus \mathcal{R}_K$
4 :     $\mathsf{AddRelationToIC}(K, u, v)$
5 : return $\mathsf{A}_K[u]$

Ideal cipher $\textsc{SimIC}^{-1}(K, v)$
1 : $\mathsf{EmbedChallengeMessage}(K)$
2 : if $\mathsf{B}_K[v] = \bot$ :
3 :     $u \leftarrow\!\$ \{0,1\}^{128} \setminus \mathcal{D}_K$
4 :     $\mathsf{AddRelationToIC}(K, u, v)$
5 : return $\mathsf{B}_K[v]$

Function $\mathsf{AddRelationToIC}(K, u, v)$
1 : $S_{\mathsf{IC}} \leftarrow S_{\mathsf{IC}} \cup \{K\}$
2 : $\mathsf{A}_K[u] \leftarrow v$ ; $\mathsf{B}_K[v] \leftarrow u$
3 : $\mathcal{D}_K \leftarrow \mathcal{D}_K \cup \{u\}$
4 : $\mathcal{R}_K \leftarrow \mathcal{R}_K \cup \{v\}$
```

Fig. 37. Adversary $\mathcal{F}_{\mathsf{OWRSA}}$ for the proof of Theorem 4.

E TOTPRF: One-time pseudorandomness of truncated SHA-1

The following definition for an intermediate security notion used in the main key exchange proofs (Appendix H) represents a version of one-time PRF security specific to how truncated SHA-1 is used in $\mathsf{MTP\text{-}KE}_{\mathsf{2st}}$ and $\mathsf{MTP\text{-}KE}_{\mathsf{3st}}$.

Definition 12. *Consider the game* $\mathrm{G}^{\mathsf{TOTPRF}}_{\mathsf{H},\mathcal{A}}$ *in Fig. 38 with the function family* H *such that* $\mathsf{H.Ev}(hk, x) = \mathsf{SHA\text{-}1}(x \parallel hk)$ *where the key length is 512, input length is 1536 and output length is 160, and an adversary* $\mathcal{D}$*. The advantage of* $\mathcal{D}$ *in breaking the* TOTPRF*-security of* H *is defined as* $\mathsf{Adv}^{\mathsf{TOTPRF}}_{\mathsf{H}}(\mathcal{D}) := 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{TOTPRF}}_{\mathsf{H},\mathcal{D}}\right] - 1$.

```
Game $\mathrm{G}^{\mathsf{TOTPRF}}_{\mathsf{H},\mathcal{D}}$
1 :  $b \leftarrow\!\!\$\ \{0,1\}$
2 :  $b' \leftarrow \mathcal{D}^{\mathrm{ROR}}$
3 :  return $b' = b$
```

```
$\mathrm{ROR}(x)$    // $|x| = 1536$
1 :  $hk \leftarrow\!\!\$\ \{0,1\}^{512}$
2 :  $y_1 \leftarrow \mathsf{H.Ev}(hk, x)$
3 :  $y_0 \leftarrow\!\!\$\ \{0,1\}^{160}$
4 :  $ax \leftarrow y_b[0:64]$
5 :  $aid \leftarrow y_b[96:160]$
6 :  return $(ax, aid)$
```

```
$\mathsf{H.Ev}(hk, x)$    // $|hk| = 512, |x| = 1536$
1 :  // $p$ represents known SHA-1 padding
2 :  $h_0 \leftarrow \mathtt{IV}_{160}\ \hat{+}\ \mathsf{SHACAL\text{-}1.Ev}(x[0:512], \mathtt{IV}_{160})$
3 :  $h_1 \leftarrow h_0\ \hat{+}\ \mathsf{SHACAL\text{-}1.Ev}(x[512:1024], h_0)$
4 :  $h_2 \leftarrow h_1\ \hat{+}\ \mathsf{SHACAL\text{-}1.Ev}(x[1024:1536], h_1)$
5 :  $h_3 \leftarrow h_2\ \hat{+}\ \mathsf{SHACAL\text{-}1.Ev}(hk, h_2)$
6 :  $y \leftarrow h_3\ \hat{+}\ \mathsf{SHACAL\text{-}1.Ev}(p, h_3)$
7 :  return $y$
```

Fig. 38. Truncated OTPRF of the function family H such that $\mathsf{H.Ev}(hk, x) = \mathsf{SHA\text{-}1}(x \parallel hk)$.

Proposition 1. *Let* $\mathcal{D}_{\mathsf{TOTPRF}}$ *be an adversary against the* TOTPRF*-security of the function family* H *such that* $\mathsf{H.Ev}(hk, x) = \mathsf{SHA\text{-}1}(x \parallel hk)$*. Then we can build an adversary* $\mathcal{D}_{\mathsf{OTPRF}}$ *against the* OTPRF*-security of* $\mathsf{SHACAL\text{-}1}$ *such that* $\mathsf{Adv}^{\mathsf{TOTPRF}}_{\mathsf{H}}(\mathcal{D}_{\mathsf{TOTPRF}}) = \mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{OTPRF}})$.

Proof. We can proceed via a direct reduction, i.e. we build $\mathcal{D}_{\mathsf{OTPRF}}$ to perfectly simulate $\mathrm{G}^{\mathsf{TOTPRF}}_{\mathsf{H}}$ for $\mathcal{D}_{\mathsf{TOTPRF}}$ by replacing the single call to $\mathsf{SHACAL\text{-}1.Ev}(hk, h_2)$ (line 5 of H.Ev in Fig. 38) for $hk \leftarrow\!\!\$\ \{0,1\}^{512}$ with a call to $\mathcal{D}_{\mathsf{OTPRF}}$'s ROR oracle with input h_2. $\mathcal{D}_{\mathsf{OTPRF}}$ does not need to replace the remaining SHACAL-1 calls as they do not depend on any secret values and can be computed by the adversary. $\mathcal{D}_{\mathsf{OTPRF}}$ then returns the bit guess of $\mathcal{D}_{\mathsf{TOTPRF}}$, so we get $\mathsf{Adv}^{\mathsf{TOTPRF}}_{\mathsf{H}}(\mathcal{D}_{\mathsf{TOTPRF}}) = \mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{OTPRF}})$. □

F INT-PTXT: Integrity of plaintexts of HtE-SE with respect to SKDF

In Appendix F.1, we define an intermediate security notion used in the main proofs (Appendix H) that speaks to the integrity of plaintexts produced by HtE-SE when it is keyed by SKDF. In Appendices F.2 to F.4, we then define and prove a number of sub-notions that will be used to prove that this property holds, in addition to any standard or non-standard assumptions. Finally, in Appendix F.5 we state our integrity result for the HtE-SE-SKDF combination and give its proof.

F.1 Definition of INT-PTXT

Here, we define a weak notion of plaintext integrity. In Fig. 39, the message sampler is used in order to restrict the power of the adversary, i.e. not allowing it to encrypt arbitrary messages. The encryption oracle can be called as long as SAMP keeps successfully sampling new messages. The decryption oracle can be called arbitrarily many times, but only until the first failure to decrypt a ciphertext.

Definition 13. *Consider the game* $\mathsf{G}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p,\mathcal{A}}$ *in Fig. 39 with the schemes* HtE-SE, SKDF *defined in Section 4.2, the message sampler* $\mathsf{SAMP}[\cdot,\cdot,g,p]$ *defined in Definition 6, and an adversary* $\mathcal{A}$*. The advantage of* $\mathcal{A}$ *in breaking the* INT-PTXT*-security of* HtE-SE *with respect to* $\mathsf{SKDF}, \mathsf{SAMP}[\cdot,\cdot,g,p]$ *is defined as* $\mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p}(\mathcal{A}) := \Pr\left[\mathsf{G}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p,\mathcal{A}}\right]$.

```
Game $\mathsf{G}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p,\mathcal{A}}$
1: $st \leftarrow \varepsilon$; $\mathcal{M} \leftarrow \emptyset$
2: $n \leftarrow\!\!\$\ \{0,1\}^{128}$
3: $(n_s, st_{\mathcal{A}}) \leftarrow\!\!\$\ \mathcal{A}_1(n)$
4: if $|n_s| \neq 128$:
5:     return false
6: $n_n \leftarrow\!\!\$\ \{0,1\}^{256}$
7: $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
8: $\mathcal{A}_2^{\textsc{Enc},\textsc{Dec}}(st_{\mathcal{A}})$
9: return false

$\textsc{Enc}(aux)$    // $aux \in \{0,1\}^*$
1: $(st, m, x) \leftarrow\!\!\$\ \mathsf{SAMP}[n, n_s, g, p](st, aux)$
2: if $m = \perp$: return $\perp$
3: $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
4: $c \leftarrow\!\!\$\ \mathsf{HtE\text{-}SE.Enc}(k_{se}, m)$
5: return $(m, x, c)$

$\textsc{Dec}(c)$
1: $m \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c)$
2: if $m = \perp$: abort(false)
3: if $m \notin \mathcal{M}$: abort(true)
4: return $m$
```

Fig. 39. Weak plaintext integrity of HtE-SE for keys derived by SKDF and messages sampled by $\mathsf{SAMP}[\cdot,\cdot,g,p]$.

Simplified variant of the INT-PTXT **definition.** We define a slightly simpler variant of the INT-PTXT definition below. We will argue that it is equivalent to the original definition, and our security proof in Appendix F.5 will be stated with respect to it.

Definition 14. *Consider the game* $\mathsf{G}^{\mathsf{INT\text{-}PTXT}^*}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},n_s,\mathsf{Samp},\mathcal{A}}$ *in Fig. 40 with the schemes* HtE-SE, SKDF *defined in Section 4.2, the message sampler* Samp *defined in Definition 6, and an adversary* $\mathcal{A}$*. The advantage of* $\mathcal{A}$ *in breaking the* INT-PTXT**-security of* HtE-SE *with respect to* SKDF, Samp *is defined as* $\mathsf{Adv}^{\mathsf{INT\text{-}PTXT}^*}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},n_s,\mathsf{Samp}}(\mathcal{A}) := \Pr\left[\mathsf{G}^{\mathsf{INT\text{-}PTXT}^*}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},n_s,\mathsf{Samp},\mathcal{A}}\right]$.

```
Game $\mathsf{G}^{\mathsf{INT\text{-}PTXT}^*}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},n_s,\mathsf{Samp},\mathcal{A}}$
1: $st \leftarrow \varepsilon$; $\mathcal{M} \leftarrow \emptyset$
2: $n_n \leftarrow\!\!\$\ \{0,1\}^{256}$
3: $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
4: $\mathcal{A}^{\textsc{Enc},\textsc{Dec}}$
5: return false

$\textsc{Enc}(aux)$    // $aux \in \{0,1\}^*$
1: $(st, m, x) \leftarrow\!\!\$\ \mathsf{Samp}(st, aux)$
2: if $m = \perp$: return $\perp$
3: $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
4: $c \leftarrow\!\!\$\ \mathsf{HtE\text{-}SE.Enc}(k_{se}, m)$
5: return $(m, x, c)$

$\textsc{Dec}(c)$
1: $m \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c)$
2: if $m = \perp$: abort(false)
3: if $m \notin \mathcal{M}$: abort(true)
4: return $m$
```

Fig. 40. Weak plaintext integrity of HtE-SE for keys derived by SKDF on input n_s and messages sampled by Samp.

Given an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against the INT-PTXT security of HtE-SE, we can express its advantage as an expected value of the advantage of $\mathcal{A}_2$ against the INT-PTXT* security of HtE-SE as follows:

$$\mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p}(\mathcal{A}) = \mathbb{E}_{n \leftarrow\$\{0,1\}^{128};\ (n_s, st_{\mathcal{A}}) \leftarrow\$\mathcal{A}_1(n)}[\mathsf{Adv}^{\mathsf{INT\text{-}PTXT}^*}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},n_s,\mathsf{SAMP}[n,n_s,g,p]}(\mathcal{A}_2(st_{\mathcal{A}}))].$$

Here the expected value is expressed over the randomness in $n \leftarrow\!\!\$\ \{0,1\}^{128}$; $(n_s, st_{\mathcal{A}}) \leftarrow\!\!\$\ \mathcal{A}_1(n)$, and $\mathcal{A}_2(st_{\mathcal{A}})$ can be thought of as adversary $\mathcal{A}_2$ with some hardcoded input $st_{\mathcal{A}}$. This relation allows us to prove the INT-PTXT* security of HtE-SE, and use the obtained upper bound for its advantage as the upper bound for the INT-PTXT security of HtE-SE. Note that this is meaningful only because our proof for INT-PTXT* is generic with respect to the actual value of n_s, i.e. there is no advantage to be gained from choosing a "bad" value of n_s by $\mathcal{A}_1$.

F.2 SKDF is an OTPRF

Here, we show that the function SKDF is a one-time PRF.

Proposition 2. *Let $\mathcal{D}_{\mathsf{OTPRF}}$ be an adversary against the* OTPRF*-security of* SKDF*. Then we can build an adversary $\mathcal{D}_{\mathsf{3TPRF}}$ against the* 3TPRF*-security of* SHACAL-1 *(Definition 5) such that $\mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SKDF}}(\mathcal{D}_{\mathsf{OTPRF}}) = \mathsf{Adv}^{\mathsf{3TPRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{3TPRF}})$.*

Proof. We proceed by way of a straightforward reduction. Given an adversary $\mathcal{D}_{\mathsf{OTPRF}}$, we directly build an adversary $\mathcal{D}_{\mathsf{3TPRF}}$ that simulates the game $\mathrm{G}^{\mathsf{OTPRF}}_{\mathsf{SKDF},\mathcal{D}_{\mathsf{OTPRF}}}$ (shown in expanded form in Fig. 41) for $\mathcal{D}_{\mathsf{OTPRF}}$, providing an oracle RORSIM and outputting the bit guess of $\mathcal{D}_{\mathsf{OTPRF}}$. The RORSIM oracle first makes a ROR call in its own 3TPRF game to obtain the values $k', (r_0, r_1, r_2)$. Then, it computes the output $y \leftarrow h_0 \parallel h_1 \parallel h' \parallel k'$ where:

$$h_i \leftarrow \mathtt{IV}_{160} \dotplus r_i \text{ for } i \in \{0,1,2\}$$
$$h' \leftarrow h_2 \dotplus \mathsf{SHACAL\text{-}1.Ev}(\mathtt{pad}', h_2).$$

Note that RORSIM outputs the "real" k' regardless of the bit in the 3TPRF game, however if $b = 0$ the value is random and unrelated to the remaining outputs, just like in the OTPRF game. Hence, $\mathcal{D}_{\mathsf{3TPRF}}$ wins whenever $\mathcal{D}_{\mathsf{OTPRF}}$ wins.

```
Game $\mathrm{G}^{\mathsf{OTPRF}}_{\mathsf{SKDF},\mathcal{D}}$
1 : $b \leftarrow\$ \{0,1\}$
2 : $b' \leftarrow\$ \mathcal{D}^{\mathrm{ROR}}()$
3 : return $b' = b$

ROR($x$)    // $|x| = 128$
1 : $k \leftarrow\$ \{0,1\}^{256}$
2 : $h_0 \leftarrow \mathtt{IV}_{160} \dotplus \mathsf{SHACAL\text{-}1.Ev}(k \parallel x \parallel \mathtt{pad}, \mathtt{IV}_{160})$
3 : $h_1 \leftarrow \mathtt{IV}_{160} \dotplus \mathsf{SHACAL\text{-}1.Ev}(x \parallel k \parallel \mathtt{pad}, \mathtt{IV}_{160})$
4 : $h_2 \leftarrow \mathtt{IV}_{160} \dotplus \mathsf{SHACAL\text{-}1.Ev}(k \parallel k, \mathtt{IV}_{160})$
5 : $h' \leftarrow h_2 \dotplus \mathsf{SHACAL\text{-}1.Ev}(\mathtt{pad}', h_2)$
6 : $y_1 \leftarrow h_0 \parallel h_1 \parallel h' \parallel k[0:32]$
7 : $y_0 \leftarrow\$ \{0,1\}^{512}$
8 : return $y_b$
```

Fig. 41. Game $\mathrm{G}^{\mathsf{OTPRF}}_{\mathsf{SKDF},\mathcal{D}}$ with an expanded definition of SKDF, with different but known SHA-1 padding $\mathtt{pad}, \mathtt{pad}'$.

F.3 UPREF: Prefix unpredictability of SKDF

Here, we define an intermediate notion that requires SKDF to have unpredictable prefixes. In Fig. 42 we require that for a 512-bit key $k_{se} = k \parallel iv$ derived by SKDF (which can be thought of as containing two 256-bit halves k, iv), it is hard to predict k even if iv is known. The adversary is assumed to know n_s, while n_n is secret.

Definition 15. *Consider the game $\mathrm{G}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s,\mathcal{A}}$ in Fig. 42 with the function* SKDF *defined in Section 4.2, a parameter n_s of size 128 bits, and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the* UPREF*-security of* SKDF *is defined as $\mathsf{Adv}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s,\mathcal{A}}\right]$.*

```
Game $\mathrm{G}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s,\mathcal{A}}$
1 :  $\mathsf{win} \leftarrow \mathtt{false}$
2 :  $n_n \leftarrow\!\!\$\ \{0,1\}^{256}$
3 :  $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
4 :  $\mathcal{A}^{\mathrm{GUESS}}(k_{se}[256:512])$
5 :  return $\mathsf{win}$

GUESS($pref$)    // $|pref| = 256$
1 :  if $pref = k_{se}[0:256]$ :
2 :      $\mathsf{win} \leftarrow \mathtt{true}$
```

Fig. 42. Prefix unpredictability of keys derived by SKDF on input n_s.

We now prove that this notion is satisfied if SKDF is a one-time PRF.

Proposition 3. *Let $\mathcal{A}$ be an adversary against the* UPREF*-security of* SKDF *with respect to n_s, making* $\mathsf{n_{Guess}}$ *queries to its* GUESS *oracle. Then we can build an adversary $\mathcal{D}$ against the* OTPRF*-security of* SKDF *such that* $\mathsf{Adv}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SKDF}}(\mathcal{D}) + \mathsf{n_{Guess}} \cdot 2^{-256}$.

Proof. We build $\mathcal{D}$ as follows: it sets $\mathsf{win} \leftarrow \mathtt{false}$, and obtains $k_{se} \leftarrow\!\!\$\ \mathrm{ROR}(n_s)$ using its own oracle from the OTPRF game. Then it simulates the UPREF game for $\mathcal{A}$, where GUESSSIM behaves exactly as GUESS in the original game. Finally, it outputs $b' \leftarrow \mathsf{win} = \mathtt{true}$ as its guess of the challenge bit b. Observe that if $b = 1$, $k_{se} = \mathsf{SKDF}(n_n, n_s)$ for random n_n, i.e. $\mathcal{D}$ perfectly simulates the UPREF game for $\mathcal{A}$. Otherwise, the key k_{se} is a random value, and so $\mathcal{A}$ can only set win to $\mathtt{true}$ by guessing. We can write

$$\begin{aligned}\mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SKDF}}(\mathcal{D}) &= \Pr\left[b' = 1 \mid b = 1\right] - \Pr\left[b' = 1 \mid b = 0\right] \\ &= \Pr\left[\mathrm{G}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s,\mathcal{A}}\right] - \Pr\left[\mathsf{win} = \mathtt{true} \,\middle|\, k_{se}[0:256] \leftarrow\!\!\$\ \{0,1\}^{256}\right] \\ &= \mathsf{Adv}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s}(\mathcal{A}) - \frac{\mathsf{n_{Guess}}}{2^{256}}.\end{aligned}$$

□

F.4 USUFF: Suffix unpredictability of SKDF

Here, we define an intermediate notion that expresses that an adversary cannot predict the suffix of the keys computed via SKDF, even in the presence of evaluation oracles using some ideal permutations. In Fig. 43, the adversary $\mathcal{A}$ has unrestricted access to X, X^{-1}, EVAL and EVAL^{-1}, Here X is an ideal permutation and EVAL evaluates the function $\zeta = \oplus_{k_1} \circ \mathrm{X} \circ \oplus_{k_0}$ on an input p. The adversary $\mathcal{A}$ wins if it can find $k_1 \in \{0,1\}^{128}$, hence the name "suffix", such that c was never involved in an EVAL or EVAL^{-1} query but $k_1 \oplus c$ was queried to X^{-1} or output by X.

The motivation for this notion comes from the presence of IGE mode in the proof of Proposition 5. There, we sample IVs for IGE and provide the adversary with oracle access to IGE encryption and decryption oracles that take as input only one-block-long plaintexts or ciphertexts respectively. This is captured by oracles EVAL and EVAL^{-1}. We also provide the adversary with oracle access to the underlying block cipher with a hardcoded key, explaining why we here only deal with an ideal permutation rather than an ideal cipher. This is captured by oracles X and X^{-1}. The adversary wins if it can find some value $c^\star$ in the range of IGE encrypt algorithm, such that $c^\star$ was never returned by a prior call to IGE encryption.[41] In particular, what we call k_0, k_1 here are actually IV values in IGE mode, but we adopted those names here to match more closely with the Even-Mansour scheme which appears in our analysis.

[41] We also have that $c^\star$ was never queried into IGE decryption. We do not need this in our proof of Proposition 5 but allows us to appeal to known results in the literature below.

Definition 16. *Consider the game* $G^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s,\mathcal{A}}$ *in Fig. 43 with the function* SKDF *defined in Section 4.2, a parameter n_s of size 128 bits, and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the* USUFF*-security of* SKDF *is defined as* $\mathsf{Adv}^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s}(\mathcal{A}) := \Pr\left[G^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s,\mathcal{A}}\right]$.

```
$G^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s,\mathcal{A}}$
1: $\mathcal{C} \leftarrow \emptyset$
2: $n_n \leftarrow\$ \{0,1\}^{256}$
3: $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
4: $k_0 \leftarrow k_{se}[256:384]$
5: $k_1 \leftarrow k_{se}[384:512]$
6: $c^\star \leftarrow\$ \mathcal{A}^{\textsc{Eval},\textsc{Eval}^{-1},\mathsf{X},\mathsf{X}^{-1}}$
7: $\mathsf{win}_0 \leftarrow (c^\star \notin \mathcal{C})$
8: $\mathsf{win}_1 \leftarrow (\mathsf{B}[c^\star \oplus k_1] \neq \bot)$
9: return $\mathsf{win}_0 \wedge \mathsf{win}_1$

$\textsc{Eval}(p)$   // $|p| = 128$
1: $c \leftarrow \mathsf{X}(p \oplus k_0) \oplus k_1$
2: $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
3: return $c$

$\textsc{Eval}^{-1}(c)$   // $|c| = 128$
1: $p \leftarrow \mathsf{X}^{-1}(c \oplus k_1) \oplus k_0$
2: $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
3: return $p$

$\mathsf{X}(u)$   // $|u| = 128$
1: if $\mathsf{A}[u] = \bot$:
2:     $\mathsf{A}[u] \leftarrow\$ \{0,1\}^{128} \setminus \mathsf{A}$
3:     $\mathsf{B}[\mathsf{A}[u]] \leftarrow u$
4: return $\mathsf{A}[u]$

$\mathsf{X}^{-1}(v)$   // $|v| = 128$
1: if $\mathsf{B}[v] = \bot$:
2:     $\mathsf{B}[v] \leftarrow\$ \{0,1\}^{128} \setminus \mathsf{B}$
3:     $\mathsf{A}[\mathsf{B}[v]] \leftarrow v$
4: return $\mathsf{B}[v]$
```

Fig. 43. Suffix unpredictability of keys derived by SKDF on input n_s.

Below, we establish that USUFF is hard, if SKDF is a one-time PRF. We first replace the outputs of SKDF with uniformly random values and then appeal to the security of the Even-Mansour construction [EM97] in the ideal cipher model. We give the Even-Mansour security game in Fig. 44. It has been established in [EM97] that no adversary succeeds with probability great than $O(\mathsf{n}_\mathsf{X} \cdot \mathsf{n}_{\textsc{Eval}}/2^{128})$. We may simplify this to $\varepsilon = O(t^2/2^{128})$.

