On the Concrete Security of TLS 1.3 PSK Mode

Hannah Davis[*] Denis Diemert[†] Felix Günther[‡] Tibor Jager[§]

August 27, 2023

Abstract

The pre-shared key (PSK) handshake modes of TLS 1.3 allow for the performant, low-latency resumption of previous connections and are widely used on the Web and by resource-constrained devices, e.g., in the Internet of Things. Taking advantage of these performance benefits with optimal and theoretically-sound parameters requires tight security proofs. We give the first tight security proofs for the TLS 1.3 PSK handshake modes.

Our main technical contribution is to address a gap in prior tight security proofs of TLS 1.3 which modeled either the entire key schedule or components thereof as independent random oracles to enable tight proof techniques. These approaches ignore existing interdependencies in TLS 1.3's key schedule, arising from the fact that the same cryptographic hash function is used in several components of the key schedule and the handshake more generally. We overcome this gap by proposing a new abstraction for the key schedule and carefully arguing its soundness via the indifferentiability framework. Interestingly, we observe that for one specific configuration, PSK-only mode with hash function SHA-384, it seems difficult to argue indifferentiability due to a lack of domain separation between the various hash function usages. We view this as an interesting insight for the design of protocols, such as future TLS versions.

For all other configurations however, our proofs significantly tighten the security of the TLS 1.3 PSK modes, confirming standardized parameters (for which prior bounds provided subpar or even void guarantees) and enabling a theoretically-sound deployment.

[*] Department of Computer Science & Engineering, University of California San Diego, La Jolla, CA, USA. Email: h3davis@eng.ucsd.edu. URL: https://cseweb.ucsd.edu/~h3davis/. Supported in part by NSF grant CNS-1717640. Some of this work was done while Hannah Davis was visiting ETH Zurich.

[†] Chair for IT Security and Cryptography, Bergische Universität Wuppertal, Wuppertal, Germany. Email: denis.diemert@uni-wuppertal.de. URL: https://itsc.uni-wuppertal.de/en/group-members/denis-diemert.html.

[‡] Department of Computer Science, ETH Zürich, Zürich, Switzerland. Email: mail@felixguenther.info. URL: https://www.felixguenther.info. Supported in part by Research Fellowship grant GU 1859/1-1 of the German Research Foundation (DFG).

[§] Chair for IT Security and Cryptography, Bergische Universität Wuppertal, Wuppertal, Germany. Email: tibor.jager@uni-wuppertal.de. URL: https://www.tiborjager.de. Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement 802823.

A preliminary version of this paper appears in the proceedings of the *41st Annual International Confer*
on the Theory and Applications of Cryptology and Information Security (Eurocrypt 2022), DOI: 10.1007/9
3-030-78375-4_18. This is the full version.

Contents

1 Introduction **4**

2 The TLS 1.3 Pre-shared Key Handshake Protocol **7**

3 Code-based MSKE Model for PSK Modes **10**
3.1 Key Exchange Syntax 10
3.2 Key Exchange Security 11

4 Indifferentiability Background **15**

5 Key-Schedule Indifferentiability **17**
5.1 Indifferentiability for the TLS 1.3 Key Schedule in Three Steps 18
5.1.1 Step 1: Domain-separating the Transcript Hash 19
5.1.2 Step 2: Applying the Indifferentiability of HMAC 21
5.1.3 Step 3: Applying Indifferentiability to the TLS Key Schedule 22

6 Modularizing Handshake Encryption **26**
6.1 Handshake Encryption as a Modular Transformation 26

7 Tight Security of the TLS 1.3 PSK Modes **30**
7.1 TLS 1.3 PSK-only/PSK-(EC)DHE as a MSKE Protocol 30
7.2 Tight Security Analysis of TLS 1.3 PSK-(EC)DHE 32
7.3 Full Security Bound for TLS 1.3 PSK-(EC)DHE and PSK-only 46

8 Evaluation **48**
8.1 Evaluation Details 48

A Collision Resistance of Random Oracles **55**

B A Careful Discussion of Domain Separation **55**
B.1 PSK-only mode with `SHA256` 58
B.2 Pre-shared key with Diffie–Hellmann mode with `SHA256` 59
B.3 Pre-shared key with Diffie–Hellmann mode with `SHA384` 60
B.4 PSK-only mode with `SHA384` 61
B.5 Repairing Domain Separation for TLS 1.3-like Protocols 62

1 Introduction

The *Transport Layer Security* (TLS) protocol is probably the most widely-used cryptographic protocol. It provides a secure channel between two endpoints (*client* and *server*) for arbitrary higher-layer application protocols. Its most recent version, TLS 1.3 [54], specifies two different "modes" for the initial handshake establishing a secure session key: the main handshake mode based on a Diffie–Hellman key exchange and public-key authentication via digital signatures, and a *pre-shared key* (PSK) mode, which performs authentication based on symmetric keys. The latter is mainly used for two purposes:

Session resumption. Here, a prior TLS connection established a secure channel along with a pre-shared key PSK, usually via a full handshake. Subsequent TLS resumption sessions use this key for authentication and key derivation. For example, modern web browsers typically establish multiple TLS connections when loading a web site. Using public-key authentication only in an initial session and PSK-mode in subsequent ones minimizes the number of relatively expensive public-key computations and significantly improves performance for both clients and servers.

Out-of-band establishment. PSKs can also be established out-of-band, e.g., by manual configuration of devices or with a separate key establishment protocol. This enables secure communication in settings where a complex public-key infrastructure (PKI) is unsuitable, such as IoT applications.

TLS 1.3 provides two variants of the PSK handshake mode: *PSK-only* and *PSK-(EC)DHE*. The PSK-only mode is purely based on symmetric-key cryptography. This makes TLS accessible to resource-constrained low-cost devices, and other applications with strict performance requirements, but comes at the cost of not providing *forward secrecy* [34], since the latter is not achievable with static symmetric keys.[1] The PSK-(EC)DHE mode in turn achieves forward secrecy by additionally performing an (elliptic-curve) Diffie–Hellman key exchange, authenticated via the PSK (i.e., still avoiding inefficient public-key signatures). This compromise between performance and security is the suggested choice for TLS 1.3 session resumption on the Internet.

Concrete security and tightness. Classical, complexity-theoretic security proofs considered the security of cryptosystems *asymptotically*. They are satisfied with security reductions running in polynomial time and having non-negligible success probability. However, it is well-known that this only guarantees that a sufficiently large security parameter exists *asymptotically*, but it does not guarantee that a deployed real-world cryptosystem with standardized parameters—such as concrete key lengths, sizes of algebraic groups, moduli, etc.—can achieve a certain expected security level. In contrast, a *concrete security* approach makes all bounds on the running time and success probability of adversaries explicit, for example, with a bound of the form

$$\mathrm{Adv}(\mathcal{A}) \leq f(\mathcal{A}) \cdot \mathrm{Adv}(\mathcal{B}),$$

where f is a function of the adversary's resources and $\mathcal{B}$ is an adversary against some underlying cryptographic hardness assumption.

The concrete security approach makes it possible to determine concrete deployment parameters that are supported by a formal security proof. As an intuitive toy example, suppose we want to achieve "128-bit security", that is, we want a security proof that guarantees (for any $\mathcal{A}$ in a certain class of adversaries) that $\mathrm{Adv}(\mathcal{A}) \leq 2^{-128}$. Suppose we have a cryptosystem with a reduction that loses "40 bits of security" because we can only prove a bound of $f(\mathcal{A}) \leq 2^{40}$. This means that we have to instantiate the scheme with an underlying hardness assumption that achieves $\mathrm{Adv}(\mathcal{B}) \leq 2^{-168}$ for any $\mathcal{B}$ in order to upper bound $\mathrm{Adv}(\mathcal{A})$ by 2^{-128} as desired. Hence, the 40-bit security loss of the bound is compensated by larger parameters that provide "168-bit security".

This yields a theoretically-sound choice of deployment parameters, but it might incur a very significant performance loss, as it requires the choice of larger groups, moduli, or key lengths. For example, the size of an elliptic curve group scales quadratically with the expected bit security, so we would have to choose $|\mathbb{G}| \approx 2^{2 \cdot 168} = 2^{336}$ instead of the optimal $|\mathbb{G}| \approx 2^{2 \cdot 128} = 2^{256}$. The performance penalty is even more significant for finite field groups, RSA or discrete logarithms "modulo p". This could lead to parameters

[1] See [2, 10] for recent work discussing symmetric key exchange and forward secrecy.

which are either too large for practical use, or too small to be supported by the formal security analysis of the cryptosystem. We demonstrate this below for security proofs of TLS.

Even worse, for a given security proof the concrete loss ℓ may not be a constant, as in the above example, but very often ℓ depends on other parameters, such as the number of users or protocol sessions, for example. This makes it difficult to choose theoretically-sound parameters when bounds on these other parameters are not exactly known at the time of deployment. If then a concrete value for ℓ is estimated too small (e.g., because the number of users is underestimated), then the derived parameters are not backed by the security analysis. If ℓ is chosen too large, then it incurs an unnecessary performance overhead.

Therefore we want to have *tight* security proofs, where ℓ is a small constant, independent of any parameters that are unknown when the cryptosystem is deployed. This holds in particular for cryptosystems and protocols that are designed to maximize performance, such as the PSK modes of TLS 1.3 for session resumption or resource-constrained devices.

Previous analyses of the TLS handshake protocol and their tightness. TLS 1.3 is the first TLS version that was developed in a close collaboration between academia and industry. Early TLS 1.3 drafts were inspired by the OPTLS design by Krawczyk and Wee [47], and several draft revisions as well as the final TLS 1.3 standard in RFC 8446 [54] were analyzed by many different research groups, including computational/reductionist analyses of the full and PSK modes in [22, 24, 29, 25]. All reductions in these papers are however highly non-tight, having up to a quadratic security loss in the number of TLS sessions and adversary can interact with. For example, [19] explains that for "128-bit security" and plausible numbers of users and sessions, an RSA modulus of more than 10,000 bits would be necessary to compensate the loss of previous security proofs for TLS, even though 3072 bits are usually considered sufficient for "128-bit security" when the loss of reductions is not taken into account. Likewise, [15] argues that the tightness loss to the underlying Diffie–Hellman hardness assumption lets these bounds fail to meet the standardized elliptic curves' security target, and for large-scale adversary even yields completely vacuous bounds.

Recently, Davis and Günther [15] and Diemert and Jager [19] gave new, tight security proofs for the TLS 1.3 full handshake based on Diffie–Hellman key exchange and digital signatures (not PSKs). However, their results required very strong assumptions. One is that the underlying digital signature scheme is tightly secure in a multi-user setting with adaptive corruptions. While such signature schemes do exist [3, 33, 18, 36], this is not known for any of the signature schemes standardized for TLS 1.3, which are subject to the tightness lower bounds of [4] as their public keys uniquely determine the matching secret key.

Even more importantly, both [15] and [19] modeled the TLS key schedule or components thereof as *independent* random oracles. This was done to overcome the technical challenge that the Diffie–Hellman secret and key shares need to be *combined* in the key derivation to apply their tight security proof strategy, following Cohn-Gordon et al. [13], yet in TLS 1.3 those values enter key derivation through *separate* function calls. But neither work provided formal justification for their modeling, and both neglected to address potential dependencies between the use of a hash function in the key schedule and elsewhere in the protocol.

Our contributions

In this paper, we describe a new perspective on TLS 1.3, which enables a modular security analysis with tight security proofs.

New abstraction of the TLS 1.3 key schedule. We first describe a new abstraction of the TLS 1.3 key schedule used in the PSK modes (in Section 2), where different steps of the key schedule are modeled as *independent* random oracles (12 random oracles in total). This makes it significantly easier to rigorously analyze the security of TLS 1.3, since it replaces a significant part of the complexity of the protocol with what the key schedule intuitively provides, namely "as-good-as-independent cryptographic keys", deterministically derived from pre-shared keys, Diffie–Hellman values (in PSK-(EC)DHE mode), protocol messages, and the randomness of communicating parties.

Most importantly, in contrast to prior works on TLS 1.3's tightness that abstracted (parts of or the entire) key schedule as random oracles [19, 15] to enable the tight proof technique of Cohn-Gordon et al. [13], we support this new abstraction formally. Using the *indifferentiability* framework of Maurer et al. [51] in its recent adaptation by Bellare et al. [5] that treats *multiple* random oracles, in Section 5 we prove

our abstraction *indifferentiable* from TLS 1.3 with *only* the underlying cryptographic hash function modeled as a random oracle, and this proof is *tight*. This accounts for possible interdependencies between the use of a hash function in multiple contexts, which were not considered in [19, 15].

Identifying a lack of domain separation. A noteworthy subtlety is that, to our surprise, we identify that for a certain choice of TLS 1.3 PSK mode and hash function (namely, PSK-only mode with SHA384), a lack of *domain separation* [5] in the protocol does *not* allow us to prove indifferentiability for this case. We discuss the details of why domain separation is achieved for all but this case in Appendix B.

This gap could be closed by more careful domain separation in the key schedule, which we consider an interesting insight for designers of future versions of TLS or other protocols. Concretely, the ideal domain separation method would be to add a unique prefix or suffix to each hash function call made by the protocol. However, existing standard primitives like HMAC and HKDF do not permit the use of such labels, so this advice is not practical for TLS 1.3 or similar protocols. For these, a combination of labels (where possible) and padding for domain separation seems advisable, where the padding ensures that the protocol's direct hash calls have strictly longer inputs than the internal hash calls in HMAC and HKDF. We outline this method in more detail in Appendix B.5.

Modularization of record layer encryption. Like most of the prior computational TLS 1.3 analyses [22, 29, 25, 19], we use a *multi-stage key exchange* (MSKE) security model [28] to capture the complex and fine-grained security aspects of TLS 1.3. These aspects include cleverly distinguishing between "external" keys established in the handshake for subsequent use (by, e.g., application data encryption, resumption, etc.) and "internal" keys, used within the handshake itself (in TLS 1.3 for encrypting most of the handshake through the protocol's record layer) to avoid complex security models such as the ACCE model [38] which monolithically treat handshake and record-layer encryption.

As a generic simplification step for MSKE models, we show (in Section 6) that for a certain class of *transformations* using the internal keys, we can even avoid the somewhat involved handling of internal keys altogether. We use this to simplify our analysis of the TLS 1.3 handshake (treating the TLS 1.3 record-layer encryption as such transformation). The result itself however is not specific to TLS 1.3, but general and of independent interest; it furthermore is *tight*.

Tight security of TLS 1.3 PSK modes. We leverage the new perspective on the TLS 1.3 key schedule and the fact that we can ignore record-layer encryption to give our main results: the first *tight* security proofs for the PSK-only and PSK-(EC)DHE handshake modes of TLS 1.3.

Evaluation. Finally, we evaluate our new bounds and prior ones from [25] over a wide range of fully concrete resource parameters, following the approach of Davis and Günther [15]. Our bounds improve on previous analyses of the PSK-only handshake by between 15 and 53 bits of security, and those of the PSK-(EC)DHE handshake by 60 and 131 bits of security across all our parameters evaluated.

Further related work and scope of our analysis

Several previous works gave security proofs for the previous protocol version TLS 1.2 [38, 45, 31, 46, 49, 8], including its PSK-modes [49]; all reductions in these works are highly non-tight.

Brzuska et al. [11] recently proposed a stand-alone security model for the TLS 1.3 key schedule, likewise aiming at a new abstraction perspective on the latter to support formal protocol analysis. While their treatment focuses solely on the key schedule and only briefly argues its application to a key exchange security result, it is more general and covers the negotiation of parameters [26, 7] and agile usage of various algorithms.

Our focus is on the TLS 1.3 PSK modes. Hence, our abstraction of the key schedule and the careful indifferentiability treatment is tailored to that mode and cannot be directly translated to the full handshake (without PSKs). We are confident that our approach can be adapted to achieve similar results for the full handshake, but leave revisiting the results in [19, 15] in that way to future work.

Like many previous cryptographic analyses [38, 45, 22, 24, 29, 25, 19, 15] of the TLS handshake, our work focuses on the "cryptographic core" of the TLS 1.3 PSK handshake modes (in particular, we consider

fixed parameters like the Diffie–Hellman group, TLS ciphersuite, etc.). Our abstraction of the key schedule is designed for easy composition with our tight key exchange proof, and our indifferentiability treatment is important confirmation of that abstraction's soundness. We do not consider, e.g., ciphersuite and version negotiation [26] or backwards compatibility issues in settings where multiple TLS versions are used in parallel, such as [39]. We also do not treat the security of the TLS record layer; instead we explain how to avoid the necessity to do so in order to achieve more modular security analyses, and we refer to compositional results [28, 22, 35, 25, 19] treating the combined security when subsequent protocols use the session keys established in an MSKE protocol.

Numerous authenticated key exchange protocols [33, 13, 50, 37, 36] were recently proposed that can be proven (almost) tightly secure. However, these protocols were specifically designed to be tightly secure and none is standardized.

2 The TLS 1.3 Pre-shared Key Handshake Protocol

Overview. We consider the pre-shared key mode of TLS 1.3, used in a setting where both client and server already share a common secret, a so-called *pre-shared key* (PSK). A PSK is a cryptographic key which may either be manually configured, negotiated out-of-band, or (and most commonly) be obtained from a prior and possibly not PSK-based TLS session to enable fast *session resumption.* The TLS 1.3 PSK handshake comes in two flavors: PSK-only, where security is established from the pre-shared key alone, and PSK-(EC)DHE, which includes an (finite-field or elliptic-curve) Diffie–Hellman key exchange for added forward secrecy. Both PSK handshakes essentially consist of two phases (cf. Figure 1).

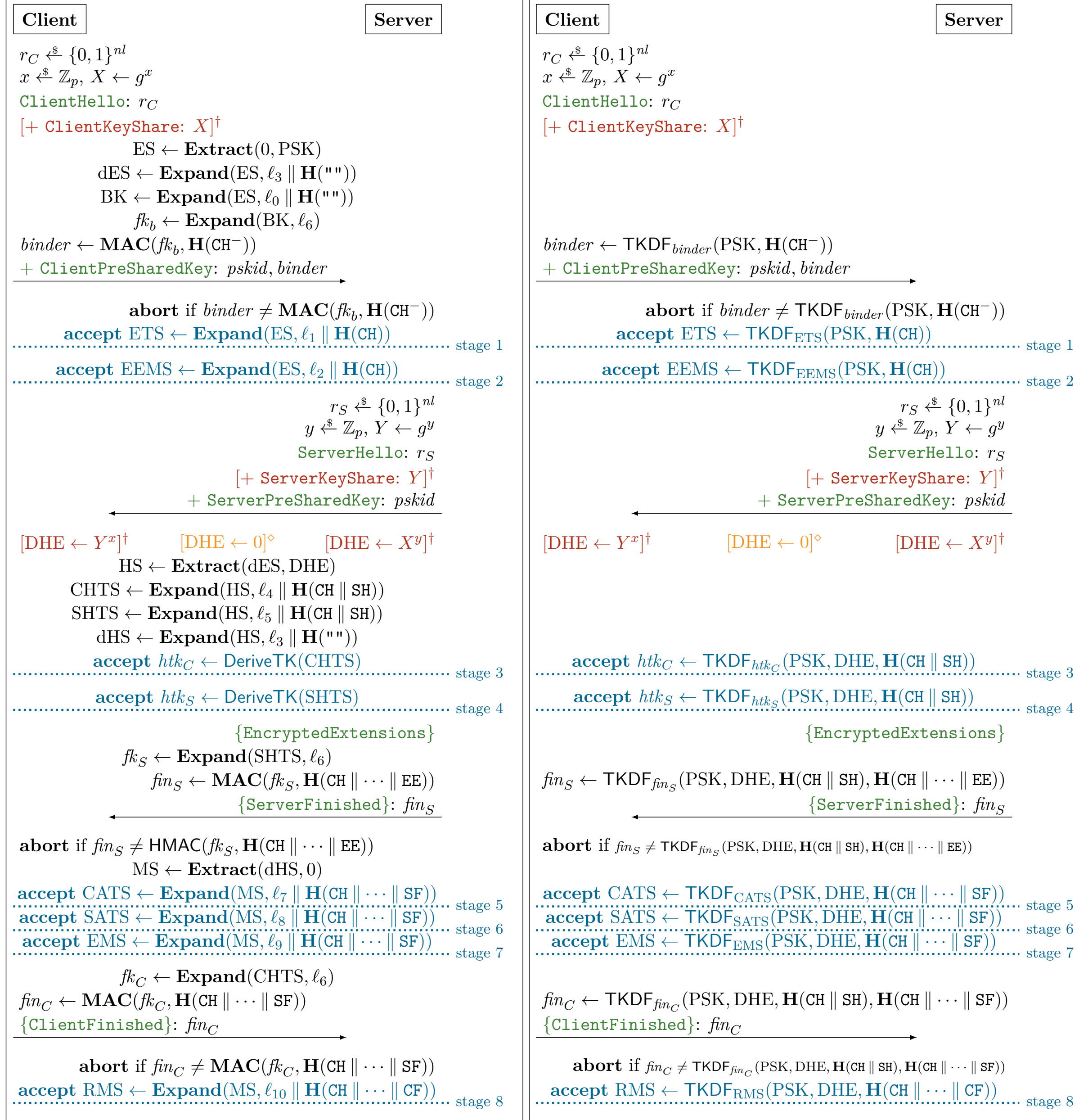
1. The client sends a random nonce and a list of offered pre-shared keys to the server, where each key is identified by a (unique) identifier *pskid*.[2] The server then selects one *pskid* from the list, and responds with another random nonce and the selected *pskid*. In PSK-(EC)DHE mode, client and server additionally perform a Diffie–Hellman key exchange, sending group elements along with the nonces and PSK identifiers. In both modes, the client also sends a so-called binder value, which applies a *message authentication code* (MAC) to the client's nonce and *pskid* (and the Diffie–Hellman share in PSK-(EC)DHE mode) and binds the PSK handshake to the (potential) prior handshake in which the used pre-shared key was established (see [14, 42] for analysis rationale behind the binder value).
2. Then client and server derive *unauthenticated* cryptographic keys from the PSK and the established Diffie–Hellman key (the latter only in (EC)DHE mode, of course). This includes, for instance, the *client* and *server handshake traffic keys* (htk_C and htk_S) used to encrypt the subsequent handshake messages, as well as *finished keys* (fk_C and fk_S) used to compute and exchange *finished messages.* The finished messages are MAC tags over all previous messages, ensuring that client and server have received all previous messages exactly as they were sent.

   After verifying the finished messages, client and server "accept" *authenticated* cryptographic keys, including the *client* and *server application traffic secret* (CATS and SATS), the *exporter master secret* (EMS), and the *resumption master secret* (RMS) for future session resumptions.

Detailed specification. For our proofs we will need fully-specified descriptions for each of the TLS 1.3 PSK and PSK-(EC)DHE handshake protocols. Pseudocode for these protocols can be found in Figure 1, where we let $(\mathbb{G}, p, g)$ be a cyclic group of prime order p such that $\mathbb{G} = \langle g \rangle$.

The two descriptions on the left and right in Figure 1 show the same protocol, but they use different abstractions to highlight how we capture the complex way TLS 1.3 calls its hash function. This one hash function is used in some places to condense transcripts, in others to help derive session keys, and in still others as part of a message authentication code. We call this function $\mathbf{H}$, and let its output length be hl bits so that we have $\mathbf{H}\colon \{0,1\}^* \to \{0,1\}^{hl}$. Depending on the choice of ciphersuite, TLS 1.3 instantiates $\mathbf{H}$ with either `SHA256` or `SHA384` [52]. In our security analysis, we will model $\mathbf{H}$ as a random oracle.

[2] In this work, we do not consider negotiation of pre-shared keys in situations where client and server share multiple keys, but focus on the case where client and server share only one PSK and the client therefore offers only a single *pskid*. However, we expect that our results extend to the general case as well.



Legend

MSG: Y	message MSG sent, containing Y	CH$^-$	partial ClientHello up to (incl.) *pskid*
+ MSG	extension sent within previous message	ℓ_x	label value, distinct for distinct x
{MSG}	MSG sent AEAD-encrypted with htk_C/htk_S		
[...]†	present only in PSK-(EC)DHE		
[...]$^\diamond$	present only in PSK		

DeriveTK(HTS) := **Expand**(HTS, ℓ_{11} ‖ Th(""), hl) ‖ **Expand**(HTS, ℓ_{12} ‖ Th(""), ivl)
(traffic key computation, deriving a hl-bit key and a ivl-bit IV)

Figure 1: TLS 1.3 PSK and PSK-(EC)DHE handshake modes with (optional) 0-RTT keys (stages 1 and 2), with detailed key schedule (left) and our representation of the key schedule through functions TKDF$_x$ (right), explained in the text. Centered computations are executed by both client and server with their respective messages received, and possibly at different points in time. Dotted lines indicate the derivation of session (stage) keys together with their stage number. The labels ℓ_x are distinct for distinct index x, see Table 1 for their definition.

Value	Label	Value	Label
dES	$\ell_3 =$ `"derived"`	htk_C	$\ell_{11} =$ `"key"` & $\ell_{12} =$ `"iv"`
BK	$\ell_0 =$ `"ext binder"` / `"res binder"`	htk_S	$\ell_{11} =$ `"key"` & $\ell_{12} =$ `"iv"`
fk_b	$\ell_6 =$ `"finished"`	fk_S	$\ell_6 =$ `"finished"`
ETS	$\ell_1 =$ `"c e traffic"`	CATS	$\ell_7 =$ `"c ap traffic"`
EEMS	$\ell_2 =$ `"e exp master"`	SATS	$\ell_8 =$ `"s ap traffic"`
CHTS	$\ell_4 =$ `"c hs traffic"`	EMS	$\ell_9 =$ `"exp master"`
SHTS	$\ell_5 =$ `"s hs traffic"`	fk_C	$\ell_6 =$ `"finished"`
dHS	$\ell_3 =$ `"derived"`	RMS	$\ell_{10} =$ `"res master"`

Table 1: Definitions of the short labels used in Figure 1. We simplify the labeling of **Expand** in our presentation. In the specification each **Expand** is not only labeled by $\ell \parallel H$ for some label ℓ and some hash H, but it is prefixed by the output length of the respecitive **Expand** call and the constant label "`tls13` ". As the output length for all of the above calls is equal (namely, the output length hl of $\mathbf{H}$), we leave this constant prefix out to reduce complexity.

On the left-hand side of Figure 1, we distinguish three named subroutines of TLS 1.3 which use $\mathbf{H}$ for different purposes:

- A message authentication code $\mathbf{MAC}\colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$, which calls $\mathbf{H}$ via the HMAC function $\mathbf{MAC}(K, M) := \mathsf{HMAC}[\mathbf{H}](K, M)$ where

$$\mathsf{HMAC}[\mathbf{H}](K, M) := \mathbf{H}((K \parallel 0^{bl-hl}) \oplus \mathsf{opad}) \parallel \mathbf{H}((K \parallel 0^{bl-hl} \oplus \mathsf{ipad}) \parallel M))$$

  Here opad and ipad are bl-bit strings, where each byte of opad and ipad is set to the hexadecimal value `0x5c`, resp. `0x36`. We have $bl = 512$ when `SHA256` is used and $bl = 512$ for `SHA384`. When modeling `SHA256` resp. `SHA384` as a random oracle, we keep the corresponding value of bl.