```
$G^{\mathsf{EM}}_{\mathcal{B}}$
1: $\mathcal{S}, \mathsf{n}_{\textsc{Eval}}, \mathsf{n}_\mathsf{X} \leftarrow \emptyset, 0, 0$
2: $k_0 \leftarrow\$ \{0,1\}^{128}$
3: $k_1 \leftarrow\$ \{0,1\}^{128}$
4: $p^\star, c^\star \leftarrow\$ \mathcal{B}^{\textsc{Eval},\textsc{Eval}^{-1},\mathsf{X},\mathsf{X}^{-1}}$
5: $\mathsf{win}_0 \leftarrow ((p^\star, c^\star) \notin \mathcal{S})$
6: $\mathsf{win}_1 \leftarrow (\textsc{Eval}(p^\star) = c^\star)$
7: return $\mathsf{win}_0 \wedge \mathsf{win}_1$

$\textsc{Eval}(p)$   // $|p| = 128$
1: $c \leftarrow \mathsf{X}(p \oplus k_0) \oplus k_1$
2: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(p, c)\}$
3: $\mathsf{n}_{\textsc{Eval}} \leftarrow \mathsf{n}_{\textsc{Eval}} + 1$
4: return $c$

$\textsc{Eval}^{-1}(c)$   // $|c| = 128$
1: $p \leftarrow \mathsf{X}^{-1}(c \oplus k_1) \oplus k_0$
2: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(p, c)\}$
3: $\mathsf{n}_{\textsc{Eval}} \leftarrow \mathsf{n}_{\textsc{Eval}} + 1$
4: return $p$

$\mathsf{X}(u)$   // $|u| = 128$
1: if $\mathsf{A}[u] = \bot$ :
2:     $\mathsf{A}[u] \leftarrow\$ \{0,1\}^{128} \setminus \mathsf{A}$
3:     $\mathsf{B}[\mathsf{A}[u]] \leftarrow u$
4: $\mathsf{n}_\mathsf{X} \leftarrow \mathsf{n}_\mathsf{X} + 1$; return $\mathsf{A}[u]$

$\mathsf{X}^{-1}(v)$   // $|v| = 128$
1: if $\mathsf{B}[v] = \bot$ :
2:     $\mathsf{B}[v] \leftarrow\$ \{0,1\}^{128} \setminus \mathsf{B}$
3:     $\mathsf{A}[\mathsf{B}[v]] \leftarrow v$
4: $\mathsf{n}_\mathsf{X} \leftarrow \mathsf{n}_\mathsf{X} + 1$; return $\mathsf{B}[v]$
```

Fig. 44. Even-Mansour security game.

Proposition 4. *Let $\mathcal{A}$ be an adversary against* $G^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s,\mathcal{A}}$ *that makes* $\mathsf{n}_{\textsc{Eval}}$ *queries to* Eval *or* $\textsc{Eval}^{-1}$ *and* n_X *queries to* X *or* X^{-1} *and runs in time t. Then we can build an adversary $\mathcal{D}$ against the* OTPRF*-security of* SKDF *(Appendix F.2) such that*

$$\mathsf{Adv}^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SKDF}}(\mathcal{D}) + O\left(\frac{(3\,\mathsf{n}_\mathsf{X}+1)\cdot \max(\mathsf{n}_{\textsc{Eval}}, 3)}{2^{128}}\right) + \mathsf{negl}(\lambda).$$

Proof. We proceed in a series of game hops. G_0 is the $G^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s,\mathcal{A}}$ game. In G_1 we replace the output of SKDF with random values. This change is detectable by an adversary with advantage at most $\mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SKDF}}(\mathcal{D})$ for any PPT adversary $\mathcal{D}$.

In G_2, we are now playing a variant of the game in Fig. 44. We construct an adversary $\mathcal{B}$ against $G^{\mathsf{EM}}_{\mathcal{B}}$. Note that in this game, the exact same four oracles are used. This means that $\mathcal{B}$ can simulate $G^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s,\mathcal{A}}$ using the oracles provided in the Even-Mansour game, forwarding queries and keeping track of query response pairs. Let $\mathcal{Q}_{\text{EVAL}}$ be the set of query-response pairs submitted to EVAL or EVAL^{-1}, and $\mathcal{Q}_{\mathsf{X}}$ be the set of query-response pairs submitted to X or X^{-1}. We normalise the pairs to always store (p, c) for $c = \text{EVAL}(p)$ and (u, v) for $u = \mathsf{X}^{-1}(v)$.

Eventually, after t steps the adversary $\mathcal{A}$ outputs some $c^\star$. Assume $\mathcal{A}$ succeeded. If $\mathcal{Q}_{\text{EVAL}}$ does not contain at least three pairs, then $\mathcal{B}$ picks random p_i and queries $\text{EVAL}(x)$ and records the output as c_i. Otherwise it picks random pairs from $\mathcal{Q}_{\text{EVAL}}$ and calls them (p_0, c_0), (p_1, c_1), and (p_2, c_2). From the winning condition win_1 we know that there exists at least one value j such that $v_j = c^\star \oplus k_1$. The adversary $\mathcal{B}$ will use this fact to recover k_0 and k_1 that will enable it to win the Even-Mansour game.

Our new adversary $\mathcal{B}$ now loops through all pairs $(u_i, v_i) \in \mathcal{Q}_{\mathsf{X}}$. For each v_i, $\mathcal{B}$ calculates a key guess $k_1^{(i)} := v_i \oplus c^\star$ and uses (p_0, c_0) to calculate the corresponding value for $k_0^{(i)} := \mathsf{X}^{-1}(c_0 \oplus k_1^{(i)}) \oplus p_0$. It then checks if this key guess is the correct one by calculating $\mathsf{X}(p_1 \oplus k_0^{(i)}) \oplus k_1^{(i)}$ and comparing it to c_1. If it matches, $\mathcal{B}$ uses $k_0^{(i)}, k_1^{(i)}$ and $\mathsf{X}()$ to return $(p^\star, c^\star)$ such that $\text{EVAL}(p^\star) = c^\star$. Note that there is a negligible probability that more than one value of v_i passes the test. In that case, we can use a third (p_2, c_2) to check which of the possible resulting keys is the correct one.

Finally, we note that by [EM97] the success probability of $\mathcal{B}$ is bounded by $O(\mathsf{n}'_{\mathsf{X}} \cdot \mathsf{n}'_{\text{EVAL}}/2^{128})$ where $\mathsf{n}'_{\mathsf{X}} \leq 3\,\mathsf{n}_{\mathsf{X}} + 1$ and $\mathsf{n}'_{\text{EVAL}} \leq \max(\mathsf{n}_{\text{EVAL}}, 3)$. □

F.5 Proof for INT-PTXT of HtE-SE and SKDF

Proposition 5. *Let $n_s \in \{0,1\}^{128}$. Let $\mathsf{Samp} = \mathsf{SAMP}[n, n_s, g, p]$ be the message sampler given in Definition 6, instantiated with any n, g, p. Let* SKDF, HtE-SE *be as defined in Section 4.2. We model* AES-256 *in* HtE-SE *as an ideal cipher. Let $\mathcal{A}_{\text{INT-PTXT}^*}$ be any adversary against the* INT-PTXT*-*security of* HtE-SE *with respect to* SKDF, *n_s, and* Samp. *Let $t_{total} \geq 1$ be the number of 128-bit blocks of* AES-256 *data that are encrypted or decrypted in total across all the* ENC *and* DEC *oracle queries made by $\mathcal{A}_{\text{INT-PTXT}^*}$. Then we can build adversaries $\mathcal{A}_{\mathsf{SPR}}$ against the* SPR-*security of* SHA-1 *with respect to* Samp *(Definition 7), $\mathcal{A}_{\mathsf{UPREF}}$ against the* UPREF-*security of* SKDF *(Definition 15), and $\mathcal{A}_{\mathsf{USUFF}}$ against the* USUFF-*security of* SKDF *(Definition 16) such that*

$$\begin{aligned}\mathsf{Adv}^{\text{INT-PTXT}^*}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},n_s,\mathsf{Samp}}(\mathcal{A}_{\text{INT-PTXT}^*}) \leq \frac{t_{total}^2}{2^{128}} &+ \mathsf{Adv}^{\mathsf{SPR}}_{\mathsf{SHA\text{-}1},\mathsf{Samp}}(\mathcal{A}_{\mathsf{SPR}}) \\ &+ \mathsf{Adv}^{\mathsf{UPREF}}_{\mathsf{SKDF},n_s}(\mathcal{A}_{\mathsf{UPREF}}) + \mathsf{Adv}^{\mathsf{USUFF}}_{\mathsf{SKDF},n_s}(\mathcal{A}_{\mathsf{USUFF}}).\end{aligned}$$

Assume $\mathcal{A}_{\text{INT-PTXT}^*}$ makes n_{ENC} queries to its oracle ENC, *and n_{IC} queries in total to its oracles* IC, IC^{-1}. *Then $\mathcal{A}_{\mathsf{SPR}}$ makes n_{ENC} queries to its oracle* NEWMSG; *$\mathcal{A}_{\mathsf{UPREF}}$ makes n_{IC} queries to its oracle* GUESS; *$\mathcal{A}_{\mathsf{USUFF}}$ makes n_{ENC} queries to its oracle* EVAL, *and t_{total} queries in total to its oracles* X *and* X^{-1}.

Intuitively, this proof contains two high-level steps. First, we argue that an adversary is unlikely to produce a plaintext forgery that reuses the first block of a ciphertext obtained from a prior encryption query. Then we argue that placing any other value in the first block of the attempted ciphertext forgery – will produce a plaintext that looks uniformly random, and hence fails the integrity check. The latter claim relies on the use of secret IV values for the IGE encryption and decryption operations (in HtE-SE).

Proof. This proof uses games G_0–G_{11} in Figs. 45, 47 and 49. In all figures: the code highlighted in green was added for transitions between games, and does not affect the functionality of the initial game in the corresponding figure. The code highlighted in gray generally rewrites the code of the previous game *from the previous figure* in an equivalent way.

In the proof, we let $\mathcal{P}(\ell)$ denote the set of all bit-string permutations $\pi\colon \{0,1\}^\ell \to \{0,1\}^\ell$ (the inverse of π is then π^{-1}).

G_0. Game G_0 is functionally equivalent to game $G^{\text{INT-PTXT}^*}_{\text{HtE-SE},\text{SKDF},n_s,\text{Samp},\mathcal{A}_{\text{INT-PTXT}^*}}$. The former expands the code of algorithms HtE-SE.Enc, HtE-SE.Dec, and further expands the code of algorithms AES-256-IGE.Enc, AES-256-IGE.Dec (in the ideal cipher model). Highlighted in gray is the expanded code of AES-256-IGE, and also the code parsing k_{se} into parts (because it was immediately moved from oracles ENC, DEC into the main body of the game). We have

$$\Pr[G_0] = \Pr[G^{\text{INT-PTXT}^*}_{\text{HtE-SE},\text{SKDF},n_s,\text{Samp},\mathcal{A}_{\text{INT-PTXT}^*}}] = \mathsf{Adv}^{\text{INT-PTXT}^*}_{\text{HtE-SE},\text{SKDF},n_s,\text{Samp}}(\mathcal{A}_{\text{INT-PTXT}^*}).$$

```
Games G0–G2
$st \leftarrow \varepsilon$ ; $\mathcal{M} \leftarrow \emptyset$ ; $n_n \leftarrow\$ \{0,1\}^{256}$
$k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$ ; $k \leftarrow k_{se}[0:256]$
$c_0 \leftarrow k_{se}[256:384]$ ; $p_0 \leftarrow k_{se}[384:512]$
$\mathcal{A}^{\text{ENC},\text{DEC},\text{IC},\text{IC}^{-1}}_{\text{INT-PTXT}^*}$ ; return false

ENC($aux$)    // $aux \in \{0,1\}^*$
$(st, m, x) \leftarrow\$ \mathsf{Samp}(st, aux)$
if $m = \perp$ : return $\perp$
$\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
$\ell \leftarrow (128 - (160 + |m|)) \bmod 128$
$r \leftarrow\$ \{0,1\}^\ell$    // pad to block length
$p \leftarrow \mathsf{SHA\text{-}1}(m) \,\|\, m \,\|\, r$
// Parse $p$ into 128-bit blocks $p_1 \,\|\, \ldots \,\|\, p_t$.
for $i = 1, \ldots, t$ :
    $c_i \leftarrow \mathrm{IC}(k, p_i \oplus c_{i-1}) \oplus p_{i-1}$
$c \leftarrow c_1 \,\|\, \ldots \,\|\, c_t$
$\mathsf{preImg}[c_1] \leftarrow m$ ; return $(m, x, c)$

IC($i, u$)    // $|i| = 256$, $|u| = 128$
if $\mathsf{P}[i] = \perp$ : $\mathsf{P}[i] \leftarrow\$ \mathcal{P}(128)$
$\pi \leftarrow \mathsf{P}[i]$ ; return $\pi(u)$

IC$^{-1}$($i, v$)    // $|i| = 256$, $|v| = 128$
if $\mathsf{P}[i] = \perp$ : $\mathsf{P}[i] \leftarrow\$ \mathcal{P}(128)$
$\pi \leftarrow \mathsf{P}[i]$ ; return $\pi^{-1}(v)$

DEC($c$)
// Parse $c$ into 128-bit blocks $c_1 \,\|\, \ldots \,\|\, c_t$.
$p_1 \leftarrow \mathrm{IC}^{-1}(k, c_1 \oplus p_0) \oplus c_0$
for $i = 2, \ldots, t$ :
    $p_i \leftarrow \mathrm{IC}^{-1}(k, c_i \oplus p_{i-1}) \oplus c_{i-1}$
$p \leftarrow p_1 \,\|\, \ldots \,\|\, p_t$
for $i = 0, \ldots, 15$ :    // bytes of padding
    $m \leftarrow p[160 : |p| - i \cdot 8]$
    if $\mathsf{SHA\text{-}1}(m) = p[0:160]$ :
        if $\mathsf{preImg}[c_1] \neq \perp$ :
            $m_{\text{ENC}} \leftarrow \mathsf{preImg}[c_1]$
            if $m = m_{\text{ENC}}$ :
                return $m_{\text{ENC}}$    // G1–G2
            elseif $m \neq m_{\text{ENC}}$ :
                $\mathsf{bad}_0 \leftarrow$ true
                $\omega \leftarrow \mathsf{SHA\text{-}1}(m_{\text{ENC}}) \,\|\, m_{\text{ENC}}$    // G2
                for $i' = 0, \ldots, 15$ :    // G2
                    if $\omega = p[0 : |p| - i' \cdot 8]$ :    // G2
                        return $m_{\text{ENC}}$    // G2
                abort(false)    // G2
        if $m \notin \mathcal{M}$ : abort(true)
        return $m$
abort(false)
```

Fig. 45. Games G_0–G_2 for the proof of Proposition 5.

$G_0 \to G_1$. Game G_1 rewrites game G_0 in a functionally equivalent way, moving a single **return** statement up (in oracle DEC) and hence skipping a conditional statement that leads to **abort**(false) (this condition would have failed because $m = m_{\mathsf{ENC}}$ is an honestly forwarded message, i.e. $m \in \mathcal{M}$). We have

$$\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] = 0.$$

The only purpose of G_1 is to later allow us to rewrite game G_2 in an equivalent way into G_3.

$\mathrm{G}_1 \rightarrow \mathrm{G}_2$. In Fig. 46 we build adversary $\mathcal{A}_{\mathsf{SPR}}$ that breaks SPR security of SHA-1 whenever bad_0 is set in game G_1, so

$$\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \Pr[\mathsf{bad}_0^{\mathrm{G}_1}] \leq \Pr[\mathrm{G}^{\mathsf{SPR}}_{\mathsf{SHA\text{-}1,Samp},\mathcal{A}_{\mathsf{SPR}}}] = \mathsf{Adv}^{\mathsf{SPR}}_{\mathsf{SHA\text{-}1,Samp}}(\mathcal{A}_{\mathsf{SPR}}).$$

In game G_2, the code below $\mathsf{bad}_0 \leftarrow \mathtt{true}$ returns m_{ENC} if and only if this message was previously encrypted in a prior call to ENC. Otherwise, it calls **abort**(`false`), which halts the game with **return** `false` (meaning $\mathcal{A}$ lost the game). At the high level, this change eliminates any ambiguity in what could happen if DEC is queried on a ciphertext that contains the first block from a prior ENC query. Now either the decryption fails, or the corresponding message from ENC must be returned; no other message (and in particular a forgery) can be produced this way.

```
Adversary $\mathcal{A}_{\mathsf{SPR}}^{\textsc{NewMsg}}$
  $st \leftarrow \varepsilon$ ; $\mathcal{M} \leftarrow \emptyset$ ; $n_n \leftarrow\!\!\$\ \{0,1\}^{256}$
  $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$ ; $k \leftarrow k_{se}[0:256]$
  $c_0 \leftarrow k_{se}[256:384]$ ; $p_0 \leftarrow k_{se}[384:512]$
  $\mathcal{A}_{\mathsf{INT\text{-}PTXT^*}}^{\textsc{Enc},\textsc{Dec},\mathrm{IC},\mathrm{IC}^{-1}}$

$\textsc{Enc}(aux)$    // $aux \in \{0,1\}^*$
  $(m, x) \leftarrow \textsc{NewMsg}(aux)$
  if $m = \perp$ : return $\perp$
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
  $\ell \leftarrow (128 - (160 + |m|)) \bmod 128$
  $r \leftarrow\!\!\$\ \{0,1\}^{\ell}$    // pad to block length
  $p \leftarrow \mathsf{SHA\text{-}1}(m) \parallel m \parallel r$
  // Parse $p$ into 128-bit blocks $p_1 \parallel \ldots \parallel p_t$.
  for $i = 1, \ldots, t$ :
    $c_i \leftarrow \mathrm{IC}(k, p_i \oplus c_{i-1}) \oplus p_{i-1}$
  $c \leftarrow c_1 \parallel \ldots \parallel c_t$
  $\mathsf{preImg}[c_1] \leftarrow m$ ; return $(m, x, c)$

$\mathrm{IC}(i, u)$    // $|i| = 256$, $|u| = 128$
$\mathrm{IC}^{-1}(i, v)$    // $|i| = 256$, $|v| = 128$
// These oracles are identical to the
// corresponding oracles in game $\mathrm{G}_1$ of Fig. 45.
```

```
$\textsc{Dec}(c)$
  // Parse $c$ into 128-bit blocks $c_1 \parallel \ldots \parallel c_t$.
  $p_1 \leftarrow \mathrm{IC}^{-1}(k, c_1 \oplus p_0) \oplus c_0$
  for $i = 2, \ldots, t$ :
    $p_i \leftarrow \mathrm{IC}^{-1}(k, c_i \oplus p_{i-1}) \oplus c_{i-1}$
  $p \leftarrow p_1 \parallel \ldots \parallel p_t$
  for $i = 0, \ldots, 15$ :    // bytes of padding
    $m \leftarrow p[160 : |p| - i \cdot 8]$
    if $\mathsf{SHA\text{-}1}(m) = p[0:160]$ :
      if $\mathsf{preImg}[c_1] \neq \perp$ :
        $m_{\mathsf{ENC}} \leftarrow \mathsf{preImg}[c_1]$
        if $m = m_{\mathsf{ENC}}$ :
          return $m_{\mathsf{ENC}}$
        elseif $m \neq m_{\mathsf{ENC}}$ :
          $out \leftarrow (m_{\mathsf{ENC}}, m)$
          abort$(out)$
      if $m \notin \mathcal{M}$ : abort$(\perp)$
      return $m$
  abort$(\perp)$
```

Fig. 46. Adversary $\mathcal{A}_{\mathsf{SPR}}$ for the proof of Proposition 5. It simulates G_1 for $\mathcal{A}_{\mathsf{INT\text{-}PTXT^*}}$.

$\mathrm{G}_2 \rightarrow \mathrm{G}_3$. Game G_3 is functionally equivalent to game G_2, so

$$\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] = 0.$$

In particular, the code of oracle DEC is essentially duplicated twice depending on whether $\mathsf{preImg}[c_1] = \bot$. If $\mathsf{preImg}[c_1] = \bot$, then oracle DEC in G_3 runs the same code as in G_2. But if $\mathsf{preImg}[c_1] \neq \bot$, then $\mathcal{A}$ reused c_1 from a prior ENC query and that allows to introduce the following changes (without changing the functionality): (i) we can use $(p_1, m_{\text{ENC}}) \leftarrow \mathsf{preImg}[c_1]$ instead of explicitly using c_0, p_0 to derive p_1; these values are meant to be secret, so minimizing their usage now will help us in future steps; (ii) the **abort** (`true`) instruction cannot be reached during this call to DEC (so no further changes to this conditional branch will be needed in subsequent security games); (iii) the only message potentially returned by the current DEC call is some m_{ENC} that was encrypted in a prior ENC call (note that HtE-SE does not prevent an attacker from possibly mauling the ciphertext without changing the underlying plaintext; this possibility is covered here, but is not essential in our proof).

Besides the above changes, we also introduce new X and X^{-1} oracles that lazily sample and evaluate a random permutation. We set $\mathrm{IC}(k_{se}[0:256], x) := \mathsf{X}(x)$ and $\mathrm{IC}^{-1}(k_{se}[0:256], y) := \mathsf{X}^{-1}(y)$, and we call $\mathsf{X}, \mathsf{X}^{-1}$ directly from oracles ENC, DEC instead of calling $\mathrm{IC}(k_{se}[0:256], \cdot), \mathrm{IC}^{-1}(k_{se}[0:256], \cdot)$.

$\mathbf{G}_3 \to \mathbf{G}_4$. In G_4 we break the consistency between the ideal cipher oracles and X, X^{-1}. Meaning $\mathrm{IC}(k_{se}[0:256], x) := \mathsf{X}(x)$ and $\mathrm{IC}^{-1}(k_{se}[0:256], y) := \mathsf{X}^{-1}(y)$ is no longer true, and $\mathcal{A}$ can learn nothing useful even by querying the ideal cipher oracles on $k_{se}[0:256]$. This transition requires that $\mathcal{A}$ cannot learn the value $k_{se}[0:256]$, even if it happens to know the other half of k_{se} (there is no intuition for why it could know this; we just cannot get around giving this info to $\mathcal{A}$ for free in this transition). Formally, we build an UPREF adversary in Fig. 48 that wins whenever bad_1 is set in game G_4, so

$$\Pr[G_3] - \Pr[G_4] \leq \Pr[\mathsf{bad}_1^{G_4}] \leq \Pr[G^{\mathsf{UPREF}}_{\mathsf{SKDF}, n_s, \mathcal{A}_{\mathsf{UPREF}}}] = \mathsf{Adv}^{\mathsf{UPREF}}_{\mathsf{SKDF}, n_s}(\mathcal{A}_{\mathsf{UPREF}}).$$

$\mathbf{G}_4 \to \mathbf{G}_5$. In G_5 we do the PRP-to-PRF switch inside X, X^{-1} (simultaneously). Each call to either of X, X^{-1} fills up one new entry in each of A and B. There is a total of t_{total} such calls. In the first call, no collision can happen. In the second call, a collision could happen with probability at most $\frac{1}{2^{128}}$. In the t_{total}-th call a collision could happen with probability at most $\frac{t_{total}-1}{2^{128}}$. We have

$$\Pr[G_4] - \Pr[G_5] \leq \Pr[\mathsf{bad}_2^{G_5}] \leq \frac{1 + 2 + \ldots + (t_{total} - 1)}{2^{128}} = \frac{(t_{total} - 1) \cdot t_{total}}{2 \cdot 2^{128}}.$$

This is a standard birthday bound, i.e. $\mathsf{bb}(0, t_{total}, 0, 2^{128})$ as per the notation from Appendix D.1.

$\mathbf{G}_5 \to \mathbf{G}_6$. In G_6 we remove the dead code from oracles IC, IC^{-1}. And we rewrite/expand the code in the first branch of oracle DEC in a functionally equivalent way. We have

$$\Pr[G_5] - \Pr[G_6] = 0.$$

The code of AES-256-IGE is now evaluated separately in two conditional branches, depending on whether $\mathsf{B}[c_1 \oplus p_0] = \bot$. If $\mathsf{B}[c_1 \oplus p_0] = \bot$ then the first ciphertext block of c has never never been queried to X^{-1} and not previously returned by X. So it should decrypt into a uniformly random plaintext block, and subsequently cause the entire decrypted ciphertext be indistinguishable from a uniformly random string. We show this in the next steps.

$\mathbf{G}_6 \to \mathbf{G}_7$. In G_7 we rewrite $p_i \leftarrow \mathsf{X}^{-1}(v_i) \oplus c_{i-1}$ into the equivalent $p_i \leftarrow\$ \{0,1\}^{128}$, while maintaining proper bookkeeping of A and B. (No bookkeeping of sets $\mathcal{D}, \mathcal{R}$ is necessary here, because these sets are only updated, and never used beyond that.) This transformation is possible because the inverse $\mathsf{X}^{-1}(v_i)$ was not yet specified (i.e. $\mathsf{B}[v_i] = \bot$). We have

$$\Pr[G_6] - \Pr[G_7] = 0.$$

$\mathbf{G}_7 \to \mathbf{G}_8$. We now show that $\mathsf{B}[v_i] \neq \bot$ is unlikely to happen, meaning we can change the code in this conditional branch. We do this by bounding the probability of bad_3 being set in game G_8 (rather than in G_7), where every plaintext block p_i is sampled independently, uniformly at random. The best chance of triggering the bad_3 flag would be if $\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}$ queried its oracle DEC once, on a ciphertext containing t_{total} 128-bit blocks. This would guarantee that the conditions $\mathsf{preImg}[c_1] = \bot$ and $\mathsf{B}[c_1 \oplus p_0] = \bot$ would hold. This means that the DEC oracle would successively sample t_{total} random values p_i in an attempt to obtain a collision. We use the birthday bound (using the notation from Appendix D.1)

$$\Pr[G_7] - \Pr[G_8] \leq \Pr[\mathsf{bad}_3^{G_8}] \leq \mathsf{bb}(0, t_{total}, 0, 2^{128}) \leq \frac{(t_{total} - 1) \cdot t_{total}}{2 \cdot 2^{128}}.$$

$\mathbf{G}_8 \to \mathbf{G}_9$. In G_8 when DEC was called for $\mathsf{B}[c_1 \oplus p_0] = \bot$ we sampled the entire plaintext uniformly at random. The probability of the first 160 bits of such a plaintext containing the SHA-1 hash (or, more generally, any deterministic function) of a fixed message that is contained in subsequent blocks of this plaintext is $\frac{1}{2^{160}}$. There are 16 different ways to parse out a message from the plaintext, depending on the number of padding blocks appended to it. Let $\mathsf{n}_{\mathrm{DEC}}$ be the maximum number of DEC queries that might be made by $\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}$ during which the conditions $\mathsf{preImg}[c_1] = \bot$ and $\mathsf{B}[c_1 \oplus p_0] = \bot$ are true. Then we can upper bound the probability of setting the bad_4 flag in game G_9 as follows:

$$\Pr[G_8] - \Pr[G_9] \leq \Pr[\mathsf{bad}_4^{G_9}] \leq \mathsf{n}_{\mathrm{DEC}} \cdot 16 \cdot \frac{1}{2^{160}} \leq t_{total} \cdot 16 \cdot \frac{1}{2^{160}},$$

where the latter inequality holds because $\mathsf{n}_{\mathrm{DEC}} \leq t_{total}$ is trivially true.

$\mathbf{G}_9 \to \mathbf{G}_{10}$. Game G_{10} reverts the prior PRP-to-PRF switch, by replacing the **abort**(`false`) calls in oracles X and X^{-1} with the original pseudocode lines that were used before game G_5. An **abort**(`false`) call causes the adversary to immediately lose the game. The advantage of an adversary could only increase by replacing such a call with anything else. We have

$$\Pr[G_9] - \Pr[G_{10}] \leq 0.$$

$\mathbf{G}_{10} \to \mathbf{G}_{11}$. Finally, we build USUFF adversary in Fig. 50 that bounds the probability of bad_5 being set in game G_{11}. We define G_{11} to call **abort**(`false`) whenever $\mathsf{preImg}[c_1] = \bot$ and $\mathsf{B}[c_1 \oplus p_0] \neq \bot$. These two conditions together mean that the adversary queried a ciphertext whose first block c_1 is not equal to the first block of some prior challenge-encryption ciphertext, yet the ideal cipher mapping already maps the output value $c_1 \oplus p_0$ to its corresponding input value. (In particular, in this case we cannot use the above argument that c_1 should decrypt into a uniformly random plaintext block.) Intuitively, this is very unlikely to occur because it should be hard for the adversary to learn the value of p_0. The USUFF notion captures this intuition. In game G_{11}, the conditional branch for $\mathsf{B}[c_1 \oplus p_0] = \bot$ always returns **abort**(`false`), whereas the higher-level conditional branch for $\mathsf{preImg}[c_1] \neq \bot$ simply does not use any secret values (c_0, p_0 in particular). This allows our USUFF adversary to trivially simulate oracle DEC of game G_{11}. We have

$$\Pr[G_{10}] - \Pr[G_{11}] \leq \Pr[\mathsf{bad}_5^{G_{11}}] \leq \Pr[G^{\mathsf{USUFF}}_{\mathsf{SKDF}, n_s, \mathcal{A}_{\mathsf{USUFF}}}] = \mathsf{Adv}^{\mathsf{USUFF}}_{\mathsf{SKDF}, n_s}(\mathcal{A}_{\mathsf{USUFF}}).$$

$\mathbf{G}_{11}$. In game G_{11} the instruction **abort**(`true`) can no longer be reached, so $\mathcal{A}$ can never win. We have

$$\Pr[G_{11}] = 0.$$

To conclude the proof, we express

$$\mathsf{Adv}^{\mathsf{INT\text{-}PTXT}^*}_{\mathsf{HtE\text{-}SE}, \mathsf{SKDF}, n_s, \mathsf{Samp}}(\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}) = \sum_{i=0}^{10} (\Pr[G_i] - \Pr[G_{i+1}]) + \Pr[G_{11}],$$

and observe that $\Pr[\mathsf{bad}_4^{G_9}] \leq \frac{t_{total}}{2^{128}}$.

```
Games G3–G5
1 :  $st \leftarrow \varepsilon$ ; $\mathcal{M} \leftarrow \emptyset$ ; $n_n \leftarrow\!\!\$ \{0,1\}^{256}$
2 :  $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$ ; $k \leftarrow k_{se}[0:256]$
3 :  $c_0 \leftarrow k_{se}[256:384]$ ; $p_0 \leftarrow k_{se}[384:512]$
4 :  $\mathcal{D} \leftarrow \emptyset$ ; $\mathcal{R} \leftarrow \emptyset$
5 :  $\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}^{\mathrm{ENC},\mathrm{DEC},\mathrm{IC},\mathrm{IC}^{-1}}$ ; return false

ENC(aux)    // $aux \in \{0,1\}^*$
1 :  $(st, m, x) \leftarrow\!\!\$ \mathsf{Samp}(st, aux)$
2 :  if $m = \bot$ : return $\bot$
3 :  $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
4 :  $\ell \leftarrow (128 - (160 + |m|)) \bmod 128$
5 :  $r \leftarrow\!\!\$ \{0,1\}^{\ell}$ ; $p \leftarrow \mathsf{SHA\text{-}1}(m) \,\|\, m \,\|\, r$
6 :  // Parse $p$ into 128-bit blocks $p_1 \| \ldots \| p_t$.
7 :  for $i = 1, \ldots, t$ :
8 :    $c_i \leftarrow \mathsf{X}(p_i \oplus c_{i-1}) \oplus p_{i-1}$
9 :  $c \leftarrow c_1 \| \ldots \| c_t$
10 : $\mathsf{preImg}[c_1] \leftarrow (p_1, m)$ ; return $(m, x, c)$

X(u)    // $|u| = 128$
1 :  if $\mathsf{A}[u] = \bot$ :
2 :    $\mathsf{A}[u] \leftarrow\!\!\$ \{0,1\}^{128}$
3 :    if $\mathsf{A}[u] \in \mathcal{R}$ :
4 :      $\mathrm{bad}_2 \leftarrow$ true
5 :      $\mathsf{A}[u] \leftarrow\!\!\$ \{0,1\}^{128} \setminus \mathcal{R}$    // G3–G4
6 :      abort(false)    // G5
7 :    $\mathsf{B}[\mathsf{A}[u]] \leftarrow u$
8 :    $\mathcal{D} \leftarrow \mathcal{D} \cup \{u\}$ ; $\mathcal{R} \leftarrow \mathcal{R} \cup \{\mathsf{A}[u]\}$
9 :  return $\mathsf{A}[u]$

X^{-1}(v)    // $|v| = 128$
1 :  if $\mathsf{B}[v] = \bot$ :
2 :    $\mathsf{B}[v] \leftarrow\!\!\$ \{0,1\}^{128}$
3 :    if $\mathsf{B}[v] \in \mathcal{D}$ :
4 :      $\mathrm{bad}_2 \leftarrow$ true
5 :      $\mathsf{B}[v] \leftarrow\!\!\$ \{0,1\}^{128} \setminus \mathcal{D}$    // G3–G4
6 :      abort(false)    // G5
7 :    $\mathsf{A}[\mathsf{B}[v]] \leftarrow v$
8 :    $\mathcal{D} \leftarrow \mathcal{D} \cup \{\mathsf{B}[v]\}$ ; $\mathcal{R} \leftarrow \mathcal{R} \cup \{v\}$
9 :  return $\mathsf{B}[v]$

DEC(c)
1 :  // Parse $c$ into 128-bit blocks $c_1 \| \ldots \| c_t$.
2 :  if $\mathsf{preImg}[c_1] = \bot$ :
3 :    for $i = 1, \ldots, t$ :
4 :      $p_i \leftarrow \mathsf{X}^{-1}(c_i \oplus p_{i-1}) \oplus c_{i-1}$
5 :    $p \leftarrow p_1 \| \ldots \| p_t$
6 :    for $i = 0, \ldots, 15$ :
7 :      $m \leftarrow p[160 : |p| - i \cdot 8]$
8 :      if $\mathsf{SHA\text{-}1}(m) = p[0:160]$ :
9 :        if $m \notin \mathcal{M}$ : abort(true)
10 :       return $m$
11 : elseif $\mathsf{preImg}[c_1] \neq \bot$ :
12 :   $(p_1, m_{\mathrm{ENC}}) \leftarrow \mathsf{preImg}[c_1]$
13 :   for $i = 2, \ldots, t$ :
14 :     $p_i \leftarrow \mathsf{X}^{-1}(c_i \oplus p_{i-1}) \oplus c_{i-1}$
15 :   $p \leftarrow p_1 \| \ldots \| p_t$
16 :   $\omega \leftarrow \mathsf{SHA\text{-}1}(m_{\mathrm{ENC}}) \,\|\, m_{\mathrm{ENC}}$
17 :   for $i = 0, \ldots, 15$ :
18 :     if $\omega = p[0 : |p| - i \cdot 8]$ :
19 :       return $m_{\mathrm{ENC}}$
20 : abort(false)

IC(i, u)    // $|i| = 256$, $|u| = 128$
1 :  if $i = k$ :
2 :    $\mathrm{bad}_1 \leftarrow$ true
3 :    return $\mathsf{X}(u)$    // G3
4 :  if $\mathsf{P}[i] = \bot$ : $\mathsf{P}[i] \leftarrow\!\!\$ \mathcal{P}(128)$
5 :  $\pi \leftarrow \mathsf{P}[i]$ ; return $\pi(u)$

IC^{-1}(i, v)    // $|i| = 256$, $|v| = 128$
1 :  if $i = k$ :
2 :    $\mathrm{bad}_1 \leftarrow$ true
3 :    return $\mathsf{X}^{-1}(v)$    // G3
4 :  if $\mathsf{P}[i] = \bot$ : $\mathsf{P}[i] \leftarrow\!\!\$ \mathcal{P}(128)$
5 :  $\pi \leftarrow \mathsf{P}[i]$ ; return $\pi^{-1}(v)$
```

Fig. 47. Games G3–G5 for the proof of Proposition 5.

```
Adversary $\mathcal{A}_{\mathsf{UPREF}}^{\textsc{Guess}}(iv)$
$st \leftarrow \varepsilon$ ; $\mathcal{M} \leftarrow \emptyset$
$c_0 \leftarrow iv[0:128]$ ; $p_0 \leftarrow iv[128:256]$
$\mathcal{D} \leftarrow \emptyset$ ; $\mathcal{R} \leftarrow \emptyset$
$\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}^{\textsc{Enc},\textsc{Dec},\mathrm{IC},\mathrm{IC}^{-1}}$

$\mathrm{IC}(i,u)$    // $|i| = 256$, $|u| = 128$
$\textsc{Guess}(i)$
if $\mathsf{P}[i] = \perp$ : $\mathsf{P}[i] \leftarrow\!\!\$\ \mathcal{P}(128)$
$\pi \leftarrow \mathsf{P}[i]$ ; return $\pi(u)$

$\mathrm{IC}^{-1}(i,v)$    // $|i| = 256$, $|v| = 128$
$\textsc{Guess}(i)$
if $\mathsf{P}[i] = \perp$ : $\mathsf{P}[i] \leftarrow\!\!\$\ \mathcal{P}(128)$
$\pi \leftarrow \mathsf{P}[i]$ ; return $\pi^{-1}(v)$

$\textsc{Enc}(aux)$    // $aux \in \{0,1\}^*$
$\mathsf{X}(u)$    // $|u| = 128$
$\mathsf{X}^{-1}(v)$    // $|v| = 128$
// These oracles are identical to the
// corresponding oracles in game $G_4$ of Fig. 47.

$\textsc{Dec}(c)$
// Parse $c$ into 128-bit blocks $c_1 \| \ldots \| c_t$.
if $\mathsf{preImg}[c_1] = \perp$ :
    for $i = 1, \ldots, t$ :
        $p_i \leftarrow \mathsf{X}^{-1}(c_i \oplus p_{i-1}) \oplus c_{i-1}$
    $p \leftarrow p_1 \| \ldots \| p_t$
    for $i = 0, \ldots, 15$ :
        $m \leftarrow p[160 : |p| - i \cdot 8]$
        if $\mathsf{SHA\text{-}1}(m) = p[0:160]$ :
            if $m \notin \mathcal{M}$ : abort$(\perp)$
            return $m$
elseif $\mathsf{preImg}[c_1] \neq \perp$ :
    $(p_1, m_{\mathrm{ENC}}) \leftarrow \mathsf{preImg}[c_1]$
    for $i = 2, \ldots, t$ :
        $p_i \leftarrow \mathsf{X}^{-1}(c_i \oplus p_{i-1}) \oplus c_{i-1}$
    $p \leftarrow p_1 \| \ldots \| p_t$
    $\omega \leftarrow \mathsf{SHA\text{-}1}(m_{\mathrm{ENC}}) \| m_{\mathrm{ENC}}$
    for $i = 0, \ldots, 15$ :
        if $\omega = p[0 : |p| - i \cdot 8]$ :
            return $m_{\mathrm{ENC}}$
abort$(\perp)$
```

Fig. 48. Adversary $\mathcal{A}_{\mathsf{UPREF}}$ for the proof of Proposition 5. It simulates G_4 for $\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}$.

```
Games $G_6$–$G_{11}$
$st \leftarrow \varepsilon$ ; $\mathcal{M} \leftarrow \emptyset$ ; $n_n \leftarrow\!\!\$ \{0,1\}^{256}$
$k_{se} \leftarrow$ SKDF.Ev$(n_n, n_s)$ ; $k \leftarrow k_{se}[0:256]$
$c_0 \leftarrow k_{se}[256:384]$ ; $p_0 \leftarrow k_{se}[384:512]$
$\mathcal{D} \leftarrow \emptyset$ ; $\mathcal{R} \leftarrow \emptyset$
$\mathcal{A}_{\text{INT-PTXT}^*}^{\text{Enc,Dec,IC,IC}^{-1}}$ ; return false
```

```
Enc$(aux)$   // $aux \in \{0,1\}^*$
$(st, m, x) \leftarrow\!\!\$$ Samp$(st, aux)$
if $m = \perp$ : return $\perp$
$\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
$\ell \leftarrow (128 - (160 + |m|)) \bmod 128$
$r \leftarrow\!\!\$ \{0,1\}^{\ell}$ ; $p \leftarrow$ SHA-1$(m) \,\|\, m \,\|\, r$
// Parse $p$ into 128-bit blocks $p_1 \| \ldots \| p_t$.
for $i = 1, \ldots, t$ :
    $c_i \leftarrow$ X$(p_i \oplus c_{i-1}) \oplus p_{i-1}$
$c \leftarrow c_1 \| \ldots \| c_t$
preImg$[c_1] \leftarrow (p_1, m)$ ; return $(m, x, c)$
```

```
X$(u)$   // $|u| = 128$
if A$[u] = \perp$ :
    A$[u] \leftarrow\!\!\$ \{0,1\}^{128}$
    if A$[u] \in \mathcal{R}$ :
        abort(false)   // $G_6$–$G_9$
        A$[u] \leftarrow\!\!\$ \{0,1\}^{128} \setminus \mathcal{R}$   // $G_{10}$–$G_{11}$
    B[A$[u]$] $\leftarrow u$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{u\}$ ; $\mathcal{R} \leftarrow \mathcal{R} \cup \{$A$[u]\}$
return A$[u]$
```

```
X$^{-1}(v)$   // $|v| = 128$
if B$[v] = \perp$ :
    B$[v] \leftarrow\!\!\$ \{0,1\}^{128}$
    if B$[v] \in \mathcal{D}$ :
        abort(false)   // $G_6$–$G_9$
        B$[v] \leftarrow\!\!\$ \{0,1\}^{128} \setminus \mathcal{R}$   // $G_{10}$–$G_{11}$
    A[B$[v]$] $\leftarrow v$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{$B$[v]\}$ ; $\mathcal{R} \leftarrow \mathcal{R} \cup \{v\}$
return B$[v]$
```

```
IC$(i, u)$   // $|i| = 256$, $|u| = 128$
if P$[i] = \perp$ : P$[i] \leftarrow\!\!\$ \mathcal{P}(128)$
$\pi \leftarrow$ P$[i]$ ; return $\pi(u)$
```

```
Dec$(c)$
// Parse $c$ into 128-bit blocks $c_1 \| \ldots \| c_t$.
if preImg$[c_1] = \perp$ :
    not-queried-before $\leftarrow$ (B$[c_1 \oplus p_0] = \perp$)
    if B$[c_1 \oplus p_0] \neq \perp$ :
        bad$_5 \leftarrow$ true
        abort(false)   // $G_{11}$
        for $i = 1, \ldots, t$ :
            $p_i \leftarrow$ X$^{-1}(c_i \oplus p_{i-1}) \oplus c_{i-1}$
    else :   // B$[c_1 \oplus p_0] = \perp$
        for $i = 1, \ldots, t$ :
            $v_i \leftarrow c_i \oplus p_{i-1}$
            if B$[v_i] = \perp$ :
                $p_i \leftarrow$ X$^{-1}(v_i) \oplus c_{i-1}$   // $G_6$
                $p_i \leftarrow\!\!\$ \{0,1\}^{128}$   // $G_7$–$G_{11}$
                B$[v_i] \leftarrow p_i \oplus c_{i-1}$   // $G_7$–$G_{11}$
                A[B$[v_i]$] $\leftarrow v_i$   // $G_7$–$G_{11}$
            else :   // B$[v_i] \neq \perp$
                bad$_3 \leftarrow$ true
                $p_i \leftarrow$ X$^{-1}(v_i) \oplus c_{i-1}$   // $G_6$–$G_7$
                $p_i \leftarrow\!\!\$ \{0,1\}^{128}$   // $G_8$–$G_{11}$
    $p \leftarrow p_1 \| \ldots \| p_t$
    for $i = 0, \ldots, 15$ :
        $m \leftarrow p[160 : |p| - i \cdot 8]$
        if SHA-1$(m) = p[0:160]$ :
            if not-queried-before :
                bad$_4 \leftarrow$ true
                abort(false)   // $G_9$–$G_{11}$
            if $m \notin \mathcal{M}$ : abort(true)
            return $m$
elseif preImg$[c_1] \neq \perp$ :
    $(p_1, m_{\text{ENC}}) \leftarrow$ preImg$[c_1]$
    for $i = 2, \ldots, t$ :
        $p_i \leftarrow$ X$^{-1}(c_i \oplus p_{i-1}) \oplus c_{i-1}$
    $p \leftarrow p_1 \| \ldots \| p_t$
    $\omega \leftarrow$ SHA-1$(m_{\text{ENC}}) \,\|\, m_{\text{ENC}}$
    for $i = 0, \ldots, 15$ :
        if $\omega = p[0 : |p| - i \cdot 8]$ :
            return $m_{\text{ENC}}$
abort(false)
```

```
IC$^{-1}(i, v)$   // $|i| = 256$, $|v| = 128$
if P$[i] = \perp$ : P$[i] \leftarrow\!\!\$ \mathcal{P}(128)$
$\pi \leftarrow$ P$[i]$ ; return $\pi^{-1}(v)$
```

Fig. 49. Games G_6–G_{11} for the proof of Proposition 5.

```
Adversary $\mathcal{A}_{\mathsf{USUFF}}^{\textsc{Eval},\mathrm{X},\mathrm{X}^{-1}}$
1 :  $st \leftarrow \varepsilon$ ; $\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}^{\textsc{Enc},\textsc{Dec},\mathrm{IC},\mathrm{IC}^{-1}}$

$\textsc{Enc}(aux)$    // $aux \in \{0,1\}^*$
1 :  $(st, m, x) \leftarrow\!\!\$\ \mathsf{Samp}(st, aux)$
2 :  if $m = \perp$ : return $\perp$
3 :  $\ell \leftarrow (128 - (160 + |m|)) \bmod 128$
4 :  $r \leftarrow\!\!\$\ \{0,1\}^{\ell}$ ; $p \leftarrow \mathsf{SHA\text{-}1}(m) \parallel m \parallel r$
5 :  // Parse $p$ into 128-bit blocks $p_1 \parallel \ldots \parallel p_t$.
6 :  $c_1 \leftarrow \textsc{Eval}(p_1)$
7 :  for $i = 2, \ldots, t$ :
8 :    $c_i \leftarrow \mathrm{X}(p_i \oplus c_{i-1}) \oplus p_{i-1}$
9 :  $c \leftarrow c_1 \parallel \ldots \parallel c_t$
10 : $\mathsf{preImg}[c_1] \leftarrow (p_1, m)$ ; return $(m, x, c)$

$\mathrm{IC}(i, u)$    // $|i| = 256$, $|u| = 128$
$\mathrm{IC}^{-1}(i, v)$    // $|i| = 256$, $|v| = 128$
// These oracles are identical to the
// corresponding oracles in game $\mathrm{G}_{11}$ of Fig. 49.

$\textsc{Dec}(c)$
1 :  // Parse $c$ into 128-bit blocks $c_1 \parallel \ldots \parallel c_t$.
2 :  if $\mathsf{preImg}[c_1] = \perp$ :
3 :    abort$(c_1)$
4 :  else :    // $\mathsf{preImg}[c_1] \neq \perp$
5 :    $(p_1, m_{\mathrm{ENC}}) \leftarrow \mathsf{preImg}[c_1]$
6 :    for $i = 2, \ldots, t$ :
7 :      $p_i \leftarrow \mathrm{X}^{-1}(c_i \oplus p_{i-1}) \oplus c_{i-1}$
8 :    $p \leftarrow p_1 \parallel \ldots \parallel p_t$
9 :    $\omega \leftarrow \mathsf{SHA\text{-}1}(m_{\mathrm{ENC}}) \parallel m_{\mathrm{ENC}}$
10 :   for $i = 0, \ldots, 15$ :
11 :     if $\omega = p[0 : |p| - i \cdot 8]$ :
12 :       return $m_{\mathrm{ENC}}$
13 : abort$(\perp)$
```

Fig. 50. Adversary $\mathcal{A}_{\mathsf{USUFF}}$ for the proof of Proposition 5. It simulates G_{11} for $\mathcal{A}_{\mathsf{INT\text{-}PTXT}^*}$.

G EUF-CMA: Existential unforgeability of MTProto 1.0 encryption

In Appendix G.1, we define an intermediate security notion used in the proof for $\mathsf{MTP\text{-}KE_{3st}}$ (Appendix H.2) that speaks to the unforgeability of outputs produced by CHv1. In Appendices G.2 and G.3, we then define and prove a number of sub-notions that will be used to prove that this property holds, in addition to any standard or non-standard assumptions. In Appendix G.4 we state the main result and give its proof.

G.1 Definition of EUF-CMA

Here, we define a notion of message unforgeability for CHv1. This may seem counterintuitive at first sight, as CHv1 is defined as a symmetric encryption scheme (albeit restricted for a particular message type). However, in stage 3 of $\mathsf{MTP\text{-}KE_{3st}}$ it is used in such a way that the decrypted plaintext must always match a given message, essentially mandating that the output of CHv1 functions as a message authentication code. The game Fig. 51 captures this usage scenario by explicitly requiring (mid, m) as input to the VFY oracle. For a successful forgery, the adversary must query VFY with a message m that was never used as part of a query to EVAL, but which passes all the checks imposed by CHv1.Dec as well as matches the decrypted message.

Definition 17. *Consider the game* $\mathrm{G}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1},\mathcal{A}}$ *in Fig. 51 with the MTProto 1.0 channel* CHv1 *defined in Fig. 8, and an adversary* $\mathcal{A}$*. The advantage of* $\mathcal{A}$ *in breaking the* EUF-CMA*-security of* CHv1 *is defined as* $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1}}(\mathcal{A}) := \Pr\left[\mathrm{G}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1},\mathcal{A}}\right]$.

```
Game $\mathrm{G}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1},\mathcal{A}}$
1: $\mathsf{T} \leftarrow []$
2: $\mathcal{M} \leftarrow \emptyset$
3: $\mathcal{A}^{\text{EVAL},\text{VFY}}()$
4: return false

GetKey($kid$)
1: if $\mathsf{T}[kid] = \perp$:
2:     $\mathsf{T}[kid] \leftarrow\!\$ \{0,1\}^{1024}$
3: return $\mathsf{T}[kid]$

EVAL($kid, (mid, m)$)   // $|mid| = 64, |m| = 288$
1: $ak_{\mathsf{v1}} \leftarrow\!\$ \mathsf{GetKey}(kid)$
2: $\mathcal{M} \leftarrow \mathcal{M} \cup \{(kid, m)\}$
3: $msk, c \leftarrow\!\$ \mathsf{CHv1.Enc}(ak_{\mathsf{v1}}, (mid, m))$
4: return $msk, c$

VFY($kid, (mid, m), (msk, c)$)   // $|mid| = 64, |m| = 288$
1: $ak_{\mathsf{v1}} \leftarrow\!\$ \mathsf{GetKey}(kid)$
2: $out \leftarrow \mathsf{CHv1.Dec}(ak_{\mathsf{v1}}, (msk, c))$
3: if $out = \perp$: return 0
4: $mid', m' \leftarrow out$
5: if $mid' \neq mid \vee m' \neq m$: return 0
6: if $(kid, m) \notin \mathcal{M}$: abort(true)
7: return 1
```

Fig. 51. (Weak) existential unforgeability of CHv1 (Fig. 8) as used in the key exchange.

G.2 KDFv1 is a PRF

Here, we show that KDFv1, the key derivation function of CHv1, is a PRF under a suitable assumption on SHACAL-1. Figure 52 shows the PRF game with an expanded version of KDFv1. We can reduce the PRF-security of KDFv1 to a property of SHACAL-1 that we refer to as the 4PRF security with leakage (Appendix C.1).