- $\mathbf{Extract}, \mathbf{Expand}\colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$, two subroutines for *extracting* and *expanding* key material in the key schedule, following the HKDF key derivation paradigm of Krawczyk [41, 44]. These functions are defined
  - $\mathbf{Extract}(K, M) := \mathsf{HKDF.Extract}(K, M) = \mathbf{MAC}(K, M)$.
  - $\mathbf{Expand}(K, M) := \mathsf{HKDF.Expand}(K, M) = \mathbf{MAC}(K, M \parallel \texttt{0x01})$.[3]

Despite the new naming conventions, this abstraction closely mimics the TLS 1.3 standard: **MAC**, **Extract**, and **Expand** can be read as more generic ways of referring to the HMAC, HKDF.Extract, and HKDF.Expand algorithms [43, 44].

The right-hand side of Figure 1 separates the key derivation functions for each first-class key as well as the binder and finished MAC values derived. This way of modeling TLS 1.3 makes it easier to establish key independence for the many keys computed in the key schedule, as we will see in Section 5. We introduce 11 functions TKDF_{binder}, $\mathsf{TKDF}_{\mathrm{ETS}}$, $\mathsf{TKDF}_{\mathrm{EEMS}}$, TKDF_{htk_C}, TKDF_{fin_C}, TKDF_{htk_S}, TKDF_{fin_S}, $\mathsf{TKDF}_{\mathrm{CATS}}$, $\mathsf{TKDF}_{\mathrm{SATS}}$, $\mathsf{TKDF}_{\mathrm{EMS}}$, and $\mathsf{TKDF}_{\mathrm{RMS}}$ (indexed by the value they derive) and use them to abstract away many intermediate computations. Note that we are not changing the protocol, though: we define each TKDF function to capture the same steps it replaces.

Take as an example TKDF_{fin_S}, the function used to derive the MAC in the `ServerFinished` message. In the prior abstraction, a session would first use the key schedule to derive a finished key fk_S from the hashed transcript and the secrets PSK and DHE. It would then call **MAC**, keyed with fk_S, to generate the

[3] HKDF.Expand [44] is defined for any output length (given as third parameter). In TLS 1.3, **Expand** always derives at most hl bits, which can be trimmed from a hl-bit output; we hence in most places omit the output length parameter.

$\mathsf{TKDF}_{\mathit{fin}_S}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{h}_1, \mathsf{h}_2)$:

```
1 ES ← Extract(0, PSK)
2 dES ← Expand(ES, ℓ3 ‖ Th(""))
3 HS ← Extract(dES, DHE)
4 SHTS ← Expand(HS, ℓ5 ‖ h1)
5 fk_S ← Expand(SHTS, ℓ6)
6 fin_S ← MAC(fk_S, h2)
7 return fin_S
```

Figure 2: Definition of $\mathsf{TKDF}_{\mathit{fin}_S}$, deriving the `ServerFinished` MAC.

`ServerFinished` message authentication code on the hashed transcript and encrypted extensions. Accordingly, we define $\mathsf{TKDF}_{\mathit{fin}_S} : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$ as in Figure 2. In the protocol, $\mathsf{TKDF}_{\mathit{fin}_S}$ takes inputs the pre-shared key PSK and Diffie–Hellman secret DHE and hash digests $\mathsf{h}_1 = \mathsf{Th}(\mathtt{CH} \,\|\, \mathtt{SH})$ and $\mathsf{h}_2 = \mathsf{Th}(\mathtt{CH} \,\|\, \cdots \,\|\, \mathtt{EE})$, and it outputs a MAC tag for the `ServerFinished` message. The remaining key derivation functions are defined the same way; we give their signatures below for completeness.

1. $\mathsf{TKDF}_{binder}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
2. $\mathsf{TKDF}_{\mathrm{ETS}}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
3. $\mathsf{TKDF}_{\mathrm{EEMS}}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
4. $\mathsf{TKDF}_{htk_C}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
5. $\mathsf{TKDF}_{\mathit{fin}_C}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
6. $\mathsf{TKDF}_{htk_S}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
7. $\mathsf{TKDF}_{\mathit{fin}_S}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
8. $\mathsf{TKDF}_{\mathrm{CATS}}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
9. $\mathsf{TKDF}_{\mathrm{SATS}}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
10. $\mathsf{TKDF}_{\mathrm{EMS}}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
11. $\mathsf{TKDF}_{\mathrm{RMS}}[\mathsf{RO}_{\mathsf{HMAC}}] \quad : \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$

Note that the definition of the 11 functions induces a lot of redundancy as we derive every value independently and therefore compute intermediate values (e.g., ES, dES, and HS) multiple times over the execution of the handshake. However, this is only conceptual. Since the computations of these intermediate values are deterministic, the intermediate values will be the same for the same inputs and could be cached.

3 Code-based MSKE Model for PSK Modes

We formalize security of the TLS 1.3 PSK modes in a game-based multi-stage key exchange (MSKE) model, adapted primarily from that of Dowling et al. [25]. We fully specify our model in pseudocode in Figures 3 and 4. We adopt the explicit authentication property from the model of Davis and Günther [15] and capture forward secrecy by following the model of Schwabe et al. [56].

3.1 Key Exchange Syntax

In our security model, the adversary interacts with *sessions* executing a key exchange protocol KE. For the definition of the security experiment it will be useful to have a unified, generic interface to the algorithms implementing KE, which can then be called from the various procedures defining the security experiment to run KE. Therefore, we first formalize a general syntax for protocols.

We assume that pairs of users share long-term symmetric keys (pre-shared keys), which are chosen

uniformly at random from a set KE.PSKS.[4] We allow users to share multiple pre-shared keys, maintained in a list pskeys, and require that each user uses any key only in a fixed role (i.e., as client *or* server) to avoid the Selfie attack [27]. We do not cover PSK negotiation; each session will know at the start of the protocol which key it intends to use.

New sessions are created via the algorithm Activate. This algorithm takes as input the new session's own user, identified by some ID u, the user ID *peerid* of the intended communication partner, a pre-shared key PSK, and a role identifier—initiator (client) or responder (server)—that determines whether the session will send or receive the first protocol message. It returns the new session π_u^i, which is identified by its user ID u and a unique index i so that a single user can execute many sessions.

Existing sessions send and receive messages by executing the algorithm Run. The inputs to Run are an existing session π_u^i and a message m it has received. The algorithm processes the message, updates the state of π_u^i, and returns the next protocol message m' on behalf of the session. Run also maintains the status of π_u^i, which can have one of three values: running when it is awaiting the next protocol message, accepted when it has established a session key, and rejected if the protocol has terminated in failure.

In a multi-stage protocol, sessions accept multiple session keys while running; we identify each with a numbered *stage*. A protocol may accept several stages/keys while processing a single message, and TLS 1.3 does this. In order to handle each stage individually, our model adds artificial pauses after each acceptance to allow the adversary to interact with the sessions upon each stage accepting (beyond, as usual, each message exchanged). When a session π_u^i accepts in stage s while executing Run, we require Run to set the status of π_u^i to $\mathsf{accepted}_s$ and terminate. We then define a special "continue" message. When session π_u^i in state $\mathsf{accepted}_s$, receives this message it calls Run again, updates its status to $\mathsf{running}_{s+1}$ and continues processing from the point where it left off.

3.2 Key Exchange Security

We define key exchange security via a real-or-random security game, formalized through Figures 3 and 4.

Game oracles. In this security game, the adversary $\mathcal{A}$ has access to seven oracles: Initialize, NewSecret, Send, RevSessionKey, RevLongTermKey, Test, and Finalize, as well as any random oracles the protocol defines. The game begins with a call to Initialize, which samples a challenge bit b. It ends when the adversary calls Finalize with a guess b' at the challenge bit. We say the adversary "wins" the game if Finalize returns true.

The adversary can establish a random pre-shared key between two users by calling NewSecret.[5] It can corrupt existing users' pre-shared keys via the oracle RevLongTermKey. The Send oracle creates new protocol sessions and processes protocol messages on the behalf of existing sessions. The RevSessionKey oracle reveals a session's accepted session key. Finally, the Test oracle servers as the challenge oracle: it returns the real session key of a target session or an independent one sampled randomly from the session key space KE.KS[s] of the respective stage s, depending on the value of the challenge bit b.

Protocol properties. Keys established in different stages possess different security attributes, which are defined as part of the key exchange protocol: replayability, forward secrecy level, and authentication level. Certain stages, whose indices are tracked in a list INT, produce "internal" keys intended for use only within the key exchange protocol; these keys may only be Tested at the time of acceptance of this particular key, but not later. This is because otherwise such keys may be trivially distinguishable from random, e.g., via trial decryption, due to the fact that they are used within the protocol. To avoid a trivial distinguishing attack, we force the rest of the protocol execution to be consistent with the result of such a Test. That is,

[4]While our results can be generalized to any distribution on KE.PSKS (based on its min-entropy), for simplicity, we focus on the uniform distribution in this work.

[5]Our model stipulates that pre-shared keys are sampled uniformly random and honestly. One could additionally allow the registration of biased or malicious PSKs, akin to models treating, e.g., the certification of public keys [9]. While this would yield a theoretically stronger model, we consider a simpler model reasonable, because we expect most PSKs used in practice to be random keys established in prior protocol sessions. Furthermore, we consider tightness as particularly interesting when "good" PSKs are used, since low-entropy PSKs might decrease the security below what is achieved by (non)-tight security proofs, anyway.

$G^{\mathsf{MSKE}}_{\mathsf{KE},\mathcal{A}}$

INITIALIZE:

```
1  time ← 0;
2  b ←$ {0,1}
```

NEWSECRET$(u, v, pskid)$:

```
3  time ← time + 1
4  if pskeys[(u, v, pskid)] ≠ ⊥
5     return ⊥
6  pskeys[(u, v, pskid)] ←$ KE.PSKS
7  revpsk_(u,v,pskid) ← ∞
8  return pskid
```

SEND(u, i, m):

```
9  time ← time + 1
10 if π_u^i = ⊥ then
11    (peerid, pskid, role) ← m
12    if role = initiator
13       then psk ← pskeys[(u, peerid, pskid)]
14       else psk ← pskeys[(peerid, u, pskid)]
15    (π_u^i, m') ←$ Activate(u, peerid, psk, role)
16 else
17    (π_u^i, m') ←$ Run(u, π_u^i.psk, π_u^i, m)
18 if π_u^i.status = accepted_{π_u^i.stage} then
19    stage ← π_u^i.stage
20    π_u^i.accepted[stage] ← time
21    if repr[π_u^i.sid[stage]] ≠ ⊥ then
22       π_u^i.skey[stage] ← repr[π_u^i.sid[stage]]
23    π_u^i.untampered[stage] ← ∃π_v^j with π_v^j.cid_{π_u^i.role}[stage] =
   π_u^i.cid_{π_u^i.role}[stage]
24 return m'
```

REVSESSIONKEY(u, i, s):

```
25 time ← time + 1
26 if π_u^i = ⊥ or π_u^i.accepted[s] = ∞ then
27    return ⊥
28 π_u^i.revealed[s] ← true
29 return π_u^i.skey[s]
```

REVLONGTERMKEY$(u, v, pskid)$:

```
30 time ← time + 1
31 revpsk_(u,v,pskid) ← time
32 return pskeys[(u, v, pskid)]
```

TEST(u, i, s):

```
33 time ← time + 1
34 if π_u^i = ⊥ or π_u^i.accepted[s] = ∞ or π_u^i.tested[s]
   then
35    return ⊥
36 if s ∈ INT
         and ∃π_v^j : π_v^j.sid[s] = π_u^i.sid[s]
         and π_v^j.accepted[s] < ∞
         and π_v^j.status ≠ accepted_s then
37    return ⊥
   // can only test internal keys if all sessions having accepted
   that key have not moved on with the protocol
38 π_u^i.tested[s] ← time
39 T ← T ∪ {(π_u^i, s)}
40 k_0 ← π_u^i.skey[s]
41 k_1 ←$ KE.KS[s]
42 if s ∈ INT then
      ∀π_v^j : π_v^j.sid[s] = π_u^i.sid[s]
         and π_v^j.status = accepted_s
43       π_v^j.skey[s] ← k_b
44    repr[π_u^i.sid[s]] ← k_b
45 return k_b
```

FINALIZE(b'):

```
46 if ¬Sound then
47    return 1
48 if ¬ExplicitAuth then
49    return 1
50 if ¬Fresh then
51    b' ← 0
52 return [[b = b']]
```

RO(i, X):

```
53 time ← time + 1
54 return RO_i(X)
```

Figure 3: Multi-stage key exchange (MSKE) security game for a key exchange protocol KE with pre-shared keys. Predicates Fresh, ExplicitAuth, and Sound are defined in Figure 4. The functions RO_i correspond to the (independent) random oracles available to the adversary.

Fresh:

1 for each $(\pi_u^i, s) \in \mathcal{T}$
2   $\mathsf{t_{Test}} \leftarrow \pi_u^i.\mathsf{tested}[s]$
3   if $\pi_u^i.\mathsf{revealed}[s]$ then
4   return false // tested session may not be revealed
5   if $\exists \pi_v^j \neq \pi_u^i : \pi_v^j.sid[s] = \pi_u^i.sid[s]$
    and $(\pi_v^j.\mathsf{tested}[s]$ or $\pi_v^j.\mathsf{revealed}[s])$ then
6   return false // tested session's partnered session may not be tested or revealed
7   if $\pi_u^i.\mathsf{accepted}[\mathsf{FS}[s, \mathsf{fs}]] < \mathsf{t_{Test}}$
8   if $\mathsf{revpsk}_{(u, \pi_u^i.peerid, \pi_u^i.pskid)} < \pi_u^i.\mathsf{accepted}[\mathsf{FS}[s, \mathsf{fs}]]$ and $\neg \pi_u^i.\mathsf{untampered}[\mathsf{FS}[s, \mathsf{fs}]]$ then
9   return false // Sessions with forward secrecy are fresh if they attained fs before their PSK was corrupted, or if they have a contributive partner (no tampering).
10   else if $\pi_u^i.\mathsf{accepted}[\mathsf{FS}[s, \mathsf{wfs2}]] < \mathsf{t_{Test}}$
11   if $\mathsf{revpsk}_{(u, \pi_u^i.peerid, \pi_u^i.pskid)}$ and $\neg \pi_u^i.\mathsf{untampered}[\mathsf{FS}[s, \mathsf{wfs2}]]$ then
12   return false // Sessions with weak forward secrecy 2 are fresh if the PSK was never corrupted, or if they have a contributive partner.
13   else if $\mathsf{revpsk}_{\{u, \pi_u^i.peerid\}, \pi_u^i.pskid}$ then
14   return false // Sessions with no forward secrecy are fresh if the PSK was never corrupted.
15 return true

ExplicitAuth:

1 if $\forall \pi_u^i, s$:
   $s' \leftarrow \mathsf{EAUTH}[\pi_u^i.role, s]$
  $\pi_u^i.\mathsf{accepted}[s'] < \infty$
   and $\pi_u^i.\mathsf{accepted}[s] < \infty$
   and $\pi_u^i.\mathsf{accepted}[s'] < \mathsf{revpsk}_{(u, \pi_u^i.peerid, \pi_u^i.pskid)}$
  // all sessions accepting in explicitly authenticated stages whose PSK was not corrupted before acceptance of the stage at which explicit authentication was (perhaps retroactively) established...
     $\implies \exists \pi_v^j : \pi_u^i.sid[s'] = \pi_v^j.sid[s']$
      and $\pi_u^i.peerid = v$
      and $\pi_u^i.pskid = \pi_v^j.pskid$
  // ... have a partnered session in that stage ...
  // ... agreeing on the peerid and pre-shared key...
     and $(\pi_v^j.\mathsf{accepted}[s] < \mathsf{time} \implies \pi_v^j.sid[s] = \pi_u^i.sid[s])$
  // ... and partnered in stage s (upon acceptance)
2   return true

Sound:

1 if $\exists s$, distinct π_u^i, π_v^j, π_w^k with $\pi_u^i.sid[s] = \pi_v^j.sid[s] = \pi_w^k.sid[s] \neq \bot$
  and $\mathsf{REPLAY}[s] = \mathsf{false}$ then
2   return false
  // no triple sid match, except for replayable stages
3 if $\exists \pi_u^i, \pi_v^j$, s with
   $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$ and
   $\pi_u^i.role = \pi_v^j.role$ and
   $(\mathsf{REPLAY}[s] = \mathsf{false}$ or $\pi_u^i.role = \mathsf{initiator})$ then
4   return false
  // partnering implies different roles (except for responders in replayable stages)
5 if $\exists \pi_u^i, \pi_v^j$, s with
   $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$ and
  $(\pi_u^i.cid_{\mathsf{initiator}}[s] \neq \pi_v^j.cid_{\mathsf{initiator}}[s]$ or $\pi_u^i.cid_{\mathsf{responder}}[s] \neq \pi_v^j.cid_{\mathsf{responder}}[s])$
6   return false
  // partnering implies matching cids
  if $\exists \pi_u^i, \pi_v^j$ and $s \neq t$ such that
   $\pi_u^i.sid[s] = \pi_v^j.sid[t]$
7   return false
  // different stages implies different sids
8 if $\exists \pi_u^i, \pi_v^j$, s with
   $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$
   and $\pi_u^i.peerid \neq v$
   or $\pi_v^j.peerid \neq u$ or $\pi_u^i.pskid \neq \pi_v^j.pskid$ then
  // partnering implies agreement on peer IDs and PSKs
9   return false
10 if $\exists \pi_u^i, \pi_v^j$, s with
   $\pi_u^i.\mathsf{accepted}[s] < \mathsf{time}$
   and $\pi_v^j.\mathsf{accepted}[s] < \mathsf{time}$
   and $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$,
   but $\pi_u^i.skey[s] \neq \pi_v^j.skey[s]$ then
  // partnering implies same key
11   return false
12 return true

Figure 4: Predicates Fresh, ExplicitAuth, and Sound for the MSKE pre-shared key model.

a tested internal key is replaced in the protocol with whatever the Test returns to the adversary (which is either the real internal key or an independent random key). The remaining stages produce "external" keys which may be tested at any time after acceptance.

For some protocols, it may be possible that a trivial replay attack can achieve that several sessions agree on the same session key for stage s, but this is not considered an "attack". For example, in TLS 1.3 PSK an adversary can always replay the `ClientHello` message to multiple sessions of the same server, which then all derive the same ETS and EEMS keys (cf. Figure 1). To specify that such a replay is not considered a protocol weakness, and thus should not be considered a valid "attack", the protocol specification may define $\mathsf{REPLAY}[s]$ to true for a stage s. $\mathsf{REPLAY}[s]$ is set to false by default.

As we focus on protocols which rely on (pre-authenticated) pre-shared keys, our model encodes that all protocol stages are at least *implicitly* mutually authenticated in the sense of Krawczyk [40], i.e., a session is guaranteed that any established key can only be known by the intended partner. Some stages will further be *explicitly* authenticated, either immediately upon acceptance or retroactively upon acceptance of a later state. Additionally, the stage at which explicit authentication is achieved may differ between the initiator and responder roles. For each stage s and role r, the key exchange protocol specification states in $\mathsf{EAUTH}[r, s]$ the stage t from whose acceptance stage s derives explicit authentication for the session in role r. Note that the stage-s key is not authenticated until both stages s and $\mathsf{EAUTH}[r, s]$ have been accepted. If the stage-s key will never be explicitly authenticated for role r, we set $\mathsf{EAUTH}[r, s] = \infty$.

We use a predicate ExplicitAuth (cf. Figure 4) to require the existence of an honest partner for explicitly authenticated stages upon both parties' completion of the protocol, except when the session's pre-shared key was corrupted prior to accepting the explicitly-authenticating stage (as in that case, we anticipate the adversary can trivially forge any authentication mechanism).

Motivated by TLS 1.3, it might be the case that initiator and responder sessions achieve slightly different guarantees of authentication. While responders in TLS 1.3 are guaranteed the existence of an honest partner in any explicitly authenticated stage, initiators cannot guarantee that their partner has received their final message. This issue was first raised by FGSW [30] and led to their definitions of "full" and "almost-full" key confirmation; it was then extended to "full" and "almost-full" explicit authentication by DFW [16]. Our definitions for responders and initiators respectively resemble the latter two notions most closely, but we rely on session identifiers instead of "key confirmation identifiers".

We consider three levels of forward secrecy inspired by the KEMTLS work of Schwabe, Stebila, and Wiggers [56]: no forward secrecy, weak forward secrecy 2 (wfs2), and full forward secrecy (fs). As for authentication, each stage may retroactively upgrade its level of forward secrecy upon the acceptance of later stages, and forward secrecy may be established at different stages for each role. For each stage s and role r, the stage at which wfs2, resp. fs, is achieved is stated in $\mathsf{FS}[r, s, \mathsf{wfs2}]$, resp. $\mathsf{FS}[r, s, \mathsf{fs}]$, by the key exchange protocol.

The definition of weak forward secrecy 2 states that a session key with wfs2 should be indistinguishable as long as (1) that session has received the relevant messages from an honest partner (formalized via matching contributive identifiers below, we say: "has an honest contributive partner") or (2) the pre-shared key was never corrupted. Full forward secrecy relaxes condition (2) to forbid corruption of the pre-shared key only before acceptance of the stage that retroactively provides full forward secrecy. We capture these notions of forward secrecy in a predicate Fresh(cf. Figure 4), which uses the log of events to check whether any tested session key is trivially distinguishable (e.g., through the session or its partnered being revealed, or forward secrecy requirements violated). With forward secrecy encoded in Fresh, our long-term key corruption oracle (RevLongTermKey), unlike in the model of [25], handles all corruptions the same way, regardless of forward secrecy.

Session and game variables. Sessions π_u^i and the security game itself maintain several variables; we indicate the former in *italics*, the latter in sans-serif font.

The game uses a counter time, initialized to 0 and incremented with any oracle query the adversary makes, to order events in the game log for later analysis. When we say that an event happens at a certain "time", we mean the current value of the time counter. The list pskeys contains, as discussed above, all pre-shared keys, indexed by a tuple $(u, v, pskid)$ containing the two users' IDs (u using the key only in the initiator role, v only in the reponder role), and a unique string identifier. The list revpsk, indexed like pskeys, tracks

the time of each pre-shared key corruption, initialized to $\mathsf{revpsk}_{(u,v,pskid)} \leftarrow \infty$. (In boolean expressions, we write $\mathsf{revpsk}_{(u,v,pskid)}$ as a shorthand for $\mathsf{revpsk}_{(u,v,pskid)} \neq \infty$.)

Each session π_u^i, identified by (adversarially chosen) user ID and a unique session ID, furthermore tracks the following variables:

- *status* $\in \{\mathsf{running}_s, \mathsf{accepted}_s, \mathsf{rejected}_s \mid s \in [1, \dots, \mathsf{STAGES}]\}$, where STAGES is the total number of stages of the considered protocol. The status should be $\mathsf{accepted}_s$ immediately after the session accepts the stage-s key, $\mathsf{rejected}_s$ after it rejects stage s (but may continue running; e.g., rejecting 0-RTT data), and $\mathsf{running}_s$ for some stage s otherwise.
- *peerid*. The identity of the session's intended communication partner.
- *pskid*. The identifier of the session's pre-shared key.
- $\mathsf{accepted}[s]$. For each stage s, the time (i.e., the value of the time counter) at which the stage s key was accepted. Initialized to ∞.
- $\mathsf{revealed}[s]$. A boolean denoting whether the stage s key has been leaked through a RevSessionKey query. Initialized to false.
- $\mathsf{tested}[s]$. The time at which the stage s key was tested. Initialized to ∞ before any Test query occurs. (In boolean expressions, we write $\mathsf{tested}[s]$ as a shorthand for $\mathsf{tested}[s] \neq \infty$.)
- $sid[s]$. The session identifier for each stage s, used to match honest communication partners within each stage.
- $skey[s]$. The key accepted at each stage.
- $cid_{\mathsf{initiator}}[s]$ and $cid_{\mathsf{responder}}[s]$. The contributive identifiers for each stage s, where $cid_{role}[s]$ identifies the communication part that a session in role *role* must have honestly received in order to be allowed to be tested in certain scenarios (cf. the freshness definition in the Fresh predicate). Unlike prior models, each session maintains a contributive identifiers for each role; one for itself and one for its intended partner. This enables more fine-grained testing of session stages in our model.

The predicate Sound (cf. Figure 4) captures that variables are properly assigned, in particular that session identifiers uniquely identify a partner session (except for replayable stages) and that partnering implies agreement on (distinct) roles, contributive identifiers, peer identities and the pre-shared key used, as well as the established session key.

Definition 3.1 (Multi-stage key exchange security)**.** *Let* KE *be a key exchange protocol and* $G^{\mathsf{MSKE}}_{\mathsf{KE},\mathcal{A}}$ *be the key exchange security game defined in Figures 3 and 4. We define*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) := 2 \cdot \max_{\mathcal{A}} \Pr\left[\mathsf{Game}^{\mathsf{MSKE}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1\right] - 1,$$

where the maximum is taken over all adversaries, denoted $(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})$-MSKE-adversaries*, running in time at most* t *and making at most* q_{NS}*,* q_{S}*,* q_{RS}*,* q_{RL}*,* q_{T}*, resp.* q_{RO} *queries to their respective oracles* NewSecret*,* Send*,* RevSessionKey*,* RevLongTermKey*,* Test*, and* RO.

4 Indifferentiability Background

In the random oracle model, we treat hash functions like `SHA256` as uniformly sampled random functions. Honest parties and adversaries alike access these functions via additional oracles in the security game. These are the *random oracles*. These random functions will be sampled from a set called a *function space* at the start of a security game. Alternatively, the random oracle can *lazily sample* responses to each query as they are needed. While we typically use the latter (lazily-sampled) model in key exchange security proofs, we will focus on the former conceptual view here.

Let us give an example. When we model the TLS 1.3 protocol in the ROM, we will equip our protocol definition with a function space parameter FS. We set this parameter according to the portion of the protocol we wish to model as a random oracle. If we wish to replace the hash function H with a random oracle RO_H,

Game $\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim},\mathsf{SS},\mathsf{ES}}$

```
INITIALIZE():
1  b <-$ {0,1}
2  RO_SS <-$ SS
3  RO_ES <-$ ES
4  state <-$ ε

FINALIZE(b'):
5  return b'

PUB(i, Y):
6  if b = 0 then
7     (z, state) <- Sim[PRIV](i, Y, state)
8     return z
9  else return RO_SS(i, Y)

PRIV(i, X):
10 if b = 0 then return RO_SS(i, X)
11 else return C[RO_ES](i, X)
```

Figure 5: The game $\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim},\mathsf{SS},\mathsf{ES}}$ measuring indifferentiability of a construct $\mathbf{C}$ that transforms function space SS into ES. The game is parameterized by a simulator Sim.

then we would set FS to be the set of all functions the set of all functions with domain $\{0,1\}^*$ and range $\{0,1\}^{hl}$. The KE security game would sample $\mathsf{RO_H}$ from FS in its INITIALIZE routine, then provide oracle access to $\mathsf{RO_H}$ to all parties. This notation also captures protocols which use multiple random oracles. If we wish to use two independent random oracles, say RO_1 and RO_2, then we would define an *arity*-2 function space FS, which is a set of tuples each containing two functions. Let FS_1, resp. FS_2 be the set from which RO_1, RO_2 should be drawn. Then we set $\mathsf{FS} = \{(F_1, F_2) : F_1 \in \mathsf{FS}_1 \text{ and } F_2 \in \mathsf{FS}_2\}$. We call FS_1 and FS_2 the subspaces of FS. A security game provides access to F_1 and F_2 through a single oracle RO that takes two arguments; the first is the index of the function to be queried and the second is the contents of the query. So $\mathsf{RO}(i, X)$ will return $F_i(X)$. We can also cast an arity-1 function space in this notation by identifying each function F with the tuple (F), but we will typically omit the parentheses and index argument when only one random oracle is used.

Indifferentiability was originally developed by Maurer, Renner, and Holenstein [51], and it has been used to prove security for hash functions built from public compression functions. More generally, it gives a framework to show the security of a transition between any two function spaces. We'll call these spaces SS (for "starting space") and ES (for "ending space"). A *construction* of ES from SS is an algorithm $\mathbf{C}$ that outputs elements of ES given an oracle $\mathsf{RO_{SS}} \in \mathsf{SS}$. We may use the notation $\mathbf{C} : \mathsf{SS} \to \mathsf{ES}$. We then say that $\mathbf{C}$ is "indifferentiable" if for any function $\mathsf{RO_{SS}}$ sampled from SS, $\mathbf{C}[\mathsf{RO}]$ behaves indistinguishably from a function $\mathsf{RO_{ES}}$ sampled from ES. Indifferentiability requires this behavior to hold even when the adversary can access *both* $\mathbf{C}[\mathsf{RO_{SS}}]$ *and* $\mathsf{RO_{SS}}$ without any restriction. Once we have an indifferentiable construction between two function spaces, we can use the indifferentiability "composition theorem" to prove that (almost) any protocol is as secure when it uses $\mathbf{C}[\mathsf{RO_{SS}}]$ as its random oracle as when it uses $\mathsf{RO_{ES}}$.[6]

How do we check whether a construction $\mathbf{C}$ is indifferentiable? From the earlier intuition, we set up a security game with two worlds. In one world, often called the "real world", the adversary has oracle access to $\mathsf{RO_{SS}}$ (drawn from SS) and $\mathbf{C}[\mathsf{RO_{SS}}]$. In the other, the "ideal world", it has oracle access to $\mathsf{RO_{ES}}$, a random oracle sampled from ES. The adversary's task is then to return a bit indicating which world it is in.

This intuition is obviously incomplete: the adversary can distinguish between worlds just by counting its oracles. We need a second oracle in the ideal world. This second oracle, PUB, must behave indistinguishably from $\mathsf{RO_{SS}}$, but its responses must also be consistent with the view of $\mathsf{RO_{ES}}$ (accessed via the first oracle, PRIV) as a construction of PUB. The algorithm that does this is called a "simulator". Every construction requires a different simulator Sim, so we make it a parameter of the definition. We can now give pseudocode for the full indifferentiability security game, shown in Figure 5.

Definition 4.1 (Indifferentiability)**.** *Let* SS *and* ES *be function spaces, and let* $\mathbf{C}$ *be a construction of* ES *from* SS*. Then for any simulator* Sim *and any adversary* $\mathcal{D}$ *which makes* q_{PRIV} *queries to the* PRIV *oracle*

[6] As Ristenpart, Shacham, and Shrimpton [55] showed, indifferentiability composition does not cover what they call "multi-stage games," meaning games in which the adversary is split into distinct algorithms with restricted communication. Our multi-stage AKE security game is actually a "single-stage" game in the RSS terminology; indifferentiability composition does apply to our results without issue.

and q_{PUB} *queries to the* Pub *oracle, the indifferentiability advantage of* $\mathcal{D}$ *is*

$$\mathrm{Adv}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim},\mathsf{q}_{\text{PRIV}},\mathsf{q}_{\text{PUB}}}(\mathcal{D}) := \Pr[\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathcal{D}) \Rightarrow 1 | b = 1] - \Pr[\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathcal{D}) \Rightarrow 1 | b = 0].$$

Indifferentiability is useful because of the following theorem of Maurer et al. [51]. In our presentation, we consider only the authenticated key exchange game, although the theorem applies equally well to any single-stage game [55].

Theorem 4.2. *Let* KE *be a key exchange protocol using function space* ES*. Let* $\mathbf{C}$ *be an indifferentiable construct of* ES *from* SS *with respect to simulator* Sim*, and let* t' *be the runtime of* Sim *on a single query. We define* KE' *to be the following key exchange protocol with function space* SS*:* KE' *runs* KE*, but wherever* KE *would call its random oracle,* KE' *instead computes* $\mathbf{C}$ *using its own random oracle. For any adversary* $\mathcal{A}$ *against the* MSKE *security of* KE' *with runtime* $t_{\mathcal{A}}$ *and making* q *random oracle queries, there exists an adversary* $\mathcal{B}$ *and a distinguisher* $\mathcal{D}$ *with runtime approximately* $t_{\mathcal{A}} + q \cdot t$ *such that*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}'}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathcal{D}).$$

Proof. Adversary $\mathcal{B}$ is a wrapper for $\mathcal{A}$ whenever $\mathcal{A}$ makes a query to its random oracle RO, $\mathcal{B}$ responds by running the simulator with its own random oracle. The distinguisher $\mathcal{D}$ simulates the $\mathsf{KE} - Sec$ game of KE for $\mathcal{A}$, with two differences: instead of an RO, it gives $\mathcal{A}$ oracle access to Pub, and where KE would query its own RO, it instead queries Priv. We claim that when $b = 1$ in the indifferentiability game (the real world), $\mathcal{D}$ perfectly simulates the MSKE game of KE' for $\mathcal{A}$. This works because the Priv oracle computes $\mathbf{C}$ for KE', and the Pub oracle is indeed an RO as $\mathcal{A}$ expects. When $b = 0$, $\mathcal{D}$ perfectly simulates MSKE of KE for $\mathcal{B}$. The Pub oracle answers all of $\mathcal{A}$'s queries using the simulator, so it properly executes the wrapper code that makes up $\mathcal{B}$. The rest of the simulation is honest, down to the random oracle accessed via Priv. □

5 Key-Schedule Indifferentiability

In this section we will argue that the key schedule of TLS 1.3 PSK modes, where the underlying cryptographic hash function is modeled as a random oracle (i.e., the left-hand side of Figure 1 with the underlying hash function modeled as a random oracle), is *indifferentiable* [51] from a key schedule that uses *independent* random oracles for each step of the key derivation (i.e., the right-hand side of Figure 1 with all TKDF_x functions modeled as independent random oracles). We stress that this step not only makes our main security proof in Section 7 significantly simpler and cleaner, but also it puts the entire protocol security analysis on a firmer theoretical ground than previous works.

In their proof of tight security, Diemert and Jager [19] previously modeled the TLS 1.3 key schedule as four independent random oracles. Davis and Günther [15] concurrently modeled the functions HKDF.Extract and HKDF.Expand used by the key schedule as two independent random oracles. Neither work provided formal justification for their modeling. Most importantly, both neglected potential dependencies between the use of the hash function in multiple contexts in the key schedule and elsewhere in the protocol. In particular, no construction of HKDF.Extract and HKDF.Expand as independent ROs from one hash function could be indifferentiable, because HKDF.Extract and HKDF.Expand both call HMAC directly on their inputs, with HKDF.Expand only adding a counter byte. Hence, the two functions are inextricably correlated by definition. We do not claim that the analyses of [19, 15] are incorrect or invalid, but merely point out that their modeling of independent random oracles is currently not justified and might not be formally reachable if one only wants to treat the hash function itself as a random oracle. This is undesirable because the gap between an instantiated protocol and its abstraction in the random oracle model can camouflage serious attacks, as Bellare et al. [5] found for the NIST PQC KEMs. Their attacks exploited dependencies between functions that were also modeled as independent random oracles but instantiated with a single hash function.

In contrast, in this section we will show that our modeling of the TLS 1.3 key schedule is indifferentiable from the key schedule when the underlying cryptographic hash function is modeled as a random oracle. To this end, we will require that inputs to the hash function do not appear in multiple contexts. For instance, a protocol transcript might collide with a Diffie–Hellman group element or an internal key (i.e., both might be represented by exactly the same bit string, but in different contexts). For most parameter settings, we can

rule out such collisions by exploiting serendipitous formatting, but for one choice of parameters (the PSK-only handshake using SHA384 as hash function), an adversary could conceivably force this type of collision to occur; see Appendix B for a detailed discussion. While this does not lead to any known attack on the handshake, it precludes our indifferentiability approach for that case.

Insights for the design of cryptographic protocols. One interesting insight for protocol designers that results from our attempt of closing this gap with a careful indifferentiability-based analysis is that proper domain separation might enable a cleaner and simpler analysis, whereas a lack of domain separation leads to uncertainty in the security analysis. No domain separation means stronger assumptions in the best case, and an insecure protocol in the worst case, due to the potential for overlooked attack vectors in the hash functions. A simple prefix can avoid this with hardly any performance loss.

Indifferentiability of the TLS 1.3 key schedule. Via the indifferentiability framework, we replace the complex key schedule of TLS 1.3 with 12 independent random oracles: one for each first-class key and MAC tag, and one more for computing transcript hashes. In short, we relate the security of TLS 1.3 as described in the left-hand side of Figure 1 to that of the simplified protocol on the right side of Figure 1 with the key derivation and MAC functions TKDF_x and modeled as independent random oracles. We prove the following theorem, which formally justifies our abstraction of the key exchange protocol by reducing its security to that of the original key exchange game.

Theorem 5.1. *Let* $\mathsf{RO}_\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^{hl}$ *be a random oracle. Let* KE *be the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol described on the left hand side of Figure 1 with* $\mathbf{H} := \mathsf{RO}_\mathsf{H}$ *and* **MAC**, **Extract**, *and* **Expand** *defined from* **H** *as in Section 2. Let* KE' *be the corresponding (PSK-only or PSK-(EC)DHE) handshake protocol on the right hand side of Figure 1, with* $\mathbf{H} := \mathsf{RO}_\mathsf{Th}$ *and* $\mathsf{TKDF}_x := \mathsf{RO}_x$, *where* RO_Th, RO_{binder}, $\ldots$, $\mathsf{RO}_{\mathrm{RMS}}$ *are random oracles with the appropriate signatures (cf. Section 5.1.3 for the signature details). Then,*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}'}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}} + \frac{2q_{\mathrm{RO}}^2}{2^{hl}} + \frac{8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.$$

We establish this result via three modular steps in the indifferentiability framework introduced by Maurer, Renner, and Holenstein [51]. More specifically we will leverage a recent generalization proposed by Bellare, Davis, and Günther (BDG) [5], which in particular formalizes indifferentiability for constructions of *multiple* random oracles.

5.1 Indifferentiability for the TLS 1.3 Key Schedule in Three Steps

We move from the left of Figure 1 to the right via three steps. Each step introduces a new variant of the TLS 1.3 protocol with a different set of random oracles by changing how we implement **H**, **MAC**, **Expand**, **Extract**, and eventually the whole key schedule. Then we view the prior implementations of these functions as constructions of new, independent random oracles. We prove security for each intermediate protocol in two parts: first, we bound the indifferentiability advantage against that step's construction; then we apply the indifferentiability composition theorem based on [51] (cf. Section 4, Theorem 4.2) to bound the multi-stage key exchange (MSKE) security of the new protocol.

We give a brief description of each step; all details and formal theorem statements and proofs can be found in Sections 5.1.1, 5.1.2, and 5.1.3, respectively.

From one random oracle to two. TLS 1.3 calls its hash function **H**, which we initially model as random oracle RO_H, for two purposes: to hash protocol transcripts, and as a component of **MAC**, **Extract**, and **Expand** which are implemented using $\mathsf{HMAC}[\mathbf{H}]$. Our eventual key exchange proof needs to make full use of the random oracle model for the latter category of hashes, but we require only collision resistance for transcript hashes.

Our first intermediate handshake variant, KE_1, replaces **H** with two new functions: Th for hashing transcripts, and Ch for use within **MAC**, **Extract**, or **Expand**. While KE uses the same random oracle

$\mathsf{RO_H}$ to implement Th and Ch, the KE_1 protocol instead uses two independent random oracles $\mathsf{RO_{Th}}$ and $\mathsf{RO_{HMAC}}$. To accomplish this without loss in MSKE security, we exploit some possibly unintentional domain separation in how inputs to these functions are formatted in TLS 1.3 to define a so-called *cloning functor*, following BDG [5]. Effectively, we partition the domain $\{0,1\}^*$ of $\mathsf{RO_H}$ into two sets D_{Th} and D_{Ch} such that D_{Th} contains all valid transcripts and D_{Ch} contains all possible inputs to **H** from HMAC. We then leverage Theorem 1 of [5] that guarantees composition for any scheme that only queries $\mathsf{RO_{Ch}}$ within the set D_{Ch} and $\mathsf{RO_{Th}}$ within the set D_{Th}.

We defer details on the exact domain separation to Appendix B, but highlight that the PSK-only handshake with hash function `SHA384` *fails* to achieve this domain separation and consequently this proof step cannot be applied and leaves a gap for that configuration of TLS 1.3.

From SHA to HMAC. Our second variant protocol, KE_2, rewrites the **MAC** function. Instead of computing $\mathsf{HMAC}[\mathsf{RO_{Ch}}]$, **MAC** now directly queries a new random oracle $\mathsf{RO_{HMAC}}: \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$. Since $\mathsf{RO_{Ch}}$ was only called by **MAC**, we drop it from the protocol, but we do continue to use $\mathsf{RO_{Th}}$, i.e., KE_2 uses two random oracles: $\mathsf{RO_{Th}}$ and $\mathsf{RO_{HMAC}}$. The security of this replacement follows directly from Theorem 4.3 of Dodis et al. [21], which proves the indifferentiability of HMAC with fixed-length keys.[7]

From two random oracles to 12. Finally, we apply a "big" indifferentiability step which yields 12 independent random oracles and moves us to the right-hand side of Figure 1. The 12 ROs include the transcript-hash oracle $\mathsf{RO_{Th}}$ and 11 oracles that handle each key(-like) output in TLS 1.3's key derivation, named RO_{binder}, $\mathsf{RO}_{\mathrm{ETS}}$, $\mathsf{RO}_{\mathrm{EEMS}}$, RO_{htk_C}, $\mathsf{RO}_{\mathtt{CF}}$, RO_{htk_S}, $\mathsf{RO}_{\mathtt{SF}}$, $\mathsf{RO}_{\mathrm{CATS}}$, $\mathsf{RO}_{\mathrm{SATS}}$, $\mathsf{RO}_{\mathrm{EMS}}$, and $\mathsf{RO}_{\mathrm{RMS}}$. (The signatures for these oracles are given in Appendix 5.1.3.) For this step, we view TKDF as a construction of 11 random oracles from a single underlying oracle ($\mathsf{RO_{HMAC}}$). We then give our a simulator in pseudocode and prove the indifferentiability of TKDF with respect to this simulator. Our simulator uses look-up tables to efficiently identify intermediate values in the key schedule and consistently program the final keys and MAC tags.

Combining these three steps yields the result in Theorem 5.1. In the remainder of the paper, we can therefore now work with the right-hand side of Figure 1, modeling **H** and the TKDF functions as 12 independent random oracles.

5.1.1 Step 1: Domain-separating the Transcript Hash

In the original TLS 1.3 PSK/PSK-(EC)DHE handshake, the hash function **H** is used in two different ways. It is used directly to compute digests of a *transcript* and it is used as a *component* of **MAC**, **Extract**, and **Expand**. We will argue now that these two uses are entirely distinct, and we can accordingly write two functions Th and Ch in place of the two uses of **H**, and, following BDG [5], go from modeling **H** as one random oracle to modeling Th and Ch as two independent random oracles.

We will refer to our two new random oracles as $\mathsf{RO_{Th}}$ (modeling the *transcript hash* function Th) and $\mathsf{RO_{Ch}}$ (modeling the *component hash* Ch). Because TLS 1.3 fully specifies the inputs to each hash function call, we can show that in PSK-(EC)DHE mode and in PSK-only mode when $hl = 256$, TLS 1.3 will never call the same string as an input to both Th and Ch. This is due to some fortunate coincidences of formatting in the standard, which we describe in full in Appendix B. We can therefore define two disjoint sets D_{Th} and D_{Ch} such that $D_{\mathsf{Th}} \cup D_{\mathsf{Th}} = \{0,1\}^*$ split up H's domain.

If we define the domain of $\mathsf{RO_{Th}}$ to be D_{Th} and the domain of $\mathsf{RO_{Ch}}$ to be D_{Ch}, we could prove indifferentiability using a construction called the *identity (cloning) functor* **I** from [5]. The identity functor constructs two or more random oracles $\mathsf{RO}_1, \mathsf{RO}_2, \ldots$ from $\mathsf{RO_H}$ by forwarding all RO_i queries to $\mathsf{RO_H}$ unchanged. However, the definitions of sets D_{Th} and D_{Ch} are somewhat complex, especially in PSK-only mode. We would instead prefer to define both $\mathsf{RO_{Th}}$ and $\mathsf{RO_{Ch}}$ with domains $\{0,1\}^*$. This would greatly simplify our later use of $\mathsf{RO_{Ch}}$ as a component of HMAC. Unfortunately, when the domains of $\mathsf{RO_{Th}}$ and $\mathsf{RO_{Ch}}$ overlap, the

[7] This requires PSKs to be elements of $\{0,1\}^{hl}$, which is true of resumption keys but possibly not for out-of-band PSKs.

identity functor is *not* indifferentiable. We can however still provide the desired result by turning to the read-only indifferentiability framework of Bellare, Davis, and Günther [5].

Read-only indifferentiability (a.k.a. rd-indiff) is similar to standard indifferentiability [51]. One notable change (and the one we will leverage here) is that it is parameterized by a set $\mathcal{W}$ called the "working domain." The security game places a restriction on the PRIV oracle so that it only responds to queries within $\mathcal{W}$. Read-only indifferentiability supports a broader composition thoerem than Theorem 4.2, which covers security games which call their random oracles only within the working domain. BDG prove [5, Theorem 1], which states that when $\mathcal{W}$ consists of disjoint sets like D_{Th} and D_{Ch}, the identity functor is read-only indifferentiable even when the full domains of $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ are not disjoint. Furthermore, the read-only indifferentiability advantage is upper-bounded by 0, and BDG give a simulator that runs in linear time on the length of its inputs and makes at most one query per execution. When we apply the read-only indifferentiability composition theorem, the adversary's runtime and query bounds will not increase.

We formalize this with a lemma:

Lemma 5.2. *Let* KE *be the TLS 1.3 key exchange protocol of Theorem 5.1. Let* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{\mathsf{Ch}}\colon \{0,1\}^* \to \{0,1\}^{hl}$ *be two random oracles, and let* KE_1 *be the protocol on the left-hand side of Figure 1, where*

- $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$
- $\mathbf{MAC} := \mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$

and **Expand** *and* **Extract** *are as in* KE *(using the new definition of* **MAC***). Let* D_{Th} *and* D_{Ch} *be two disjoint sets such that* $\mathsf{KE}_1.\mathsf{Run}$ *only queries* $\mathsf{RO}_{\mathsf{Th}}$*, resp.* $\mathsf{RO}_{\mathsf{Ch}}$ *in* D_{Th}*, resp.* D_{Ch}*, and* $D_{\mathsf{Th}} \cup D_{\mathsf{Ch}} = \{0,1\}^*$*. Furthermore, let* D_{Th} *have an efficient membership function.*

Let $\mathcal{A}$ *be an adversary against the* MSKE *security of* KE*, running in time* $t_{\mathcal{A}}$ *and making* q_{RO} *and* q_{S} *queries to its random oracle resp.* SEND *oracle. Then there exists an adversary* $\mathcal{B}$ *against the security of* KE'*, such that*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(\mathcal{B}).$$

Adversary $\mathcal{B}$*'s runtime is* $\mathcal{O}(t_{\mathcal{A}} + q_{\mathrm{RO}})$*, and it makes the same number of queries to each of its oracles as* $\mathcal{A}$ *in the* MSKE *game.*

Proof. The function space of KE is $\mathsf{SS} = \mathsf{AF}(\{0,1\}^*, \{0,1\}^{hl})$, and the function space of KE_1 is $\mathsf{ES} = \mathsf{AF}(\{\mathsf{Th}, \mathsf{Ch}\} \times \{0,1\}^*, \{0,1\}^{hl})$. We can construct ES from SS via a construction called the "identity functor" defined by BDG [5]. This construction is parameterized by a set $\mathcal{W} := (\{\mathsf{Th}\} \times D_{\mathsf{Th}}) \cup (\{\mathsf{Ch}\} \times D_{\mathsf{Ch}})$. To answer any query (i, s), the identity functor simply forwards s to its own oracle, regardless of whether i is Th or Ch. Because $\mathcal{W}$ is the union of two disjoint sets with efficient membership functions, the simulator Sim defined by BDG's Theorem 1 has the property that for any distinguisher $\mathcal{D}$,

$$\mathrm{Adv}^{\mathsf{rd\text{-}indiff}}_{\mathbf{I}_{\mathcal{W}},\mathcal{W},\mathsf{Sim}}(\mathcal{D}) = 0.$$

Sim works by using the membership function of D_{Th} to check which of the two oracles is being simulated; then it forwards the query to the appropriate oracle.

For this (or any) simulator, the composition theorem for read-only indifferentiability grants the existence of adversary $\mathcal{B}$ and a distinguisher $\mathcal{D}$ such that

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{rd\text{-}indiff}}_{\mathbf{I}_{\mathcal{W}},\mathcal{W},\mathsf{Sim}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(\mathcal{B}).$$

This composition theorem crucially rests on the fact that $\mathsf{KE}_1.\mathsf{Run}$ queries $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ only within $\mathcal{W}$. The lemma follows.

We require that D_{Th} and D_{Ch} are disjoint sets. We define specific choices of D_{Th} and D_{Ch} based on the low-level formatting of TLS 1.3 in Appendix B, and there we give detailed arguments that the sets are disjoint for 3 of 4 standardized settings of the PSK/PSK-(EC)DHE handshake.

In the fourth setting, PSK-only mode with hash function `SHA384`, there are no disjoint choices for D_{Th} and D_{Ch} with efficient membership functions. This is due to a lack of careful domain separation of the hash function calls in TLS 1.3. We therefore cannot apply this indifferentiability step for the PSK-only/`SHA384` handshake protocol. Any security proof of this handshake must either rely on stronger, possibly falsifiable

abstractions in the random oracle model, or use a model SHA384 as a single random oracle, with no guarantees of independence. We avoid the latter approach in order to maintain a modular and readable proof.

The second inequality follows from our choice of simulator and Theorem 1 of [5], which makes at most one query to its random oracle per execution. Their simulator, as mentioned above, must efficiently determine for every query s whether to query $\mathsf{RO}_{\mathsf{Th}}$ or $\mathsf{RO}_{\mathsf{Ch}}$. This induces the requirement that $D_{\mathsf{Th}} \cup D_{\mathsf{Ch}} = \{0,1\}^*$, so every possible query can be routed appropriately, and the requirement that D_{Th} has an efficient membership function so that the simulator is itself efficient. D_{Th} and D_{Ch} satisfy these requirements thanks to the rules given in Appendix B. □

5.1.2 Step 2: Applying the Indifferentiability of HMAC

Our next key exchange protocol, KE_2, replaces the construction $\mathsf{HMAC}[\mathsf{Ch}]$ with a single random oracle $\mathsf{RO}_{\mathsf{HMAC}}$ in the implementation of **MAC** and by extension **Extract** and **Expand**. We rely on the proof of HMAC's indifferentiability by Dodis et al. [21, Theorem 3]. As a prerequisite for this theorem, we need to restrict HMAC to keys of a fixed length less than the block length of the hash function (512 bits for SHA256 and 1024 bits for SHA384). This is consistent with HMAC's usage in TLS 1.3, where the keys are almost always of length $hl \in \{256, 384\}$. The only exception is when pre-shared keys of another length are negotiated out-of-band; we exclude this case.

Lemma 5.3. *Let* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{\mathsf{Ch}} \colon \{0,1\}^* \to \{0,1\}^{hl}$ *and* $\mathsf{RO}_{\mathsf{HMAC}} \colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$ *be random oracles. Let* KE_1 *be the TLS 1.3 key exchange protocol described in Theorem 5.2 using random oracles* $\mathsf{RO}_{\mathsf{Th}}$ *and* $\mathsf{RO}_{\mathsf{Ch}}$*. Let* KE_2 *be the key exchange protocol given on the left-hand side of Figure 1, where*

- $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$
- $\mathbf{MAC} := \mathsf{RO}_{\mathsf{HMAC}}$

and **Extract** *and* **Expand** *are defined as Section 2. Let* $\mathcal{A}$ *be an adversary against the* MSKE *security of* KE_1*, running in time* $t_{\mathcal{A}}$ *and making* q_{RO} *and* q_{S} *queries to its random oracle resp.* Send *oracle. Then there exists an adversary* $\mathcal{B}$ *against the security of* KE_2 *such that*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(\mathcal{B}) + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}}.$$

Adversary $\mathcal{B}$ *has runtime* $\mathcal{O}(t_{\mathcal{A}} + q_{\mathrm{RO}})$ *and makes the same number of queries to each of its oracles as* $\mathcal{A}$ *in the* MSKE *game.*

Proof. KE_1 uses function space ES, defined in the proof of Lemma 5.2, and KE_2 uses function space $\mathsf{ES}_2 = \mathsf{AF}((\{\mathsf{Th}\} \times \{0,1\}^*) \cup (\{\mathsf{HMAC}\} \times \{0,1\}^{hl} \times \{0,1\}^*), \{0,1\}^{hl})$. The construction $\mathbf{C}$ of ES_2 from ES simply forwards all queries to $\mathsf{RO}_{\mathsf{Th}}$. It answers $\mathsf{RO}_{\mathsf{HMAC}}$ queries with $\mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$.

For any simulator Sim, Theorem 5 grants the existence of a distinguisher $\mathcal{D}$ and an adversary $\mathcal{B}$ such that

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathcal{D}).$$

The distinguisher $\mathcal{D}$ makes up to 12 queries to Priv for each Send query made by $\mathcal{A}$, and makes one Pub query for each RO query of $\mathcal{A}$.

We consider the simulator Sim_2 defined by Dodis et al. for [20, Theorem 4.3] (the full version of [21, Theorem 3]). This simulator relies on the requirement that HMAC keys are a fixed length, and shorter than the block length of the underlying hash function. HMAC pads its keys with zero bits up to the block length, so each hash function call made by HMAC contains a segment containing the byte `0x36` for the first of the two calls and `0x5c` for the second. Sim_2 uses this segment to identify whether a particular query is intended to simulate the first or second hash function call. It answers the "first" calls with random strings and logs these responses. Then it programs the "second" calls by using its stored intermediate values to find which $\mathsf{RO}_{\mathsf{HMAC}}$ query should be simulated. We augment the simulator to forward all queries to $\mathsf{RO}_{\mathsf{Th}}$; this does not change its runtime or effectiveness. This simulator works perfectly unless there is a collision among the $2\mathsf{q}_{\mathrm{Priv}} + \mathsf{q}_{\mathrm{Pub}}$ intermediate values, which Dodis et al. bound with a birthday bound. That theorem states that for a distinguisher $\mathcal{D}$ making $12q_{\mathrm{S}}$ queries to Priv and q_{RO} queries to Pub,

$$\mathrm{Adv}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathcal{D}) \leq \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}}.$$

The lemma follows. $\square$

5.1.3 Step 3: Applying Indifferentiability to the TLS Key Schedule

In the last step, we move to the right-hand side of Figure 1 and introduce 11 new independent random oracles to model the key schedule. We start by rephrasing the TLS key schedule and message authentication codes as eleven functions $\mathsf{TKDF}_{binder}, \ldots, \mathsf{TKDF}_{\mathrm{RMS}}$ as in Section 2. This abstraction does not change any of the operations performed by the key schedule; the TKDF functions simply rename the key derivation steps already performed by KE_2. In our last key exchange protocol KE', we model each TKDF function as a independent random oracle. We name these oracles after the keys or values they derive:

1. $\mathsf{RO}_{binder}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
2. $\mathsf{RO}_{\mathrm{ETS}}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
3. $\mathsf{RO}_{\mathrm{EEMS}}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
4. $\mathsf{RO}_{htk_C}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
5. $\mathsf{RO}_{fin_C}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
6. $\mathsf{RO}_{htk_S}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
7. $\mathsf{RO}_{fin_S}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
8. $\mathsf{RO}_{\mathrm{CATS}}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
9. $\mathsf{RO}_{\mathrm{SATS}}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
10. $\mathsf{RO}_{\mathrm{EMS}}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
11. $\mathsf{RO}_{\mathrm{RMS}}[\mathsf{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$

The 12th random oracle is $\mathsf{RO}_{\mathsf{Th}}$, used to hash transcripts as in KE_1 and KE_2.