```
Game $G^{\mathsf{PRF}}_{\mathsf{KDFv1},\mathcal{D}}$
1:  $b \leftarrow\$ \{0,1\}$
2:  $\mathsf{K} \leftarrow []$
3:  $ak_{\mathsf{v1}} \leftarrow\$ \{0,1\}^{1024}$
4:  $b' \leftarrow\$ \mathcal{D}^{\mathrm{ROR}}()$
5:  return $b' = b$

$\mathrm{ROR}(msk)$    // $|msk| = 128$
1:  $x_A \leftarrow msk \| ak_{\mathsf{v1}}[0:256] \| \mathtt{pad}$
2:  $x_B \leftarrow ak_{\mathsf{v1}}[256:384] \| msk \| ak_{\mathsf{v1}}[384:512] \| \mathtt{pad}$
3:  $x_C \leftarrow ak_{\mathsf{v1}}[512:768] \| msk \| \mathtt{pad}$
4:  $x_D \leftarrow msk \| ak_{\mathsf{v1}}[768:1024] \| \mathtt{pad}$
5:  $A \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{SHACAL\text{-}1.Ev}(x_A, \mathtt{IV}_{160})$
6:  $B \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{SHACAL\text{-}1.Ev}(x_B, \mathtt{IV}_{160})$
7:  $C \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{SHACAL\text{-}1.Ev}(x_C, \mathtt{IV}_{160})$
8:  $D \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{SHACAL\text{-}1.Ev}(x_D, \mathtt{IV}_{160})$
9:  $y_1 \leftarrow A \| B \| C[32:160] \| D[0:64]$
10: if $\mathsf{K}[msk] = \perp$:
11:     $\mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$
12: $y_0 \leftarrow \mathsf{K}[msk]$
13: return $y_b$
```

Fig. 52. Pseudorandomness of KDFv1, where pad is fixed, known SHA-1 padding for an input of length 384 bits.

Proposition 6. *Let $\mathcal{D}_{\mathsf{PRF}}$ be an adversary against the* PRF*-security of* KDFv1*. Then we can construct an adversary $\mathcal{D}_{\mathsf{4PRF}}$ against the* 4PRF*-security of* SHACAL-1 *(Definition 4) such that*

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KDFv1}}(\mathcal{D}_{\mathsf{PRF}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{4PRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{4PRF}}).$$

Proof. We start by defining two games, $G'_0 = G^{\mathsf{4PRF}}_{\mathsf{SHACAL\text{-}1},\mathcal{D}}$ and G'_1, which replaces the SHACAL-1 calls in the original game with randomly generated values. In G'_1, the adversary can no longer win, as it is asked to distinguish between a random string and a fixed string masked by random values. We can thus construct an adversary $\mathcal{D}_{\mathsf{4PRF}}$ (Fig. 53) against the 3PRF security of SHACAL-1 (Fig. 14) that simulates G'_0 resp. G'_1.

```
Adversary $\mathcal{D}_{\mathsf{4PRF}}$
1:  $b \leftarrow\$ \{0,1\}$
2:  $\mathsf{K} \leftarrow []$
3:  $ak_{\mathsf{v1}} \leftarrow\$ \{0,1\}^{1024}$
4:  $b' \leftarrow\$ \mathcal{D}^{\mathrm{ROR}}_{\mathsf{PRF}}()$
5:  return $b' = b$

$\mathrm{RORSIM}(msk)$    // $|msk| = 128$
1:  $A \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{RoR}(msk, \mathtt{A})$
2:  $B \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{RoR}(msk, \mathtt{B})$
3:  $C \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{RoR}(msk, \mathtt{C})$
4:  $D \leftarrow \mathtt{IV}_{160} \hat{+} \mathsf{RoR}(msk, \mathtt{D})$
5:  $y_1 \leftarrow A \| B \| C[32:160] \| D[0:64]$
6:  if $\mathsf{K}[msk] = \perp$:
7:      $\mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$
8:  $y_0 \leftarrow \mathsf{K}[msk]$
9:  return $y_b$
```

Fig. 53. Adversary $\mathcal{D}_{\mathsf{4PRF}}$.

We have $\mathsf{Adv}^{\mathsf{4PRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{4PRF}}) = \Pr[G'_0] - \Pr[G'_1]$, and hence

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KDFv1}}(\mathcal{D}_{\mathsf{PRF}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{4PRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{4PRF}}).$$

$\square$

G.3 UNPRED: Unpredictability of SEv1 on random keys

Here, we define an intermediate notion that requires SEv1, the symmetric encryption part of CHv1, to have unpredictable outputs when random keys are used. In more detail, in Fig. 54 the adversary is given access to a CHALL oracle, which requires the adversary to submit a message *mid*, *m* together with a ciphertext *c* which upon decryption (with a random key identified by *msk*), produces the given (mid, m) as part of the plaintext. The adversary is also provided an EXPOSE oracle, which allows it to learn selected encryption keys, however it is then disallowed from calling CHALL for these keys.

Definition 18. *Consider the game* $G^{\mathsf{UNPRED}}_{\mathsf{SEv1},\mathcal{A}}$ *in Fig. 54 with the symmetric encryption scheme* SEv1 *defined in Fig. 8, and an adversary* $\mathcal{A}$. *The advantage of* $\mathcal{A}$ *in breaking the* UNPRED*-security of* SEv1 *is defined as* $\mathsf{Adv}^{\mathsf{UNPRED}}_{\mathsf{SEv1}}(\mathcal{A}) := \Pr\left[G^{\mathsf{UNPRED}}_{\mathsf{SEv1},\mathcal{A}}\right]$.

```
Game $G^{\mathsf{UNPRED}}_{\mathsf{SEv1},\mathcal{A}}$
1 : $\mathsf{K}, \mathsf{S} \leftarrow []$
2 : $\mathcal{A}^{\text{EXPOSE},\text{CHALL}}()$
3 : return false

EXPOSE$(msk)$
1 : $\mathsf{S}[msk] \leftarrow$ true
2 : return $\mathsf{K}[msk]$

CHALL$((mid, m), (msk, c))$
1 : if $\neg\mathsf{S}[msk]$ :
2 :   if $\mathsf{K}[msk] = \bot$ :
3 :     $\mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$
4 :   $k, iv \leftarrow \mathsf{K}[msk]$
5 :   $out \leftarrow \mathsf{SEv1.Dec}(k, iv, c)$
6 :   if $out \neq \bot$ :
7 :     $\_, mid', m' \leftarrow out$
8 :     if $mid' = mid \wedge m' = m$ :
9 :       abort(true)
10 : return $\bot$
```

Fig. 54. Unpredictability of SEv1 with respect to chosen inputs.

Proposition 7. *Let* $\mathcal{A}$ *be an adversary against the* UNPRED*-security of* SEv1 *(Definition 18) making* $\mathsf{n}_{\text{CHALL}}$ *queries to its* CHALL *oracle. Then* $\mathsf{Adv}^{\mathsf{UNPRED}}_{\mathsf{SEv1}}(\mathcal{A}) \leq \frac{\mathsf{n}_{\text{CHALL}}}{2^{128}}$.

Proof. We proceed along the lines of the UNPRED proof in [AMPS23, Appendix E.6]. First, we define a game G (Fig. 55) similar to $G^{\mathsf{UNPRED}}_{\mathsf{SEv1},\mathcal{A}}$, however in G the CHALL oracle only decrypts the first two blocks of *c* and does not use the input *m*. Hence, the game G is easier to win than the original game, but is indistinguishable to $\mathcal{A}$ up to that point since CHALL always returns $\bot$. We have

$$\mathsf{Adv}^{\mathsf{UNPRED}}_{\mathsf{SEv1}}(\mathcal{A}) \leq \Pr[\mathrm{G}].$$

Let $s = 00000000 \parallel 00000028$ be a fixed 64-bit string. The winning condition in G can be summarised as requiring the adversary to produce mid, c_1, c_2 such that the following holds:

$$c_0 = \mathsf{AES\text{-}256.Enc}(k, (mid \oplus c_1[0:64]) \parallel (s \oplus c_1[64:128])) \oplus c_2 \oplus \mathsf{AES\text{-}256.Dec}(k, c_1 \oplus p_0),$$

where c_0, p_0 are random 128-bit values generated as part of the IV and unknown to the adversary.

For a given *msk*, every query to CHALL can be seen as a guess for c_0. Calling EXPOSE will reveal c_0 to the adversary, but it is forbidden from calling CHALL on such *msk* afterwards. Over all *msk*s that $\mathcal{A}$ could query CHALL for, we get

$$\Pr[\mathrm{G}] \leq \frac{\mathsf{n}_{\text{CHALL}}}{2^{128}}.$$

```
Game G
1 :  $\mathsf{K}, \mathsf{S} \leftarrow []$
2 :  $\mathcal{A}^{\textsc{Expose},\textsc{Chall}}()$
3 :  return false

Expose($msk$)
1 :  $\mathsf{S}[msk] \leftarrow$ true
2 :  return $\mathsf{K}[msk]$

Chall($(mid, m), (msk, c_1 \parallel c_2)$)
1 :  if $\neg\mathsf{S}[msk]$ :
2 :      if $\mathsf{K}[msk] = \bot$ :
3 :          $\mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$
4 :      $k, iv \leftarrow \mathsf{K}[msk]$ ; $c_0 \parallel p_0 \leftarrow iv$
5 :      $p_1 \leftarrow \mathsf{AES\text{-}256.Dec}(k, c_1 \oplus p_0) \oplus c_0$
6 :      $p_2 \leftarrow \mathsf{AES\text{-}256.Dec}(k, c_2 \oplus p_1) \oplus c_1$
7 :      if $p_2 = mid \parallel 00000000 \parallel 00000028$ :
8 :          abort(true)
9 :  return $\bot$
```

Fig. 55. Game G for the proof of Proposition 7.

G.4 Proof for EUF-CMA of CHv1

Proposition 8. *Let $\mathcal{A}$ be an adversary against the* EUF-CMA*-security of the channel* CHv1 *(Definition 17) with at most $\mathsf{n_{kid}}$ long-term symmetric keys, making at most $\mathsf{n}_{\textsc{Vfy}}$ queries to the* Vfy *oracle. Then there exist adversaries $\mathcal{D}_{\mathsf{4PRF}}$ against the* 4PRF*-security of* SHACAL-1 *(Definition 4) and $\mathcal{A}_{\mathsf{UPCR}}$ against the* UPCR*-security of* Hv1 *(Definition 8) such that*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1}}(\mathcal{A}) \leq \mathsf{n_{kid}} \cdot \left(2 \cdot \mathsf{Adv}^{\mathsf{4PRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{4PRF}}) + \frac{\mathsf{n}_{\textsc{Vfy}}}{2^{128}} + \mathsf{Adv}^{\mathsf{UPCR}}_{\mathsf{Hv1}}(\mathcal{A}_{\mathsf{UPCR}})\right).$$

Proof. We proceed via a sequence of games shown in Figs. 56 and 59.

G$_0$. The game G$_0$ is equivalent to the game $\mathrm{G}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1},\mathcal{A}}$, so we have

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{CHv1}}(\mathcal{A}) = \Pr[\mathrm{G}_0].$$

G$_0$ → G$_1$. The game G$_1$, shown in Fig. 56, only allows the adversary to query Eval and Vfy for a single value of *kid*. Since *kid* only impacts the choice of the key ak_{v1}, we omit it showing it as input to Eval and Vfy oracles. More formally, given an adversary $\mathcal{A}_{\mathsf{multi}}$ playing in G$_0$ and making queries for at most $\mathsf{n_{kid}}$ *kid* values, we can build an adversary $\mathcal{A}_{\mathsf{single}}$ playing in G$_1$ which queries a single *kid* value. $\mathcal{A}_{\mathsf{single}}$ must guess the value of *kid* that will result in a forgery and use its own oracles to answer $\mathcal{A}_{\mathsf{multi}}$'s queries for this *kid*; all other queries can be fully simulated.

Note that the game G$_1$ also places the mid', m' check in Vfy earlier in the execution than in G$_0$, however the resulting output is the same. We have

$$\Pr[\mathrm{G}_0] \leq \mathsf{n_{kid}} \cdot \Pr[\mathrm{G}_1].$$

G$_1$ → G$_2$. The game G$_2$, shown in Fig. 56, replaces the real calls to KDFv1 (Fig. 8) with randomly sampled keys, which are tracked using the table K. Straightforwardly, we construct an adversary $\mathcal{D}_{\mathsf{PRF}}$ (Fig. 57) against the PRF security of KDFv1 that simulates G$_1$ resp. G$_2$ for $\mathcal{A}$. We have

$$\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KDFv1}}(\mathcal{D}_{\mathsf{PRF}}).$$

Using Proposition 6, we have

$$\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{KDFv1}}(\mathcal{D}_{\mathsf{PRF}}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{4PRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{4PRF}}).$$

```
Games G_1–G_4
1 :  $\mathsf{K} \leftarrow [\,]\,;\ \mathsf{S} \leftarrow [\,]$        // $G_3$–$G_4$
2 :  $\mathcal{M} \leftarrow \emptyset$
3 :  $ak_{v1} \leftarrow\$ \{0,1\}^{1024}$        // $G_1$
4 :  $\mathcal{A}^{\textsc{Eval},\textsc{Vfy}}()$
5 :  return false
```

```
Eval(mid, m)    // |mid| = 64, |m| = 288
1 :  $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
2 :  $r \leftarrow\$ \{0,1\}^{128}$
3 :  $msk \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
4 :  $\mathsf{S}[msk] \leftarrow \mathtt{true}$        // $G_3$–$G_4$
5 :  $k, iv \leftarrow \mathsf{KDFv1.Ev}(ak_{v1}, msk)$        // $G_1$
6 :  if $\mathsf{K}[msk] = \bot : \mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$        // $G_2$–$G_4$
7 :  $k, iv \leftarrow \mathsf{K}[msk]$        // $G_2$–$G_4$
8 :  $c \leftarrow\$ \mathsf{SEv1.Enc}(k, iv, (r, mid, m))$
9 :  return $msk, c$
```

```
Vfy((mid, m), (msk, c))    // |mid| = 64, |m| = 288
1 :  $k, iv \leftarrow \mathsf{KDFv1.Ev}(ak_{v1}, msk)$        // $G_1$
2 :  if $\mathsf{K}[msk] = \bot : \mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$        // $G_2$–$G_4$
3 :  $k, iv \leftarrow \mathsf{K}[msk]$        // $G_2$–$G_4$
4 :  $out \leftarrow \mathsf{SEv1.Dec}(k, iv, c)$
5 :  if $out = \bot$ : return 0
6 :  $r, mid', m' \leftarrow out$
7 :  if $mid' \neq mid \vee m' \neq m$ : return 0
8 :  if $\mathsf{S}[msk] = \bot$ :        // $G_3$–$G_4$
9 :      $\mathsf{bad}_0 \leftarrow \mathtt{true}$        // $G_3$–$G_4$
10 :     return 0        // $G_4$
11 :  $msk' \leftarrow \mathsf{Hv1.Ev}(r, mid', m')$
12 :  if $msk' \neq msk$ : return 0
13 :  if $m \notin \mathcal{M}$ : abort(true)
14 :  return 1
```

Fig. 56. Games G_1–G_4. Expanded code of CHv1 is shown in gray.

```
Adversary $\mathcal{D}_{\mathsf{PRF}}$
1 : $\mathcal{M} \leftarrow \emptyset$
2 : $\mathcal{A}^{\textsc{EvalSim},\textsc{VfySim}}()$
3 : return $0$

$\textsc{EvalSim}((mid, m))$   // $|mid| = 64, |m| = 288$
1 : $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
2 : $r \leftarrow\!\!\$ \{0,1\}^{128}$
3 : $msk \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
4 : $k, iv \leftarrow\!\!\$ \mathrm{ROR}(msk)$
5 : $c \leftarrow\!\!\$ \mathsf{SEv1.Enc}(k, iv, (r, mid, m))$
6 : return $msk, c$

$\textsc{VfySim}((mid, m), (msk, c))$   // $|mid| = 64, |m| = 288$
1 : $k, iv \leftarrow\!\!\$ \mathrm{ROR}(msk)$
2 : $out \leftarrow \mathsf{SEv1.Dec}(k, iv, c)$
3 : if $out = \perp$ : return $0$
4 : $r, mid', m' \leftarrow out$
5 : $msk' \leftarrow \mathsf{Hv1.Ev}(r, mid', m')$
6 : if $msk' \neq msk$ : return $0$
7 : if $mid' \neq mid \vee m' \neq m$ : return $0$
8 : if $m \notin \mathcal{M}$ : abort$(1)$
9 : return $1$
```

Fig. 57. Adversary $\mathcal{D}_{\mathsf{PRF}}$.

$G_2 \to G_3$. The game G_3, shown in Fig. 56, introduces a bad_0 event if the adversary queries $\textsc{Vfy}$ with a value of *msk* that has never been output by $\textsc{Eval}$ and *mid*, *m* which are equal to the decrypted *mid'*, *m'*; to track this event, it uses a new table S. Since nothing else in the game changes, we have

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

$G_3 \to G_4$. In the game G_4, shown in Fig. 56, $\textsc{Vfy}$ returns 0 if bad_0 is set. We reduce this to UNPRED-security (Definition 18) a notion that concerns the unpredictability of SEv1 when random keys are used. This proof step is very similar to the transition $\mathbf{G_4} \to \mathbf{G_5}$ in the integrity proof of [AMPS22], which used a comparable notion of unpredictability in the context of the MTProto 2.0 channel.

We build an adversary $\mathcal{A}_{\mathsf{UNPRED}}$ (Fig. 58) that simulates G_4 for $\mathcal{A}$: if the value of *msk* input to the $\textsc{EvalSim}$ and $\textsc{VfySim}$ oracles was previously unseen, $\mathcal{A}_{\mathsf{UNPRED}}$ calls its own oracles $\textsc{Expose}$ and $\textsc{Chall}$, and otherwise it proceeds as in the original game. If $\textsc{EvalSim}$ is called before $\textsc{VfySim}$ on some *msk*, $\textsc{Expose}$ will not produce a key, so the behaviour in this case will also be identical to the original game. If $\textsc{VfySim}$ is called first, $\mathcal{A}_{\mathsf{UNPRED}}$ will either return 0 (which happens when $\textsc{Chall}$ returns $\perp$ because $out = \perp$, $mid' \neq mid$ or $m' \neq m$) or win in its own game (because $\mathcal{A}$ triggers the condition that would have set bad_0 in G_4). We have

$$\Pr[G_3] - \Pr[G_4] \leq \Pr[\mathsf{bad}_0] \leq \mathsf{Adv}^{\mathsf{UNPRED}}_{\mathsf{SEv1}}(\mathcal{A}_{\mathsf{UNPRED}}).$$

Using Proposition 7, we can write

$$\mathsf{Adv}^{\mathsf{UNPRED}}_{\mathsf{SEv1}}(\mathcal{A}_{\mathsf{UNPRED}}) \leq \frac{\mathsf{n}_{\textsc{Vfy}}}{2^{128}},$$

where $\mathsf{n}_{\textsc{Vfy}}$ is the number of queries the adversary makes to its $\textsc{Chall}$ resp. $\textsc{Vfy}$ oracle.

$G_4 \to G_5$. The game G_5, shown in Fig. 59, is functionally equivalent to G_4. However, we introduce a number of syntactic changes. We move the check on $\mathsf{S}[msk]$ to the beginning of the game, since $\mathsf{S}[msk] = \perp$

```
Adversary $\mathcal{A}_{\mathsf{UNPRED}}$
1 : $\mathsf{K}, \mathsf{S} \leftarrow []$
2 : $\mathcal{M} \leftarrow \emptyset$
3 : $\mathcal{A}^{\textsc{EvalSim},\textsc{VfySim}}()$
4 : return

$\textsc{EvalSim}(mid, m)$    // $|mid| = 64, |m| = 288$
1 : $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
2 : $r \leftarrow\!\!\$ \{0,1\}^{128}$
3 : $msk \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
4 : if $\mathsf{S}[msk] = \perp$ :
5 :     $\mathsf{K}[msk] \leftarrow \textsc{Expose}(msk)$
6 : if $\mathsf{K}[msk] = \perp$ :
7 :     $\mathsf{K}[msk] \leftarrow\!\!\$ \{0,1\}^{512}$
8 : $k, iv \leftarrow \mathsf{K}[msk]$
9 : $c \leftarrow\!\!\$ \mathsf{SEv1.Enc}(k, iv, (r, mid, m))$
10 : $\mathsf{S}[msk] \leftarrow \mathtt{true}$
11 : return $msk, c$

$\textsc{VfySim}((mid, m), (msk, c))$    // $|mid| = 64, |m| = 288$
1 : if $\mathsf{S}[msk] = \perp$ :
2 :     $\textsc{Chall}((mid, m), (msk, c))$
3 :     return 0
4 : if $\mathsf{K}[msk] = \perp$ :
5 :     $\mathsf{K}[msk] \leftarrow\!\!\$ \{0,1\}^{512}$
6 : $k, iv \leftarrow \mathsf{K}[msk]$
7 : $out \leftarrow \mathsf{SEv1.Dec}(k, iv, c)$
8 : if $out = \perp$ : return 0
9 : $r, mid', m' \leftarrow out$
10 : if $mid' \neq mid \vee m' \neq m$ : return 0
11 : $msk' \leftarrow \mathsf{Hv1.Ev}(r, mid', m')$
12 : if $msk' \neq msk$ : return 0
13 : if $m \notin \mathcal{M}$ : abort
14 : return 1
```

Fig. 58. Adversary $\mathcal{A}_{\mathsf{UNPRED}}$.

causes the VFY oracle to return 0 regardless of the order it is processed in. We also change the order of the calls to SEv1 and Hv1 such that only the first block of *c* is decrypted before the *msk* check. The *msk* check itself is done using the input *mid*, *m* rather than the decrypted values mid', m' – this is equivalent, since the oracle can only return 1 if these values match. In more detail, instead of decrypting, using mid', m' to check *msk* and then checking $mid', m' = mid, m$, we use *mid*, *m* to check *msk*, then decrypt and check $mid', m' = mid, m$. There is nothing for the adversary gain since all checks must pass, no matter the order. Finally, we set the flag $\mathtt{bad}_1$ if, after the *msk* check, it is the case that $m \notin \mathcal{M}$. We have

$$\Pr[G_5] = \Pr[G_4].$$

$\mathbf{G}_5 \rightarrow \mathbf{G}_6$. The game G_6, shown in Fig. 59, returns $\mathtt{false}$ after setting $\mathtt{bad}_1$. We can reduce the probability that $\mathtt{bad}_1$ will be set to the collision resistance of Hv1 under unpredictable prefixes (Definition 8). We build an adversary $\mathcal{A}_{\mathsf{UPCR}}$ (Fig. 60) that simulates G_6 for $\mathcal{A}$. To generate *msk* during EVALSIM, it calls its own oracle EVAL, which also provides it with a value *r*. It then uses *r* to construct the ciphertext *c* under a key it knows. During VFYSIM, whenever $\mathcal{A}$ would have set $\mathtt{bad}_1$, $\mathcal{A}_{\mathsf{UPCR}}$ can use the decrypted value *r*

```
Games $G_5$–$G_6$
1 : $\mathsf{K}, \mathsf{S} \leftarrow []$
2 : $\mathcal{M} \leftarrow \emptyset$
3 : $\mathcal{A}^{\textsc{Eval},\textsc{Vfy}}()$
4 : return false

$\textsc{Eval}(mid, m)$    // $|mid| = 64, |m| = 288$
1 : $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
2 : $r \leftarrow\$ \{0,1\}^{128}$
3 : $msk \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
4 : $\mathsf{S}[msk] \leftarrow$ true
5 : if $\mathsf{K}[msk] = \bot$ : $\mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$
6 : $k, iv \leftarrow \mathsf{K}[msk]$
7 : $c \leftarrow\$ \mathsf{SEv1.Enc}(k, iv, (r, mid, m))$
8 : return $msk, c$

$\textsc{Vfy}((mid, m), (msk, c))$    // $|mid| = 64, |m| = 288$
1 : if $\mathsf{S}[msk] = \bot$ : return 0
2 : $c_1 \| \ldots \| c_5 \leftarrow c$
3 : $k, iv \leftarrow \mathsf{K}[msk]$
4 : $r, \_, \_ \leftarrow \mathsf{SEv1.Dec}^*(k, iv, c_1)$
5 : $msk' \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
6 : if $msk' \neq msk$ : return 0
7 : if $m \notin \mathcal{M}$ :
8 :     $\mathsf{bad}_1 \leftarrow$ true
9 :     abort(false)                                  // $G_6$
10 : $out \leftarrow \mathsf{SEv1.Dec}(k, iv, c)$
11 : if $out = \bot$ : return 0
12 : $r', mid', m' \leftarrow out$
13 : if $mid' \neq mid \vee m' \neq m$ : return 0
14 : if $m \notin \mathcal{M}$ : abort(true)
15 : return 1
```

Fig. 59. Games G_5–G_6. The lines that are shared by all games but formally differ from G_4 are also highlighted in green. We use SEv1.Dec* as shorthand for executing SEv1.Dec to only decrypt the first ciphertext block (which always succeeds).

as well as the input mid, m as a solution to its own game. This is because to reach this point in the game, $msk = \mathsf{Hv1.Ev}(r, mid, m)$ is such that there was at least one previous call to EVALSIM (and therefore to EVAL) that produced this msk, so $msk \in \mathcal{H}$; but $m \notin \mathcal{M}$, hence the winning condition of the UPCR game is satisfied. We have

$$\Pr[G_5] - \Pr[G_6] \leq \Pr[\mathsf{bad}_1] \leq \mathsf{Adv}^{\mathsf{UPCR}}_{\mathsf{Hv1}}(\mathcal{A}_{\mathsf{UPCR}}).$$

```
Adversary $\mathcal{A}_{\mathsf{UPCR}}$
    $\mathsf{K}, \mathsf{S} \leftarrow []$
    $\mathcal{M} \leftarrow \emptyset$
    $\mathcal{A}^{\text{EVALSIM},\text{VFYSIM}}()$
    return $\perp$
```

```
EVALSIM$(mid, m)$    // $|mid| = 64, |m| = 288$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{m\}$
    $r, msk \leftarrow \text{EVAL}(mid, m)$
    $\mathsf{S}[msk] \leftarrow \mathtt{true}$
    if $\mathsf{K}[msk] = \perp$ : $\mathsf{K}[msk] \leftarrow\$ \{0,1\}^{512}$
    $k, iv \leftarrow \mathsf{K}[msk]$
    $c \leftarrow\$ \mathsf{SEv1.Enc}(k, iv, (r, mid, m))$
    return $msk, c$
```