Now we can state Lemma 5.4.

Lemma 5.4. *Let KE_2 be the key exchange protocol of Lemma 5.3, and let KE' be the key exchange protocol of Theorem 5.1.*

For any adversary $\mathcal{A}$ against the MSKE security of KE_2, with runtime t and making q_{RO} random oracle queries and q_{S} queries to Send*, there exists adversary $\mathcal{B}$ against the MSKE security of KE' such that*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(\mathcal{B}) + \frac{2\mathsf{q}^2_{\mathrm{PUB}}}{2^{hl}} + \frac{8(\mathsf{q}_{\mathrm{PUB}} + 6\mathsf{q}_{\mathrm{PRIV}})^2}{2^{hl}}.$$

Adversary $\mathcal{B}$ runs in time at most $t + q_{\mathrm{RO}} t_{\mathbb{G}}$, where $t_{\mathbb{G}}$ is the time to perform one group operation in the Diffie–Hellman group $\mathbb{G}$. It makes no more queries to each of the oracles in the MSKE game than does $\mathcal{A}$.

Proof. We view TKDF as defined in Section 2 as a construction of the function space ES' of KE': the arity-12 function space whose first subspace is $\mathsf{AF}(\{0,1\}^*, \{0,1\}^{hl})$ and whose remaining 11 subspaces are the spaces of all functions with the domains and ranges specified in the above list. This TKDF construction takes an oracle from ES_2, the function space of KS_2.

As in the prior two steps, we consider a particular simulator Sim (cf. Figure 5.1.3) and rely on Theorem 5 for the existence of a distinguisher $\mathcal{D}$ and an adversary $\mathcal{B}$ such that

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}'}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{indiff}}_{\mathsf{TKDF},\mathsf{Sim}}(\mathcal{D}).$$

The distinguisher $\mathcal{D}$ will make no more than 12 queries to Priv for each Send query made by $\mathcal{A}$ and one query to Pub per RO query.

Via a sequence of code-based games, we will show that the indifferentiability advantage of any distinguisher $\mathcal{D}$ making $\mathsf{q}_{\mathrm{PRIV}}$ queries to the Priv oracle and $\mathsf{q}_{\mathrm{PUB}}$ queries to the Pub oracle is

$$\mathrm{Adv}^{\mathsf{indiff}}_{\mathsf{TKDF},\mathsf{SS},\mathsf{ES},\mathsf{Sim}}(\mathcal{D}) \leq \frac{2\mathsf{q}^2_{\mathrm{PUB}}}{2^{hl}} + \frac{8(\mathsf{q}_{\mathrm{PUB}} + 6\mathsf{q}_{\mathrm{PRIV}})^2}{2^{hl}}.$$

We give fully specified pseudocode for each of our games.

First, we explain the high-level strategy of our simulator. Our simulator takes two inputs: an index $i \in \{\mathsf{Th}, \mathsf{HMAC}\}$ and a string $s \in \{0,1\}^*$. When $i = \mathsf{Th}$, the simulator simulates $\mathsf{RO}_{\mathsf{Th}}(s)$ easily; it simply forwards the query to its own random oracle $\mathsf{RO}_{\mathsf{Th}}$. When $i = \mathsf{HMAC}$, the simulator will parse s into a key $K \in \{0,1\}^{hl}$ and a context string $Y \in \{0,1\}^*$ and simulate $\mathsf{RO}_{\mathsf{HMAC}}(K, Y)$. This simulation should be compatible with a view of the random oracles RO_x as computing $\mathsf{TKDF}_x[\mathsf{RO}_{\mathsf{HMAC}}]$.

Initially, Sim randomly samples the response y to any simulated $\mathsf{RO}_{\mathsf{HMAC}}$ query from $\{0,1\}^{hl}$. Repeated queries are cached in a table M. Next, Sim checks whether the query could be part of an attempt to compute $\mathsf{TKDF}_x[\mathsf{Sim}]$ for some x. If so, it may have to program its response for consistency with RO_x, or it may store its response in a lookup table T to enable future programming.

The only values that need programming are the first-class keys and MAC values. These are all outputs of **Expand**[$\mathsf{RO}_{\mathsf{HMAC}}$]. Sim can tell if a particular $\mathsf{RO}_{\mathsf{HMAC}}$ query is made by **Expand** by checking its formatting. The inputs Y of all **Expand**'s queries in the key schedule start with 3 bytes of fixed values and a label ℓ between 8 and 18 bytes long that starts with the string `"tls13"`. They end with a 1 byte counter that TLS 1.3 fixes to `0x01`. Sim pattern-matches this label to determine which key is being derived. It has a subroutine $\mathcal{L}$ to translate the few labels which are used in the last derivation step for multiple keys.

Whenever Sim detects the label of an intermediate key derivation query like the **Expand** calls used to compute ES, HS, or MS, it stores the response to this query in table T under the name of the key in question. If $\mathcal{D}$ computes TKDF honestly, these tables will allow the simulator to backtrack through the execution to identify all of the inputs to TKDF. Inputs to $\mathsf{RO}_{\mathsf{HMAC}}$ queries made by $\mathsf{HKDF.Extract}$ do not contain labels, so some tables contain multiple intermediate values. Even without labels, each intermediate value should only appear in one key derivation except in the unlikely event of a collision in $\mathsf{RO}_{\mathsf{HMAC}}$.

The first game in our sequence is Game_0 which is the "ideal world" setting of the indifferentiability game. Here, Priv queries are answered using a random function RO drawn from ES, and Pub queries are answered with $\mathsf{Sim}[\mathsf{RO}]$.

In Game_1 (cf. Figure 5.1.3), we set a bad flag bad_C and abort whenever Sim samples a random answer y that collides with the input or output of any previous simulator query. We track these inputs and outputs in a list L. For each new query, there are at most $2\mathsf{q}_{\text{Pub}}$ points to collide with. Since y is sampled uniformly from $\{0,1\}^{hl}$, the probability of such a collision over all queries is at most $\frac{2\mathsf{q}_{\text{Pub}}^2}{2^{hl}}$ by a birthday and union bound). Then

$$|\Pr[\mathsf{Game}_1] - \Pr[\mathsf{Game}_0]| \leq \frac{2\mathsf{q}_{\text{Pub}}^2}{2^{hl}}.$$

In Game_2 (Figure5.1.3), the Finalize oracle computes $\mathsf{TKDF}[\mathsf{RO}_{\mathsf{HMAC}}]$ on the input to every query to the Priv oracle, using Pub as its hash function. It discards the results of this computation, so this change can affect the outcome of the game only if one of the additional Pub queries sets the bad_C flag. The TKDF function queries its oracle at most 6 times per execution, so there are no more than $6\mathsf{q}_{\text{Priv}}$ new queries. There are now a total of $\mathsf{q}_{\text{Pub}} + 6\mathsf{q}_{\text{Priv}}$ queries to Pub, so the probability that bad_C is set increases by another birthday bound.

$$|\Pr[\mathsf{Game}_2] - \Pr[\mathsf{Game}_1]| \leq \frac{2(\mathsf{q}_{\text{Pub}} + 6\mathsf{q}_{\text{Priv}})^2}{2^{hl}}.$$

The next step is the most subtle. In Game_3 (Figure 5.1.3), we move the new computations of TKDF from the Finalize oracle into Priv. When Priv is called with index i and input X, it still returns $\mathsf{RO}_i(X)$. First, however, it computes $\mathsf{TKDF}_i[\text{Pub}](X)$. It discards the result of this computation, so the behavior of the Priv oracle does not change in the adversary's view.

However, queries to Priv now run the simulator Sim. They can update its state and set the global bad_C flag. This has two consequences. First, the changed order of Pub queries may cause bad_C to be set in Game_3 when it was not set in Game_2, or vice versa. Second, queries to Priv in Game_3 can add entries to the reverse lookup table T. These new entries can be used to satisfy the conditions the simulator uses to check if a full execution of TKDF has been completed. Then the simulator in Game_3 may program responses that were not programmed in Game_2.

We claim that despite the changed order of the queries, Game_3 and Game_2 behave identically in the adversary's view except when one of them would set the bad_C flag, assuming that the same random coins

$\mathsf{Sim}(i, s)$

$\mathsf{Sim}[\mathsf{RO}](i, s)$:

1 if $M[s] \neq \bot$
2   then return $M[s]$
3 if $i = \mathsf{Th}$ then return $\mathsf{RO}_{\mathsf{Th}}(K \| Y)$
// If not, this query should simulate $\mathsf{RO}_{\mathsf{HMAC}}$
4 $K,\ Y \leftarrow s$
// Randomly sample a response
5 $y \xleftarrow{\$} \{0,1\}^{hl}$
6 if $Y = 0$
7   $T_{\text{PSK}}[y] \leftarrow K$
8 else if $K = 0$
9   $T_{\text{dHS}}[y] \leftarrow Y$
10 else if $T_{fk_b/fk_C/fk_S}[K] \neq \bot$
11   $\text{ES} \leftarrow T_{\text{ES}}[T_{\text{BK/CHTS/SHTS}}[K]]$
12   $\text{PSK} \leftarrow T_{\text{PSK}}[\text{ES}]$
13   if $\text{PSK} \neq \bot$
14     $y \leftarrow \mathsf{RO}_{binder}(\text{PSK}, Y)$
15     $\text{HTS} \leftarrow T_{\text{BK/CHTS/SHTS}}[K]$
16   $(\ell', \text{HS}, \mathsf{H}_2) \leftarrow T_{\text{HS}/d}[\text{HTS}]$
17   $(\text{dES}, \text{DHE}) \leftarrow T_{\text{dES/DHE}}[\text{HS}]$
18   $\text{PSK} \leftarrow T_{\text{PSK}}[T_{\text{ES/HS}}[\text{dES}]]$
19   if $\text{PSK} \neq \bot$
20     $y \leftarrow \mathsf{RO}_{\ell'[1]}(\text{PSK}, \text{DHE}, \mathsf{H}_2, Y)[\mathcal{L}(\ell)]$
21 else $T_{\text{dES/DHE}}[y] \leftarrow (K, Y)$
22 if $(Y[0\ldots2] \neq hl)$
  $\vee Y[2] < 8) \vee (Y[2] > 18)$
  $\vee (Y[3\ldots9] \neq \text{"}\mathsf{tls13}\text{"})$
  $\vee (Y[|Y|-1] \neq 1)$
// This query does not match $\mathsf{HKDF.Expand}$ formatting.
23     $M[s] \leftarrow y$
24   return y
// Parse the **Expand** formatting to find the label.
25 $\text{len}_\ell \leftarrow Y[2]$
26 $\ell \leftarrow Y[3\ldots(3+\text{len}_\ell)]$
27 $d \leftarrow Y[(3+\text{len}_\ell)\ldots|Y|]$
. . . // continued in next column

$\mathsf{Sim}[\mathsf{RO}](i, s)$ // continued:

28 if $\ell = \ell_{binder}$ and $d = \mathsf{H}(\texttt{""})$
29   $T_{\text{ES}}[y] \leftarrow K$
30 else if $\ell = \ell_{\text{dES/dHS}}$ and $d = \mathsf{H}(\texttt{""})$
31   $T_{\text{ES/HS}}[y] \leftarrow K$
32 else if $\ell \in \{\ell_{\text{CHTS}}, \ell_{\text{SHTS}}\}$
33   $T_{\text{HS}/d}[y] \leftarrow (\mathcal{L}(\ell), K, d)$
34 else if $\exists k \in \{\text{ETS}, \text{EEMS}\}$ with $\ell = \ell_k$ and $T_{\text{PSK}}[K] \neq \bot$
35   $y \leftarrow \mathsf{RO}_k(T_{\text{PSK}}[K], d)$
36 else if $\exists k \in \{\text{CATS}, \text{SATS}, \text{EMS}, \text{RMS}\}$ with $\ell = \ell_k$
37   $(\text{dES}, \text{DHE}) \leftarrow T_{\text{dES/DHE}}[T_{\text{ES/HS}}[T_{\text{dHS}}[K]]]$
38   $\text{PSK} \leftarrow T_{\text{PSK}}[T_{\text{ES/HS}}[\text{dES}]]$
39   if $\text{PSK} \neq \bot$
40     $y \leftarrow \mathsf{RO}_k(\text{PSK}, \text{DHE}, d)$
41 else if $\ell = \ell_{fk}$ and $d = \texttt{""}$
42   $T_{\text{BK/CHTS/SHTS}}[y] \leftarrow K$
43 else if $\ell \in \{\texttt{"tls13 key"}, \texttt{"tls13 iv"}\}$
44   and $d = \mathsf{H}(\texttt{""})$
45   $(\ell', \text{HS}, \mathsf{H}_2) \leftarrow T_{\text{HS}/d}[K]$
46   $(\text{dES}, \text{DHE}) \leftarrow T_{\text{dES/DHE}}[\text{HS}]$
47   $\text{PSK} \leftarrow T_{\text{PSK}}[T_{\text{ES/HS}}[\text{dES}]]$
48   if $\text{PSK} \neq \bot$
49     $y \leftarrow \mathsf{RO}_{\ell'[0]}(\text{PSK}, \text{DHE}, \mathsf{H}_2)[\mathcal{L}(\ell)]$

50 $M[s] \leftarrow y$
51 return y

Label translator $\mathcal{L}(\ell)$:

52 if $\ell = \ell_{\text{CHTS}}$
53   return htk_C, `ClientFinished`
54 if $\ell = \ell_{\text{SHTS}}$
55   return htk_S, `ServerFinished`
56 if $\ell = \texttt{"tls13 key"}$
57   return 0
58 if $\ell = \texttt{"tls13 iv"}$
59   return 1
60 return $\bot$

Figure 6: Simulator Sim used in the proof of Lemma 5.4.

are used in both games. Let E denote the event that bad_C is set either when $\mathcal{A}$ plays Game_2 or when $\mathcal{A}$ plays Game_3. Differences between the two games about when this flag is set are obviously irrelevant unless event E occurs.

The argument that PUB responses are identical in both games except when event E occurs is more subtle. Assume event E does not occur. There must be a first adversarial query to PUB that gives different responses in Game_3 and Game_2, all oracles behave identically in both games. We name this query Q. Both games sample the same random responses, so query Q has its response programmed by the simulator in at least one of the two games.

The simulator decides whether to program based on the entries of reverse lookup table T, so we consider the differences in this table between our two games. Let T_2 be the table in Game_2 at the time when Query Q is made, and let T_3 be the table at the same point in Game_4. Entries in the reverse lookup table are indexed by randomly sampled values y, so they cannot be overwritten by later queries unless event E occurs. Furthermore, until query Q is made, every PUB query in Game_2 that updates T gives the identical response in Game_3, so every entry in T_2 is also an entry in T_3. Therefore any query which is programmed in Game_2, up to and including query Q, will be programmed to the same response in Game_3. The contrapositive statement says that any response which is randomly sampled in Game_3 will be also be randomly sampled in Game_2.

It follows that query Q must have a randomly sampled response in Game_2 but be programmed in Game_3. There must exist a sequence of entries in T_3 that correspond to a full execution of $\mathsf{TKDF}[\text{PUB}]$ on some input. We name the queries that created these entries $Q_1, \ldots, Q_i$. In each execution, our simulator either stores an entry in T, or it programs the response y, never both. Therefore queries $Q_1, \ldots Q_i$ have randomly sampled responses. By the definition of TKDF, the output of each query Q_j is contained in the input of the next query Q_{j+1}. The output of Q_i is contained in the input of Q, so we identify query Q with Q_{i+1}.

In Game_2, one of the entries in the sequence is not present in T_2. Therefore one of the queries $Q_1, \ldots, Q_i$ is not made before query Q in Game_2. This query, Q_j must have been one of the FINALIZE queries of Game_2 that were moved earlier in Game_3. It will therefore be made in FINALIZE, after all of the other queries, including Q_{j+1}. The randomly sampled output of Q_j will collide with the input of earlier query Q_{j+1}, setting bad_C and causing event E to occur.

The difference in advantage in Game_3 and Game_2 is therefore bounded by the probability of event E. Both games make $\mathsf{q}_{\text{PUB}} + 6\mathsf{q}_{\text{PRIV}}$ queries to PUB, each of which sets bad_C is set with probability at most $\frac{2(\mathsf{q}_{\text{PUB}}+6\mathsf{q}_{\text{PRIV}})}{2^{hl}}$. By a union bound,

$$|\Pr[\mathsf{Game}_3] - \Pr[\mathsf{Game}_2]| \leq \frac{4(\mathsf{q}_{\text{PUB}} + 6\mathsf{q}_{\text{PRIV}})^2}{2^{hl}}.$$

Pseudocode for the last three games is given in Figure 5.1.3. Now we adjust PRIV in Game_4 to return the result of $\mathbf{C}[\text{PUB}]$ instead of querying RO. Unless bad_C is set, $\mathsf{TKDF}[\text{PUB}](r, X) = \mathsf{RO}_r(X)$. The function TKDF makes sequential queries to PUB that are properly formatted, so our Sim will program the last query in the sequence for consistency with the appropriate RO. This programming occurs every time $\mathsf{TKDF}[\text{PUB}]$ is called, unless the last query is a repeated query. In that case, it will be answered using table M instead of RO. However, if the queries in the sequence occur out of order, they will always cause bad_C to be set because the output of a later query will match the input to an earlier query. Then the adversary wins in Game_4 with the same likelihood as Game_3, unless bad_C is set. If bad_C is set, both games have a win probability of 0 thanks to the check in the FINALIZE oracle, so

$$\Pr[\mathsf{Game}_4] = \Pr[\mathsf{Game}_3].$$

Starting with Game_5, we stop returning 0 in FINALIZE when bad_C is set. This increases the win probability by at most $\Pr[\mathsf{Game}_4 \text{ sets } \mathsf{bad}_C] \leq \frac{2(\mathsf{q}_{\text{PUB}}+6\mathsf{q}_{\text{PRIV}})^2}{2^{hl}}$, by the same birthday and union bounds over the $\mathsf{q}_{\text{PUB}} + 6\mathsf{q}_{\text{PRIV}}$ queries to PUB.

$$|\Pr[\mathsf{Game}_5] - \Pr[\mathsf{Game}_4]| \leq \frac{2(\mathsf{q}_{\text{PUB}} + 6\mathsf{q}_{\text{PRIV}})^2}{2^{hl}}.$$

From Game_4 onward, all queries to $\mathsf{RO}_{\mathsf{HMAC}}$ are made by Sim. In Game_6, therefore, we can inline the lazily sampled $\mathsf{RO}_{\mathsf{HMAC}}$ oracle as part of the simulator. Repeated queries to Sim are cached, so the random oracle does not need to maintain its own lookup table. Now all responses from PUB are randomly sampled from $\{0,1\}^{hl}$, regardless of the contents of table T. The table and the conditional statements used to maintain it

are now redundant bookkeeping, as is the unused bad_C flag after Game_5. We eliminate all of this code from Game_6 without detection by the adversary. Then

$$\Pr[\mathsf{Game}_6] = \Pr[\mathsf{Game}_5].$$

The remaining code of Sim just implements random oracles $\mathsf{RO}_{\mathsf{HMAC}}$ and $\mathsf{RO}_{\mathsf{Th}}$. Consequently Game_6 is identical to the ideal indifferentiability game for the TKDF construction. Collecting bounds proves the theorem.

□

We have now established that in order to give a (tight) security proof for TLS 1.3 PSK-only and PSK-(EC)DHE, it suffices to prove (tight) security of the protocol on the right-hand side of Figure 1.

6 Modularizing Handshake Encryption

Next will argue that using "internal" keys to encrypt handshake messages on the TLS 1.3 record-layer does not impact the security of other keys established by the handshake. Theorem 6.2 below formulates our argument in a general way, applicable to any multi-stage key exchange protocol, so that future analyses of similar protocols might take advantage of this modularity as well.

Intuitively, we argue as follows. Let KE_2 be a protocol that provides multiple different stages with different external keys (i.e., none of the keys is used in the protocol, e.g., to encrypt messages), and let KE_1 be the same protocol, except that some keys are "internal" and used, e.g., to encrypt certain protocol messages. We argue that either using "internal" keys in KE_1 does not harm the security of *other* keys of KE_1, or KE_2 cannot be secure in the first place. This will establish that we can prove security of a variant TLS 1.3 *without* handshake encryption (in an accordingly simpler model), and then lift this result to the actual TLS 1.3 protocol *with* handshake encryption and the handshake traffic keys treated as "internal" keys.

Theorem 6.1. *Let KE_1 be the TLS 1.3 PSK-only resp. PSK-(EC)DHE mode* with *handshake encryption (i.e., with internal stages $\mathsf{KE}_1.\mathsf{INT} = \{3, 4\}$) as specified on the right-hand side in Figure 1. Let KE_2 be the same mode* without *handshake encryption (i.e., $\mathsf{KE}_1.\mathsf{INT} = \emptyset$ and AEAD-encryption/decryption of messages is omitted). Let $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ be the AEAD encryption resp. decryption algorithms deployed in TLS 1.3 and $\mathsf{K}_{\mathsf{Transform}} = \mathsf{KE}_1.\mathsf{INT} = \{3, 4\}$. Then we have*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(t + t_{\mathrm{AEAD}} \cdot q_{\mathrm{S}}, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}} + q_{\mathrm{S}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}),$$

where t_{AEAD} is the maximum time required to execute AEAD encryption or decryption of TLS 1.3 messages.

For TLS 1.3 this means that we will not consider any security guarantees provided by the additional encryption of handshake messages. We consider this as reasonable for PSK-mode ciphersuites, because the main purposes of handshake message encryption in TLS 1.3 is to hide the identities of communicating parties, e.g., in digital certificates, cf. [1]. In PSK mode there are no such identities. The *pskid* might be viewed as a string that could identify communicating parties, but it is sent unencrypted in the `ClientHello` message, anyway, the encryption of subsequent handshake messages would not contribute to its protection.

6.1 Handshake Encryption as a Modular Transformation

Formally, let $\mathsf{KE}_2 = (\mathsf{KGen}, \mathsf{Activate}, \mathsf{Run})$ be a key exchange protocol with no internal keys. We define another key exchange protocol KE_1 which is parameterized by two functions $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ and a list $\mathsf{K}_{\mathsf{Transform}} \subseteq \{1, \ldots, \mathsf{STAGES}\}$, where STAGES is the number of stages of KE_2. KE_1 inherits its key generation and activation algorithms from KE_2. In its $\mathsf{KE}_1.\mathsf{Run}$ algorithm, described in Figure 11, it essentially applies $\mathsf{Transform}_{\mathsf{Recv}}$ to a message before calling $\mathsf{KE}_2.\mathsf{Run}$, and then $\mathsf{Transform}_{\mathsf{Send}}$ to the returned message, to transform the protocol messages as they pass over a wire. This transformation may be, for instance, the encryption and decryption of messages of KE_2 using an internal key.

In addition to the messages, both algorithms take as input the list of stages that have been accepted by the current session, its role (initiator or responder) in the protocol, and a list of the keys from all stages

Game Game_0

$\textsc{Initialize}()$:

1 $b \leftarrow 0$
2 $\mathsf{RO} \xleftarrow{\$} \mathsf{ES}$
3 $state \xleftarrow{\$} \varepsilon$

$\mathsf{Sim}(i, s, state)$:

1 if $i = \mathsf{Th}$ then return $\mathsf{RO}_{\mathsf{Th}}, (s)$
2 $T, M \leftarrow state$
3 if $M[s] \neq \bot$
4 then return $M[s]$
5 $K, Y \leftarrow s$
6 $y \leftarrow \mathsf{Sim}[\mathsf{RO}](K, Y, T)$
7 $M[s] \leftarrow y$
8 return y

$\mathsf{Sim}[\mathsf{RO}](K, Y, T)$:

// Randomly sample a response
9 $y \xleftarrow{\$} \{0,1\}^{hl}$
10 if $Y = 0$
11 $T_{\mathrm{PSK}}[y] \leftarrow K$
12 else if $K = 0$
13 $T_{\mathrm{dHS}}[y] \leftarrow Y$
14 else if $T_{fk_b/fk_C/fk_S}[K] \neq \bot$
15 $\mathrm{ES} \leftarrow T_{\mathrm{ES}}[T_{\mathrm{BK/CHTS/SHTS}}[K]]$
16 $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[\mathrm{ES}]$
17 if $\mathrm{PSK} \neq \bot$
18 $y \leftarrow \mathsf{RO}_{binder}(\mathrm{PSK}, Y)$
19 $\mathrm{HTS} \leftarrow T_{\mathrm{BK/CHTS/SHTS}}[K]$
20 $(\ell', \mathrm{HS}, \mathsf{H}_2) \leftarrow T_{\mathrm{HS}/d}[\mathrm{HTS}]$
21 $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[\mathrm{HS}]$
22 $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
23 if $\mathrm{PSK} \neq \bot$
24 $y \leftarrow \mathsf{RO}_{\ell'[1]}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{H}_2, Y)[\mathcal{L}(\ell)]$
25 else $T_{\mathrm{dES/DHE}}[y] \leftarrow (K, Y)$
26 if $(Y[0\ldots2] \neq hl)$
$\vee Y[2] < 8) \vee (Y[2] > 18)$
$\vee (Y[3\ldots9] \neq \text{"tls13"})$
$\vee (Y[|Y|-1] \neq 1)$
// This query does not match HKDF.Expand formatting.
27 return y
// Parse the **Expand** formatting to find the label.
28 $\mathrm{len}_\ell \leftarrow Y[2]$
29 $\ell \leftarrow Y[3\ldots(3+\mathrm{len}_\ell)]$
30 $d \leftarrow Y[(3+\mathrm{len}_\ell)\ldots|Y|]$
... // continued in next column

$\mathsf{Sim}[\mathsf{RO}](K, Y, T)$ // ...continued:

31 if $\ell = \ell_{binder}$ and $d = \mathsf{H}(\texttt{""})$
32 $T_{\mathrm{ES}}[y] \leftarrow K$
33 else if $\ell = \ell_{\mathrm{dES/dHS}}$ and $d = \mathsf{H}(\texttt{""})$
34 $T_{\mathrm{ES/HS}}[y] \leftarrow K$
35 else if $\ell \in \{\ell_{\mathrm{CHTS}}, \ell_{\mathrm{SHTS}}\}$
36 $T_{\mathrm{HS}/d}[y] \leftarrow (\mathcal{L}(\ell), K, d)$
37 else if $\exists k \in \{\mathrm{ETS}, \mathrm{EEMS}\}$ with $\ell = \ell_k$ and $T_{\mathrm{PSK}}[K] \neq \bot$
38 $y \leftarrow \mathsf{RO}_k(T_{\mathrm{PSK}}[K], d)$
39 else if $\exists k \in \{\mathrm{CATS}, \mathrm{SATS}, \mathrm{EMS}, \mathrm{RMS}\}$ with $\ell = \ell_k$
40 $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[T_{\mathrm{ES/HS}}[T_{\mathrm{dHS}}[K])]]$
41 $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
42 if $\mathrm{PSK} \neq \bot$
43 $y \leftarrow \mathsf{RO}_k(\mathrm{PSK}, \mathrm{DHE}, d)$
44 else if $\ell = \ell_{fk}$ and $d = \texttt{""}$
45 $T_{\mathrm{BK/CHTS/SHTS}}[y] \leftarrow K$
46 else if $\ell \in \{\texttt{"tls13 key"}, \texttt{"tls13 iv"}\}$
47 and $d = \mathsf{H}(\texttt{""})$
48 $(\ell', \mathrm{HS}, \mathsf{H}_2) \leftarrow T_{\mathrm{HS}/d}[K]$
49 $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[\mathrm{HS}]$
50 $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
51 if $\mathrm{PSK} \neq \bot$
52 $y \leftarrow \mathsf{RO}_{\ell'[0]}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{H}_2)[\mathcal{L}(\ell)]$

53 return y

$\textsc{Pub}(i, s)$:

1 $(z, state) \leftarrow \mathsf{Sim}(i, s, state)$
2 return z

$\textsc{Priv}(r, X)$:

1 return $\mathsf{RO}_r(X)$

$\textsc{Finalize}(b')$:

1 return b'

Figure 7: Indiff game instantiated with simulator Sim, also Game Game_0 in the proof of Lemma 5.4.

Games Game_1

```
Sim(i, s, state):
1 if i = Th then return RO_Th(s)
2 T, M, L ← state
3 if M[s] ≠ ⊥
4   then return M[s]
5 K, Y ← s
6 y ← Sim[RO](K, Y, T, L)
7 M[s] ← y
8 L ← L ∪ {y, s}
9 return y
```

```
Sim[RO](K, Y, T, L):
10 y ←$ {0,1}^hl
11 if y ∈ L or ∃t ∈ L such that y ∈ t
12   bad_C ← true
   ...

Finalize(b'):
1 if bad_C then return 0
2 return b'
```

Figure 8: Game Game_1 in the proof of Lemma 5.4.

Game Game_2

```
Priv(r, X):
1 Q ← Q ⋃ {(r, X)}
2 return RO_r(X)

Finalize(b'):
1 for (r, X) ∈ Q do
2   z ← TKDF_r[Pub](X)
3 if bad_C then return 0
4 return b'
```

Game Game_3

```
Priv(r, X):
1 z ← TKDF_r[Pub](X)
2 return RO_r(X)

Finalize(b'):
1 if bad_C then return 0
2 return b'
```

Figure 9: Games Game_2 and Game_3 in the proof of Lemma 5.4.

Games $\boxed{\mathsf{Game}_4}$, Game_5

```
Priv(r, X):
1 z ← TKDF_r[Pub](X)
2 return z

Finalize(b'):
1 [if bad_C then return 0]
2 return b'
```

Game Game_6

```
Sim[RO](i, s, T):
1 y ←$ {0,1}^hl
2 return y
```

Figure 10: Games Game_4, Game_5, and Game_6 in the proof of Lemma 5.4.

```
KE_1.Run(u, π_u^i, psk, m):
1 keys ← (π_u^i.skey[stage] for stage ∈ K_Transform)
2 acc ← (π_u^i.accepted[stage] ≠ ∞ for stage in [1...STAGES])
3 m̃ ← Transform_Recv(keys, π_u^i.role, acc, m)
4 (π_u^i, m̃') ← KE_2.Run(u, π_u^i, psk, m̃)
5 keys ← (π_u^i.skey[stage] for stage ∈ K_Transform)
6 acc ← (π_u^i.accepted[stage] ≠ ∞ for stage in [1...STAGES])
7 m' ← Transform_Send(keys, π_u^i.role, acc, m̃')
8 return (π_u^i, m')
```

Figure 11: Key exchange KE_1 built by transforming protocol messages of KE_2.

in $\mathsf{K}_{\mathsf{Transform}}$. In the security game for KE_1, the stages in $\mathsf{K}_{\mathsf{Transform}}$ will produce internal keys; all other keys remain external.

Although $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ change the messages as they pass over the wire, the way that the messages are processed after receipt by $\mathsf{KE}_2.\mathsf{Run}$ must not change. In particular, $\mathsf{KE}_2.\mathsf{Run}$, internally run within $\mathsf{KE}_1.\mathsf{Run}$, still expects messages of the same format and content; also, KE_1 defines its session and contributive identifiers, as well as all other session-specific information in the same way as KE_2.

Correctness. Not all choices of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ are "good choices". For example, if mauling overwrites critical pieces of the protocol messages, then no honest session would ever accept a key. The resulting key exchange KE_2 would be vacuously "secure" because it would be unusable.

For our perspective to be meaningful, we therefore need a correctness property that guarantees that two honest parties executing KE_1 with no adversarial interference will accept at all stages. Informally, we wish that if two sessions honestly executing KE_2 will accept keys for stage s with probability p, then two sessions honestly executing KE_1 will accept keys for stage s with probability close to p. This property only needs to hold when the protocol messages are relayed honestly, with no changes or delivery failures beyond those caused by the application of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$.

We do not give a formal definition or proof of correctness for TLS 1.3, but we note that in TLS 1.3, the transformation algorithms are AEAD encryption and decryption. Since decryption failures cannot occur in the standardized AEAD algorithms if messages are honestly relayed (due to their perfect correctness), received messages will always match their corresponding sent message, and correctness of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ follows.

Security. We wish KE_1 to be secure if KE_2 is secure. This should be independent of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$, i.e., should hold even if $\mathsf{Transform}_{\mathsf{Send}}$ leaks its keys and fully overwrites all protocol messages. The following theorem established this result, using that the keys used for the transformation are internal and $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ have no access to other privileged information. Therefore, their behavior can be mimicked by a reduction to the security of KE_2 as long as KE_2 has "public session matching" for the stages in $\mathsf{K}_{\mathsf{Transform}}$ of KE_1, i.e., session partnering (or matching) for those stages is decidable from the publicly exchanged messages.[8]

Theorem 6.2. *Let KE_2 be a key exchange protocol with STAGES stages, $\mathsf{KE}_2.\mathsf{INT}$ being empty, and public session matching. Let $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ be algorithms as above and $\mathsf{K}_{\mathsf{Transform}} \subseteq \{1, \ldots, \mathsf{STAGES}\}$. Define key exchange KE_1 such that $\mathsf{KE}_1.\mathsf{Run}$ is described in Figure 11, $\mathsf{KE}_1.\mathsf{INT} = \mathsf{K}_{\mathsf{Transform}}$, and all other attributes of KE_1 are identical to those of KE_2.*

Let $\mathcal{A}$ be an adversary with running time t against the multi-stage key exchange security of KE_1, making q_{S} queries to the Send *oracle. Then there exists an adversary $\mathcal{B}$ with running time $\approx t + q_{\mathrm{S}} m$, where m is the maximum running time of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$, such that*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(\mathcal{B}).$$

$\mathcal{B}$ makes at most q_{S} queries to RevSessionKey *in addition to queries made by $\mathcal{A}$ and the same number of queries as $\mathcal{A}$ to all other oracles in the MSKE game.*

Proof. Adversary $\mathcal{B}$ runs adversary $\mathcal{A}$ and relays all of its queries to the appropriate oracles in its own MSKE game, except for Send queries. It maintains the time time of the MSKE game itself, incrementing it once per query. For each session π_u^i, it maintains a list $keys_u^i$ that is initially empty and a list acc_u^i in which $acc_u^i[stage]$ is initially false for each $stage \in \mathsf{K}_{\mathsf{Transform}}$.

When $\mathcal{A}$ makes a query $\mathrm{Send}(u, i, m)$, $\mathcal{B}$ first checks for each $stage \in \mathsf{K}_{\mathsf{Transform}}$ with $acc_u^i[stage] = \mathsf{false}$ whether $\pi_u^i.\mathsf{accepted}[stage] \neq \infty$. For each $stage$ which satisfies this condition, $\mathcal{B}$ checks whether $\pi_u^i.\mathsf{tested}[stage]$ or $\pi_u^i.\mathsf{revealed}[stage]$ is true and if π_u^i has a partnered session (matching $sid[stage]$) which has been tested or revealed. (The latter check for partnering is possible because KE_1 has public session matching.) If any of these conditions is true, then $\mathcal{B}$ knows $\pi_u^i.skey[stage]$. Otherwise, it makes an extra

[8] The property of "public session matching" has already already come up when considering the composition of (regular or multi-stage) key exchange protocols with subsequent symmetric-key protocols [12, 22, 23, 35].

query $\textsc{RevSessionKey}(u, i, stage)$ and adds the response to $keys_u^i$. Then it marks $acc_u^i[stage] \leftarrow \mathsf{true}$ and computes $\tilde{m} \leftarrow \mathsf{Transform}_{\mathsf{Recv}}(keys_u^i, \pi_u^i.role, acc_u^i, m)$. It queries its own $\textsc{Send}$ oracle on the tuple $(u, i, \tilde{m})$ and captures the response $\tilde{m}'$. Then it returns $m' \leftarrow \mathsf{Transform}_{\mathsf{Send}}(keys_u^i, \pi_u^i.role, acc_u^i, \tilde{m}')$ to $\mathcal{A}$.

$\mathcal{B}$ perfectly simulates KE_1 for $\mathcal{A}$, so we wish that if $\mathcal{A}$ wins its simulated game, $\mathcal{B}$ should also win its game. $\mathcal{A}$ can win the MSKE game in one of three ways: it can violate the Sound predicate, it can violate the $\mathsf{ExplicitAuth}$ predicate, or it can satisfy the Fresh predicate and guess the secret bit b. All of the variables tracked by the $\mathsf{ExplicitAuth}$ and Sound predicates are maintained by the MSKE game for KE_1, not by $\mathcal{B}$. Therefore $\mathcal{A}$ wins the simulated game by violating Sound or $\mathsf{ExplicitAuth}$ only if Sound or $\mathsf{ExplicitAuth}$ is violated in the MSKE game for KE_2. In this case, $\mathcal{B}$ also wins.

If $\mathcal{A}$ wins by guessing the secret bit b, the story is more complicated. The bit b is chosen by the MSKE game, so if $\mathcal{A}$ guesses correctly, then so will $\mathcal{B}$. However, a correct guess only matters if the queries do not violate the Fresh predicate. Even if $\mathcal{A}$ did not violate the Fresh predicate, $\mathcal{B}$ makes up to q_{S} additional $\textsc{RevSessionKey}$ queries. Each of these could cause Fresh to be set to false. We claim that none of these queries violate the Fresh predicate.

The Fresh predicate requires that no session be both tested and revealed. $\mathcal{B}$ only reveals keys that have not already been tested, so the only worry is that $\mathcal{A}$ will test this key later. However, all keys that $\mathcal{B}$ reveals are in $\mathsf{K}_{\mathsf{Transform}}$, which is a subset of $\mathsf{KE}_1.\mathsf{INT}$, meaning they are internal keys. These keys cannot be tested if any session which has accepted it has moved on with the protocol. Since $\mathcal{B}$ only reveals a key when a session has both accepted that key and received the next protocol message, it will have moved on and $\mathcal{A}$ can not make any later $\textsc{Test}$ queries on a key that $\mathcal{B}$ has revealed.

The next condition of Fresh is that a tested session's partner cannot be tested or revealed. $\mathcal{B}$ ensures that such a $\textsc{Test}$ query does not occur before the $\textsc{RevSessionKey}$ query. Again, the $\textsc{Test}$ query cannot happen after the $\textsc{RevSessionKey}$ query because the session whose key was revealed has moved on with the protocol. Since all the revealed keys are internal in the simulated game, $\mathcal{A}$ cannot test them after this point.

The remaining three conditions of the Fresh predicate establish different levels of forward secrecy. They check for the existence of a contributive partner. We want to exclude the situation that a contributive partner exists in $\mathcal{A}$'s simulated game, but not in $\mathcal{B}$'s game. However, contributive identifiers are defined identically in KE_1 and KE_2. Therefore if two sessions π_u^i and π_v^j have matching contributive identifiers in the simulated game for KE_2, they will also have matching identfiers in the game for KE_1.

It is therefore not possible for $\mathcal{A}$ to win its simulated MSKE game unless $\mathcal{B}$ also wins its MSKE game, and the theorem follows. □

7 Tight Security of the TLS 1.3 PSK Modes

In this section, we apply the insights gained in Sections 5 and 6 to obtain tight security bounds for both the PSK-only and the PSK-(EC)DHE mode of TLS 1.3. To that end, we first present the protocol-specific properties of the TLS 1.3 PSK-only and PSK-(EC)DHE modes such that they can be viewed as multi-stage key exchange (MSKE) protocols as defined in Section 3. Then, we prove tight security bounds in the MSKE model in Theorem 7.1 for the TLS 1.3 PSK-(EC)DHE mode and in Theorem 7.7 for the TLS 1.3 PSK-only mode.

7.1 TLS 1.3 PSK-only/PSK-(EC)DHE as a MSKE Protocol

We begin by capturing the TLS 1.3 PSK-only and PSK-(EC)DHE modes, specified in Figure 1, formally as MSKE protocols. To this end, we must explicitly define the variables discussed in Section 3. In particular, we have to define the stages themselves, which stages are internal and which replayable, the session and contributive identifiers, when stages receive explicit authentication, and when stages become forward secret.

Stages. The TLS 1.3 PSK-only/PSK-(EC)DHE handshake protocol has eight stages (i.e., $\mathsf{STAGES} = 8$), corresponding to the keys ETS, EEMS, htk_S, htk_C, CATS, SATS, EMS, and RMS in that order. The set INT of internal keys contains htk_C and htk_S, the handshake traffic encryption keys. Stages ETS and EEMS are replayable: $\mathsf{REPLAY}[s]$ is true for $s \in \{1, 2\}$ and false for all others.

Session and contributive identifiers. The session and contributive identifiers for stage$s$ are tuples $(label_s, ctxt)$, where $label_s$ is a unique label identifying stage s, and $ctxt$ is the transcript that enters key's derivation. The session identifiers $(sid[s])_{s \in \{1,\dots,8\}}$ are defined as follows:[9]

$$
\begin{aligned}
sid[1] &= \big(\text{“ETS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\big)\,, \\
sid[2] &= \big(\text{“EEMS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\big)\,, \\
sid[3] &= \big(\text{“}htk_C\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK})\big)\,, \\
sid[4] &= \big(\text{“}htk_S\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK})\big)\,, \\
sid[5] &= \big(\text{“CATS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\big)\,, \\
sid[6] &= \big(\text{“SATS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\big)\,, \\
sid[7] &= \big(\text{“EMS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\big)\,, \text{ and} \\
sid[8] &= \big(\text{“RMS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF}, \mathtt{CF})\big)\,.
\end{aligned}
$$

To make sure that a server that received `ClientHello`, `ClientKeyShare`†, and `ClientPreSharedKey` untampered can be tested in stages 3 and 4, even if the sending client did not receive the server's answer, we set the contributive identifiers of stages 3 and 4 such that cid_{role} reflects the messages that a session in role *role* must have honestly received for testing to be allowed. Namely, we let clients (resp. servers) upon sending (resp. receving) the messages $(\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})$ set

$$
\begin{aligned}
cid_{\mathsf{responder}}[3] &= \big(\text{“}htk_C\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\big) \text{ and} \\
cid_{\mathsf{responder}}[4] &= \big(\text{“}htk_S\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\big)\,.
\end{aligned}
$$

Further, when the client receives (resp. the server sends) the message $(\mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK})$, they set

$$cid_{\mathsf{initiator}}[3] = sid[3] \quad \text{and} \quad cid_{\mathsf{initiator}}[4] = sid[4].$$

For all other stages $s \in \{1, 2, 5, 6, 7, 8\}$, $cid_{\mathsf{initiator}}[s] = cid_{\mathsf{responder}}[s] = sid[s]$ is set upon acceptance of the respective stage (i.e., when $sid[s]$ is set as well).

Explicit authentication. For initiator sessions, all stages achieve explicit authentication when the `ServerFinished` message is verified successfully. This happens right before stage 5 (i.e., CATS) is accepted. That is, upon accepting stage 5 all previous stages receive explicit authentication retroactively and all following stages are explicitly authenticated upon acceptance. Formally, we set $\mathsf{EAUTH}[\mathsf{initiator}, s] = 5$ for all stages $s \in \{1, \dots, 8\}$.

For responder session, all stages receive explicit authentication upon (successful) verification of the `ClientFinished` message. This occurs right before the acceptance of stage 8 (i.e., RMS). Similar to initiators, responders receive explicit authentication for all stages upon acceptance of stage 8 since this is the last stage of the protocol. Accordingly, we set $\mathsf{EAUTH}[\mathsf{responder}, s] = 8$ for all stages $s \in \{1, \dots, 8\}$.

Forward secrecy. Only keys dependent on a Diffie–Hellman secret achieve forward secrecy, so all stages s of the PSK-only handshake have $\mathsf{FS}[r, s, \mathsf{fs}] = \mathsf{FS}[r, s, \mathsf{wfs2}] = \infty$ for both roles $r \in \{\mathsf{initiator}, \mathsf{responder}\}$. In the PSK-(EC)DHE handshake, full forward secrecy is achieved at the same stage as explicit authentication for all keys except ETS and EEMS, which are never forward secret. That is, for both roles r and stages $s \in \{3, \dots, 8\}$ we have $\mathsf{FS}[r, s, \mathsf{fs}] = \mathsf{EAUTH}[r, s]$. All keys except ETS and EEMS possess weak forward secrecy 2 upon acceptance, so we set $\mathsf{FS}[r, s, \mathsf{wfs2}] = s$ for stages $s \in \{3, \dots, 8\}$. Finally, as stages 1 and 2 (i.e., ETS and EEMS) never achieve forward secrecy we set $\mathsf{FS}[r, s, \mathsf{fs}] = \mathsf{FS}[r, s, \mathsf{wfs2}] = \infty$ for both roles r and stages $s \in \{1, 2\}$.

[9]Components marked with † are only part of the TLS 1.3 PSK-(EC)DHE handshake.

7.2 Tight Security Analysis of TLS 1.3 PSK-(EC)DHE

We now come to the tight MSKE security result for the TLS 1.3 PSK-(EC)DHE handshake.

Theorem 7.1. *Let* $\texttt{TLS1.3-PSK-(EC)DHE}$ *be the TLS 1.3 PSK-(EC)DHE handshake protocol (with optional 0-RTT) as specified on the right-hand side in Figure 1 without handshake encryption. Let* $\mathbb{G}$ *be the Diffie–Hellman group of order* p*. Let* nl *be the length in bits of the nonce, let* hl *be the output length in bits of* $\mathbf{H}$*, and let the pre-shared key space be* $\mathsf{KE.PSKS} = \{0,1\}^{hl}$*. We model the functions* $\mathbf{H}$ *and* TKDF_x *for each* $x \in \{binder, \ldots, \mathrm{RMS}\}$ *as* 12 *independent random oracles* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{binder}, \ldots, \mathsf{RO}_{\mathrm{RMS}}$*. Then,*

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\texttt{TLS1.3-PSK-(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p}$$
$$+ \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}} + \frac{4(t + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p}.$$

Remark 7.2. Our MSKE model from Section 3 assumes pre-shared keys to be uniformly random sampled from $\mathsf{KE.PSKS}$, where here $\mathsf{KE.PSKS} = \{0,1\}^{hl}$. This matches how pre-shared keys are derived for session resumption, as well as our analysis of domain separation, which assumes pre-shared keys to be of length hl.

Remark 7.3. Our bound is easily adapted to any distribution on $\{0,1\}^{hl}$ in order to accommodate out-of-band pre-shared keys that satisfy the length requirement but do not have full entropy. Expectedly, lower-entropy PSK distributions result in weaker bounds, due to the increased chance for collisions between PSKs as well as the adversary guessing a PSK.

Remark 7.4. In order to deal with small subgroup attacks, note that we assume that implementations properly validate received key shares by checking for membership in the appropriate prime-order group. This has to be done explicitly for NIST curves (`secp256r1`, `secp384r1`, and `secp521r1` in TLS 1.3 [54, Section 4.2.8.2]). Curves like `x25519` and `x448` rule out small subgroup attacks implicitly, with a mechanism called "clamping". In our proof we treat Diffie–Hellman groups as prime-order groups with uniform exponents in $\mathbb{Z}_p$, as common in the cryptographic literature. However, we stress that clamping as in RFC 7748 [48] makes exponents non-uniform over $\mathbb{Z}_p$. Hence, we implicitly assume that this difference in the DH key generation is indistinguishable for the adversary.

Proof. To prove our bound, we make an incremental series of changes to the key exchange security game $G_{\texttt{TLS1.3-PSK-(EC)DHE},\mathcal{A}}$. We divide the proof into three phases reflecting the three ways of the adversary to win the security game.

1. We establish that the adversary cannot violate the predicate Sound.
2. We establish the same for the predicate $\mathsf{ExplicitAuth}$.
3. Finally, we ensure that all TEST queries return uniformly random keys independent of the challenge bit b if predicate Fresh is not violated.

We can then conclude that the adversary cannot do better than random guessing to win the game, i.e., its advantage is 0.

GAME 0 (Initial game). The initial game $\mathsf{Game}_0^{\mathcal{A}}$ is the key exchange security game $G_{\texttt{TLS1.3-PSK-(EC)DHE},\mathcal{A}}$ played for the TLS 1.3 PSK-(EC)DHE handshake (with optional 0-RTT) as specified in Figure 1 (right), but without handshake encryption. Note that the functions $\mathbf{H}$ and TKDF_x for $x \in \{binder, \ldots, \mathrm{RMS}\}$ are modeled as 12 independent random oracles $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{binder}, \ldots, \mathsf{RO}_{\mathrm{RMS}}$. We implement random oracle RO_x by a look-up table ROList_x assigning inputs to outputs. We assume that every look-up table implementing a random oracle is stored in a data structure that enables constant time access when indexed either by random oracle inputs or by random oracle outputs, using two hash tables, for instance. By definition, we have

$$\Pr[\mathsf{Game}_0^{\mathcal{A}} \Rightarrow 1] = \mathrm{Adv}^{\mathsf{MSKE}}_{\texttt{TLS1.3-PSK-(EC)DHE}}(\mathcal{A}).$$

Phase 1: Ensuring Predicate Sound cannot be violated

GAME 1 (Exclude collisions of nonces and group elements). In $\mathsf{Game}_1^{\mathcal{A}}$, we eliminate collisions among nonces and group elements computed by honest sessions via two new flags:

- bad_C is set when two honest sessions choose the same nonce and group element, and
- bad_O is set when an honest responder samples some nonce and group element that have already been received by another session. We view this nonce and group element as having been chosen by an adversarial session.

If either bad_C or bad_O is set, the game returns 0 from FINALIZE.

By the well-known identical-until-bad-lemma [6, Lemma 2], we get

$$\Pr[\mathsf{Game}_0^{\mathcal{A}} \Rightarrow 1] \leq \Pr[\mathsf{Game}_1^{\mathcal{A}} \Rightarrow 1] + \Pr[\mathsf{Game}_1^{\mathcal{A}} \text{ sets } \mathsf{bad}_C] + \Pr[\mathsf{Game}_1^{\mathcal{A}} \text{ sets } \mathsf{bad}_O]. \tag{1}$$

Let us separately analyze the probabilities that $\mathsf{Game}_1^{\mathcal{A}}$ sets the flags bad_C and bad_O. Each SEND query causes at most one session to uniformly and independently sample a nonce $r \xleftarrow{\$} \{0,1\}^{nl}$ and a group element $g \xleftarrow{\$} \mathbb{G}$. If the bad_C flag is set, we have that there exists some SEND query that creates a session π_u^i using $\mathsf{Activate}$. This new session samples nonce and group element(r, g) which were previously sampled by another session $\pi_{u'}^{i'}$. That is, the probability for bad_C to be set is the probability of a collision among the (up to) q_{S} pairs of uniformly and independently sampled nonces and group elements; we can use the birthday bound to bound the probability of setting bad_C by

$$\Pr[\mathsf{Game}_1^{\mathcal{A}} \text{ sets } \mathsf{bad}_C] \leq \frac{q_{\mathrm{S}}^2}{2^{nl} \cdot p}. \tag{2}$$

where q_{S} is the number of SEND queries.

Next, if the game sets bad_O, we have that there is a SEND query which creates a new session π_v^j. This session samples a nonce $r_S \xleftarrow{\$} \{0,1\}^{nl}$ and a group element $Y \xleftarrow{\$} \mathbb{G}$, which were already received by another session π_u^i. There are at most q_{S} sessions, so there are no more than q_{S} received pairs which which (r_S, Y) can collide. Since π_v^j samples its nonce and group element uniformly and independently at random from $\{0,1\}^{nl} \times \mathbb{G}$, we get by the union bound that the probability that π_v^j samples one of the already received pairs is bounded from above by $q_{\mathrm{S}}/(2^{nl} \cdot p)$. Overall, we again get by the union bound that there is such a collision for any π_v^j with probability

$$\Pr[\mathsf{Game}_1^{\mathcal{A}} \text{ sets } \mathsf{bad}_O] \leq q_{\mathrm{S}} \cdot \frac{q_{\mathrm{S}}}{2^{nl} \cdot p} = \frac{q_{\mathrm{S}}^2}{2^{nl} \cdot p}. \tag{3}$$

Combining Equations (1)–(3), we get

$$\Pr[\mathsf{Game}_0^{\mathcal{A}^*} \Rightarrow 1] \leq \Pr[\mathsf{Game}_1^{\mathcal{A}^*} \Rightarrow 1] + \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p}. \tag{4}$$

GAME 2 (Exclude binder collisions). In game $\mathsf{Game}_2^{\mathcal{A}}$, we let the adversary lose if there is a collision among the binder values computed by any honest session. Whenever two distinct queries to RO_{binder} return the same value, we set a flag bad_{binder} and return 0 from FINALIZE.

To implement this, we add a table $\mathsf{CollList}_{binder}$ to the random oracle RO_{binder} (this table is currently redundant to the table implementing RO_{binder}, but will be useful in later game hops, where we will introduce changes such that it is not guaranteed anymore that all *binder* values will be contained in the RO_{binder} table). Whenever RO_{binder} computes a binder value $b = \mathsf{RO}_{binder}(\mathrm{PSK}, ctxt)$, we log $\mathsf{CollList}_{binder}[b] \leftarrow (\mathrm{PSK}, ctxt)$. Now, whenever RO_{binder} computes some binder b for some tuple s and $\mathsf{CollList}_{binder}[b]$ is not empty, there has to be a tuple $s' = (\mathrm{PSK}, ctxt)$ with $\mathsf{RO}_{binder}(psk, ctxt) = b$ queried before and we have found a collision if $s \neq s'$. In this case we set bad_{binder}.

Again by the identical-until-bad-lemma,

$$\Pr[\mathsf{Game}_1^{\mathcal{A}} \Rightarrow 1] \leq \Pr[\mathsf{Game}_2^{\mathcal{A}} \Rightarrow 1] + \Pr[\mathsf{Game}_2^{\mathcal{A}} \text{ sets } \mathsf{bad}_{binder}].$$

To bound the probability that the game sets flag bad_{binder}, we construct a reduction $\mathcal{B}_1$ to the collision-resistance of RO_{binder}. The reduction $\mathcal{B}_1$ simulates Game 2 for adversary $\mathcal{A}$. It implements all oracles itself except for RO_{binder}. $\mathcal{B}_1$ will need to query its own oracle RO_{binder} at most once per RO query and once per SEND query, so it makes $q_{\mathrm{RO}} + q_{\mathrm{S}}$ queries in total. If the flag bad_{binder} would be set in Game 2, which can be checked efficiently using $\mathsf{CollList}_{binder}$ as described before, then the reduction has found a collision (s, s') with $s \neq s'$ such that $\mathsf{RO}_{binder}(s) = \mathsf{RO}_{binder}(s')$. Reduction $\mathcal{B}_1$ then outputs (s, s') and wins the collision-resistance game.