```
VFYSIM$((mid, m), (msk, c))$    // $|mid| = 64, |m| = 288$
    if $\mathsf{S}[msk] = \perp$ : return 0
    $c_1 \| \ldots \| c_5 \leftarrow c$
    $k, iv \leftarrow \mathsf{K}[msk]$
    $r, \_, \_ \leftarrow \mathsf{SEv1.Dec}^*(k, iv, c_1)$
    $msk' \leftarrow \mathsf{Hv1.Ev}(r, mid, m)$
    if $msk' \neq msk$ : return 0
    if $m \notin \mathcal{M}$ :
        abort$(r, mid, m)$
    $out \leftarrow \mathsf{SEv1.Dec}(k, iv, c)$
    if $out = \perp$ : return 0
    $r', mid', m' \leftarrow out$
    if $mid' \neq mid \vee m' \neq m$ : return 0
    return 1
```

Fig. 60. Adversary $\mathcal{A}_{\mathsf{UPCR}}$.

G_6. In the game G_6, the winning condition in VFY can never be reached by the adversary. We have

$$\Pr[G_6] = 0.$$

H Main proofs

H.1 Proof for the two-stage protocol

Here, we provide a proof for Theorem 1.

Proof. We proceed via a sequence of games. In the whole proof, we do not explicitly show the TL schema wrappers that format every message on the byte level unless it is relevant. For brevity, we shorten certain stage 1 messages to only refer to the cryptographically relevant variables, e.g. we write $n, n_s, ...$ for $n, n_s, prod', \mathcal{F}$. When we describe modified SEND queries, we assume that the behaviour still follows the prescribed order of messages in $\mathsf{MTP\text{-}KE_{2st}}$ (which the TL schema encodes and enables to check), i.e. processing out-of-order messages results in the receiver session rejecting.

To simplify the presentation, in all of the games we omit the predicate Auth, since $\mathsf{MTP\text{-}KE_{2st}}$ does not have non-testable stages. We also remove the NEWSECRET oracle, as $\mathsf{MTP\text{-}KE_{2st}}$ does not use long-term symmetric keys.

$\mathbf{G}_0$. The game G_0 is equivalent to the game $\mathrm{G}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MTP\text{-}KE_{2st}},\mathcal{U}_{\mathsf{role}},\mathcal{A}}$, so we have

$$\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MTP\text{-}KE_{2st}},\mathcal{U}_{\mathsf{role}}}(\mathcal{A}) = \mathsf{Adv}^{\mathrm{G}_0}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) = 2 \cdot \Pr[\mathrm{G}_0] - 1.$$

From now on, we omit displaying $\mathcal{U}_{\mathsf{role}}$ in the advantage terms.

$\mathbf{G}_0 \to \mathbf{G}_1$. The game G_1 is amended to set the bad_0 flag and return 0 if there are two honest initiator sessions that collide on n and n_n, as shown in Fig. 61. By Lemma 1, we have

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

By the birthday bound, we have

$$\Pr[\mathsf{bad}_0] \leq \frac{\mathsf{n}_\mathsf{S}^2}{2 \cdot 2^{128+256}}$$

and hence

$$\mathsf{Adv}^{\mathrm{G}_0}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \frac{\mathsf{n}_\mathsf{S}^2}{2^{384}} + \mathsf{Adv}^{\mathrm{G}_1}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

To argue that in G_1, the predicate Sound (as defined in Fig. 4) is always satisfied, we need to show the following properties:

1. *At most two sessions can be partnered.*
   First, note that the public key pk is part of $\mathsf{sid}.1$, which means that different responders have different session identifiers by default. However, the matching of session identifiers could be impacted by collisions in the initiator-chosen nonces n, n_n. The nonce n is part of $\mathsf{sid}.1$ in plaintext, but n_n appears in encrypted form. We have that if $n_n \neq n'_n$, then the corresponding $c_0 \neq c'_0$ by the correctness of TOAEP^+ (for honestly produced c_0 under the same public key pk). Further, since $\mathsf{sid}.1$ is a prefix of $\mathsf{sid}.2$, a collision in stage 2 implies a collision in stage 1. The game G_1 returns 0 if there is a collision in n, n_n. Without such a collision, three sessions cannot accept the same sid in any stage.

   The remaining soundness properties thus assume that "partnered sessions" always mean only a single pair.

2. *Partnered sessions must have different roles.*
   The TL schema defines different message headers for each protocol message, thereby ensuring that there can be no role confusion.

3. *Session identifiers cannot match across different stages.*
   This holds trivially as each stage adds at least one message to the identifier.

```
G_0–G_1
$\mathcal{L}_\mathsf{K}, \mathcal{K}_\mathsf{pub} \leftarrow \mathsf{Init}(\mathcal{U}_\mathsf{role})$
$\mathcal{L}_\mathsf{S} \leftarrow [\,]\,;\ \mathcal{C}_\mathsf{pub} \leftarrow \emptyset$
$b_\mathsf{test} \leftarrow\!\!\$\ \{0,1\}$
$lost \leftarrow \mathtt{false}$
$b'_\mathsf{test} \leftarrow \mathcal{A}^{\textsc{NewSession},\ldots,\textsc{Test}}(\mathcal{K}_\mathsf{pub})$
$\mathbf{if}\ \exists s, s' \in \mathcal{L}_\mathsf{S}, n \in \{0,1\}^{128}, n_n \in \{0,1\}^{256} : (s \neq s'$
    $\wedge\ s.\mathsf{uid.role} = s'.\mathsf{uid.role} = \mathtt{I} \wedge s.\mathsf{sskey}.1 = s'.\mathsf{sskey}.1 = n_n$
    $\wedge\ s.\mathsf{sid}.1 = (\_, n, \_, \_) \wedge s'.\mathsf{sid}.1 = (\_, n, \_, \_)) :$
      $\mathsf{bad}_0 \leftarrow \mathtt{true}$
      $\mathbf{return}\ 0$    // G_1
$\mathbf{if}\ \neg\mathsf{Sound} :$
    $\mathbf{return}\ 1$
$\mathbf{if}\ \neg\mathsf{Fresh} :$
    $lost \leftarrow \mathtt{true}$
$\mathbf{return}\ b'_\mathsf{test} = b_\mathsf{test} \wedge lost = \mathtt{false}$
```

Fig. 61. Games G_0–G_1 for the proof of Multi-Stage-security of $\mathsf{MTP\text{-}KE_{2st}}$.

4. *Partnered sessions must agree on contributive identifiers.*
   $\mathsf{MTP\text{-}KE_{2st}}$ always sets $s.cid.i = s.sid.i$ upon acceptance.

5. *Partnered sessions must output the same session key.*
   In the first stage, the session id includes the public key pk of the responder, the unencrypted nonces n, n_s exchanged between the client and the server and the encrypted ciphertext c_0 which carries n_n. An honest session of a user U such that $U.\mathsf{role} = \mathtt{I}$ will only include c_0 in its session id if it had produced it as $c_0 \leftarrow \mathsf{TOAEP}^+.\mathsf{Enc}(pk, m_0)$ where m_0 includes n, n_s and n_n. A partnered session owned by a user V such that $V.\mathsf{role} = \mathtt{R}$ will only include c_0 in its session id if it can decrypt it to recover n and n_s. By the correctness of TOAEP^+, this means that V will also recover the correct n_n.

   In the second stage, the session key is fully determined by the values $g^a \bmod p$ and $g^b \bmod p$.

   Therefore, in both stages, the session id determines all the inputs from which the session key is derived.

6. *Responder-only authentication.*
   Let s, s' denote the partnered sessions, with $s.\mathsf{uid} = U$, $s'.\mathsf{uid} = V$. We want to show that if $s.\mathsf{role} = \mathtt{I}$ and $s'.\mathsf{role} = \mathtt{R}$, then U had set its partner identity correctly as $s.\mathsf{vid} = V$. This is ensured via the inclusion of pk in $s.\mathsf{sid} = s'.\mathsf{sid}$, as it is associated with V during the key setup for each session.

Thus, in the games that follow, we may assume that $\mathsf{Sound} = \mathtt{true}$.

$\mathbf{G}_1 \rightarrow \mathbf{G}_2$. The game G_2 differs from G_1 in that it only allows the adversary to submit a single Test query. We can build a reduction from the adversary $\mathcal{A}_\mathsf{multi}$ making at most $\mathsf{n_T} \leq \mathsf{M} \cdot \mathsf{n_S} = 2\mathsf{n_S}$ Test queries in G_1 to an adversary $\mathcal{A}_\mathsf{single}$ that makes a single query in G_2. $\mathcal{A}_\mathsf{single}$ first guesses a value $t \in \{1, \ldots, \mathsf{n_T}\}$, and then answers the first $t-1$ Test queries of $\mathcal{A}_\mathsf{multi}$ by calling Reveal, the t-th Test query by calling its own Test oracle and the remaining queries by sampling a random session key.

For stage 1 queries, $\mathcal{A}_\mathsf{single}$ can do this simulation perfectly since $sid.1$ can be computed from the transcript of each accepting session and so $\mathcal{A}_\mathsf{single}$ can keep track of which sessions are partnered, ensuring consistent outputs for queries for partnered sessions. In more detail, this means that $\mathcal{A}_\mathsf{single}$ will answer any Test query for a session whose partner was already tested with $\perp$, keep track of which sessions are considered revealed as well as ensure that the stage 1 session keys are set consistently for sessions whose partners were tested (since we have $\mathsf{USE}.1 = \texttt{internal}$).

For stage 2 queries, *sid*.2 contains the plaintext values $g^a \bmod p$, $g^b \bmod p$, so $\mathcal{A}_{\mathsf{single}}$ must first obtain the stage 1 session key before it can match partnered sessions. First, observe that since *sid*.1 is contained within *sid*.2, those sessions that were not partnered in stage 1 will not be partnered in stage 2, so we only need to consider the case where we know that the sessions in question agree on n, n_s and n_n. Then, suppose that $\mathcal{A}_{\mathsf{multi}}$ tests a given session with the label *label* at stage 2. If it tests or reveals the same session at stage 1, such a query would mark the stage 2 session key as revealed due to key dependence, causing the *lost* flag to be set to `true` at the end of the execution, which $\mathcal{A}_{\mathsf{single}}$ can simulate as described below. We can now consider the other case, i.e. $\mathcal{A}_{\mathsf{multi}}$ does not test or reveal the same session at stage 1. Before answering a SEND query for *label* that would set st_{exec} to $\texttt{accepted}_2$ (which must happen before the session can be tested at stage 2), $\mathcal{A}_{\mathsf{single}}$ can call $\text{REVEAL}(label, 1)$ in its own game to obtain n_n and decrypt the ciphertexts containing the values $g^a \bmod p$, $g^b \bmod p$ needed to determine *sid*.2. The REVEAL query does not introduce any differences from the simulated game, enabling $\mathcal{A}_{\mathsf{single}}$ to determine partnered sessions at the time when each session accepts.

Finally, to avoid a mismatch arising if $\mathcal{A}_{\mathsf{multi}}$ causes the *lost* flag to be set `true`, $\mathcal{A}_{\mathsf{single}}$ makes a REVEAL query for the particular session that would have set the flag in G_1 (either because of the conditions within the simulated TEST queries when $j \neq t$ or the check at the end of the game which searches for sessions where the same key was tested and revealed).

Following [DFGS21, Appendix A], we get

$$2 \cdot \Pr[G_2] - 1 \geq \frac{1}{\mathsf{n_T}} \cdot (2 \cdot \Pr[G_1] - 1)$$

and hence

$$\mathsf{Adv}^{G_1}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}_{\mathsf{multi}}) \leq 2\mathsf{n_S} \cdot \mathsf{Adv}^{G_2}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}_{\mathsf{single}}).$$

From now on, we will refer to the tested session s_{test}, and to the adversary $\mathcal{A}_{\mathsf{single}}$ as $\mathcal{A}$.

Recall that regardless of the tested stage, we aim for responder-only authentication, but forward secrecy is only expected from stage 2. We now split our analysis depending on whether the adversary tests s_{test} during stage 1 or stage 2. These are disjoint events, hence:

$$\mathsf{Adv}^{G_2}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \mathsf{Adv}^{G_2,\text{stage 1 tested}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) + \mathsf{Adv}^{G_2,\text{stage 2 tested}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

Stage 1. We can assume that the session s_{test} is owned by either an honest initiator whose intended partner is honest or an honest responder partnered with an honest initiator after stage 1. This is because there is no forward secrecy for this stage, so we only need to consider honest intended partners and honest responders, i.e. if the public-key keypair used by s_{test} is corrupted at any point, $s_{\mathsf{test}}.\mathsf{sskey}.1$ is always considered revealed. We further split our analysis into two cases: Case A in which s_{test} is not partnered with any session at stage 1 (and therefore $s_{\mathsf{test}}.\mathsf{role} = \texttt{I}$, since testing a responder without a partner sets the *lost* flag to `true`), and Case B in which s_{test} does have a stage 1 partner. The games $G_{A.0}$ and $G_{B.0}$ are otherwise equivalent to G_2 where $\mathcal{A}$ tests stage 1. We have:

$$\mathsf{Adv}^{G_2,\text{stage 1 tested}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \mathsf{Adv}^{G_{A.0}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) + \mathsf{Adv}^{G_{B.0}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

Case A**: Honest initiator with no partner**. First, note that initiators may accept at stage 1 without there being a corresponding partnered session because $\mathsf{MTP\text{-}KE_{2st}}$ only provides implicit authentication at stage 1. However, we may assume that s_{test}'s intended partner identified by its public key is honest.

$\mathbf{G}_{A.0}$. The game $G_{A.0}$ is equivalent to the game G_2 where $\mathcal{A}$ tests stage 1 and s_{test} is an honest initiator session without a partner.

$\mathbf{G}_{\mathrm{A}.0} \to \mathbf{G}_{\mathrm{A}.1}$. Let $\mathsf{n_{pk}}$ be the number of public-key keypairs generated in the game, which is equivalent to the number of users with the role of a responder. Consider the game $\mathrm{G}_{\mathrm{A}.1}$, which is equivalent to $\mathrm{G}_{\mathrm{A}.0}$ except that at the beginning, the game guesses which of the $\mathsf{n_{pk}}$ potential responders will be chosen in s_{test} as intended partner and aborts when the guess is wrong. That is, if the guessed user is V_{test}, the game aborts when $s_{\mathsf{test}}.\mathsf{vid} \neq V_{\mathsf{test}}$.

We have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{A}.0}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \mathsf{n_{pk}} \cdot \mathsf{Adv}^{\mathrm{G}_{\mathrm{A}.1}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

$\mathbf{G}_{\mathrm{A}.1} \to \mathbf{G}_{\mathrm{A}.2}$. Let the game $\mathrm{G}_{\mathrm{A}.2}$ be as the game $\mathrm{G}_{\mathrm{A}.1}$ with the following difference in the handling of SEND queries during stage 1: $\textsc{Send}(s_{\mathsf{test}}.\mathsf{label}, (n, n_s, ...))$ for some n, n_s sets a random session key $s_{\mathsf{test}}.\mathsf{sskey}.1 \leftarrow\$ \{0,1\}^{256}$ instead of using n_n.

Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE_{2st}}$, we build an adversary $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ against the IND-CCA-security of TOAEP^+ that simulates $\mathrm{G}_{\mathrm{A}.1}$ or $\mathrm{G}_{\mathrm{A}.2}$ depending on the challenge bit b_{chall} in its game. $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ samples a random bit $b_{\mathsf{test}} \leftarrow\$ \{0,1\}$ at the start and inserts the public key pk^* given to it in the IND-CCA game as the public key of V_{test}. It otherwise generates all key pairs as normal for the Multi-Stage-security game. Thereafter, the adversary only modifies the SEND oracle as follows:

– Upon receiving $\textsc{Send}(s_{\mathsf{test}}.\mathsf{label}, (n, n_s, ...))$ for some n, n_s, instead of running $\mathsf{MTP\text{-}KE_{2st}}$ the adversary $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ samples two independent values $n_n \leftarrow\$ \{0,1\}^{256}$, $r \leftarrow\$ \{0,1\}^{256}$ and computes

$$m_0^0 \leftarrow prod', p', q', n, n_s, n_n, dc$$
$$m_0^1 \leftarrow prod', p', q', n, n_s, r, dc$$

  for suitable values of p', q', dc according to $\mathsf{MTP\text{-}KE_{2st}}$.

  Then, $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ computes $c_0^* \leftarrow \textsc{Enc}(m_0^0, m_0^1)$ using its own oracle. It sets $s.\mathsf{sskey}.1 \leftarrow n_n$ and continues as in the usual Multi-Stage game, returning c_0^*, $\texttt{running}_1$ as output to $\mathcal{A}$.

– Upon receiving $\textsc{Send}(s.\mathsf{label}, (n', n_s', ..., c_0))$ for a session $s \neq s_{\mathsf{test}}$ such that $s.\mathsf{cid}.1 = (pk^*, n', n_s')$ for some n', n_s', it computes $m_0 \leftarrow \textsc{Dec}(c_0)$, and if $m_0 \neq \perp$ and n', n_s' are included in m_0, it sets $s.\mathsf{sskey}.1$ as the correct substring of m_0 and returns $\texttt{paused}$ to $\mathcal{A}$. Otherwise, it returns $\perp$, $\texttt{rejected}_1$ to $\mathcal{A}$. Note that if $c_0 = c_0^*$, we will necessarily get $m_0 = \perp$, but since s_{test} has no partner, this means that at least one of the n, n_s values in $s.\mathsf{cid}.1 = (pk^*, n, n_s)$ differs from n', n_s' and so $\mathsf{MTP\text{-}KE_{2st}}$ would have also output $\perp$ in this case.

$\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ outputs its own guess as $b'_{\mathsf{chall}} \leftarrow b'_{\mathsf{test}} = b_{\mathsf{test}} \wedge lost = \texttt{false}$ where b'_{test} is the bit guess of $\mathcal{A}_{\mathsf{KE}}$ and the *lost* flag is set under the same conditions as in both games $\mathrm{G}_{\mathrm{A}.1}, \mathrm{G}_{\mathrm{A}.2}$.

If the challenge bit $b_{\mathsf{chall}} = 0$ in the IND-CCA game, then the tested session will produce $c_0 = \mathsf{TOAEP}^+.\mathsf{Enc}(pk^*, m_0^1)$ (where m_0^0 contains n_n) just as in the game $\mathrm{G}_{\mathrm{A}.1}$. Otherwise, if $b_{\mathsf{chall}} = 1$, then the ciphertext $c_0 = \mathsf{TOAEP}^+.\mathsf{Enc}(pk^*, m_0^1)$ will be independent of the stage 1 session key n_n, capturing the behaviour of the game $\mathrm{G}_{\mathrm{A}.2}$. We have $\Pr\left[b'_{\mathsf{chall}} = 1 \mid b_{\mathsf{chall}} = 1\right] = \Pr[\mathrm{G}_{\mathrm{A}.1}]$ and $\Pr\left[b'_{\mathsf{chall}} = 1 \mid b_{\mathsf{chall}} = 0\right] = \Pr[\mathrm{G}_{\mathrm{A}.2}]$, hence

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{A}.1}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{G}_{\mathrm{A}.2}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{TOAEP}^+}(\mathcal{A}_{\mathsf{IND\text{-}CCA}}).$$

$\mathbf{G}_{\mathrm{A}.2}$. Observe that if the adversary makes a $\textsc{Test}(s_{\mathsf{test}}, 1)$ query in $\mathrm{G}_{\mathrm{A}.2}$, it will always receive a random key irrespective of the challenge bit b_{test}. Since the agreed session key is now independent of the protocol messages, the adversary can only win by guessing and we have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{A}.2}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) = 0.$$

Case B**: Partner of tested session exists**. We now consider the case that the tested session does have a partner at stage 1.

$\mathbf{G}_{\mathrm{B}.0}$. The game $\mathrm{G}_{\mathrm{B}.0}$ is equivalent to the game G_2 where $\mathcal{A}$ tests stage 1 and s_{test} is an honest session with a partner.

$\mathbf{G}_{\mathrm{B}.0} \to \mathbf{G}_{\mathrm{B}.1}$. Consider the game $\mathrm{G}_{\mathrm{B}.1}$, which is equivalent to $\mathrm{G}_{\mathrm{B}.0}$ except that at the beginning, the game guesses which of the n_{pk} potential responders will s_{test} choose as intended partner, as well as which of the n_{S} sessions will be partnered with s_{test} after stage 1 and aborts when the guess is wrong. That is, if the guessed user is V_{test} and the guessed session is s'_{test}, the game aborts if $s_{\mathsf{test}}.\mathsf{vid} \neq V_{\mathsf{test}}$ or $s'_{\mathsf{test}}.\mathsf{sid}.1 \neq s_{\mathsf{test}}.\mathsf{sid}.1$.

We have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{B}.0}}_{\mathsf{MTP\text{-}KE}_{2\mathsf{st}}}(\mathcal{A}) \leq \mathsf{n}_{\mathsf{pk}} \cdot \mathsf{n}_{\mathsf{S}} \cdot \mathsf{Adv}^{\mathrm{G}_{\mathrm{B}.1}}_{\mathsf{MTP\text{-}KE}_{2\mathsf{st}}}(\mathcal{A}).$$

Note that in this game, s_{test} and s'_{test} may be either an initiator and a responder or vice versa (correct roles are ensured by the soundness predicate). In the pair $(s_{\mathsf{test}}, s'_{\mathsf{test}})$, denote the initiator by $s_{\mathtt{I}}$ and the responder by $s_{\mathtt{R}}$.

$\mathbf{G}_{\mathrm{B}.1} \to \mathbf{G}_{\mathrm{B}.2}$. Let the game $\mathrm{G}_{\mathrm{B}.2}$ be as the game $\mathrm{G}_{\mathrm{B}.1}$ with the following difference in the handling of SEND queries during stage 1: $\mathrm{SEND}(s_{\mathtt{I}}.\mathsf{label}, (n, n_s, ...))$ (for some n, n_s) sets a random session key $s_{\mathtt{I}}.\mathsf{sskey}.1 \leftarrow\!\!\$ \{0,1\}^{256}$ instead of using the generated n_n. For consistency, the game ensures that $s_{\mathtt{R}}.\mathsf{sskey}.1 \leftarrow s_{\mathtt{I}}.\mathsf{sskey}.1$.

Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$, we build an adversary $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ against the IND-CCA of TOAEP^+ that simulates $\mathrm{G}_{\mathrm{B}.1}$ or $\mathrm{G}_{\mathrm{B}.2}$ depending on the challenge bit in its game. $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ inserts the public key pk^* given to it in the IND-CCA game as the public key of $s_{\mathtt{R}}.\mathsf{uid}$. Thereafter, the adversary only modifies the SEND oracle as follows:

– Upon receiving a query $\mathrm{SEND}(s_{\mathtt{I}}.\mathsf{label}, (n, n_s, ...))$ for n such that $s_{\mathtt{I}}.\mathsf{cid}.1 = (pk^*, n)$ and some n_s, instead of running $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$ the adversary $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ samples two independent values $n_n \leftarrow\!\!\$ \{0,1\}^{256}$, $r \leftarrow\!\!\$ \{0,1\}^{256}$ and computes

$$m_0^0 \leftarrow prod', p', q', n, n_s, n_n, dc$$
$$m_0^1 \leftarrow prod', p', q', n, n_s, r, dc$$

  for suitable values of p', q', dc according to $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$.

  Then, $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ computes $c_0^* \leftarrow \mathrm{ENC}(m_0^0, m_0^1)$ using its own oracle. It sets $s_{\mathtt{I}}.\mathsf{sskey}.1 \leftarrow n_n$ and continues as in the usual Multi-Stage game, returning c_0^* as part of its output to $\mathcal{A}$.

– Upon receiving $\mathrm{SEND}(s_{\mathtt{R}}.\mathsf{label}, (n, n_s, ..., c_0^*))$ for n, n_s such that $s_{\mathtt{R}}.\mathsf{cid}.1 = (pk^*, n, n_s)$, the adversary sets $s_{\mathtt{R}}.\mathsf{sskey}.1 \leftarrow n_n$ from the previous modified SEND query for $s_{\mathtt{I}}$ and returns `paused` to $\mathcal{A}$ (if there was no such query, it returns $\perp$, $\mathtt{rejected}_1$ just like $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$).

– Upon receiving $\mathrm{SEND}(s.\mathsf{label}, (n', n'_s, ..., c_0))$ for a session $s = s_{\mathtt{R}}$ with $c_0 \neq c_0^*$ or a session $s \neq s_{\mathtt{R}}$ such that $s.\mathsf{cid}.1 = (pk^*, n', n'_s)$, it computes $m_0 \leftarrow \mathrm{DEC}(c_0)$, sets $s.\mathsf{sskey}.1$ as the correct substring of m_0 and returns `paused` to $\mathcal{A}$ (if $m_0 = \perp$ or n', n'_s are not included in m_0, it returns $\perp$, $\mathtt{rejected}_1$).

$\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ outputs its own guess as $b'_{\mathsf{chall}} \leftarrow b'_{\mathsf{test}} = b_{\mathsf{test}} \wedge lost = \mathtt{false}$ where b'_{test} is the bit guess of $\mathcal{A}_{\mathsf{KE}}$ and the *lost* flag is set under the same conditions as in both games $\mathrm{G}_{\mathrm{B}.1}, \mathrm{G}_{\mathrm{B}.2}$.

If the challenge bit $b_{\mathsf{chall}} = 0$ in the IND-CCA game, then the tested session will exchange $c_0 = \mathsf{TOAEP}^+.\mathsf{Enc}(pk^*, m_0^0)$ (where m_0^0 contains n_n) with its partnered session just as in the game $\mathrm{G}_{\mathrm{B}.1}$. Otherwise, if $b_{\mathsf{chall}} = 1$, then the ciphertext $c_0 = \mathsf{TOAEP}^+.\mathsf{Enc}(pk^*, m_0^1)$ will be independent of the stage 1 session key n_n, capturing the behaviour of the game $\mathrm{G}_{\mathrm{B}.2}$. We have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{B}.1}}_{\mathsf{MTP\text{-}KE}_{2\mathsf{st}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{G}_{\mathrm{B}.2}}_{\mathsf{MTP\text{-}KE}_{2\mathsf{st}}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{TOAEP}^+}(\mathcal{A}_{\mathsf{IND\text{-}CCA}}).$$

$\mathbf{G}_{\mathrm{B}.2}$. Observe that if the adversary makes a $\mathrm{TEST}(s_{\mathsf{test}}, 1)$ query in $\mathrm{G}_{\mathrm{B}.2}$, it will always receive a random key irrespective of the challenge bit b_{test}, so that it can only win by guessing and we have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{B}.2}}_{\mathsf{MTP\text{-}KE}_{2\mathsf{st}}}(\mathcal{A}) = 0.$$

Stage 2. Consider the game G_2 where the adversary makes a $\textsc{Test}(s_{\mathsf{test}}, 2)$ query. We consider forward secrecy, so the adversary is allowed to make a $\textsc{Corrupt}$ query for the tested session s_{test} or its partner after $s_{\mathsf{test}}.\mathsf{st}_{\mathsf{exec}} = \mathtt{accepted}_2$ is set (the $\textsc{Corrupt}$ query can be issued before or after the $\textsc{Test}$ query). We can assume that s_{test} is owned either by an honest initiator whose intended partner may only be corrupted after s_{test} completes stage 2, or by a responder who may only be corrupted after completing stage 2 and who is partnered with an honest initiator.

We will again consider two cases: Case C in which $s_{\mathsf{test}}.\mathsf{uid.role} = \mathtt{I}$ and s_{test} does not have a contributive partner established during stage 1 (in particular, there does not exist a session $s'_{\mathsf{test}} \neq s_{\mathsf{test}}$ such that $s'_{\mathsf{test}}.\mathsf{cid}.1 = s_{\mathsf{test}}.\mathsf{cid}.1 = (pk, n, n_s)$ for some pk, n, n_s), and Case D in which s_{test} does have a contributive partner. The games $G_{C.0}$ and $G_{D.0}$ are otherwise equivalent to G_2 where $\mathcal{A}$ tests stage 1.

$$\mathsf{Adv}^{G_2,\text{stage 2 tested}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \mathsf{Adv}^{G_{C.0}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) + \mathsf{Adv}^{G_{D.0}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

Case C: Honest initiator with no partner.

$\mathbf{G}_{C.0}$. This game is equivalent to G_2 where $\mathcal{A}$ tests stage 2 and s_{test} is an honest initiator session without a contributive partner in stage 1.

$\mathbf{G}_{C.0} \to \mathbf{G}_{C.1}$. Similarly to $G_{A.1}$, take the game $G_{C.1}$, which is equivalent to $G_{C.0}$ except that at the beginning, the game guesses which of the $\mathsf{n_{pk}}$ potential responders will s_{test} choose as intended partner and aborts when the guess is wrong.

We have

$$\mathsf{Adv}^{G_{C.0}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \mathsf{n_{pk}} \cdot \mathsf{Adv}^{G_{C.1}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

Let V_{test} be the intended partner, i.e. $s_{\mathsf{test}}.\mathsf{vid} = V_{\mathsf{test}}$.

$\mathbf{G}_{C.1} \to \mathbf{G}_{C.2}$. Next, consider the game $G_{C.2}$, which is equivalent to $G_{C.1}$ except that the game sets the flag bad_1 to $\mathtt{true}$ and aborts if s_{test} recovers $m_1 \neq \perp$ such that n, n_s from $s_{\mathsf{test}}.\mathsf{sid}.1$ appear in m_1 as $m_1[32:160]$, $m_1[160:288]$ respectively. We have

$$\Pr[G_{C.1}] - \Pr[G_{C.2}] \leq \Pr\left[\mathsf{bad}_1^{G_{C.2}}\right].$$

IND-CCA. To bound $\Pr\left[\mathsf{bad}_1^{G_{C.2}}\right]$, we first use an additional game hop to a game $G_{C.2^*}$. This intermediate game differs from $G_{C.2}$ in that $\textsc{Send}(s_{\mathsf{test}}.\mathsf{label}, (n, n_s, ...))$ (for some n, n_s) sets a random session key $s_{\mathsf{test}}.\mathsf{sskey}.1 \leftarrow\!\$ \{0,1\}^{256}$ instead of using n_n.

Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE_{2st}}$, we build an adversary $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ against the IND-CCA of TOAEP^+ that simulates $G_{C.2}$ or $G_{C.2^*}$ depending on the challenge bit in its game. The construction is the same as in Case A, in particular as in the transition between the games $G_{A.1}$ and $G_{A.2}$.

In contrast to the above transition, however, here the simulation is not perfect: if $\mathcal{A}$ makes a $\textsc{Corrupt}$ query for V_{test}, $\mathcal{A}_{\mathsf{IND\text{-}CCA}}$ cannot reveal the private key corresponding to the challenge public key pk^* as it does not possess it. However, we only consider what happens before the flag bad_1 is set and in this time frame, the adversary cannot issue a $\textsc{Corrupt}$ query without eventually losing.

We have

$$\Pr\left[\mathsf{bad}_1^{G_{C.2}}\right] \leq \Pr\left[\mathsf{bad}_1^{G_{C.2^*}}\right] + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{TOAEP}^+}(\mathcal{A}_{\mathsf{IND\text{-}CCA}}).$$

Integrity of plaintexts. Starting from $G_{C.2^*}$, we can bound the probability of abort due to bad_1 using the integrity of plaintexts of $\mathsf{HtE\text{-}SE}$. Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE_{2st}}$ in $G_{C.2^*}$, we build an adversary $\mathcal{A}_{\mathsf{INT\text{-}PTXT}} = (\mathcal{A}_1, \mathcal{A}_2)$ that plays in the INT-PTXT game (Fig. 39) with $\mathsf{HtE\text{-}SE}$, SKDF and $\mathsf{SAMP}[\cdot, \cdot, g, p]$.

$\mathcal{A}_1(n)$ starts simulating the game $\mathrm{G}_{\mathrm{C}.2^*}$ for $\mathcal{A}$. During stage 1, it makes s_{test} set the nonce n it was given as input instead of generating its own. If $\mathcal{A}$ makes a $\textsc{Send}(s_{\mathsf{test}}.\mathsf{label}, (n, n_s^*, ...))$ query for some n_s^*, $\mathcal{A}_1$ responds as in $\mathrm{G}_{\mathrm{C}.2^*}$, saves the state of the simulation as st and returns n_s^*. Since s_{test} does not have a contributive partner who shares n_s^* with them, it must have been produced by a session controlled by $\mathcal{A}$.

$\mathcal{A}_2(st)$ continues the simulation for $\mathcal{A}$. On receiving $\textsc{Send}(s_{\mathsf{test}}.\mathsf{label}, (n, n_s^*, ..., c_1))$, $\mathcal{A}_2$ calls its own oracle via $\textsc{Dec}(c_1)$ to obtain m_1. If $m_1 \neq \perp$, $\mathcal{A}_2$ is aborted by its own game, immediately winning since it has not made any $\textsc{Enc}$ queries. Note that the $\textsc{Dec}$ oracle will never return $m_1 = \perp$, as in that case it aborts the adversary who loses the game at that point.

The oracles of the INT-PTXT game use an encryption key derived from a fresh value independent of the stage 1 key n_n, matching the behaviour of $\mathrm{G}_{\mathrm{C}.2^*}$. In any case, $\mathcal{A}$ cannot learn the real n_n and notice the discrepancy before stage 2 ends since a) issuing a $\textsc{Reveal}$ query for stage 1 would invalidate it from testing stage 2 due to key dependence, and b) $\textsc{Corrupt}$ queries are only permitted once the protocol accepts stage 2, but the abort condition that we are analysing must happen before such acceptance as it is a result of a $\textsc{Dec}$ query by $\mathcal{A}_{\mathsf{INT\text{-}PTXT}}$. Further, though the real n_n would also be used later in the protocol to compute h, the execution never reaches that point. Thus, whenever $\mathcal{A}$ sets the bad_1 flag and thus triggers an abort, $\mathcal{A}_{\mathsf{INT\text{-}PTXT}}$ wins in its own game. We have

$$\Pr\left[\mathsf{bad}_1^{\mathrm{G}_{\mathrm{C}.2^*}}\right] \leq \mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p}(\mathcal{A}_{\mathsf{INT\text{-}PTXT}}),$$

hence we can write

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{C}.1}}_{\mathsf{MTP\text{-}KE}_{\mathsf{2st}}}(\mathcal{A}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{TOAEP}^+}(\mathcal{A}_{\mathsf{IND\text{-}CCA}}) + 2 \cdot \mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p}(\mathcal{A}_{\mathsf{INT\text{-}PTXT}}) + \mathsf{Adv}^{\mathrm{G}_{\mathrm{C}.2}}_{\mathsf{MTP\text{-}KE}_{\mathsf{2st}}}(\mathcal{A})$$

$\mathrm{G}_{\mathrm{C}.2}$. Notice that in $\mathrm{G}_{\mathrm{C}.2}$, $\mathcal{A}$ cannot win anymore: either the game aborts, or s_{test} outputs $m_1 = \perp$ and $\texttt{rejected}_2$. In either case, $\mathcal{A}$ cannot issue its $\textsc{Test}$ query anymore, so we have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{C}.2}}_{\mathsf{MTP\text{-}KE}_{\mathsf{2st}}}(\mathcal{A}) = 0.$$

Case D**: Contributive partner of tested session exists**.

$\mathrm{G}_{\mathrm{D}.0}$. This game is equivalent to G_2 where $\mathcal{A}$ tests stage 2 and s_{test} does have a contributive partner.

$\mathrm{G}_{\mathrm{D}.0} \to \mathrm{G}_{\mathrm{D}.1}$. Similarly to $\mathrm{G}_{\mathrm{B}.1}$, take the game $\mathrm{G}_{\mathrm{D}.1}$, which is equivalent to $\mathrm{G}_{\mathrm{D}.0}$ except that at the beginning, the game guesses which of the n_{pk} potential responders will s_{test} choose as intended partner, as well as which of the n_{S} sessions will be partnered with s_{test} after stage 1 and aborts when the guess is wrong.

We have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{D}.0}}_{\mathsf{MTP\text{-}KE}_{\mathsf{2st}}}(\mathcal{A}) \leq \mathsf{n}_{\mathsf{pk}} \cdot \mathsf{n}_{\mathsf{S}} \cdot \mathsf{Adv}^{\mathrm{G}_{\mathrm{D}.1}}_{\mathsf{MTP\text{-}KE}_{\mathsf{2st}}}(\mathcal{A}).$$

Let pk, n, n_s be the values included in $s_{\mathsf{test}}.\mathsf{cid}.1$ resp. $s'_{\mathsf{test}}.\mathsf{cid}.1$. In this game, s_{test} and s'_{test} may be either an initiator and a responder or vice versa. In the pair $(s_{\mathsf{test}}, s'_{\mathsf{test}})$, denote the initiator by $s_{\mathtt{I}}$ and the responder by $s_{\mathtt{R}}$.

Next, we consider the possibility that the protocol will involve a number of "retries". A retry happens when the computed auth key identifier aid is not unique among the identifiers of other accepted sessions of $s_{\mathtt{R}}.\mathsf{uid}$ and both $s_{\mathtt{I}}, s_{\mathtt{R}}$ revert to a previous state, forcing $s_{\mathtt{I}}$ to sample a new $b \leftarrow\$ \{0,1\}^{2048}$. The sessions keep track of the number of retries in the field rid, which is part of m_2, and reject if $rid \geq \mathsf{rid}_{\mathsf{max}}$, the maximum number of retries allowed in the protocol. Since the aid value is only 64 bits, we cannot exclude the possibility of collisions, especially as the number of completed sessions grows. Further, it is possible that the adversary may be able to manipulate the hash h that is used by the responder to indicate a retry due to collisions in the 64-bit "auth key aux hash" value ax, which is the only input used to compute h