Therefore, we have that

$$\Pr[\mathsf{Game}_1^{\mathcal{A}} \Rightarrow 1] \leq \Pr[\mathsf{Game}_2^{\mathcal{A}} \Rightarrow 1] + \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{RO}_{binder}}(q_{\mathrm{RO}} + q_{\mathrm{S}}). \tag{5}$$

GAME 3 (Exclude collisions of pre-shared keys). In game $\mathsf{Game}_3^{\mathcal{A}}$, we set a flag bad_{PC} and return 0 from FINALIZE whenever the NEWSECRET oracle samples a previously sampled pre-shared key (again). Formally, we set bad_{PC} if there exist two distinct tuples $(u, v, pskid)$ and $(u', v', pskid')$ with $\mathsf{pskeys}[(u, v, pskid)] = \mathsf{pskeys}[(u', v', pskid')]$. By the identical-until-bad-lemma,

$$\Pr[\mathsf{Game}_2^{\mathcal{A}} \Rightarrow 1] \leq \Pr[\mathsf{Game}_3^{\mathcal{A}} \Rightarrow 1] + \Pr[\mathsf{Game}_3^{\mathcal{A}} \text{ sets } \mathsf{bad}_{PC}].$$

Since the pre-shared keys are uniformly distributed[10] on $\{0,1\}^{hl}$, by the birthday bound

$$\Pr[\mathsf{Game}_3^{\mathcal{A}} \text{ sets } \mathsf{bad}_{PC}] \leq \frac{q_{\mathrm{NS}}^2}{2^{hl}}.$$

Conclusion of Phase 1. At this point, we argue that in Game 3 and any subsequent games, adversary $\mathcal{A}$ cannot violate the Sound predicate without also causing FINALIZE to return 0. If any Sound check fails, one of the checks we have added to the FINALIZE oracle will also fail. According to the definition of the MSKE game, there are six events that cause the predicate Sound to be violated (see Figure 4). In the following, we argue why each of these events cannot occur in Game 3 and thus $\mathsf{Sound} = \mathsf{true}$ needs to hold from Game 3 on.

1. *There are three honest sessions that have the same session identifier at any non-replayable stage.*

   Since the only replayable stages are stages 1 (ETS) and 2 (EEMS), consider any later stage $s \geq 3$. Recall that session identifiers sid for all stages $s \geq 3$ contain a `ClientHello` message containing the initiator session's nonce and group element and a `ServerHello` message containing the responder session's nonce and group element (see Section 7.1). Every session's sid therefore contains its own randomly sampled nonce-group element pair. For three sessions to accept the same $sid[s]$ for $s \geq 3$, there must be two honest sessions who have sampled the same nonce and group element. Due to Game 1, this would trigger the bad_C flag, leading FINALIZE to return 0.

2. *There are two sessions with the same session identifier in some non-replayable stage that have the same role.*

   Session identifiers $sid[s]$ for $s \geq 3$ as defined by TLS 1.3 (see Section 7.1) contain only one pair of nonce and group element per initiator and responder. If two honest sessions share a sid and a role, they must also share a nonce and group element. This case would also trigger the bad_C flag.

3. *There are two sessions with the same session identifier in some stage that do not share the same contributive identifier in that stage.*

   Once a session holds both a contributive identifier and a session identifier for the same stage, both are equal by our definition (see Section 7.1) of the session and contributive identifiers for TLS 1.3. This case will therefore never occur.

[10] As mentioned in Remark 7.3, this term has to be adapted for a different distribution on $\{0,1\}^{hl}$, i.e., for any distribution $\mathcal{D}$ on $\{0,1\}^{hl}$, the denominator would change to 2^{α}, where α is the min-entropy of $\mathcal{D}$.

4. *There are two sessions that hold the same session identifier for different stages.*

   This is impossible as the session identifier of stage s begins with the unique label $label_s$ for stage s.

5. *There are two honest sessions with the same session identifier in some stage that disagree on the identity of their peer or their pskid.*

   Two sessions which hold the same session identifier must necessarily agree on the value of the *binder*, which is part of the `ClientHello` message. In Game 2, we required that FINALIZE returns 0 if two queries to the oracle RO_{binder} collide. The two sessions must therefore also agree on the pre-shared key, which they obtained from the list pskeys. From Game 3, we have that FINALIZE returns 0 if any two distinct entries in pskeys contain the same value. Therefore two sessions can obtain the same pre-shared key from pskeys only if they hold the same tuple $(u, v, pskid)$, meaning they agree on both the peer identities and the pre-shared key identity.

6. *Sessions with the same session identifier in some stage do not hold the same key in that stage.*

   We have just established that two sessions with the same session identifier must agree on the peer identities and *pskid* (contained in `CPSK` and `SPSK`), meaning they also share the same PSK. Session identifiers for stages whose keys are derived from a Diffie–Hellman secret DHE must include both Diffie–Hellman shares X and Y (contained in `CKS` and `SKS`). These shares uniquely determine DHE. Besides that the session identifier also contains the context required to derive the respective stage keys, which then uniquely determines the stage key. Therefore, agreement on a session identifier implies agreement on a stage key.

Phase 2: Ensuring Predicate ExplicitAuth cannot be violated

GAME 4 (Exclude transcript hash collisions). In $\mathsf{Game}_4^{\mathcal{A}}$, we let the adversary lose if two distinct queries to $\mathsf{RO}_{\mathsf{Th}}$ lead to colliding outputs. This ensures that each transcript has a unique hash. When such a collision occurs, we set a new flag $\mathsf{bad}_{\mathsf{H}}$ and let the game return 0 from FINALIZE.

As in Game 2, we introduce a table $\mathsf{CollList}_{\mathsf{Th}}$ to random oracle $\mathsf{RO}_{\mathsf{Th}}$. Whenever it computes a hash $d = \mathsf{RO}_{\mathsf{Th}}(s)$ for some string s, we log $\mathsf{CollList}_{\mathsf{Th}}[d] \leftarrow s$. This table then is used to set $\mathsf{bad}_{\mathsf{Th}}$ as in Game 2.

Analogously to Game 2, we can construct a reduction $\mathcal{B}_2$ to the collision-resistance of $\mathsf{RO}_{\mathsf{Th}}$. As it simulates Game 4, the adversary $\mathcal{B}_2$ will need to make one query to its $\mathsf{RO}_{\mathsf{Th}}$ oracle for each $\mathsf{RO}_{\mathsf{Th}}$ query of $\mathcal{A}$ and up to 6 $\mathsf{RO}_{\mathsf{Th}}$ queries for the up to 6 *distinct* transcript hash values computed in a protocol step per SEND query of $\mathcal{A}$; in total $q_{\mathrm{RO}} + 6q_{\mathrm{S}}$ queries.

Therefore, we have that

$$\Pr[\mathsf{Game}_4^{\mathcal{A}} \text{sets } \mathsf{bad}_{\mathsf{H}}] \leq \mathrm{Adv}_{\mathsf{RO}_{\mathsf{Th}}}^{\mathsf{CR}}(q_{\mathrm{RO}} + 6q_{\mathrm{S}})$$

and it follows that

$$\Pr[\mathsf{Game}_3^{\mathcal{A}} \Rightarrow 1] \leq \Pr[\mathsf{Game}_4^{\mathcal{A}} \Rightarrow 1] + \mathrm{Adv}_{\mathsf{RO}_{\mathsf{Th}}}^{\mathsf{CR}}(q_{\mathrm{RO}} + 6q_{\mathrm{S}}).$$

GAME 5 (Abort if adversary guess a uncorrupted PSK). In $\mathsf{Game}_5^{\mathcal{A}}$, we make the adversary lose when it queries any random oracle on a pre-shared key PSK *before* that key has been corrupted via REVLONGTERMKEY.

We introduce some bookkeeping in order to implement this change. First, we add a reverse look-up table $\mathsf{PSKList}$ that is maintained by the NEWSECRET oracle. When NEWSECRET$(u, v, pskid)$ samples a fresh pre-shared key PSK, we log the tuple under index PSK as $\mathsf{PSKList}[\mathrm{PSK}] \leftarrow (u, v, pskid)$. Note that the pre-shared keys might repeat, so we may have multiple entries in $\mathsf{PSKList}$ indexed by a single PSK. Second, we add a time log T to the 12 random oracles RO_x. Each random oracle query containing a pre-shared key PSK now creates an entry $\mathsf{T}[\mathrm{PSK}] \leftarrow \mathsf{time}$, where time is the counter maintained by the key exchange experiment, unless $\mathsf{T}[\mathrm{PSK}]$ already exists.

The actual check whether the adversary queries any random oracle with a PSK before it was corrupted is performed by the FINALIZE oracle. We set a flag $\mathsf{bad}_{\mathrm{PSK}}$ if $\mathsf{T}(\mathrm{PSK}) \leq \mathsf{revpsk}_{(u,v,pskid)}$ for any PSK $\in \mathsf{T}$ and $(u, v, pskid) \in P[\mathrm{PSK}]$. If the $\mathsf{bad}_{\mathrm{PSK}}$ flag was set during this process, the FINALIZE oracle returns 0.

Next, let us analyze the probability that the game is lost due to flag $\mathsf{bad}_{\mathrm{PSK}}$ being set. Each random oracle query could hit one out of q_{NS} many pre-shared keys. Before a given pre-shared key is corrupted or queried to a random oracle, the adversary knows nothing about its value. Since we assume that pre-shared

keys are sampled uniformly at random from $\{0,1\}^{hl}$, the probability to hit a specific one is at most 2^{-hl}.[11] By the union bound, we obtain that the probability that the adversary hits any of the pre-shared keys in a single random oracle query is upper-bounded by $q_{\mathrm{NS}} \cdot 2^{-hl}$. Thus, the probability that $\mathsf{bad}_{\mathrm{PSK}}$ is set in response to any of the q_{RO} many random oracle queries overall is limited by $q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} \cdot 2^{-hl}$. This follows again by applying the union bound.

Hence, we get by the identical-until-bad lemma,

$$\begin{aligned}\Pr[\mathsf{Game}_4^{\mathcal{A}} \Rightarrow 1] &\leq \Pr[\mathsf{Game}_5^{\mathcal{A}} \Rightarrow 1] + \Pr[\mathsf{Game}_5^{\mathcal{A}} \text{ sets } \mathsf{bad}_{\mathrm{PSK}}] \\ &\leq \Pr[\mathsf{Game}_5^{\mathcal{A}} \Rightarrow 1] + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}}.\end{aligned}$$

In the next two games, we change the way that partnered sessions compute their session keys, *binder* values, and `Finished` MAC tags. Since we have established in Phase 1 that partnered sessions will always share the same key, we can compute these keys only once and let partnered sessions copy the results. This will make it easier to maintain consistency between partners as we change the way we compute keys and tags. This approach follows the tight key exchange security proof techniques of Cohn-Gordon et al. [13].

Game 6 (Log session keys and MAC tags). First, we will store all session keys in a look-up table SKEYS under their session identifiers. Sessions will be able to use this table to easily check if they share a session identifier with another honest session and thus share a key with a partner.

Honest sessions π_u^i in the initiator role will derive the keys ETS, EEMS, and RMS before their partners. In Game 6, when an initiator session accepts in stage 1 (ETS), 2 (EEMS), or 8 (RMS) it creates a new entry in SKEYS, i.e.,

$$\mathsf{SKEYS}[\pi_u^i.\mathit{sid}[s]] \leftarrow \pi_u^i.\mathit{skey}[s]$$

for $s \in \{1,2,8\}$. Honest responder sessions π_v^j will derive the keys htk_S, htk_C, CATS, SATS, and EMS before their partners. These sessions also log their keys in S under the appropriate session identifier:

$$\mathsf{SKEYS}[\pi_v^j.\mathit{sid}[s]] \leftarrow \pi_v^j.\mathit{skey}[s]$$

for $s \in \{3,\ldots,7\}$.

Note that no two sessions will ever log keys in table SKEYS under the same sid. From Sound, we know that only one initiator and one responder session may have the same session identifier $\mathit{sid}[s]$ in any stage s. Note that for the replayable stages 1 and 2 (ETS and EEMS) we only log once because the messages will only be logged by the initiator that output the replayed messages and not by the receivers that are receiving them.

We also store *binder*, fin_C and fin_S MAC tags. When any honest session queries RO_x with $x \in \{\mathit{binder}, \mathit{fin}_C, \mathit{fin}_S\}$, it logs the response in a second look-up table, TAGS, indexed by x and the inputs to RO_x. That is, for a query $(\mathrm{PSK}, \mathrm{DHE}, d_1, d_2)$ to $\mathsf{RO}_{\mathit{fin}_S}$, we log

$$\mathsf{TAGS}[\mathit{fin}_S, \mathrm{PSK}, \mathrm{DHE}, d_1, d_2] \leftarrow \mathsf{RO}_{\mathit{fin}_S}(\mathrm{PSK}, \mathrm{DHE}, d_1, d_2).$$

Since Game 6 only introduces book-keeping steps, we have that

$$\Pr[\mathsf{Game}_5^{\mathcal{A}} \Rightarrow 1] = \Pr[\mathsf{Game}_6^{\mathcal{A}} \Rightarrow 1].$$

Game 7 (Copy session keys and MAC tags from partnered session). In this game, we change the way the sessions compute their keys and MAC tags. Namely, if a session has an honest partner in stage s, instead of computing a key itself, it copies the stage-s key already computed by the partner via the table SKEYS introduced in Game 6. Concretely, the sessions compute their keys depending on their role as follows.

[11] Note that at this point, we use that the pre-shared key distribution is uniform. As already mentioned before, for any distribution $\mathcal{D}$ on $\{0,1\}^{hl}$, the probability would be $2^{-\alpha}$, where α is the min-entropy of $\mathcal{D}$.

Honest server sessions. An honest server session π_v^j, upon receiving $(\texttt{CH}, \texttt{CKS}, \texttt{CPSK})$, sets its session identifier for stages 1 (ETS) and 2 (EEMS). It then checks whether keys have been logged in SKEYS under $\pi_v^j.sid[1]$ and $\pi_v^j.sid[2]$. If such log entries exist, then π_v^j has an honest partner in stages 1 and 2, and copies the keys ETS and EEMS from SKEYS when they would instead be computed directly.

Analogously, upon receiving $\texttt{CF}$, π_v^j uses SKEYS to check whether there is an honest client session that shares the same stage-8 (RMS) session identifier $\pi_v^j.sid[8]$, and it copies the RMS key if this is the case. If there are no entries in SKEYS under the appropriate session identifiers, π_v^j proceeds as in Game 6 and computes its keys using the random oracles.

Honest client sessions. An honest client session π_u^i, upon receiving $(\texttt{SH}, \texttt{SKS}, \texttt{SPSK})$, sets its session identifiers for stages 3–7, which identify the keys htk_S, htk_C, CATS, SATS and EMS. It then searches for entries in SKEYS indexed by $\pi_u^i.sid[s]$ for $s \in \{3, \ldots, 7\}$. If these entries are present for stage s, then π_v^i copies the stage-s keys from SKEYS instead of computing them itself. Otherwise, π_u^i proceeds as in Game 6 and computes the keys using the random oracle in each case.

Computation of MAC tags. Finally, all honest sessions (both client and server) which would query RO_x to compute $x \in \{binder, fin_C, fin_S\}$ in Game 6 first check the look-up table TAGS to see if their query has already been logged. If so, they copy the response from TAGS instead of making the query to RO_x.

It remains to argue that the procedure of copying the keys in partnered sessions described in this game is consistent with computing the keys in Game 6. Recall that sessions which are partnered in stage s must agree on the stage-s key, since the Sound predicate (Property 6) cannot be violated. Consider a session π_u^i which accepts the stage-s key $\pi_u^i.skey[s]$. By Sound, any other session π_v^j in Game 6 which accepts in stage s with $\pi_v^j.sid[s] = \pi_u^i.sid[s]$ must set its stage-s key equal to $\pi_u^i.skey[s]$. Although in Game 7 the session π_v^j may copy $\pi_u^i.skey[s]$ from table SKEYS instead of deriving it directly, the value of $\pi_v^j.skey[s]$ does not change between the two games.

Sessions may also copy queries from look-up table TAGS instead of making the appropriate random oracle query themselves. However, table TAGS simply caches the response to random oracle queries and does not change them. Hence, the view of the adversary is identical. This implies that

$$\Pr[\mathsf{Game}_6^{\mathcal{A}} \Rightarrow 1] = \Pr[\mathsf{Game}_7^{\mathcal{A}} \Rightarrow 1].$$

With the next two games, we finalize Phase 2. First, we postpone the sampling of the pre-shared key to the RevLongTermKey oracle such that only corrupted sessions hold pre-shared keys. As a consequence of this change, we can no longer compute session keys and MAC tags using the random oracles. We will instead sample these uniformly at random from their respective range and only program the random oracles upon corruption of the corresponding pre-shared key. After this change, we can show that in order to break explicit authentication, the adversary must predict a uniformly random `Finished` MAC tag, which is unlikely.

Game 8 (Postpone PSK sampling until after corruption). In this game, we postpone the sampling of pre-shared keys from the NewSecret oracle to the RevLongTermKey oracle (if the pre-shared key gets corrupted) or the Finalize oracle (if the key remains uncorrupted).

Since we now do not have a PSK anymore for uncorrupted sessions, we cannot use the random oracle to compute keys or MAC tags in those sessions, but instead sample them uniformly at random. If the corresponding pre-shared key is corrupted later and a PSK is chosen (in RevLongTermKey), we will retroactively program the affected random oracles to ensure consistency.

Concretely, we change the implementation of the game as follows. When NewSecret receives a query $(u, v, pskid)$, we set $\mathsf{pskeys}[(u, v, pskid)]$ to a special symbol $\star$ instead of a randomly chosen pre-shared key. The $\star$ serves as a placeholder and signalizes that the NewSecret oracle already received a query $(u, v, pskid)$, but no PSK has been chosen yet. We add $(u, v, pskid)$ to the set $\mathsf{PSKList}[\star]$ to keep track of all tuples with an undefined PSK.

We let honest sessions whose pre-shared key has not been sampled (yet) but equals $\star$ sample their session keys as well as *binder* and `Finished` MAC tags uniformly at random. Due to the changes introduced in Game 7 we do not need to ensure consistency when sampling, as we sample each value once and partnered sessions copy the suitable value from the tables SKEYS and TAGS. (When sessions would log MAC tags in TAGS under their pre-shared keys in Game 7, those with no pre-shared key instead use the tuple $(u, v, pskid)$

in this game.) We further log the respective random oracle query that sessions would normally have used for the computation in a look-up table $\mathsf{PrgList}_x$ for later programming of the respective random oracle RO_x. Sessions which would log their RO-derived values in tables SKEYS and TAGS now log their randomly chosen values instead. That is, if a session in Game 7 would issue a query $(\star, \mathrm{DHE}, ctxt)$ (where DHE might be $\perp$) to random oracle RO_x to compute a value k, in Game 8 it chooses k uniformly at random from RO_x's range and logs

$$\mathsf{PrgList}_x[(u, v, pskid), \mathrm{DHE}, ctxt] \leftarrow k$$

in the look-up table $\mathsf{PrgList}_x$, where $(u, v, pskid)$ uniquely identifies the used PSK. Note that the table $\mathsf{PrgList}_x$ is closely related to the random oracle table ROList_x for RO_x. Table $\mathsf{PrgList}_x$ is always used when there is no PSK defined for a session, i.e., it has not (yet) been corrupted. Therefore, we need to make sure that if the PSK (identified by $(u, v, pskid)$) gets corrupted we are able to reprogram RO_x. Using $\mathsf{PrgList}_x$ we can upon corruption of the pre-shared key associated with $(u, v, pskid)$ efficiently look-up the entries we need to program from $\mathsf{PrgList}_x$ and transfer them to the random oracle table ROList_x after PSK has been set. We will discuss the precise process below when we describe how to adapt the RevLongTermKey oracle.

We must be particularly careful when $x = binder$, because we still wish to set the bad_{binder} flag when two randomly chosen binder values collide. Therefore, honest sessions still record the sampled binder values in list $\mathsf{CollList}_{binder}$, so that the bad_{binder} flag is set as before. This ensures that the probability of setting the flag does not change.

We also need to adapt the corruption oracle RevLongTermKey. Upon a query $(u, v, pskid)$ for which $\mathsf{pskeys}[(u, v, pskid)] = \star$, we perform the following additional steps: First, we sample a fresh pre-shared key $\mathrm{PSK} \xleftarrow{\$} \mathsf{KE.PSKS}$ and update pskeys, i.e., set $\mathsf{pskeys}[(u, v, pskid)] \leftarrow \mathrm{PSK}$. Next, we need to reprogram the random oracles using the lists R_x to ensure consistency. Thus, for all x we update the random oracle tables ROList_x for RO_x using $\mathsf{PrgList}_x$. For every entry $\mathsf{PrgList}_x[((u, v, pskid), \mathrm{DHE}, ctxt)] = k$, we set

$$\mathsf{ROList}_x[\mathrm{PSK}, \mathrm{DHE}, ctxt] \leftarrow k$$

where ROList_x is the random oracle table of RO_x. Lastly, we remove $(u, v, pskid)$ from the set $\mathsf{PSKList}[\star]$ and add it to $\mathsf{PSKList}[\mathrm{PSK}]$.

To be able to still set $\mathsf{bad}_{\mathrm{PSK}}$, we also make sure that in the Finalize procedure every pre-shared key is defined before the check against the random oracle time log T introduced in Game 5. We sample a pre-shared key for every tuple $(u, v, pskid) \in P[\star]$, setting $\mathsf{pskeys}[(u, v, pskid)] \xleftarrow{\$} \mathsf{KE.PSKS}$, and update the reverse look-up table PSKList accordingly. As a result, also uncorrupted sessions now have a pre-shared key defined and we can check the condition for $\mathsf{bad}_{\mathrm{PSK}}$ being set as introduced in Game 5.

The changes introduced in Game 8 are unobservable for the adversary as it never queries the random oracle for an uncorrupted pre-shared key, as otherwise the game would be aborted due to $\mathsf{bad}_{\mathrm{PSK}}$ introduced in Game 5. It hence does not matter whether the pre-shared key is already set before or upon corruption, because from the view of the adversary the keys (and the pre-shared key) are uniformly random bitstrings anyway up to this point. Upon corruption of a pre-shared key, we make sure by reprogramming the random oracle that all session keys and MAC tag computations are consistent with sessions that would have otherwise used this pre-shared key but derived all session keys and MAC tags without it. The change to the Finalize procedure does not affect the view of the adversary as it only retroactively defines keys on which the adversary cannot get any information about anymore. Consequently,

$$\Pr[\mathsf{Game}_7^{\mathcal{A}} \Rightarrow 1] = \Pr[\mathsf{Game}_8^{\mathcal{A}} \Rightarrow 1].$$

Game 9 (Exclude that honest sessions accept without a partner). In this game, we set a flag $\mathsf{bad}_{\mathrm{MAC}}$ and return 0 from Finalize if any session with an uncorrupted pre-shared key accepts stage 5 (htk_C) as initiator, or stage 8 (RMS) as responder, without having a partnered session. Formally, we set $\mathsf{bad}_{\mathrm{MAC}}$ if there is a session π_u^i such that $\pi_u^i.\mathsf{accepted}[s] < \mathsf{revpsk}_{(u,v,\pi_u^i.pskid)}$ with $v = \pi_u^i.peerid$ and

$$s = \begin{cases} 5 & \text{if } \pi_u^i.role = \mathsf{initiator} \\ 8 & \text{if } \pi_u^i.role = \mathsf{responder} \end{cases}$$

and there is no session π_v^j with $\pi_u^i.sid[s] = \pi_v^j.sid[s]$ when π_u^i accepts stage s.

Let us analyze the probability $\Pr[\mathsf{Game}_9^{\mathcal{A}} \text{ sets } \mathsf{bad}_{\mathrm{MAC}}]$. Consider a session π_u^i which triggers the $\mathsf{bad}_{\mathrm{MAC}}$ flag. In the following analysis, let π_u^i be an initiator. For responder sessions the arguments are analogous. The pre-shared key of session π_u^i is uncorrupted, which means that by the changes of Game 8 it has not been sampled. Therefore π_u^i either samples the `ServerFinished` MAC tag uniformly at random or copies it from table TAGS (in which case the MAC tag was uniformly sampled and logged by another honest session).

First observe that session π_u^i will not copy the `ServerFinished` MAC tag from table TAGS as this would imply that π_u^i is partnered when it accepts in stage 5. This in turn contradicts that π_u^i has triggered flag $\mathsf{bad}_{\mathrm{MAC}}$. Namely, if π_u^i would be able to copy the `ServerFinished` MAC tag from table TAGS there must have been another honest session that computed the same `ServerFinished` MAC, i.e., using the same tuple $(u, v, pskid)$, DHE secret, and transcript hash. Recall that the session identifier of stage 5 contains both the `ServerFinished` message and the transcript hashed to computed the `ServerFinished` MAC tag. Further, we have that transcript hashes are unique due to Game 4. This implies that the session that logged the `ServerFinished` MAC tag in TAGS needs to have the same stage-5 session identifier than π_u^i meaning π_u^i would be partnered in stage 5.

Thus, if π_u^i triggers $\mathsf{bad}_{\mathrm{MAC}}$, it must have sampled its `ServerFinished` MAC tag at random and the received `ServerFinished` message will match this tag with probability no more than 2^{-hl}.

Thus the probability that π_u^i triggers the flag $\mathsf{bad}_{\mathrm{MAC}}$ is bounded by 2^{-hl}. A union bound over all sessions gives

$$\Pr[\mathsf{Game}_9^{\mathcal{A}} \text{ sets } \mathsf{bad}_{\mathrm{MAC}}] \leq \frac{q_{\mathrm{S}}}{2^{hl}}.$$

Overall, we get by the identical-until-bad-lemma

$$\begin{aligned}\Pr[\mathsf{Game}_8^{\mathcal{A}} \Rightarrow 1] &\leq \Pr[\mathsf{Game}_9^{\mathcal{A}} \Rightarrow 1] + \Pr[\mathsf{Game}_9^{\mathcal{A}} \text{ sets } \mathsf{bad}_{\mathrm{MAC}}] \\ &\leq \Pr[\mathsf{Game}_9^{\mathcal{A}} \Rightarrow 1] + \frac{q_{\mathrm{S}}}{2^{hl}}.\end{aligned}$$

Conclusion of Phase 2. At this point, we argue that in Game 9 and any subsequent games, adversary $\mathcal{A}$ cannot violate the ExplicitAuth predicate without also causing FINALIZE to return 0. To this end, we argue that ExplicitAuth = true holds with certainty from Game 9 on.

The predicate ExplicitAuth is set to false if there is a session π_u^i accepting an explicitly authenticated stage s, whose pre-shared key was not corrupted before accepting the stage $s' \geq s$ in which it received (perhaps retroactively) explicit authentication, and (1) there is no honest session π_v^j partnered to π_u^i in stage s', or (2) there is an honest partner session π_v^j for π_u^i in stage s' but it accepts with a peer identity $w \neq u$, with a different pre-shared key identity than π_u^i, i.e. $\pi_v^j.pskid \neq \pi_u^i.pskid$, or with a different stage-s session identifier, i.e. $\pi_v^j.sid[s] \neq \pi_u^i.sid[s]$.

Recall that initiator (resp. responder) sessions receive explicit authentication with acceptance of stage 5 (resp. stage 8) meaning that all previous stages 1–4 (resp. stages 1–7) receive explicit authentication retroactively and all future stages 6–8 upon their acceptance. From Game 9, we have that any initiator session π_u^i accepting stage 5 (resp. any responder session accepting stage 8) with uncorrupted PSK must have a partnered session in that stage. Consequently, case (1) is impossible to achieve.