```
                                                                                         Stage 2
initiator $s_I$ (knows $pk$)                                          responder $s_R$ (has $(pk, sk), \mathcal{S}_{aid}$)
                                                                      $a \leftarrow\$ \{0,1\}^{2048}$
                                                                      $a \leftarrow\$ \{0,\dots,q-1\}$   // $G_{D.3}$
                                                                      $m_1 \leftarrow n, n_S, g, p, g^a \bmod p, \dots$
                                                                      $\mathcal{M} \leftarrow \mathcal{M} \cup \{m_1\}$   // $G_{D.2}$
                                                                      $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$
$k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$          $\xleftarrow{n, n_S, c_1}$          $c_1 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_1)$
$m_1 \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_1)$
if $m_1 \neq \bot \wedge m_1 \notin \mathcal{M}$ :
  $\mathsf{bad}_2 \leftarrow \mathtt{true}$ ; abort   // $G_{D.2}$
if $m_1 = \bot \vee n, n_S \notin m_1$ : reject
$rid \leftarrow -1$                                                   $rid \leftarrow -1$
RC: $rid \leftarrow rid + 1$ ; if $rid \geq \mathsf{rid}_{\mathsf{max}}$ : reject
$b \leftarrow\$ \{0,1\}^{2048}$
$b \leftarrow\$ \{0,\dots,q-1\}$   // $G_{D.3}$
$m_2 \leftarrow n, n_S, rid, g^b \bmod p$
$\mathcal{M} \leftarrow \mathcal{M} \cup \{m_2\}$   // $G_{D.2}$
$c_2 \leftarrow\$ \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_2)$          $\xrightarrow{n, n_S, c_2}$          RS: $rid \leftarrow rid + 1$ ; if $rid \geq \mathsf{rid}_{\mathsf{max}}$ : reject
                                                                      $m_2 \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_2)$
                                                                      if $m_2 \neq \bot \wedge m_2 \notin \mathcal{M} \wedge s_R.\mathsf{uid} \notin \mathcal{C}_{\mathsf{pub}}$ :
                                                                        $\mathsf{bad}_2 \leftarrow \mathtt{true}$ ; abort   // $G_{D.2}$
                                                                      if $m_2 = \bot \vee n, n_S, rid \notin m_2$ : reject
                                                                      if $s_I.\mathsf{stexec} = \mathsf{accepted}_2 \wedge s_I.\mathsf{cid}.2 = s_R.\mathsf{cid}.2$ :
                                                                        $aid \leftarrow \mathsf{SHA\text{-}1}(ak_{\mathsf{curr}})[96:160]$
                                                                        if $aid \notin \mathcal{S}_{aid}$ : $h \leftarrow h_1^{\mathsf{curr}}$ ; accept $ak_{\mathsf{curr}}$
                                                                        else : $h \leftarrow h_2^{\mathsf{curr}}$ ; retry from RS
~~$ak \leftarrow (g^a)^b \bmod p$~~                                   ~~$ak \leftarrow (g^b)^a \bmod p$~~
                                                                      if $rid = 0$ : $c \leftarrow\$ \{0,\dots,q-1\}$ else : $c \leftarrow\$ \{1,\dots,q-1\}$
                                                                      $ak \leftarrow (ak_{\mathsf{curr}})^c \bmod p$ ; $ak_{\mathsf{curr}} \leftarrow ak$   // $G_{D.5}$
~~$ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$~~                    $ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$
~~$aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$~~                 $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$
~~$h_1 \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$~~                        $h_1^{\mathsf{curr}} \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$
~~$h_2 \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$~~                        $h_2^{\mathsf{curr}} \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$
                                                                      if $aid \notin \mathcal{S}_{aid}$ :
                                                                        $h \leftarrow h_1^{\mathsf{curr}}$ ; accept $ak$
                                                                      else :
                                                                        $h \leftarrow h_2^{\mathsf{curr}}$ ; retry from RS
                                   $\xleftarrow{n, n_S, h}$
~~if $h = h_1$ : accept $ak$~~
~~elseif $h = h_2$ : retry from RC~~
~~else : reject~~
if $s_R.\mathsf{stexec} = \mathsf{accepted}_2 \wedge s_I.\mathsf{cid}.2 = s_R.\mathsf{cid}.2$ :
  $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$
  if $h = h_1^{\mathsf{curr}}$ : accept $ak_{\mathsf{curr}}$
  elseif $h = h_2^{\mathsf{curr}}$ : retry from RC
  else : reject
if $rid = 0$ : $c \leftarrow\$ \{0,\dots,q-1\}$ else : $c \leftarrow\$ \{1,\dots,q-1\}$
$ak \leftarrow (ak_{\mathsf{curr}})^c \bmod p$ ; $ak_{\mathsf{curr}} \leftarrow ak$
$ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$
$aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$
$h_1^{\mathsf{curr}} \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$
$h_2^{\mathsf{curr}} \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$
if $h = h_1^{\mathsf{curr}}$ : accept $ak$
elseif $h = h_2^{\mathsf{curr}}$ : retry from RC
else : reject   // $G_{D.5}$
```

$$\downarrow$$

$$\mathsf{sid}.2 = (\mathsf{sid}.1, g^a \bmod p, g^b \bmod p, h)\ ;\ \mathsf{sskey}.2 = ak[0:1024]$$

Fig. 62. Protocol transitions for Case D starting from $G_{D.2}$ (stage 2 only). Comments/strikethrough mark the first game that adds/removes the given colour-coded "block" of pseudocode. To "accept ak" means to use it to set sskey.2; ak_{curr} is a global variable initialised as g.

```
                                                                                        Stage 2
1 :  initiator $s_\mathrm{I}$ (knows $pk$)                      responder $s_\mathrm{R}$ (has $(pk, sk), \mathcal{S}_\mathsf{aid}$)
2 :  ...                                                       ...
3 :  $rid \leftarrow -1$                                       $rid \leftarrow -1$
4 :  RC: $rid \leftarrow rid + 1$ ; if $rid \geq \mathsf{rid}_\mathsf{max}$ : reject
5 :  $b \leftarrow\!\$\ \{0, \dots, q-1\}$
6 :  if $\exists s' : s'.\mathsf{uid} \notin \mathcal{C}_\mathsf{pub} \wedge g^{ab} = g^{a'b'}$ :
7 :    $\mathsf{bad}_3 \leftarrow \mathtt{true}$ ; abort
8 :  $m_2 \leftarrow n, n_S, rid, g^b \bmod p$
9 :  $\mathcal{M} \leftarrow \mathcal{M} \cup \{m_2\}$
10 : $c_2 \leftarrow\!\$\ \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_2)$   $\xrightarrow{n, n_S, c_2}$   RS: $rid \leftarrow rid + 1$ ; if $rid \geq \mathsf{rid}_\mathsf{max}$ : reject
11 :                                                           if $m_2 \neq \perp \wedge m_2 \notin \mathcal{M} \wedge s_\mathrm{R}.\mathsf{uid} \notin \mathcal{C}_\mathsf{pub}$ :
12 :                                                             $\mathsf{bad}_2 \leftarrow \mathtt{true}$ ; abort
13 :                                                           if $m_2 = \perp \vee n, n_S, rid \notin m_2$ : reject
14 :                                                           if $s_\mathrm{I}.\mathsf{st_{exec}} = \mathsf{accepted}_2 \wedge s_\mathrm{I}.\mathsf{cid}.2 = s_\mathrm{R}.\mathsf{cid}.2$ :
15 :                                                             $aid \leftarrow \mathsf{SHA\text{-}1}(ak_\mathsf{curr})[96:160]$
16 :                                                             if $aid \notin \mathcal{S}_\mathsf{aid}$ : $h \leftarrow h_1^\mathsf{curr}$ ; accept $ak_\mathsf{curr}$
17 :                                                             else : $h \leftarrow h_2^\mathsf{curr}$ ; retry from RS
18 :                                                           else :
19 :                                                             if $rid = 0$ : $c \leftarrow\!\$\ \{0, \dots, q-1\}$ else : $c \leftarrow\!\$\ \{1, \dots, q-1\}$
20 :                                                             $ak \leftarrow (ak_\mathsf{curr})^c \bmod p$
21 :                                                             $ak[1536:2048] \leftarrow\!\$\ \{0,1\}^{512}$   // $\mathrm{G_{D.6}}$
22 :                                                             $ak_\mathsf{curr} \leftarrow ak$   // $\mathrm{G_{D.5}}$
23 :                                                             $ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$
24 :                                                             $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$
25 :                                                             $ax \leftarrow\!\$\ \{0,1\}^{64}$ ; $aid \leftarrow\!\$\ \{0,1\}^{64}$   // $\mathrm{G_{D.7}}$
26 :                                                             $h_1^\mathsf{curr} \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$
27 :                                                             $h_2^\mathsf{curr} \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$
28 :                                                             if $aid \notin \mathcal{S}_\mathsf{aid}$ :
29 :                                                               $h \leftarrow h_1^\mathsf{curr}$ ; accept $ak$
30 :                                                             else :
31 :                                                               $h \leftarrow h_2^\mathsf{curr}$ ; retry from RS
32 :                                               $\xleftarrow{n, n_S, h}$
33 : if $s_\mathrm{R}.\mathsf{st_{exec}} = \mathsf{accepted}_2 \wedge s_\mathrm{I}.\mathsf{cid}.2 = s_\mathrm{R}.\mathsf{cid}.2$ :
34 :   $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$
35 :   if $h = h_1^\mathsf{curr}$ : accept $ak_\mathsf{curr}$
36 :   elseif $h = h_2^\mathsf{curr}$ : retry from RC
37 :   else : reject
38 : else :
39 :   if $rid = 0$ : $c \leftarrow\!\$\ \{0, \dots, q-1\}$ else : $c \leftarrow\!\$\ \{1, \dots, q-1\}$
40 :   $ak \leftarrow (ak_\mathsf{curr})^c \bmod p$
41 :   $ak[1536:2048] \leftarrow\!\$\ \{0,1\}^{512}$   // $\mathrm{G_{D.6}}$
42 :   $ak_\mathsf{curr} \leftarrow ak$
43 :   $ax \leftarrow \mathsf{SHA\text{-}1}(ak)[0:64]$
44 :   $aid \leftarrow \mathsf{SHA\text{-}1}(ak)[96:160]$
45 :   $ax \leftarrow\!\$\ \{0,1\}^{64}$ ; $aid \leftarrow\!\$\ \{0,1\}^{64}$   // $\mathrm{G_{D.7}}$
46 :   $h_1^\mathsf{curr} \leftarrow \mathsf{NH.Ev}(n_n, ax, 1)$
47 :   $h_2^\mathsf{curr} \leftarrow \mathsf{NH.Ev}(n_n, ax, 2)$
48 :   if $h = h_1^\mathsf{curr}$ : accept $ak$
49 :   elseif $h = h_2^\mathsf{curr}$ : retry from RC
50 :   else : reject   // $\mathrm{G_{D.5}}$
```

$$\downarrow$$

$$\mathsf{sid}.2 = (\mathsf{sid}.1, g^a \bmod p, g^b \bmod p, h) \ ; \ \mathsf{sskey}.2 = ak[0:1024]$$

Fig. 63. Protocol transitions for Case D starting from $\mathrm{G_{D.5}}$ (the last two queries of stage 2 only).

that depends on the agreed key *ak*. In the next part of the proof, we show that the protocol remains secure despite this ability.[42]

Note that at this point, we are not assured that $s_{\mathtt{I}}$ and $s_{\mathtt{R}}$ will be partnered at the end of the execution, only that at least one of them will accept and will be testable. Further, we attempt to capture all possible flows with respect to retries in the protocol: the case of "accidental" retries caused by the honest parties, the case where the adversary prevents honest retries, i.e. where $\mathcal{A}$ forges h signalling that *aid* is unique when it is not, and the case where the adversary causes at least one additional retry, again by forging an appropriate h.

$\mathbf{G}_{\text{D.1}} \to \mathbf{G}_{\text{D.2}}$. Consider the game $\text{G}_{\text{D.2}}$, which is equivalent to $\text{G}_{\text{D.1}}$ except that the game sets the flag bad_2 to `true` and aborts if $s_{\mathtt{I}}$ receives m_1 that was not produced by the session $s_{\mathtt{R}}$, or if $s_{\mathtt{R}}$ receives m_2 that was not produced by the session $s_{\mathtt{I}}$ and $s_{\mathtt{R}}.\mathsf{uid} \notin \mathcal{C}_{\mathsf{pub}}$ (see Fig. 62). Note that retries may cause $s_{\mathtt{R}}$ to process many m_2, but at most $\mathsf{rid}_{\mathsf{max}}$ such values. Conditioning on $s_{\mathtt{R}}.\mathsf{uid}$ remaining uncorrupted is important because we are not assured that the sessions will be partnered, which means that corrupting $s_{\mathtt{R}}.\mathsf{uid}$ may only mark one session's stage 2 key as revealed: e.g. if $\mathcal{A}$ forged a hash h making $s_{\mathtt{I}}$ accept prematurely, only $s_{\mathtt{R}}.\mathsf{sskey}.2$ would be marked as revealed and $s_{\mathtt{I}}.\mathsf{sskey}.2$ would remain testable. We have

$$\Pr[\text{G}_{\text{D.1}}] - \Pr[\text{G}_{\text{D.2}}] \leq \Pr\left[\mathsf{bad}_2^{\text{G}_{\text{D.2}}}\right].$$

To bound $\Pr\left[\mathsf{bad}_2^{\text{G}_{\text{D.2}}}\right]$, we first use two additional game hops to intermediate games $\text{G}_{\text{D.2}^*}$ and $\text{G}_{\text{D.2}'}$, that handle other uses of n_n in the protocol.

IND-CCA. The game $\text{G}_{\text{D.2}^*}$ differs from $\text{G}_{\text{D.2}}$ in that $\textsc{Send}(s_{\mathtt{I}}.\mathsf{label}, (n, n_s, ...))$ (for some n, n_s) sets a random session key $s_{\mathtt{I}}.\mathsf{sskey}.1 \leftarrow\!\!\$\ \{0,1\}^{256}$ instead of using n_n. For consistency, the game also sets $s_{\mathtt{R}}.\mathsf{sskey}.1 \leftarrow s_{\mathtt{I}}.\mathsf{sskey}.1$. The construction is the same as in Case B, in particular as in the transition between the games $\text{G}_{\text{B.1}}$ and $\text{G}_{\text{B.2}}$. As in Case C, here the simulation is not perfect, but we only consider what happens before the flag bad_2 is set. We have

$$\Pr\left[\mathsf{bad}_2^{\text{G}_{\text{D.2}}}\right] \leq \Pr\left[\mathsf{bad}_2^{\text{G}_{\text{D.2}^*}}\right] + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{TOAEP^+}}(\mathcal{A}_{\mathsf{IND\text{-}CCA}}).$$

Independence of SKDF *and* NH. The game $\text{G}_{\text{D.2}'}$ generates an additional random value $n'_n \leftarrow\!\!\$\ \{0,1\}^{256}$ at the end of stage 1, and uses this value instead of the original n_n to compute h via NH. This step is necessary to handle the key reuse present in the protocol in a more modular way, before reasoning about the pseudorandomness of the keys derived using SKDF keyed by n_n.

Using an adversary $\mathcal{A}$ playing in $\text{G}_{\text{D.2}^*}$ resp. $\text{G}_{\text{D.2}'}$, we build an adversary $\mathcal{D}_{\mathsf{IND\text{-}KEY}}$ against the indistinguishability of key reuse between SKDF and NH (Fig. 19). The adversary $\mathcal{D}_{\mathsf{IND\text{-}KEY}} = (\mathcal{D}_1, \mathcal{D}_2)$ can be constructed in a straightforward way: $\mathcal{D}_1$ begins simulating the protocol for $\mathcal{A}$ according to $\text{G}_{\text{D.2}^*}$ resp. $\text{G}_{\text{D.2}'}$ and first captures n_s from a $\textsc{Send}$ query input for the tested pair of sessions; $\mathcal{D}_2(y)$ continues without setting the stage 1 session key (which cannot be tested or revealed in this case), using y as the key k_{se} and computing the relevant h values using queries to its $\textsc{Eval}$ oracle. Since the "real" n_n, now computed by the IND-KEY game, is not used elsewhere in the protocol, the simulation is perfect. When the challenge bit in $\mathcal{D}_{\mathsf{IND\text{-}KEY}}$'s game is 1, $\mathcal{A}$ is playing in $\text{G}_{\text{D.2}^*}$, and when the challenge bit is 0, $\mathcal{A}$ is playing in $\text{G}_{\text{D.2}'}$. Finally, the adversary $\mathcal{D}_2$ outputs the bit guess of $\mathcal{A}$ as its own. We have

$$\Pr\left[\mathsf{bad}_2^{\text{G}_{\text{D.2}^*}}\right] \leq \Pr\left[\mathsf{bad}_2^{\text{G}_{\text{D.2}'}}\right] + \mathsf{Adv}^{\mathsf{IND\text{-}KEY}}_{\mathsf{SKDF,NH}}(\mathcal{D}_{\mathsf{IND\text{-}KEY}}).$$

[42] Note that though we cannot assume h to be unforgeable for the proof, this does not mean it is easy to forge in practice.

Integrity of plaintexts. Starting from $G_{D.2'}$, we can bound the probability of abort due to bad_2 using the integrity of plaintexts of HtE-SE. Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE}_{2st}$ in $G_{D.2'}$, we build an adversary $\mathcal{A}_{\mathsf{INT\text{-}PTXT}} = (\mathcal{A}_{\mathsf{INT\text{-}PTXT},1}, \mathcal{A}_{\mathsf{INT\text{-}PTXT},2})$ that plays in the INT-PTXT game (Fig. 39) with HtE-SE, SKDF and $\mathsf{SAMP}[\cdot,\cdot,g,p]$.

$\mathcal{A}_{\mathsf{INT\text{-}PTXT},1}(n)$ starts simulating the game $G_{D.2'}$ for $\mathcal{A}$. During stage 1, it makes $s_{\mathtt{I}}$ set n instead of generating its own nonce, and once $s_{\mathtt{R}}$ sets n_s as its server nonce, it outputs its current state st and n_s.

$\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}(st)$ continues the simulation. As in $\mathsf{MTP\text{-}KE}_{2st}$, it initialises a counter $rid = 0$ which will count the number of protocol retries. At the end of stage 1, it also generates a fresh random value $n'_n \leftarrow\$ \{0,1\}^{256}$, which will be used in case the simulated protocol requires honest values of h. The following SEND queries are also modified:

- On $\textsc{Send}(s_{\mathtt{R}}.\mathsf{label}, (n, n_s, ..., c_0))$ for some c_0, $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ decrypts c_0 to m_0 as in $\mathsf{MTP\text{-}KE}_{2st}$, sets a random stage 1 session key as in $G_{D.2^*}$, and calls its own encryption oracle $\textsc{Enc}(aux)$ with $aux = servertime$ to obtain a, m_1, c_1 instead of generating and encrypting the Diffie-Hellman share itself. However, it does also compute $k_{se} \leftarrow \mathsf{SKDF.Ev}(n_n, n_s)$ where n_n is the value obtained from m_0. It returns $(n, n_s, c_1), \mathtt{running}_2$ to $\mathcal{A}$.[43]
- On $\textsc{Send}(s_{\mathtt{I}}.\mathsf{label}, (n, n_s, c_1))$ for some c_1, if $s_{\mathtt{I}}.\mathsf{vid} \in \mathcal{C}_{\mathsf{pub}}$, then $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ uses the key k_{se} derived from n_n that was encrypted in c_0 to compute $m_1 \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_1)$, checks it and then (if not rejecting) computes $c_2 \leftarrow \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_2)$ for m_2 as in $\mathsf{MTP\text{-}KE}_{2st}$ and returns $(n, n_s, c_2), \mathtt{running}_2$ to $\mathcal{A}$.[44]

  Otherwise, if $s_{\mathtt{I}}.\mathsf{vid} \notin \mathcal{C}_{\mathsf{pub}}$, $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ calls its own decryption oracle via $\textsc{Dec}(c_1)$, which returns a message m_1 or it aborts $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$. If $m_1 \neq \perp$ and n, n_s are included in m_1, $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ calls $\textsc{Enc}(aux)$ with $aux = rid$ to obtain b, m_2, c_2 and returns $(n, n_s, c_2), \mathtt{running}_2$ to $\mathcal{A}$. Note that the $\textsc{Dec}$ oracle will never return $m_1 = \perp$, since in that case $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ is aborted, having lost the game.
- On $\textsc{Send}(s_{\mathtt{R}}.\mathsf{label}, (n, n_s, c_2))$ for some c_2, if $s_{\mathtt{I}}.\mathsf{vid} \in \mathcal{C}_{\mathsf{pub}}$, then $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ uses the key k_{se} to compute $m_2 \leftarrow \mathsf{HtE\text{-}SE.Dec}(k_{se}, c_2)$ and then proceeds as in the original game, which includes calculating an *aid* value and checking if it is unique, which can trigger a retry.

  Otherwise, if $s_{\mathtt{I}}.\mathsf{vid} \notin \mathcal{C}_{\mathsf{pub}}$, $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ calls its own decryption oracle via $\textsc{Dec}(c_2)$, which returns a message m_2 or it aborts $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$. If $m_2 \neq \perp$ and n, n_s are included in m_2, $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ proceeds as in the original game (incl. computing h using n'_n). As above, the $\textsc{Dec}$ oracle will never return $m_1 = \perp$.
- On $\textsc{Send}(s_{\mathtt{I}}.\mathsf{label}, (n, n_s, h))$ for some h, $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ proceeds as in the original game depending on the value of h. If a retry is signalled, it computes the next ciphertext $c'_2 \leftarrow \mathsf{HtE\text{-}SE.Enc}(k_{se}, m'_2)$ for m'_2 as in $\mathsf{MTP\text{-}KE}_{2st}$ and returns $(n, n_s, c'_2), \mathtt{running}_2$ to $\mathcal{A}$.

In the above, we assume that $\mathcal{A}_{\mathsf{INT\text{-}PTXT}}$ follows the parsing behaviour of $\mathsf{MTP\text{-}KE}_{2st}$. Since role confusion is excluded by the soundness predicate, we are assured that c_1 and c_2 could not be swapped. Further, not calling the INT-PTXT oracles when $s_{\mathtt{I}}.\mathsf{vid} \in \mathcal{C}_{\mathsf{pub}}$ ensures that the simulation works in the cases where $\mathcal{A}$ corrupts one of the parties before the other has finished running, since otherwise corruption of the keypair (pk, sk) would not give $\mathcal{A}$ the power to decrypt or produce HtE-SE ciphertexts.

The remainder of the analysis follows the same argument as in the transition between the games $G_{C.1}$ and $G_{C.2}$ of Case C. Thus, whenever $\mathcal{A}$ sets the bad_2 flag and thus triggers an abort in the simulated game, $\mathcal{A}_{\mathsf{INT\text{-}PTXT}}$ wins in its own game. Note that the maximum number of retries $\mathsf{rid}_{\mathsf{max}}$ is implicitly included in $\mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}$ as it upper-bounds the number of $\textsc{Enc}$ and $\textsc{Dec}$ queries.

We have
$$\Pr\left[\mathsf{bad}_2^{G_{D.2'}}\right] \leq \mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE},\mathsf{SKDF},\mathsf{SAMP},g,p}(\mathcal{A}_{\mathsf{INT\text{-}PTXT}}).$$

[43] To be more precise, since this query causes stages to change, $\mathcal{A}_{\mathsf{INT\text{-}PTXT},2}$ first pauses the execution at the end of stage 1, returns control to $\mathcal{A}$ and only resumes executing SEND when called by $\mathcal{A}$ to continue.

[44] This is to ensure that certain corruption queries still enable the adversary to use the value n_n retrieved from c_0 to compute keys and essentially produce forgeries.

Hence, overall we get

$$\mathsf{Adv}^{G_{D.1}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq 2 \cdot \mathsf{Adv}^{\mathsf{INT\text{-}PTXT}}_{\mathsf{HtE\text{-}SE,SKDF,SAMP},g,p}(\mathcal{A}_{\mathsf{INT\text{-}PTXT}}) + 2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}KEY}}_{\mathsf{SKDF,NH}}(\mathcal{D}_{\mathsf{IND\text{-}KEY}})$$
$$+ 2 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{TOAEP^+}}(\mathcal{A}_{\mathsf{IND\text{-}CCA}}) + \mathsf{Adv}^{G_{D.2}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

Note that counterintuitively, after this transition we cannot say yet that the sessions $s_{\mathtt{I}}, s_{\mathtt{R}}$ agree on the values $g^a \bmod p$, $g^b \bmod p$. It is possible that by manipulating the hashes that serve as retry indicators the adversary can cause the two sessions to derive different stage 2 keys, or cause one of the sessions to reject. However, given that we are in a responder-only authentication setting, we only consider a pair of sessions that includes an honest initiator. For such a pair, the adversary cannot manipulate the values themselves.

$G_{D.2} \to G_{D.3}$. The game $G_{D.3}$ samples the Diffie-Hellman shares from $\{0, \dots, q-1\}$ instead of $\{0,1\}^{2048}$ as in $G_{D.2}$. Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE_{2st}}$, we build an adversary $\mathcal{D}_{\mathsf{BIAS}}$ that distinguishes between the distributions $\mathsf{D}_0 = \{(g, g^x) \mid x \leftarrow\!\$\ \{0, \dots, q-1\}\}$ and $\mathsf{D}_1 = \{(g, g^y) \mid y \leftarrow\!\$\ \{0,1\}^{2048}\}$, simulating $G_{D.2}$ or $G_{D.3}$ depending on the challenge bit in its game and replacing the one instruction differing between $G_{D.2}$ and $G_{D.3}$ with the output of its challenge oracle. We have

$$\mathsf{Adv}^{G_{D.2}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \mathsf{Adv}^{G_{D.3}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathsf{BIAS}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{BIAS}}).$$

We can reduce the problem of distinguishing between the biased distributions as defined above to the problem of distinguishing short exponents from full exponents (Definition 1). We focus on the case $n = 2048$, and let c be such that $\log(n) < c < n$. Let $\mathsf{D}_2 = \{(g, g^z) \mid z \leftarrow 2^{2048-c} \cdot u, u \leftarrow\!\$\ \{0,1\}^c\}$. Using an adversary $\mathcal{D}_{\mathsf{BIAS}}$ that distinguishes between D_0 and D_1, we construct an adversary $\mathcal{D}_{\mathsf{S\text{-}EXP}}$ that distinguishes between D_0 and D_2. $\mathcal{D}_{\mathsf{S\text{-}EXP}}$, given (g, W), computes

$$W' \leftarrow g^{2^c \cdot r} \cdot W^v,$$

where $r \leftarrow\!\$\ \{0,1\}^{2048-c}$ and $v \leftarrow (2^{2048-c})^{-1} \bmod q$. It gives (g, W') to $\mathcal{D}_{\mathsf{BIAS}}$ and outputs whatever $\mathcal{D}_{\mathsf{BIAS}}$ returns. If W is from D_0, we have $W' = g^{2^c \cdot r} \cdot g^{v \cdot x}$. Since v is a constant and x is sampled uniformly at random from $\{0, \dots, q-1\}$, the distribution over (g, W') is the same as D_0. If W is from D_2, we have $W' = g^{2^c \cdot r} \cdot g^{v \cdot z} = g^{2^c \cdot r} \cdot g^{(2^{2048-c})^{-1} \cdot 2^{2048-c} \cdot u} = g^{2^c \cdot r + u}$. Since r is $(2048 - c)$ and u is c bits, the distribution over (g, W') is the same as D_1. We have

$$\mathsf{Adv}^{\mathsf{BIAS}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{BIAS}}) \leq \mathsf{Adv}^{\mathsf{S\text{-}EXP}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{S\text{-}EXP}}).$$

$G_{D.3} \to G_{D.4}$. The game $G_{D.4}$ differs from $G_{D.3}$ in that at the end of the game, it additionally sets the flag bad_3 and aborts if $s_{\mathtt{I}}$ accepted a key ak derived using a value g^b or g^a such that another session had also accepted a key derived using the same g^b or g^a, as shown in Fig. 64. We have

$$\Pr[G_{D.3}] - \Pr[G_{D.4}] \leq \Pr\left[\mathsf{bad}_3^{G_{D.4}}\right] \leq \frac{4 n_{\mathsf{S}}}{q},$$

and hence

$$\mathsf{Adv}^{G_{D.3}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}) \leq \frac{8 n_{\mathsf{S}}}{q} + \mathsf{Adv}^{G_{D.4}}_{\mathsf{MTP\text{-}KE_{2st}}}(\mathcal{A}).$$

This ensures that sessions other than $s_{\mathtt{I}}, s_{\mathtt{R}}$ cannot derive the same session key as $s_{\mathtt{I}}$.[45]

$G_{D.4} \to G_{D.5}$. The game $G_{D.5}$ differs from $G_{D.4}$ in that the game replaces the last computed ak value (and therefore the stage 2 session key) for the sessions $s_{\mathtt{I}}$, $s_{\mathtt{R}}$ by a value of the form $g^c \bmod p$ for a random

[45] This is sufficient in the setting of responder-only authentication as $s_{\mathtt{R}}$'s key can only be tested if $s_{\mathtt{R}}$ has an honest contributive partner (though it can still be revealed).