We next address the possibility of case (2). To achieve explicit authentication for stage $s \leq 8$, a responder session must have accepted stage 8. From Game 9 on, we know that π_u^i must have a partner with the same stage 8 session identifier. Observe that the transcripts contained in π_u^i's session identifiers for all stages are "sub-transcripts" of the transcript contained in the session identifier of stage 8. Therefore the partner must also have the same stage s session identifier. Property 5 of the Sound predicate then ensures that all partnered sessions agree on the peer identity and the pre-shared key identity, so ExplicitAuth is not violated by session π_u^i. The same property holds for initiator sessions accepting stages $s \leq 5$. So ExplicitAuth can only be violated if an initiator session's stage-5 partner accepts in stage $s > 5$ with a different peer identity, pre-shared key identifier, or session ID. Since peer and pre-shared key identifiers do not change after they are set, only the session identifiers may not match in stage s. The "sub-transcripts" of stage 6 (CATS) and 7 (SATS) session identifiers are identical to those of stage 5, so a partner in stage 5 will also be a partner in stages 6 and 7. Then the only way to violate predicate ExplicitAuth is to convince the stage-5 partner, a responder session, to accept a forged `ClientFinished` message and accept stage 8. This is impossible because the partner will verify the received `ClientFinished` message against the message sent by π_u^i, which

it copies from table TAGS. It follows that no session, responder or initiator, can violate the ExplicitAuth predicate.

Phase 3: Ensuring that the Challenge Bit is Idependently Random

GAME 10. In this game, we rule out that the adversary manages to guess the DHE secret of two honestly partnered session to learn about the keys they are computing. Here, we only look at those session that have a corrupted pre-shared key, because we already ruled out in Game 5 that the adversary learns something about the keys computed by these sessions. To that end, we add another flag $\mathsf{bad}_{\text{DHE}}$ to the game and return 0 from FINALIZE when it is set. Flag $\mathsf{bad}_{\text{DHE}}$ is set if the adversary ever queries a random oracle

$$\mathsf{RO}_x(\text{PSK}, \text{DHE}, \mathsf{RO}_{\mathsf{Th}}(sid[s]))$$

for $(x, s) \in \{(htk_C, 3), (htk_S, 4), (fin_S, 5)(\text{CATS}, 5), (\text{SATS}, 6), (\text{EMS}, 7), (fin_C, 8), (\text{RMS}, 8)\}$ such that

- PSK is corrupted, i.e., the adversary made a prior query REVLONGTERMKEY$(u, v, pskid)$ with $\mathsf{pskeys}[(u, v, pskid)] = \text{PSK}$,
- there are honest sessions π_u^i and π_v^j that are contributively partnered in stage s with $\pi_v^j.cid_{\pi_u^i.role}[s] = \pi_u^i.cid_{\pi_u^i.role}[s] = (\mathtt{CH}, \mathtt{CKS}, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}, \mathtt{SPSK}, \dots)$, and
- $\text{DHE} = g^{xy}$ such that $\mathtt{CKS} = g^x$ and $\mathtt{SKS} = g^y$.[12]

We bound the probability of flag $\mathsf{bad}_{\text{DHE}}$ being set via a reduction $\mathcal{B}_{\text{DHE}}$ to the strong Diffie–Hellman assumption in group $\mathbb{G}$. Reduction $\mathcal{B}_{\text{DHE}}$ simulates Game 10 for $\mathcal{A}$, and it wins the strong Diffie–Hellman whenever the simulated game would set the $\mathsf{bad}_{\text{DHE}}$ flag.

Definition 7.5. *Let $\mathbb{G}$ be a group of order p generated by g. We define*

$$\text{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathcal{B}_{\text{DHE}}}, 2q_{\text{RO}}) := \Pr\left[g^{ab} \xleftarrow{\$} \mathcal{B}_{\text{DHE}}^{\mathsf{stDH}_a(\cdot,\cdot)}(g^a, g^b) : a, b \xleftarrow{\$} \mathbb{Z}_p\right]$$

where stDH_a is a special "fixed-exponent DDH oracle" that on input (B, C) returns 1 if and only if $C = B^a$.

Construction of reduction $\mathcal{B}_{\text{DHE}}$. The reduction $\mathcal{B}_{\text{DHE}}$ gets as input a strong DH challenge $(A = g^a, B = g^b)$ as well as access to the oracle stDH_a for the Decisional Diffie–Hellman problem with the first argument fixed. Adversary $\mathcal{B}_{\text{DHE}}$ then honestly executes the INITIALIZE, REVSESSIONKEY, TEST, and NEWSECRET oracles as Game 10 would, managing all game variables itself. We explain in more detail how $\mathcal{B}_{\text{DHE}}$ answers SEND, REVLONGTERMKEY, and random oracle queries.

When $\mathcal{A}$ makes a query to the SEND oracle, $\mathcal{B}_{\text{DHE}}$ delivers the message to a protocol session in the same way as Game 10. However, the sessions themselves handle messages quite differently. At a high level, $\mathcal{B}_{\text{DHE}}$ embeds its strong DH challenges into the key shares of every initiator session and every partnered responder session. When $\mathsf{bad}_{\text{DHE}}$ is triggered, $\mathcal{B}_{\text{DHE}}$ learns the Diffie–Hellman secret DHE associated with two of these embedded key shares, and it can extract a solution to the strong DH challenge using some basic algebra. However, $\mathcal{B}_{\text{DHE}}$ must take care to appropriately program random oracles queries after corruptions, since it cannot compute Diffie–Hellman secrets for embedded key shares as it does not know the corresponding exponents. Next, we describe how client and server sessions are implemented in Game 10.

But first we explain the (constant-time accessible) look-up tables that are used (or defined) by reduction $\mathcal{B}_{\text{DHE}}$ to ensure an efficient implementation:

- The look-up table KSRnd is maintained for all sessions. It holds the random exponent τ used by the honest sessions to randomize their key share G, indexed by the session's nonce and key share (r, G) (see the implementation of the session for further details). To identify a session uniquely we use its nonce r and key share G as the index.

[12]Note that the game knows the exponents x and y used by the sessions, but the reduction constructed in the remainder will not.

- Each random oracle RO_x maintains a look-up table $\mathsf{DHEList}_x$. For each query $RO_x(\mathrm{PSK}, Z, d)$, the table stores the group element Z indexed by PSK and d.
- Each random oracle RO_x maintains a look-up table $\mathsf{RndList}_x$. It holds a tuple $(\tau, \tau', ctxt, key)$ indexed by the pair (PSK, d). The table holds all necessary information that is required to reprogram of the random oracle RO_x. The fields PSK and key can hold special values. If a PSK is uncorrupted, we cannot log the information under it because it is not defined. Therefore, we can use the tuple $(u, v, pskid)$ uniquely identifying PSK instead. Moreover, key can sometimes be an empty field, because reprogramming of that value will never occur. When this field is empty, it will not be accessed as we instead use the remaining information of $\mathsf{RndList}_x$ to solve the stDH challenge. See the remainder of the proof for details.

Implementation of honest server sessions. Consider any server session π_v^j.

1. Upon receiving $(\mathtt{CH}, \mathtt{CKS}, \mathtt{CPSK})$, the reduction $\mathcal{B}_{\mathrm{DHE}}$ first checks whether π_v^j has an honest partner in stages 1 (ETS) and 2 (EEMS) by checking for entries indexed by $\pi_v^j.sid[1]$ and $\pi_v^j.sid[2]$ in the look-up table SKEYS introduced in Game 6. If no such entries exist, then $\mathcal{B}_{\mathrm{DHE}}$ answers this and all future SEND queries just as specified in Game 10. For the rest of the discussion, we assume the entries do exist.

   Session π_v^j generates its key share $\mathtt{SKS}$ by randomizing the challenge key share B. Namely, it chooses a randomizer $\tau_v^j \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random and sets $Y \leftarrow B \cdot g^{\tau_v^j}$. Then, it logs τ_v^j under index (r_S, Y) in the look-up table KSRnd.

2. Before π_v^j outputs $(\mathtt{SH}, \mathtt{SKS}, \mathtt{SPSK})$, it computes the keys htk_C and htk_S. By Game 8, these keys are sampled randomly when PSK is uncorrupted and computed using RO_{htk_C}, resp. RO_{htk_S} otherwise. In both cases, $\mathcal{B}_{\mathrm{DHE}}$ needs to know the Diffie–Hellman secret DHE to log in table $\mathsf{PrgList}_x$ or to query RO_x for $x \in \{htk_C, htk_S\}$. However, $\mathcal{B}_{\mathrm{DHE}}$ cannot compute DHE because it does not know the discrete logarithms of either $\mathtt{CKS}$ or $\mathtt{SKS}$.

   Therefore, $\mathcal{B}_{\mathrm{DHE}}$ needs to compute the keys without knowing the DHE key using the control over the random oracles.

   If the pre-shared key has been corrupted, the adversary could potentially have already queried the random oracle RO_{htk_C} with the query π_v^j should make. To that end, $\mathcal{B}_{\mathrm{DHE}}$ first checks whether the corresponding query for htk_C was already made to RO_{htk_C}. Concretely, $\mathcal{B}_{\mathrm{DHE}}$ computes the context hash $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SPSK})$ and checks for a suitable RO_{htk_C} query using the look-up table $\mathsf{DHEList}_{htk_C}[\mathrm{PSK}, d]$ maintained in RO_{htk_C} (see above for the definition). Reduction $\mathcal{B}_{\mathrm{DHE}}$ queries $\mathsf{stDH}_a(Y, Z \cdot Y^{-\tau_u^i})$ for all $Z \in \mathsf{DHEList}_{htk_C}[\mathrm{PSK}, d]$, where τ_u^i is the randomizer used by the honest partner of π_v^j, which can be looked up from $\mathsf{KSRnd}[r_C, X]$ using π_u^i's nonce and key share. (Although this may cause several stDH_a queries in response to a single SEND query, $\mathcal{B}_{\mathrm{DHE}}$ is still efficient because it only checks random oracle queries whose context is d, and due to the lack of both nonce/group element and hash collisions d is unique to session π_u^i and its partner. Therefore each entry in $\mathsf{DHEList}_{htk_C}[\mathrm{PSK}, d]$ will be checked at most twice over the course of the entire reduction.)

   If any one of these queries is answered positively, we have by the definition of stDH_a that $Z \cdot Y^{-\tau_u^i} = Y^a$, which implies that $Z = Y^{a+\tau_u^i} = X^{b+\tau_v^j}$ by definition of Y and X, which was computed by the honest partner π_u^i that has output the $\mathtt{CH}$ message received by π_v^j. This exactly is the DHE value that π_v^j would have computed if we would have known the discrete logarithm of B. Hence, we have found the right Z value and only need to derandomize it to win the challenge. Therefore, we let $\mathcal{B}_{\mathrm{DHE}}$ submit the value

   $$Z \cdot Y^{-\tau_u^i} \cdot A^{-\tau_v^j} = Y^a \cdot A^{-\tau_v^j} = (g^a)^{b+\tau_v^j} \cdot (g^a)^{-\tau_v^j} = g^{ab}$$

   to the FINALIZE oracle as a solution to the strong Diffie–Hellman problem.

   Observe that if $\mathsf{bad}_{\mathrm{DHE}}$ is set due to a query to RO_{htk_C} in Game 10, there is a random oracle query such that one of the above stDH_a queries will be answered positively. Thus, $\mathcal{B}_{\mathrm{DHE}}$ will win if $\mathsf{bad}_{\mathrm{DHE}}$ is set. We do the same for htk_S with RO_{htk_S}.

If in the above process no query is answered positively, i.e., $\mathsf{bad}_{\mathrm{DHE}}$ will also not be set, then π_v^j samples the key $htk_C \xleftarrow{\$} \mathsf{KE.KS}[3]$ itself and logs the following information so that future RO queries can be answered appropriately:

$$\mathsf{RndList}_{htk_C}(\mathrm{PSK}, d = \mathsf{H}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK})) \leftarrow \left(\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), \bot\right).$$

Again, we do the same for htk_S.

If PSK is not corrupted, then $\mathsf{bad}_{\mathrm{DHE}}$ cannot possibly have been set and we do not need to worry about consistency with earlier random oracle queries. Therefore, we do not need to do the process described above and immediately sample htk_C and htk_S randomly as in Game 10. It logs the keys in table SKEYS under their respective session identifiers, which do not contain DHE or any unknown values. In Game 10, we added entries to $\mathsf{PrgList}_{htk_C}$ and $\mathsf{PrgList}_{htk_S}$ in order to program future random oracle queries upon corruption. The reduction cannot do this here as it does not know DHE; instead, it logs

$$\mathsf{RndList}_x[((u, v, pskid), d = \mathsf{H}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}))] \leftarrow \left(\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), \bot\right).$$

for $x \in \{htk_C, htk_S\}$. This will allow $\mathcal{B}_{\mathrm{DHE}}$ to win if a later RevLongTermKey or random oracle query triggers $\mathsf{bad}_{\mathrm{DHE}}$.

3. To compute the `ServerFinished` message $\mathcal{B}_{\mathrm{DHE}}$ proceeds exactly as in Step 2 except that it uses the random oracle RO_{fin_S} and context $\mathtt{CH} \parallel \cdots \parallel \mathtt{EE}$ through the `EncryptedExtensions`. Also, the `ServerFinished` message is computed first by the server, so $\mathcal{B}_{\mathrm{DHE}}$ does not check table SKEYS or TAGS for any entries. Reduction $\mathcal{B}_{\mathrm{DHE}}$ also cannot log the inputs to random oracle query RO_{fin_S} in table TAGS (as done since game Game 6) because it does not know DHE. Instead, it logs the derived value of fin_S in table TAGS and replaces DHE in the index of TAGS by $\left(\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{EE})\right)$. That is, if it computes fin_S for inputs PSK, d_1, and d_2, it logs

$$\mathsf{TAGS}[fin_S, \mathrm{PSK}, (\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{EE})), d_1, d_2] \leftarrow fin_S.$$

   That way, it is possible to identify DHE without knowing it. For fin_S, we keep the same notation for the sets $\mathsf{DHEList}_x$, $\mathsf{RndList}_x$ and ROList_x numbered as the corresponding random oracle RO_x.

4. Reduction $\mathcal{B}_{\mathrm{DHE}}$ proceeds exactly as for fin_S above, except that we again use different random oracles and the context $cid_{\mathrm{CATS}} = \mathtt{CH} \parallel \cdots \parallel \mathtt{SF} = cid_{\mathrm{SATS}} = cid_{\mathrm{EMS}}$, where cid_x denotes transcript contained in the contributive identifier which is prefixed by "x", and thus the hash $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SF})$. With respect to random oracles, we have $\mathsf{RO}_{\mathrm{CATS}}$ for CATS, $\mathsf{RO}_{\mathrm{SATS}}$ for SATS and $\mathsf{RO}_{\mathrm{EMS}}$ for EMS, respectively. Reduction $\mathcal{B}_{\mathrm{DHE}}$ logs the keys in table SKEYS under their respective session identifiers, which do not contain DHE or any unknown values.

   After this is done, π_v^j outputs $(\mathtt{EE}, \mathtt{SF})$.

5. Upon receiving `CF`, $\mathcal{B}_{\mathrm{DHE}}$ looks for a suitable entry for fin_C in TAGS. If there is a value fin_C consistent with π_v^j's view, $\mathcal{B}_{\mathrm{DHE}}$ terminates the session as specified if `CF` does not match the looked-up value of fin_C. Otherwise, $\mathcal{B}_{\mathrm{DHE}}$ continues to compute RMS. To this end, $\mathcal{B}_{\mathrm{DHE}}$ checks whether there is an entry in SKEYS that matches the stage-8 session identifier of π_v^j, if yes π_v^j simply copies that entry. If not, first observe that if there is no entry in SKEYS there is no honest stage-8 partner, which implies that PSK needs to be corrupted as otherwise the game would have been aborted due to $\mathsf{bad}_{\mathrm{MAC}}$ introduced in Game 9. Therefore, the adversary also would be allowed to query $\mathsf{RO}_{\mathrm{RMS}}$ to compute RMS. Thus, $\mathcal{B}_{\mathrm{DHE}}$ needs to check whether the value for RMS is already set. Here, we need to distinguish two cases. Namely, whether there is an honest contributive stage-3 partner or not.

   First note that as described in Step 1, $\mathcal{B}_{\mathrm{DHE}}$ does not embed its challenge in `SKS` if there is no honest session output the `ClientHello` received, i.e., there is no honest contributive stage-3 partner. Therefore, here $\mathcal{B}_{\mathrm{DHE}}$ can simply implement π_v^j as specified in Game 10.

   In case there is an honest contributive stage-3 partner, then $\mathcal{B}_{\mathrm{DHE}}$ proceeds as described in Step 2 for oracle $\mathsf{RO}_{\mathrm{RMS}}$ and context hash $d = \mathsf{RO}_{\mathsf{Th}}(cid_{\mathrm{RMS}}) = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{CF})$ to check whether the adversary already solved the stDH challenge for $\mathcal{B}_{\mathrm{DHE}}$. Note that the stage-3 session identifier uniquely defines the DHE key, thus if there is an honest partner and there is a respective $\mathsf{RO}_{\mathrm{RMS}}$ query, the adversary has to break stDH to submit the query.

Implementation of honest client sessions. Consider any client session π_u^i.

1. The reduction $\mathcal{B}_4$ proceeds exactly as in Game 10 until the session chooses its key share. Instead of choosing a fresh exponent as specified in Figure 1, it chooses a value $\tau_u^i \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random and sets $X \leftarrow A \cdot g^{\tau_u^i}$ as its key share in the `ClientKeyShare` message. Further, it logs τ_u^i in KSRnd indexed with (r_C, X). The rest is exactly as specified in Game 10. That is, it computes ETS and EEMS and outputs $(\mathtt{CH}, \mathtt{CKS}, \mathtt{CPSK})$.
2. Upon receiving $(\mathtt{SH}, \mathtt{SKS}, \mathtt{SPSK})$, π_u^i checks whether there is an entry

   $$\mathsf{SKEYS}[(\text{"}htk_C\text{"}, \mathtt{CH}, \ldots, \mathtt{SPSK})] \neq \bot.$$

   If this is the case, π_u^i knows that there is an honest stage-3 partner, and it copies all the keys stored under π_u^i's session identifier as defined in Game 10. If there is no suitable entry, $\mathcal{B}_{\mathrm{DHE}}$ faces the problem that it already "committed" to not knowing the discrete logarithm of π_u^i's key share X by embedding A into it and thus we are not able to compute the DHE value. Since there is no entry in SKEYS for htk_C, we know that there is no honest stage-3 partner session by definition of SKEYS. That is, no honest server session computed `SKS` and thus it must have been chosen by the adversary. If the pre-shared key is corrupted, $\mathcal{B}_{\mathrm{DHE}}$ needs to use the stDH_a oracle to check whether there already was a query issued to RO_x for $x \in \{htk_C, htk_S\}$. If this is not the case, π_u^i freshly samples random keys and remembers them for possible retroactive reprogramming of the random oracle. Concretely, we do the following for each random oracle RO_x for $x \in \{htk_C, htk_S\}$:

   First compute $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \ldots \parallel \mathtt{SPSK})$ and then query the stDH_a oracle for all $Z \in \mathsf{DHEList}_x[\mathrm{PSK}, d]$, where PSK is the pre-shared key used by π_u^i, as

   $$\mathsf{stDH}_a(Y, Z \cdot Y^{-\tau_u^i}) = 1 \iff Z = Y^a,$$

   where Y is the DH key share contained in `SPSK`. See the server session implementation above for further explanation. If there is any of these queries is answered positively, let the respective key be $\mathsf{RO}_x(\mathrm{PSK}, Z, d)$. If there is no Z that results in a positive query, let $key \xleftarrow{\$} \mathsf{KE.KS}[x]$ be sampled at random, and $\mathcal{B}_{\mathrm{DHE}}$ logs the value for possible later reprogramming of the random oracle RO_x, i.e.,

   $$\mathsf{RndList}_x[(\mathrm{PSK}, d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}))] \leftarrow \left(\tau_u^i, \bot, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), key\right).$$

   After that π_v^i either has copied the keys or chose them itself and will accept all of the stage keys among these keys.

   If the PSK of π_u^i has not been corrupted, then no "right" query can have been made and the keys be sampled randomly. However, we still need to program future "right" RO queries after a corruption. Therefore set

   $$\mathsf{RndList}_x[(\mathrm{PSK}, d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}))] \leftarrow \left(\tau_u^i, \bot, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), key\right).$$

   $\mathsf{PrgList}_x$ is not updated as in Game 10, because DHE is unknown.
3. Upon receiving $(\mathtt{EE}, \mathtt{SF})$, similar to the previous step, π_v^j checks whether there is an entry in SKEYS and TAGS (to verify `SF`) corresponding to its stage-5 session identifier. If this is the case, it copies the keys from that list. In case there is none, we have that there is no honest stage-5 partner. Here, we need to distinguish the case whether there was an honest stage-3 partner before or not.

   Namely, the adversary could corrupt the PSK, then change the `EE` output by an honest session and then compute a new `SF` message for the changed transcript. Hence, there is an honest stage-3 partner, but no stage-5 partner. In this case, $\mathcal{B}_{\mathrm{DHE}}$ again applies the approach from above (see implementation of server session, Step 2) for the random oracles RO_x for $x \in \{\mathrm{CATS}, \mathrm{SATS}, \mathrm{EMS}\}$ and the context $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SF})$ checking whether the random oracles received already a correct query which set the keys CATS, SATS and EMS. If this is the case and since there was a stage-3 partner, $\mathcal{B}_{\mathrm{DHE}}$ has embedded the DH challenge in both the client and the server, this solves the strong Diffie–Hellman problem. When there is no such query the keys are chosen at random and all necessary information for

possible retroactive programming of the random oracles RO_x is logged in the table $\mathsf{RndList}_x$. Please see above for details.

However, if there is no honest stage-3 partner, $\mathtt{SKS}$ was chosen by the adversary. Hence, $\mathcal{B}_{\text{DHE}}$ needs to apply the procedure described in the previous step (Step 2) and use the oracle stDH_a to check the random oracles RO_x for $x \in \{\text{CATS}, \text{SATS}, \text{EMS}\}$ whether they already set the keys. The important difference here is that a positive answer of the stDH_a oracle does not solve stDH, as $\mathtt{SKS}$ was chosen by the adversary. Note that $\mathcal{B}_{\text{DHE}}$ again needs to make sure that it gathers all the information needed to make retroactive programming of the random oracles possible by logging information in $\mathsf{RndList}_x$ as before.

4. π_u^i computes fin_C using the same process as above: if PSK is corrupted, it checks for RO queries in $\mathsf{DHEList}_{\mathit{fin}_C}[\text{PSK}, d]$ that could set $\mathsf{bad}_{\text{DHE}}$ when π_u^i has an honest partner in stage 8 or fix the value of fin_C when no honest partner exists. It then calls FINALIZE or sets fin_C accordingly. If no earlier RO query matches fin_C, then we sample fin_C randomly and log τ_u^i, fin_C, and the transcript in table $\mathsf{RndList}_{\mathit{fin}_C}$ under PSK and the transcript hash d. If PSK is uncorrupted, π_u^i immediately samples fin_C randomly and logs τ_u^i, fin_C, and the transcript in $\mathsf{RndList}_{\mathit{fin}_C}$ under index $((u, v, \mathit{pskid}), d)$.

   Next we compute RMS. As π_u^i is not able to compute DHE independent of there being a honest stage-3 partner or not, $\mathcal{B}_{\text{DHE}}$ need to apply the same procedure that was described before in Step 3, when there was no stage-5 partner for random oracle RO_{RMS} and context $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{CF})$. The only difference is that in case there was a stage-3 partner, FINALIZE is queried when the stDH oracle returns true, and if there is no stage-3 partner, RMS is only programmed. Then, π_u^i outputs $\mathtt{CF}$.

Besides changing the implementation of the session oracles, we also need to adapt the random oracles RO_x for $x \in \{\mathit{htk}_C, \ldots, \text{RMS}\}$ to make sure (1) $\mathcal{B}_{\text{DHE}}$§ programs the random oracle retroactively if the random oracle receives the right query and (2) to check whether the adversary computed DHE for $\mathcal{B}_{\text{DHE}}$ for honestly partnered sessions.

Implementation of random oracle RO_x. If RO_x receives a query that was already answered, it answers consistently. However, if there is a new query of the form (PSK, Z, d), it appends Z to the set $\mathsf{DHEList}_k[\text{PSK}, d]$. If $\mathsf{RndList}_k[\text{PSK}, d] \neq \perp$, then there already was a session using PSK and context hash d trying to compute a key without knowing the correct DHE secret. Therefore, $\mathcal{B}_{\text{DHE}}$ uses the stDH_a oracle to check whether Z is that secret. Let $(\tau_u^i, \tau_v^j, \mathit{ctxt}, \mathit{key})$ be the entry of $\mathsf{RndList}_k[\text{PSK}, d]$, where τ_u^i and τ_v^j denote the randomness used by the client and the server to randomize the stDH challenge, respectively, $\mathit{ctxt} = \mathtt{CH} \parallel \mathtt{CKS} \parallel \mathtt{CPSK} \parallel \mathtt{SH} \parallel \mathtt{SKS} \parallel \mathtt{SPSK} \parallel \cdots$ denotes the context such that $d = \mathsf{RO}_{\mathsf{Th}}(\mathit{ctxt})$ and key denotes the key chosen by the session since there was no random oracle fixing it. Using this information, it fetches $\mathtt{SKS} = Y$ and queries $\mathsf{stDH}_a(Y, Z \cdot Y^{-\tau_u^i})$. If this query is answered positively, $\mathcal{B}_{\text{DHE}}$ knows that the right DH value Z was queried. If $\tau_u^j = \perp$, i.e., the log in $\mathsf{RndList}_k$ was set by a client without an honestly partnered server, $\mathcal{B}_{\text{DHE}}$ needs to program the random oracle to be consistent. That is, $\mathsf{ROList}_k[\text{PSK}, Z, d] \leftarrow \mathit{key}$. Otherwise, $\mathcal{B}_{\text{DHE}}$ knows that the $\mathsf{PrgList}_x$ entry was set by an honestly partnered session, and thus Z is a randomized solution to the stDH challenge. Thus, $\mathcal{B}_{\text{DHE}}$ submits the solution $Z \cdot Y^{-\tau_u^i} \cdot A^{-\tau_v^j}$ to its stDH FINALIZE oracle.

Unless $\mathcal{B}_{\text{DHE}}$ solved the stDH challenge, the oracle outputs $\mathsf{ROList}_x[\text{PSK}, Z, d]$.

Implementation of corruption oracle REVLONGTERMKEY. Finally, $\mathcal{B}_{\text{DHE}}$ needs to handle corruptions via the REVLONGTERMKEY oracle. Since Game 8, the REVLONGTERMKEY oracle upon input (u, v, pskid) samples a fresh PSK. It then uses lists $\mathsf{PrgList}_x$ to program all the random oracles RO_x for consistency with any sessions whose pre-shared key is now PSK. Reduction $\mathcal{B}_{\text{DHE}}$ still does this, but in our reduction, the lists $\mathsf{PrgList}_x$ are no longer comprehensive. Some sessions fix the outputs of RO_x on some query without knowing the DHE input to that query. These sessions create log entries in $\mathsf{RndList}_x$, not $\mathsf{PrgList}_x$, and the entries have indices of the form $((u, v, \mathit{pskid}), d)$. $\mathcal{B}_{\text{DHE}}$ cannot use these entries to program past RO_x queries, but this is not necessary since any past RO_x query containing PSK would set the $\mathsf{bad}_{\text{PSK}}$ flag and cause the game to abort. $\mathcal{B}_{\text{DHE}}$ also cannot program future queries because we still do not know DHE. Instead, $\mathcal{B}_{\text{DHE}}$ just updates each matching entry in $\mathsf{PrgList}_x$ so that its index is (PSK, d) instead of

$((u, v, pskid), d)$. Future RO_x queries containing PSK will then handle strong DH checking and programming for $\mathcal{B}_{\mathrm{DHE}}$.