```
G_{D.3}–G_{D.4}
1 :  $\mathcal{L}_K, \mathcal{K}_{pub} \leftarrow \mathsf{Init}(\mathcal{U}_{role})$
2 :  $\mathcal{L}_S \leftarrow [\,]\,;\ \mathcal{C}_{pub} \leftarrow \emptyset$
3 :  $b_{test} \leftarrow\!\!\$\ \{0,1\}$
4 :  $lost \leftarrow \mathtt{false}$
5 :  $s_I, s_R \leftarrow\!\!\$$ guess tested session pair
6 :  $b'_{test} \leftarrow \mathcal{A}^{\text{NewSession},\ldots,\text{Test}}(\mathcal{K}_{pub})$   // Test aborts if the tested session is not $s_I$ or $s_R$
7 :  if $\exists s, s' \in \mathcal{L}_S, n \in \{0,1\}^{128}, n_n \in \{0,1\}^{256} : (s \neq s'$
8 :     $\wedge\ s.\mathsf{uid.role} = s'.\mathsf{uid.role} = \mathtt{I} \wedge s.\mathsf{sskey}.1 = s'.\mathsf{sskey}.1 = n_n$
9 :     $\wedge\ s.\mathsf{sid}.1 = (\_, n, \_, \_) \wedge s'.\mathsf{sid}.1 = (\_, n, \_, \_))$ :
10 :      $\mathsf{bad}_0 \leftarrow \mathtt{true}$
11 :      return 0
12 :  $(\ldots, g^a, g^b, \_) \leftarrow s_I.\mathsf{sid}.2$   // skip if the session rejects
13 :  if $\exists s \in \mathcal{L}_S$ :
14 :    $(s \neq s_I\ \wedge\ s.\mathsf{uid.role} = \mathtt{I}\ \wedge\ (s.\mathsf{sid}.2 = (\ldots, \_, g^b, \_) \vee s.\mathsf{sid}.2 = (\ldots, \_, g^a, \_)))$
15 :    $\vee\ (s.\mathsf{uid.role} = \mathtt{R} \wedge s.\mathsf{sid}.2 \neq s_I.\mathsf{sid}.2 \wedge (s.\mathsf{sid}.2 = (\ldots, g^a, \_, \_) \vee s.\mathsf{sid}.2 = (\ldots, g^b, \_, \_)))$ :
16 :      $\mathsf{bad}_3 \leftarrow \mathtt{true}$
17 :      return 0   // G_{D.4}
18 :  if $\neg\mathsf{Fresh}$ :
19 :    $lost \leftarrow \mathtt{true}$
20 :  return $b'_{test} = b_{test} \wedge lost = \mathtt{false}$
```

Fig. 64. Games $G_{D.3}$–$G_{D.4}$ for the proof of Multi-Stage-security of $\mathsf{MTP\text{-}KE}_{2st}$.

$c \leftarrow\!\!\$\ \{0, \ldots, q-1\}$. Note that the adversary may cause either session to complete its execution first, and by definition at most one of the sessions s_I, s_R can reject. Recall that the sessions are contributive partners in the last message exchange of stage 2 if and only if the sessions agree on the values of $g^a, g^b \bmod p$.

Let $ak_{curr} \leftarrow g$ be a global variable that will represent the shared key (before it is accepted, it may be updated several times). We will use h_1^{curr}, h_2^{curr} in a similar way, though they will only be first set during the execution of the protocol. The game behaves as follows (as depicted in Fig. 62):

- On $\text{Send}(s_R.\mathsf{label}, (n, n_s, c_2))$ for n, n_s in $s.\mathsf{cid}.2$, the game decrypts c_2 as in the original protocol to recover the value $g^b \bmod p$ (and rejects if it cannot recover one). Then:
  - *If s_I has already accepted at that point and the sessions s_I, s_R are contributive partners:*

    The game uses ak_{curr}, the auth key used by s_I, to check whether $aid = \mathsf{SHA\text{-}1}(ak_{curr})[96:160]$ is unique with respect to $s_R.\mathsf{uid}$'s $\mathcal{S}_{aid}$. If it is, it accepts $s_R.\mathsf{sskey}.2 \leftarrow s_I.\mathsf{sskey}.2 = ak_{curr}[0:1024]$ and outputs h_1^{curr}. If it is not, it goes into retry state and outputs h_2^{curr}.[46]
  - *Else:* The game computes a new auth key $ak \leftarrow (ak_{curr})^c \bmod p$ where $c \leftarrow\!\!\$\ \{0, \ldots, q-1\}$, updates ak_{curr} with this value, and uses it in the remaining protocol as before.
- On $\text{Send}(s_I.\mathsf{label}, (n, n_s, h))$ for n, n_s in $s.\mathsf{cid}.2$:
  - *If s_R has already accepted at that point and the sessions s_I, s_R are contributive partners:*

[46] Note that in the latter case, the session s_R will reject upon the next message it receives. This is because s_I cannot produce new messages, the adversary cannot forge such messages, and *rid* prevents out-of-order delivery or replays of past messages.

The game uses the auth key ak_{curr} computed by $s_{\mathtt{R}}$, and the corresponding values of h_1^{curr} and h_2^{curr} to proceed. If $h = h_1^{\mathsf{curr}}$, $s_{\mathtt{I}}$ accepts $s_{\mathtt{I}}.\mathsf{sskey}.2 \leftarrow s_{\mathtt{R}}.\mathsf{sskey}.2$; if $h = h_2^{\mathsf{curr}}$, it goes intro retry mode; otherwise it rejects.

- *Else:* The game computes a new auth key $ak \leftarrow (ak_{\mathsf{curr}})^c \bmod p$ where $c \leftarrow\$ \{0, \ldots, q-1\}$, updates ak_{curr} with this value, and uses it in the remaining protocol as before. Note that in this case, the session will not make any checks with respect to the uniqueness of the *aid* value.[47]

Note that once $s_{\mathtt{I}}$ uses a different value of $g^b \bmod p$ to set its cid than $s_{\mathtt{R}}$, the only way for the sessions to become contributive partners again is if $s_{\mathtt{R}}$ has not yet finished running and at some point receives c_2 containing the relevant $g^b \bmod p$ (in the right order). Note that if the sessions end up accepting without being partnered, only $s_{\mathtt{I}}$ can be tested since stage 2 is responder-only authenticated and TEST requires the tested session to have a contributive partner; when $s_{\mathtt{I}}$ is tested, the adversary can learn $s_{\mathtt{R}}$'s session key using a REVEAL query. Hence, if the sessions first agree on contributive identifiers and later diverge, the game will replace both of their stage 2 session keys, but with *different* values $g^c, g^{c \cdot c'} \bmod p$ for some c, c'.

Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE_{2st}}$, we build an adversary $\mathcal{D}_{\mathsf{DDH}}$ against the DDH assumption that simulates $\mathrm{G}_{\mathrm{D}.4}$ or $\mathrm{G}_{\mathrm{D}.5}$ depending on the challenge bit in its game. $\mathcal{D}_{\mathsf{DDH}}$ is given the group parameters $g \in \mathbb{Z}_p^*$ for prime p of the form $p = 2q + 1$ where q is prime[48] and g generates a cyclic subgroup of order q denoted by $\mathbb{G}$, and the values X, Y, Z as input; its task is to distinguish whether $Z = g^z$ or $Z = g^{xy}$ where $X = g^x$, $Y = g^y$ for random x, y, z. It samples a random bit $b_{\mathsf{test}} \leftarrow\$ \{0, 1\}$ at the start and precomputes the following values:

$$\begin{aligned} ak^* &\leftarrow Z \\ ax^* &\leftarrow \mathsf{SHA\text{-}1}(ak^*)[0:64] \\ aid^* &\leftarrow \mathsf{SHA\text{-}1}(ak^*)[96:160]. \end{aligned}$$

The adversary then modifies the simulated SEND oracle as follows:

- On $\mathrm{SEND}(s_{\mathtt{R}}.\mathsf{label}, (n, n_s, \ldots, c_0))$ for some c_0, $\mathcal{D}_{\mathsf{DDH}}$ decrypts c_0 to m_0 as in $\mathsf{MTP\text{-}KE_{2st}}$, sets a random stage 1 session key, and uses the given parameters g, p and the share X to construct m_1 instead of generating its own g^a. It then computes k_{se} and encrypts m_1 to c_1 under k_{se}. It returns $(n, n_s, c_1), \mathtt{running}_2$ to $\mathcal{A}$.
- On $\mathrm{SEND}(s_{\mathtt{I}}.\mathsf{label}, (n, n_s, c_1))$ for some c_1, $\mathcal{D}_{\mathsf{DDH}}$ computes k_{se} and decrypts c_1 to m_1, making all the necessary checks (in particular, it also checks whether m_1 is the same message it constructed for $s_{\mathtt{R}}$). It then uses the share Y to construct m_2 instead of generating its own g^b, and encrypts it as $c_2 \leftarrow \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_2)$. It returns $(n, n_s, c_2), \mathtt{running}_2$ to $\mathcal{A}$.
- On $\mathrm{SEND}(s_{\mathtt{R}}.\mathsf{label}, (n, n_s, c_2))$ for some c_2, $\mathcal{D}_{\mathsf{DDH}}$ decrypts c_2 to m_2, making all the necessary checks (it also checks whether m_2 is the same message it constructed for $s_{\mathtt{I}}$). First, consider the case that $rid = 0$. If $aid^* \notin \mathcal{S}_{\mathsf{aid}}$ of $s_{\mathtt{R}}$, it computes $h_1^* \leftarrow \mathsf{SHA\text{-}1}(n_n \parallel 01 \parallel ax^*)[32:160]$, accepts the key $s_{\mathtt{R}}.\mathsf{sskey}.2 \leftarrow ak^*[0:1024]$) and returns $(n, n_s, h_1^*), \mathtt{accepted}_2$ to $\mathcal{A}$. Otherwise, it computes $h_2^* \leftarrow \mathsf{SHA\text{-}1}(n_n \parallel 02 \parallel ax^*)[32:160]$ and returns $(n, n_s, h_2^*), \mathtt{running}_2$ to $\mathcal{A}$.

  If it is the case that $rid > 0$, it follows the logic as for $rid = 0$ except using $ak' \leftarrow Z^{b'}$ instead of ak^*, where b' was generated by $s_{\mathtt{I}}$ to produce m_2' (see below), to compute ax', aid', h_1' and h_2'.
- On $\mathrm{SEND}(s_{\mathtt{I}}.\mathsf{label}, (n, n_s, h))$ for some h, $\mathcal{D}_{\mathsf{DDH}}$ proceeds depending on the value of h. First, consider the case before any retries. If $rid = 0$ and $h = h_1^*$, it accepts the key $s_{\mathtt{I}}.\mathsf{sskey}.2 \leftarrow ak^*[0:1024]$ and returns $\mathtt{accepted}_2$ to $\mathcal{A}$. If $rid = 0$ and $h = h_2^*$, it computes $b' \leftarrow\$ \{1, \ldots, q-1\}$ and constructs the next ciphertext $c_2' \leftarrow \mathsf{HtE\text{-}SE.Enc}(k_{se}, m_2')$ where m_2' uses $Y^{b'}$ and returns $(n, n_s, c_2'), \mathtt{running}_2$ to $\mathcal{A}$.

[47] This represents no change from the original protocol, where a client session could potentially be fooled into accepting a non-unique *aid*, losing synchronisation with the server.

[48] Telegram uses a fixed prime p shown in `https://core.telegram.org/mtproto/auth_key`.

If it is the case that $rid > 0$, it follows the logic as for $rid = 0$ except using the previously generated $ak' \leftarrow Z^{b'}$ instead of ak^* to compute ax', aid', h'_1 and h'_2 that are used in the checks for $h = h^*_1$ and $h = h^*_2$. If it is the latter, it generates a fresh b' value to proceed as above.

$\mathcal{D}_{\mathsf{DDH}}$ outputs its own guess as $b'_{\mathsf{chall}} \leftarrow b'_{\mathsf{test}} = b_{\mathsf{test}} \wedge lost = \mathtt{false}$ where b'_{test} is the bit guess of $\mathcal{A}$ and the *lost* flag is set under the same conditions as in both games $\mathrm{G}_{\mathrm{D.4}}, \mathrm{G}_{\mathrm{D.5}}$. Note that in the case of retries, the session key will be of the form $Z^{b'}$ for an honestly generated b'.

If the challenge bit $b_{\mathsf{chall}} = 0$ in the DDH game, then the tested session will accept the key derived from $Z = g^{xy}$ just as in the game $\mathrm{G}_{\mathrm{D.4}}$. Otherwise, if $b_{\mathsf{chall}} = 1$, then the tested session will accept the key derived from $Z = g^z$, capturing the behaviour of the game $\mathrm{G}_{\mathrm{D.5}}$. We have $\Pr\left[b'_{\mathsf{chall}} = 1 \;\middle|\; b_{\mathsf{chall}} = 1\right] = \Pr[\mathrm{G}_{\mathrm{D.4}}]$ and $\Pr\left[b'_{\mathsf{chall}} = 1 \;\middle|\; b_{\mathsf{chall}} = 0\right] = \Pr[\mathrm{G}_{\mathrm{D.5}}]$, hence

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{D.4}}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{G}_{\mathrm{D.5}}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathsf{DDH}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{DDH}}).$$

$\mathrm{G}_{\mathrm{D.5}} \to \mathrm{G}_{\mathrm{D.6}}$. The game $\mathrm{G}_{\mathrm{D.6}}$ changes the following: whenever $\mathrm{G}_{\mathrm{D.5}}$ sets a given ak as $g^c \bmod p$, $\mathrm{G}_{\mathrm{D.6}}$ replaces 512 of the least significant bits of ak with a uniformly random string, i.e. $ak[1536:2048] \leftarrow\!\!\$ \{0,1\}^{512}$ (see Fig. 63).

Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE}_{2st}$, we build an adversary $\mathcal{D}_{\mathsf{LSB}}$ that distinguishes between the distributions $\mathsf{D}_0 = \{g^c \mid g^c \leftarrow\!\!\$ \mathbb{G}_{1536:2048}\} = \{g^c \bmod 2^{512} \mid g^c \leftarrow\!\!\$ \mathbb{G}\}$ and $\mathsf{D}_1 = \{s \mid s \leftarrow\!\!\$ \{0,1\}^{512}\}$, simulating $\mathrm{G}_{\mathrm{D.5}}$ or $\mathrm{G}_{\mathrm{D.6}}$ depending on the challenge bit in its game and replacing the one instruction differing between $\mathrm{G}_{\mathrm{D.5}}$ and $\mathrm{G}_{\mathrm{D.6}}$ with the output of its challenge oracle. In a straightforward manner, we get

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{D.5}}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{G}_{\mathrm{D.6}}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathsf{LSB}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{LSB}}).$$

Further, applying the result of [FPSZ06, Theorem 7] with $k = 512$, $n = 2048$, $\ell = 2047$ we can write

$$\mathsf{Adv}^{\mathsf{LSB}}_{\mathbb{G},q}(\mathcal{D}_{\mathsf{LSB}}) \leq \mathsf{SD}(\mathsf{D}_0, \mathsf{D}_1) < 2^{-499},$$

where SD denotes the statistical distance.

$\mathrm{G}_{\mathrm{D.6}} \to \mathrm{G}_{\mathrm{D.7}}$. The next game, $\mathrm{G}_{\mathrm{D.7}}$, replaces the $\mathsf{SHA\text{-}1}(ak)$ call used to compute the “auth key auxiliary hash” ax and the “auth key id” aid with a random string as shown in Fig. 63.

Using an adversary $\mathcal{A}$ against the Multi-Stage-security of $\mathsf{MTP\text{-}KE}_{2st}$, we build an adversary $\mathcal{D}_{\mathsf{H}}$ that plays in the TOTPRF game against H as shown in Fig. 38, where H is instantiated as $\mathsf{H.Ev}(hk, x) := \mathsf{SHA\text{-}1}(x \parallel hk) = \mathsf{SHA\text{-}1}(ak)$. $\mathcal{D}_{\mathsf{H}}$ thus simulates $\mathrm{G}_{\mathrm{D.6}}$ or $\mathrm{G}_{\mathrm{D.7}}$ depending on the challenge bit in its game, and replaces the generation of ax and aid with the output of a call to its ROR oracle as

$$(ax_{\mathtt{I}}, aid_{\mathtt{I}}) \leftarrow \mathrm{ROR}(ak_{\mathtt{I}}[0:1536]),$$
$$(ax_{\mathtt{R}}, aid_{\mathtt{R}}) \leftarrow \mathrm{ROR}(ak_{\mathtt{R}}[0:1536]).$$

Both of these calls implicitly pick an unknown key $hk_{\mathtt{I}}$ resp. $hk_{\mathtt{R}}$, however this does not break the simulation since this key replaces $ak_{\mathtt{I}}[1536:2048]$ resp. $ak_{\mathtt{R}}[1536:2048]$ which is never used elsewhere. $\mathcal{D}_{\mathsf{H}}$ outputs its own guess as $b'_{\mathsf{chall}} \leftarrow b'_{\mathsf{test}} = b_{\mathsf{test}} \wedge lost = \mathtt{false}$ where b_{test} was sampled by $\mathcal{D}_{\mathsf{H}}$ at the beginning of the game, b'_{test} is the bit guess of $\mathcal{A}$ and the *lost* flag is set under the same conditions as in both games $\mathrm{G}_{\mathrm{D.6}}, \mathrm{G}_{\mathrm{D.7}}$. If b_{chall} denotes the challenge bit in the TOTPRF game, we have $\Pr\left[b'_{\mathsf{chall}} = 1 \;\middle|\; b_{\mathsf{chall}} = 1\right] = \Pr[\mathrm{G}_{\mathrm{D.6}}]$ and $\Pr\left[b'_{\mathsf{chall}} = 1 \;\middle|\; b_{\mathsf{chall}} = 0\right] = \Pr[\mathrm{G}_{\mathrm{D.7}}]$, hence

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{D.6}}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{G}_{\mathrm{D.7}}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathsf{TOTPRF}}_{\mathsf{SHA\text{-}1}}(\mathcal{D}_{\mathsf{H}}).$$

Finally, we can apply Proposition 1 to get

$$\mathsf{Adv}^{G_{D.6}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) \leq \mathsf{Adv}^{G_{D.7}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) + 2 \cdot \mathsf{Adv}^{\mathsf{OTPRF}}_{\mathsf{SHACAL\text{-}1}}(\mathcal{D}_{\mathsf{OTPRF}}).$$

$\mathbf{G}_{D.7}$. The games $G_{D.7}$ and $G_{D.6}$ have together replaced both parts of the tested session key with random values, so that the response of the TEST oracle is indistinguishable by definition. We have

$$\mathsf{Adv}^{G_{D.7}}_{\mathsf{MTP\text{-}KE}_{2st}}(\mathcal{A}) = 0.$$

H.2 Proof for the three-stage protocol

Here, we provide a proof for Theorem 2.

Proof. As in the two-stage proof, we proceed via a sequence of games. Since large parts of the proof of Theorem 1 apply, we reference the relevant transitions without repeating the arguments here.

$\mathbf{G}_0$. The game G_0 is equivalent to the game $G^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MTP\text{-}KE}_{3st},\mathcal{U}_{\mathsf{role}},\mathcal{A}}$, so we have

$$\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MTP\text{-}KE}_{3st},\mathcal{U}_{\mathsf{role}}}(\mathcal{A}) = \mathsf{Adv}^{G_0}_{\mathsf{MTP\text{-}KE}_{3st}}(\mathcal{A}) = 2 \cdot \Pr[G_0] - 1.$$

From now on, we omit displaying $\mathcal{U}_{\mathsf{role}}$ in the advantage terms.

$\mathbf{G}_0 \to \mathbf{G}_1$. The game G_1 follows the changes made in G_1 in the proof of Theorem 1.

To argue that in G_1, the predicate Sound is always satisfied, we additionally need to show:

1. *Partnered sessions must output the same session key.*
   This holds trivially since $\mathsf{sskey.3} = \mathsf{sskey.2}$.
2. *Mutual authentication.*
   Let s, s' denote two partnered sessions. Since $s.\mathsf{sid.3}, s'.\mathsf{sid.3}$ include the session identifiers of previous stages, they must match on the public key pk. They also include aid. Since $\mathsf{MTP\text{-}KE}_{3st}.\mathsf{KGen}_{\mathsf{sym}}(W)$ by definition outputs unique aid values with respect to a given user W and each such user can only be associated with a single pk, we must have $s.\mathsf{kid} = s'.\mathsf{kid}$.

As before, we have

$$\mathsf{Adv}^{G_0}_{\mathsf{MTP\text{-}KE}_{3st}}(\mathcal{A}) \leq \frac{\mathsf{n}_\mathsf{S}^2}{2^{384}} + \mathsf{Adv}^{G_1}_{\mathsf{MTP\text{-}KE}_{3st}}(\mathcal{A}).$$

$\mathbf{G}_1 \to \mathbf{G}_2$. The game G_2 additionally defines a set $\mathcal{M}$, which it uses to keep track of messages produced by honest initiators during the third stage of the protocol. Each entry is of the form $((W, aid), m')$ where W is the identity of the intended responder, aid is an identifier of the long-term symmetric key used in that session, and m_{in} is the message generated by the initiator as part of the third stage. The game G_2 sets the flag bad_1 and aborts whenever a responder session of a user W receives c_{bind} which recovers aid^* and m^*_{in} such that $((W, aid^*), m^*_{\mathsf{in}}) \notin \mathcal{M}$. We have

$$\Pr[G_1] - \Pr[G_2] \leq \Pr\left[\mathsf{bad}_1^{G_2}\right].$$

We will reduce this to a notion of plaintext unforgeability satisfied by $\mathsf{CHv1}$. More formally, given an adversary $\mathcal{A}$ playing in the game G_2, we will construct an adversary $\mathcal{A}_{\mathsf{EUF\text{-}CMA}}$ against the EUF-CMA-security of $\mathsf{CHv1}$ (Definition 17). When $\mathcal{A}$ calls NEWSECRET, the adversary does not generate any ak_{v1}, but it does generate the random identifiers aid in the same way that $\mathsf{MTP\text{-}KE}_{3st}.\mathsf{KGen}_{\mathsf{sym}}$ does.

The adversary modifies the simulated SEND oracle only during the third stage:

– On SEND(s.label, `continue`) for an uncorrupted initiator session s, it proceeds as in the original protocol, but instead of encrypting m_{in} itself, $\mathcal{A}_{\mathsf{EUF\text{-}CMA}}$ calls the oracle: $c_{\mathsf{in}} \leftarrow\$ \text{EVAL}((W, aid), (mid, m_{\mathsf{in}}))$.

- On $\text{SEND}(s.\mathsf{label}, c_{\mathsf{bind}})$ for an uncorrupted responder session s, it decrypts the first layer (the MTProto 2.0 channel) as in the original protocol. $\mathcal{A}_{\mathsf{EUF\text{-}CMA}}$ reconstructs the message $m_{\mathsf{in}} \leftarrow (n_{\mathsf{b}}, aid_{\mathsf{t}}, aid, sid_{\mathsf{t}}, exp)$ where n_{b}, aid, exp come from m_{bind} and $aid_{\mathsf{t}}, sid_{\mathsf{t}}$ come from the MTProto 2.0 channel state. Then, it uses its VFY oracle: $b \leftarrow \text{VFY}((W, aid), (mid, m_{\mathsf{in}}), c_{\mathsf{in}})$ where W is the session's owner and mid likewise comes from the MTProto 2.0 channel state. $\mathcal{A}_{\mathsf{EUF\text{-}CMA}}$ then accepts if and only if $b = 1$.

Whenever $\mathcal{A}$ causes bad_1 to be set due to some $(W, aid), m_{\mathsf{in}}) \notin \mathcal{M}$, when $\mathcal{A}_{\mathsf{EUF\text{-}CMA}}$ submits this to VFY, it wins in the EUF-CMA game. We have

$$\Pr\left[\mathsf{bad}_1^{G_2}\right] \leq \mathsf{Adv}_{\mathsf{CHv1}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{A}_{\mathsf{EUF\text{-}CMA}}),$$

and hence

$$\mathsf{Adv}_{\mathsf{MTP\text{-}KE}_{2\mathsf{st}}}^{G_1}(\mathcal{A}) \leq 2 \cdot \mathsf{Adv}_{\mathsf{CHv1}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{A}_{\mathsf{EUF\text{-}CMA}}) + \mathsf{Adv}_{\mathsf{MTP\text{-}KE}_{2\mathsf{st}}}^{G_2}(\mathcal{A}).$$

We now argue that in G_2, the predicate Auth is always satisfied. A break of implicit authentication would imply that a responder session accepted a message with an *aid* that was not found in $\mathcal{M}$, while a break of almost-full key confirmation for the server would imply that a responder session accepted a message with a modified sid_{t} or a replayed *exp* timestamp; both of these are now ruled out in G_2.[49]

$\mathbf{G}_2 \to \mathbf{G}_3$. The game G_3 only allows the adversary to submit a single TEST query. This transition follows the one made in G_2 in the proof of Theorem 1, since the two stages are equivalent from the point of view of TEST query simulation and the third stage in $\mathsf{MTP\text{-}KE}_{3\mathsf{st}}$ is not testable.

As before, we have

$$\mathsf{Adv}_{\mathsf{MTP\text{-}KE}_{3\mathsf{st}}}^{G_2}(\mathcal{A}) \leq 2n_{\mathsf{S}} \cdot \mathsf{Adv}_{\mathsf{MTP\text{-}KE}_{3\mathsf{st}}}^{G_3}(\mathcal{A}).$$

From now on, the proof proceeds exactly as for $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$. This is because the changes in stage 1 and stage 2 of $\mathsf{MTP\text{-}KE}_{3\mathsf{st}}$ are superficial, and do not affect the security reductions: in stage 1 the message m_0 has an extra field *exp*, and the stage 2 session key contains a *subset* of the bits used in $\mathsf{MTP\text{-}KE}_{2\mathsf{st}}$. Finally, the addition of stage 3 does not affect the reductions either, since the simulations never reach that point in the execution. This appears as if it could arise in Case D, in the transition to $G_{D.2}$, where we need to argue about the independence of various uses of n_n; stage 3 adds another such use, since $n_n[0:64]$ is used to set the first $\mathsf{server\_salt}$ for the MTProto 2.0 channel. However, the analysis only considers the execution up to the event bad_2 being set in $G_{D.2'}$ resp. $G_{D.2^*}$, which occurs during stage 2. Viewed differently, the use of $n_n[0:64]$ after stage 2 accepts does not create a difference since a REVEAL query could always be issued for the stage 1 key at this point.

I The brittle monolith that is Telegram

In theory, the design of a cryptographic protocol has the sole purpose of achieving the protocol's security goals efficiently. In actuality, however, to achieve this goal it must also achieve the goal of allowing at least a sufficiently motivated expert to convince themselves that the protocol achieves these goals. In other words, the central insight of what is commonly referred to as "modern cryptography" is that a cryptographic design is also tasked with being easy to reason about. A fundamental paradigm of achieving this goal is modularity, where different components of the design can be reasoned about in isolation and then (generically) composed to establish overall security guarantees. This modularity is typically achieved by relying on building blocks that provide strong security guarantees on their own (as opposed to only and potentially in specific compositions) and by breaking the dependency between different components of a protocol by avoiding re-use of secret material.

Telegram's failure to achieve this design goal is the root cause for the limitations and complexity of our proofs and our seeming need to reach for unstudied assumptions on cryptographic building blocks

[49] Note that the above reduction does not directly handle replays, as replayed messages would be in the set $\mathcal{M}$. However, replays are already prevented by the protocol itself in G_2 as well as G_1 by an explicit timestamp check.

than would otherwise be necessary. We will now discuss these issues and highlight several of the main Telegram design choices and their effect on our proofs of security. We begin with mere complications, then move on to limitations and seemingly necessary ad-hoc assumptions. We finish by briefly recapping our hypothetical attack. We also discuss design choices that led to these issues and note that the same design choice often lead to several different difficulties for arguing for the security of Telegram, leading to necessary repetitions in what follows.

I.1 Proof complications

Several design choices made by Telegram introduced many otherwise avoidable complications in our proofs.

Lack of a suitable key schedule. Recall that n_n is passed into SKDF, into NH, and partially XORed with n_s to form $\mathsf{server\_salt}$.[50] These three uses of n_n are across three different $\textsc{Send}$ calls, rendering it impossible to replace values one-by-one with random values and appealing to some PRF notion to justify the changes. If instead n_n had been used solely as an input to SKDF to produce pseudorandom values, with these values replacing the three uses of n_n, then a significantly simpler proof would have been obtainable.

Similarly, the two values ax and aid are both the result of a single $\mathsf{SHA\text{-}1}$ call, which prevents the proof from manipulating them independently.

Use of a (truncated) weak hash function. Although more efficient and secure alternatives such as SHA-256 and SHA3 exist, Telegram uses the now mostly deprecated SHA-1 algorithm. SHA-1 has been shown not to be collision resistant via practical attacks [SBK$^+$17, LP20]. The use of SHA-1 to compute the key confirmation hash h complicates our proof. If a collision-resistant hash function had been used, we could have relied on this property in the first step of the proof to establish public session matching.

Further, in the calls of $\mathsf{SHA\text{-}1}(ak)$ in Fig. 12, the output of the SHA-1 hash is truncated to only 64 bits. This prevents us from using a simple PRF notion due to easy attacks even in the one-time PRF setting.

Short session identifiers. The 64-bit value output of the above-mentioned truncated hash function is *aid*. This value is used by the Telegram servers to identify sessions. On the one hand, this imposes a hard bound of 2^{64} on the number of sessions each responder can accept. On the other hand, the shortness of the value suggests that collisions between session state identifiers are likely, which complicates the proof. A longer value, even of 128 bits, would have allowed for a simpler proof.

Lack of ciphertext integrity. Telegram's MTProto relies on a custom mode of operation composing IGE-mode and SHA-1. The composition achieves neither INT-CTXT nor IND-CCA [JO16]. Had an established authenticated encryption scheme or an unforgable MAC been used, this would have simplified the proof in allowing us to declare the Diffie-Hellman shares authenticated and using the ciphertext/mac tag as part of our session identifiers. This in turn would have enabled public session matching based on transcripts.

Reliance on plaintext checking. Our proof relies on the correctness of a complex parsing behaviour and the checking of various plaintext headers and nonce values. That is, we also could not achieve modularity separating cryptographic operations and higher-level protocol operations.

In particular, for the soundness of Theorems 1 and 2 we require that all message headers are different, so there cannot be confusion about which state the protocol is in and role confusion is also ruled out. At the lowest level, to prove a property used by our integrity proof for $\mathsf{CHv1}$ (Proposition 7), we rely on the fact that SEv1 in Fig. 8 checks fixed parts of the plaintext. We also rely on this checking behaviour in the integrity proof itself (Proposition 8) in the transition between G_4 and G_5, where it allows us to rewrite the game with a different order of operations.

[50] We also note that n_n is misnamed as a nonce, since it is used as a key.

I.2 Limitations of our proof

As discussed in Section 4.5, the main limitation of our proof is that we do not model the actual connection between the initial run of $\mathsf{MTP\text{-}KE_{3st}}$ and subsequent runs of $\mathsf{MTP\text{-}KE_{2st}}$. Moreover, our model does not allow for generic composition of our theorems about $\mathsf{MTP\text{-}KE_{3st}}$ and existing results about the channel $\mathsf{MTP\text{-}CH}$. This is due to several design choices made by Telegram that prevent simple composition of the security proofs.

Key dependence. While being composed of multiple stages, $\mathsf{MTP\text{-}KE_{3st}}$ does not derive the keys in the different stages independently. This prevents us from using general composition results on key exchanges and secure channels (in the style of e.g. [Gün18]) to argue about the security of $\mathsf{MTP\text{-}KE_{3st}}$ when used in conjunction with $\mathsf{MTP\text{-}CH}$ (as analysed in [AMPS22]).

Another example is the fact that the DH value in $\mathsf{MTP\text{-}KE_{2st}}$ is used to internally derive ax and aid, and is used afterwards in $\mathsf{MTP\text{-}KE_{3st}}$ as an encryption key ak. Instead, if the DH value had been used as an input to a KDF to derive ax, aid and ak as (computationally) independent keys, a composition result would be more feasible to achieve.

Public key reuse. We do not model the fact that the public key pk of the server is used in both $\mathsf{MTP\text{-}KE_{2st}}$ and $\mathsf{MTP\text{-}KE_{3st}}$. To model this, a proof would have to consistently update it across two different games simultaneously. Using different independent keys would have allowed us to treat the two protocols separately without essentially assuming the co-dependence away.

Lack of key confirmation. We were unable to prove key confirmation for $\mathsf{MTP\text{-}KE_{2st}}$ and only proved key confirmation for the server in $\mathsf{MTP\text{-}KE_{3st}}$. Key confirmation would have been possible if h was produced using a secure MAC.

Direct use of non-uniform key material. As described in Section 4.3, MTProto uses bits of the agreed DH values directly as key material instead of using them as an input to a key derivation function. However, the existing proof for MTProto [AMPS22] assumes a uniform key distribution. This prevents us from composing our results with those of [AMPS22]. Moreover, this forces us to use a session key distribution for stage 2 which is not the uniform distribution on strings of a given size (see Section 4.3).

Retry handling. In general, it is difficult to reason about the security of a protocol without knowing the total number of exchanged messages. For example, the security bound for INT-PTXT depends on the number of encryption and decryption queries, which in turn depends on the number of retries. Two aspects of the protocol design prevent us from making an argument that the number of retries would be bounded in practice. First, there is a question of preventing adversarially-triggered retries: this would necessitate showing that NH outputs are unforgeable, which is not possible due to its short input length. Second, even if the adversary was not able to directly manipulate the flow of the protocol, it remains in control of creating new sessions, which in turn influences the size of each server's $\mathcal{S}_{\mathsf{aid}}$ that determines the likelihood of an honest retry. Thus, we were forced to assume a maximum retry number $\mathsf{rid}_{\mathsf{max}}$.

I.3 Reliance on unstudied assumptions

In Appendix C we describe several unstudied ad-hoc and new assumptions that we used in our proofs. These assumptions could have been avoided if collision-resistant hash functions (e.g. SHA-256 or SHA3) had been used instead of SHA-1 and if proper key derivation functions had been used.

We can view these assumptions as part of two groups, based on their plausibility and impliciations if they were invalidated. The first two ($\mathsf{4PRF}, \mathsf{3TPRF}$) are lower-level, expressing a pseudorandomness property of $\mathsf{SHACAL\text{-}1}$: they appear plausible due to the large key length of $\mathsf{SHACAL\text{-}1}$, but symmetric cryptanalysis would be needed to determine the concrete reduction in advantage compared to the known results on $\mathsf{SHACAL\text{-}1}$ without leakage. The remaining three ($\mathsf{SPR}, \mathsf{UPCR}, \mathsf{IND\text{-}KEY}$) are higher-level, expressing properties of SHA-1 that are variants of standard assumptions or more novel: however, it appears that breaking either of these would not be sufficient to break the key exchange protocol; there exist versions of these assumptions which if broken would be sufficient to break the protocol, but they place even stricter constraints on the adversary.

I.4 A hypothetical attack

Weak channel binding. In Section 4.5, we describe an attack on client authentication that is based on the way that a new temporary key is bound to the long-term authentication key *ak*. The attack exploits the fact that the Telegram server used to not verify the expiration time sent in the binding message. Although Telegram has addressed this specific issue by enforcing the check, the design choice to rely on such checks for session binding is brittle, and its security depends on nuanced details related to the way session key management and expiration are implemented. Instead, more robust cryptographic approaches can be used to bind between the sessions that generate the new temporary key and *ak*. For example, one approach is to calculate a MAC over the transcript of the current session's handshake using a key derived from *ak* as the MAC key.