By the considerations above, we have that if $\mathsf{bad}_{\mathrm{DHE}}$ is set the $\mathcal{B}_{\mathrm{DHE}}$ wins the strong DH challenge. The identical-until-bad-lemma gives us that

$$\begin{aligned} \Pr[\mathsf{Game}_9^{\mathcal{A}} \Rightarrow 1] &\leq \Pr[\mathsf{Game}_{10}^{\mathcal{A}} \Rightarrow 1] + \Pr[\mathsf{bad}_{\mathrm{DHE}}] \\ &\leq \Pr[\mathsf{Game}_{10}^{\mathcal{A}} \Rightarrow 1] + \mathrm{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathcal{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}}), \end{aligned} \tag{6}$$

where the number of stDH_a oracle queries is no greater than $2q_{\mathrm{RO}}$, since $\mathcal{B}_{\mathrm{DHE}}$ will query the oracle at most twice (once for each partner) for every random oracle query issued by the adversary, and $t_{\mathcal{B}_{\mathrm{DHE}}}$ with $t_{\mathcal{B}_{\mathrm{DHE}}} \approx t + 4\log(p) \cdot q_{\mathrm{RO}}$ is the running time of $\mathcal{B}_{\mathrm{DHE}}$. Note that for every stDH_a query, $\mathcal{B}_{\mathrm{DHE}}$ needs to perform one group operation and one exponentiation in $\mathbb{G}$, the latter can be done in $2\log(p)$ many group operations using, e.g., the square-and-multiply algorithm. Thus, the time to answer a single stDH_a query take approximately time $2\log(p)$ and taking this together with the bound on the number of stDH_a yields the approximate runtime $t_{\mathcal{B}_{\mathrm{DHE}}}$.

Conclusion of Phase 3. We finally argue that the adversary's probability in determining the challenge bit b in Game 10 is at most $\frac{1}{2}$ if the Fresh predicate is true. First, recall that Fresh = true implies no session can be tested and revealed in the same stage, and a tested session's partner may also be neither tested nor revealed in that stage. In the following, we refer to a session being "fresh" in a stage if this session does not violate the conditions defined in the predicate Fresh in that stage. The Fresh predicate depends on the level of forward secrecy reached at the time of each Test query. First, if a session is tested in a non-forward secret stage, it remains only fresh if the PSK was never corrupted. Second, if a session is tested in a weak forward secret 2 stage s, it remains fresh if the PSK was never corrupted or if there is a contributive partner in stage s. Lastly, if a session is tested on a forward secret stage s, it remains fresh the PSK was corrupted after forward secrecy was established for that stage (perhaps retroactively) or if there is a contributive partner.

Next, we argue for each level of forward secrecy that all tested keys in Game 10 which do not violate Fresh are uniformly and independently distributed from the view of the adversary. For the non-forward secret stages 1 (ETS) and 2 (EEMS), the adversary cannot corrupt the PSK of all sessions that it queried Test on stage 1 or 2. Since Game 8, we sample all session keys derived from uncorrupted pre-shared keys uniformly at random, or copy uniformly random keys from SKEYS. That is, the key returned by the Test query is a uniformly random key independent of the challenge bit b. Therefore, it cannot learn anything about either ETS nor EEMS of any session with an uncorrupted key, and thus the response of a Test query will be a uniformly random string independent of the challenge bit b from the view of the adversary.

All other stages, i.e., stages 3–8, are weak forward secret 2 upon acceptance and become forward secret as soon as the session achieves explicit authentication. If the pre-shared key is never corrupted, we have by the same arguments given for the non-forward secret stages that the adversary receives a uniformly random key in response to the Test query independent of the challenge bit.

It remains to argue that the same is true if there is a contributive partner and the PSK is corrupted. In this case, the adversary would need to make a random oracle query that triggers $\mathsf{bad}_{\mathrm{DHE}}$ introduced in Game 10 and would cause Finalize to return 0. Without such a query the respective key is just a uniformly and independently distributed bitstring from the adversary's view. Hence, without losing the game, the adversary cannot learn anything about a weak forward secret 2 key, and thus it does not learn anything from the response of the Test query.

Since forward secret stages are weak forward secret 2 until explicit authentication is established, we only consider the case that a session that is tested on a weak forward secret 2 stage was corrupted after forward secrecy has been (retroactively) established. As we only establish forward secrecy after explicit authentication has been achieved, we can be sure due to ExplicitAuth never beeing violated that there is a partnered session for that stage. Hence, there also is a contributive partner and by the same arguments as given before the adversary would trigger $\mathsf{bad}_{\mathrm{DHE}}$ and lose the game before it can learn something about the session.

Overall, we have that the adversary in Game 10 cannot gain any information on the challenge bit b without violating any of the predicates Sound, ExplicitAuth, or Fresh. Thus, the probability that Finalize

and thus Game 10 returns 1 is no greater than 1/2. Formally,

$$\Pr[\mathsf{Game}_{10}^{\mathcal{A}} \Rightarrow 1] \leq \frac{1}{2}.$$

Collecting all the terms, we get the final bound

$$\begin{aligned}
&\mathrm{Adv}^{\mathsf{MSKE}}_{\mathtt{TLS1.3\text{-}PSK\text{-}(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{RO}_{binder}}(q_{\mathrm{RO}} + q_{\mathrm{S}}) + \frac{q_{\mathrm{NS}}^2}{2^{hl}} + \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{RO}_{\mathsf{Th}}}(q_{\mathrm{RO}} + 6q_{\mathrm{S}}) \\
&\qquad + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}} + \frac{q_{\mathrm{S}}}{2^{hl}} + \mathrm{Adv}^{\mathsf{stDH}}_{\mathbb{G}}(t_{\mathcal{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}}).
\end{aligned}$$

Applying the result of Appendix A, we can make the collision resistance terms explicit

$$\begin{aligned}
&\mathrm{Adv}^{\mathsf{MSKE}}_{\mathtt{TLS1.3\text{-}PSK\text{-}(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{NS}}^2}{2^{hl}} + \frac{(q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}} + \frac{q_{\mathrm{S}}}{2^{hl}} \\
&\qquad + \mathrm{Adv}^{\mathsf{stDH}}_{\mathbb{G}}(t_{\mathcal{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}}).
\end{aligned}$$

Further, applying the GGM bound for the strong Diffie–Hellman problem proven by Davis and Günther in [15], we get the final result

$$\begin{aligned}
&\mathrm{Adv}^{\mathsf{MSKE}}_{\mathtt{TLS1.3\text{-}PSK\text{-}(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{NS}}^2}{2^{hl}} + \frac{(q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}} + \frac{q_{\mathrm{S}}}{2^{hl}} \\
&\qquad + \frac{4(t + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p} \\
&\quad = \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}} \\
&\qquad + \frac{4(t + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p}.
\end{aligned}$$

□

7.3 Full Security Bound for TLS 1.3 PSK-(EC)DHE and PSK-only

We can finally combine the results of Sections 5, 6, and our key exchange bound above to produce fully concrete bounds for the TLS 1.3 PSK-(EC)DHE and PSK-only handshake protocols as specified on the left-hand side of Figure 1. This bound applies to the protocol *with handshake traffic encryption* and *internal keys* when *only modeling as random oracle* RO_{H} the hash function $\mathbf{H}$.

First, we define three variants of the TLS 1.3 PSK handshake:

- KE_0, as defined in Theorem 5.1 with handshake traffic encryption and one random oracle RO_{H}. (This is the variant we want to obtain our overall result for.)
- KE_1, as defined in Theorem 5.1 with handshake traffic encryption and 12 random oracles $\mathsf{RO}_{\mathsf{Th}}$, RO_{binder}, …, $\mathsf{RO}_{\mathrm{RMS}}$.
- KE_2: as defined in Theorem 6.1, with no handshake traffic encryption and 12 random oracles $\mathsf{RO}_{\mathsf{Th}}$, RO_{binder}, …, $\mathsf{RO}_{\mathrm{RMS}}$.

Theorem 5.1 grants that

$$\begin{aligned}
\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_0}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq\ &\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&+ \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}} + \frac{2q_{\mathrm{RO}}^2}{2^{hl}} + \frac{8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.
\end{aligned}$$

Next, we apply Theorem 6.1, yielding the bound

$$\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(t + t_{\mathrm{AEAD}} \cdot q_{\mathrm{S}}, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}} + q_{\mathrm{S}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}),$$

where t_{AEAD} is the maximum time required to execute AEAD encryption or decryption of TLS 1.3 messages.

Theorem 7.1 then finally and entirely bounds the advantage against the MSKE security of KE_2. Collecting these bounds gives

$$\begin{aligned}
\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_0}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) &\leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}} + \frac{2q_{\mathrm{RO}}^2}{2^{hl}} + \frac{8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}} \\
&\leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_2}(t + t_{\mathrm{AEAD}} \cdot q_{\mathrm{S}}, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}} + q_{\mathrm{S}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2 + 2q_{\mathrm{RO}}^2 + 8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}} \\
&\leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}} \\
&\quad + \frac{4(t + t_{\mathrm{AEAD}} \cdot q_{\mathrm{S}} + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p} \\
&\quad + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2 + 2q_{\mathrm{RO}}^2 + 8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.
\end{aligned}$$

This yields the following overall result for the MSKE security of the TLS 1.3 PSK-(EC)DHE handshake protocol.

Corollary 7.6. *Let* `TLS1.3-PSK-(EC)DHE` *be the TLS 1.3 PSK-(EC)DHE handshake protocol as specified on the left-hand side in Figure 1. Let* $\mathbb{G}$ *be the Diffie–Hellman group of order* p*. Let* nl *be the length in bits of the nonce, let* hl *be the output length in bits of* $\mathbf{H}$*, and let the pre-shared key space be* $\mathsf{KE.PSKS} = \{0,1\}^{hl}$*. Let* $\mathbf{H}$ *be modeled as a random oracle* $\mathsf{RO}_{\mathbf{H}}$*. Then,*

$$\begin{aligned}
&\mathrm{Adv}^{\mathsf{MSKE}}_{\texttt{TLS1.3-PSK-(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}} \\
&\qquad + \frac{4(t + t_{\mathrm{AEAD}} \cdot q_{\mathrm{S}} + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p} \\
&\qquad + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2 + 2q_{\mathrm{RO}}^2 + 8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.
\end{aligned}$$

Our tight security proof for the TLS 1.3 PSK-(EC)DHE handshake given in Section 7.2 can be adapted to the PSK-only handshake. The structure and resulting bounds are largely the same between the two modes, with a couple of significant changes. Naturally, we have no Diffie–Hellman group, no key shares in the `ClientHello` or `ServerHello` messages, and no reduction to the strong Diffie–Hellman problem. Without the reduction to stDH, we cannot achieve forward secrecy for any key: an adversary in possession of the pre-shared key can compute all session keys.

The security proof for the TLS 1.3 PSK-only handshake uses the same sequence of games Game_0 to Game_9 (excluding the reduction to the strong Diffie–Hellman problem in Game_{10}). There only is a difference in Game_1, in which we exclude collisions of nonces and group elements sampled by honest session to compute there `Hello` messages. Since we do not have any key shares in the PSK-only mode, the session will consequently also not sample a group elements. Thus, the bound for Game_0 changes to

$$\Pr[\mathsf{Game}_0 \Rightarrow 1] \leq \Pr[\mathsf{Game}_1 \Rightarrow 1] + \frac{2q_{\mathrm{S}}^2}{2^{nl}}.$$

The rest of the arguments follow similarly as given in Section 7.2. We obtain the following result.

Theorem 7.7. *Let* TLS1.3-PSK *be the TLS 1.3 PSK-only handshake protocol as specified on the right-hand side in Figure 1 without handshake encryption. Let functions* $\mathbf{H}$ *and* TKDF_x *for each* $x \in \{binder, \ldots, \mathrm{RMS}\}$ *be modeled as* 12 *independent random oracles* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{binder}, \ldots, \mathsf{RO}_{\mathrm{RMS}}$*. Let nl be the length in bits of the nonce, let hl be the output length in bits of* $\mathbf{H}$*, and let the pre-shared key space* KE.PSKS *be the set* $\{0,1\}^{hl}$*. Then,*

$$\begin{aligned}&\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{TLS1.3\text{-}PSK}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})\\ &\quad\leq \frac{2q_{\mathrm{S}}^2}{2^{nl}} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}}.\end{aligned}$$

From this we obtain the following overall result for the TLS 1.3 PSK-only mode via the same series of arguments as in Section 7.3.

Corollary 7.8. *Let* TLS1.3-PSK *be the TLS 1.3 PSK-only handshake protocol as specified on the left-hand side in Figure 1. Let nl be the length in bits of the nonce, let hl be the output length in bits of* $\mathbf{H}$*, and let the pre-shared key space be* $\mathsf{KE.PSKS} = \{0,1\}^{hl}$*. Let* $\mathbf{H}$ *be modeled as a random oracle* RO_{H}*. Then,*

$$\begin{aligned}&\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{TLS1.3\text{-}PSK}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})\\ &\quad\leq \frac{2q_{\mathrm{S}}^2}{2^{nl}} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}}\\ &\qquad + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2 + 2q_{\mathrm{RO}}^2 + 8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.\end{aligned}$$

8 Evaluation

Asymptotically, our tighter security bounds improve on prior analysis of TLS 1.3 by a quadratic factor. We evaluate ours and prior bounds over a wide range of fully concrete resource parameters, following the approach of Davis and Günther [15]. The full range of evaluated parameters is given in Tables 2 and 3 below, along with reasoning for how we chose the various ranges of resource parameters. The tables show that while the prior PSK-(EC)DHE bound by Dowling et al. [25] meets the target security goals in a number of configurations, there are at least some settings for all elliptic-curve groups in which the targeted security is not met. Our bounds do significantly better than the target in all configurations we considered. The gap for the PSK-only handshake is less significant as the loosest prior reduction for TLS 1.3 was to the Diffie–Hellman problem.

Overall, our bounds improve on previous analyses of the PSK-only handshake by 15 to 53 bits of security, and those of the PSK-(EC)DHE handshake by 60 to 131 bits of security, across all our parameters evaluated.

8.1 Evaluation Details

We will briefly explain the reasoning behind each of our specific resource parameter estimates. An adversary in the MSKE game (cf. Definition 3.1) is limited in its runtime t, the number of pre-shared keys $\#N$ and protocol sessions $\#S$ it can interact with, and the number of random oracle queries $\#RO$ it can make. This last quantity captures offline work the adversary spends on computing the hash function $\mathbf{H}$, which in our analysis we model as random oracle. The choice of ciphersuite enters the bound through the length of symmetric session keys and pre-shared keys. For the PSK-(EC)DHE handshake, the bound also depends on the underlying Diffie–Hellman group.

Runtime $t \in \{2^{40}, 2^{60}, 2^{80}\}$. We consider a range of adversarial runtimes from easily achievable (2^{40} operations) to state-scaled computational power (2^{80} operations).

Random oracle queries $\#RO \in \{2^{40}, 2^{60}, 2^{80}\}$. The number of random oracle queries models the number of hash function computations an adversary is capable of computing. Accordingly, we scale the number of RO queries with the runtime, always setting $\#RO = t/2^{10}$.

Number of pre-shared keys $\#N \in \{2^{25}, 2^{35}\}$. The world's largest certificate authority Let's Encrypt reports $\approx 2^{27.5}$ active certificates for fully-qualified domains.[13] While not every *user* of TLS 1.3 will perform resumption, our model counts the number of *pre-shared keys*, where typically users may hold many pre-shared keys, with servers regularly issuing several PSKs per full-handshake connection for later resumption. We hence estimate that the number of pre-shared keys accessible to a globally-scaled adversary may well exceed the reported number of (server) certificates.

Number of sessions $\#S \in \{2^{35}, 2^{45}, 2^{55}\}$. We use the same estimates as Davis and Günther [15], based on Google's and Firefox's usage reports.[14] With a daily browser user base of 2 billion ($\approx 2^{31}$) and an HTTPS traffic encryption rate in the range of 76–98%, we estimate an adversary could encounter up to 2^{55} distinct sessions over an extended time period. Note that although the PSK handshakes are less commonly used by browsers than the full TLS 1.3 handshake, they are frequently used by embedded and low-powered devices which do not appear in these reports. Naturally, we do not allow the number of sessions to exceed the adversary's runtime t.

Diffie–Hellman groups. There are ten Diffie–Hellman groups standardized for use with the PSK-(EC)DHE handshake: five elliptic-curve groups and five finite-field groups. We reduce to the security of the strong Diffie–Hellman assumption in each of these groups. Davis and Günther gave a proof of hardness in the generic group model (GGM) for the strong DH problem. This result is a good heuristic for elliptic-curve groups, but not for finite-field ones because they are vulnerable to index-calculus based attacks not covered by the GGM. The elliptic-curve groups are more efficient and more widely used than finite-field groups, so we restrict our analysis to these groups: `secp256r1`, `x25519`, `secp384r1`, `x448`, `secp521r1`. For each group, we give in Table 3 the order p and the expected security level b in bits. We use the security level b to determine the choice of hash function and the target security level for the entire PSK-(EC)DHE handshake.

Ciphersuite and symmetric lengths. Our bounds reduce to the collision resistance of the random oracle $\mathsf{RO}_{\mathtt{Th}}$, which models the handshake's hash function. The choice of hash function also determines the length of the session and resumption keys. TLS 1.3 has five ciphersuites, all of which set the hash function to be either `SHA256` or `SHA384`. For PSK-(EC)DHE mode, we select `SHA256` as the hash function whenever a curve with 128-bit security is used and we select `SHA384` for higher-security curves. As our results of Section 5 only apply to PSK-only mode when `SHA256` is the hash function, we always use `SHA256` and a target-security level of 128 bits.

Acknowledgments

We thank the anonymous reviewers of Eurocrypt 2022 for their helpful comments. We thank Robert Merget for pointing out an omission of `ClientHello` message fields when discussion domain separation in an earlier version of this work, cf. Appendix B.

References

[1] G. Arfaoui, X. Bultel, P.-A. Fouque, A. Nedelcu, and C. Onete. The privacy of the TLS 1.3 protocol. *PoPETs*, 2019(4):190–210, Oct. 2019. 26

[2] G. Avoine, S. Canard, and L. Ferreira. Symmetric-key authenticated key exchange (SAKE) with perfect forward secrecy. In S. Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 199–224. Springer, Heidelberg, Feb. 2020. 4

[3] C. Bader, D. Hofheinz, T. Jager, E. Kiltz, and Y. Li. Tightly-secure authenticated key exchange. In Y. Dodis and J. B. Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 629–658. Springer, Heidelberg, Mar. 2015. 5

[13] https://letsencrypt.org/stats/

[14] https://transparencyreport.google.com/, https://telemetry.mozilla.org/

Adversary resources					PSK-only	
t	$\#N$	$\#S$	$\#RO$	Target $t/2^b$	DFGS [25]	Us (Cor. 7.8)
2^{40}	2^{25}	2^{35}	2^{30}	2^{-88}	$\approx 2^{-158}$	$\approx 2^{-173}$
2^{40}	2^{35}	2^{35}	2^{30}	2^{-88}	$\approx 2^{-150}$	$\approx 2^{-173}$
2^{60}	2^{25}	2^{35}	2^{50}	2^{-68}	$\approx 2^{-119}$	$\approx 2^{-152}$
2^{60}	2^{25}	2^{45}	2^{50}	2^{-68}	$\approx 2^{-109}$	$\approx 2^{-151}$
2^{60}	2^{25}	2^{55}	2^{50}	2^{-68}	$\approx 2^{-99}$	$\approx 2^{-133}$
2^{60}	2^{35}	2^{35}	2^{50}	2^{-68}	$\approx 2^{-119}$	$\approx 2^{-152}$
2^{60}	2^{35}	2^{45}	2^{50}	2^{-68}	$\approx 2^{-109}$	$\approx 2^{-151}$
2^{60}	2^{35}	2^{55}	2^{50}	2^{-68}	$\approx 2^{-99}$	$\approx 2^{-133}$
2^{80}	2^{25}	2^{35}	2^{70}	2^{-48}	$\approx 2^{-79}$	$\approx 2^{-112}$
2^{80}	2^{25}	2^{45}	2^{70}	2^{-48}	$\approx 2^{-69}$	$\approx 2^{-112}$
2^{80}	2^{25}	2^{55}	2^{70}	2^{-48}	$\approx 2^{-59}$	$\approx 2^{-112}$
2^{80}	2^{35}	2^{35}	2^{70}	2^{-48}	$\approx 2^{-79}$	$\approx 2^{-112}$
2^{80}	2^{35}	2^{45}	2^{70}	2^{-48}	$\approx 2^{-69}$	$\approx 2^{-112}$
2^{80}	2^{35}	2^{55}	2^{70}	2^{-48}	$\approx 2^{-59}$	$\approx 2^{-112}$

Table 2: Concrete advantages of a key exchange adversary with given resources t (running time), $\#N$ (number of pre-shared keys), $\#S$ (number of sessions), and $\#RO$ (number of random oracle queries) in breaking the security of the TLS 1.3 PSK-only handshake protocol with a ciphersuite targeting 128-bit security. Numbers based on the prior bounds by Dowling et al. [25] and our bound for PSK-only in Corollary 7.8. "Target" indicates the maximal advantage $t/2^b$ tolerable for a given bound on t when aiming for the bit security level $b = 128$; entries in green-shaded cells meet that target. We assume that the ciphersuite uses SHA256 as its hash function (see Appendix B for further explanation).

[4] C. Bader, T. Jager, Y. Li, and S. Schäge. On the impossibility of tight cryptographic reductions. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 273–304. Springer, Heidelberg, May 2016. 5

[5] M. Bellare, H. Davis, and F. Günther. Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 3–32. Springer, Heidelberg, May 2020. 5, 6, 17, 18, 19, 20, 21, 62

[6] M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. https://eprint.iacr.org/2004/331. 33

[7] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin. Downgrade resilience in key-exchange protocols. In *2016 IEEE Symposium on Security and Privacy*, pages 506–525. IEEE Computer Society Press, May 2016. 6

[8] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella Béguelin. Proving the TLS handshake secure (as it is). In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 235–255. Springer, Heidelberg, Aug. 2014. 6

[9] C. Boyd, C. Cremers, M. Feltz, K. G. Paterson, B. Poettering, and D. Stebila. ASICS: Authenticated key exchange security incorporating certification systems. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 381–399. Springer, Heidelberg, Sept. 2013. 11

[10] C. Boyd, G. T. Davies, B. de Kock, K. Gellert, T. Jager, and L. Millerjord. Symmetric key exchange with full forward security and robust synchronization. In *ASIACRYPT 2021*, 2021. To appear. Available as Cryptology ePrint Archive, Report 2021/702. https://ia.cr/2021/702. 4

[11] C. Brzuska, A. Delignat-Lavaud, C. Egger, C. Fournet, K. Kohbrok, and M. Kohlweiss. Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Report 2021/467, 2021. https://eprint.iacr.org/2021/467. 6

[12] C. Brzuska, M. Fischlin, B. Warinschi, and S. C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM CCS 2011*, pages 51–62. ACM Press, Oct. 2011. 29

Adversary resources						PSK-(EC)DHE	
t	$\#N$	$\#S$	$\#RO$	Curve (bit security b, group order p)	Target $t/2^b$	DFGS [25]	Us (Cor. 7.6)
2^{40}	2^{25}	2^{35}	2^{30}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-88}	$\approx 2^{-92}$	$\approx 2^{-167}$
2^{40}	2^{35}	2^{35}	2^{30}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-88}	$\approx 2^{-82}$	$\approx 2^{-167}$
2^{40}	2^{25}	2^{35}	2^{30}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-88}	$\approx 2^{-92}$	$\approx 2^{-163}$
2^{40}	2^{35}	2^{35}	2^{30}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-88}	$\approx 2^{-82}$	$\approx 2^{-163}$
2^{40}	2^{25}	2^{35}	2^{30}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-152}	$\approx 2^{-220}$	$\approx 2^{-294}$
2^{40}	2^{35}	2^{35}	2^{30}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-152}	$\approx 2^{-210}$	$\approx 2^{-294}$
2^{40}	2^{25}	2^{35}	2^{30}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-184}	$\approx 2^{-220}$	$\approx 2^{-301}$
2^{40}	2^{35}	2^{35}	2^{30}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-184}	$\approx 2^{-210}$	$\approx 2^{-301}$
2^{40}	2^{25}	2^{35}	2^{30}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-216}	$\approx 2^{-220}$	$\approx 2^{-301}$
2^{40}	2^{35}	2^{35}	2^{30}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-216}	$\approx 2^{-210}$	$\approx 2^{-301}$
2^{60}	2^{25}	2^{35}	2^{50}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-61}$	$\approx 2^{-132}$
2^{60}	2^{25}	2^{45}	2^{50}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-40}$	$\approx 2^{-132}$
2^{60}	2^{25}	2^{55}	2^{50}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-12}$	$\approx 2^{-127}$
2^{60}	2^{35}	2^{35}	2^{50}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-60}$	$\approx 2^{-132}$
2^{60}	2^{35}	2^{45}	2^{50}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-32}$	$\approx 2^{-132}$
2^{60}	2^{35}	2^{55}	2^{50}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-2}$	$\approx 2^{-127}$
2^{60}	2^{25}	2^{35}	2^{50}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-57}$	$\approx 2^{-128}$
2^{60}	2^{25}	2^{45}	2^{50}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-37}$	$\approx 2^{-128}$
2^{60}	2^{25}	2^{55}	2^{50}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-12}$	$\approx 2^{-123}$
2^{60}	2^{35}	2^{35}	2^{50}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-57}$	$\approx 2^{-128}$
2^{60}	2^{35}	2^{45}	2^{50}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-32}$	$\approx 2^{-128}$
2^{60}	2^{35}	2^{55}	2^{50}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-2}$	$\approx 2^{-123}$
2^{60}	2^{25}	2^{35}	2^{50}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-189}$	$\approx 2^{-259}$
2^{60}	2^{25}	2^{45}	2^{50}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-168}$	$\approx 2^{-259}$
2^{60}	2^{25}	2^{55}	2^{50}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-140}$	$\approx 2^{-254}$
2^{60}	2^{35}	2^{35}	2^{50}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-188}$	$\approx 2^{-259}$
2^{60}	2^{35}	2^{45}	2^{50}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-160}$	$\approx 2^{-259}$
2^{60}	2^{35}	2^{55}	2^{50}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-130}$	$\approx 2^{-254}$
2^{60}	2^{25}	2^{35}	2^{50}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-200}$	$\approx 2^{-280}$
2^{60}	2^{25}	2^{45}	2^{50}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-170}$	$\approx 2^{-279}$
2^{60}	2^{25}	2^{55}	2^{50}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-140}$	$\approx 2^{-261}$
2^{60}	2^{35}	2^{35}	2^{50}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-190}$	$\approx 2^{-280}$
2^{60}	2^{35}	2^{45}	2^{50}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-160}$	$\approx 2^{-279}$
2^{60}	2^{35}	2^{55}	2^{50}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-130}$	$\approx 2^{-261}$
2^{60}	2^{25}	2^{35}	2^{50}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-200}$	$\approx 2^{-280}$
2^{60}	2^{25}	2^{45}	2^{50}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-170}$	$\approx 2^{-279}$
2^{60}	2^{25}	2^{55}	2^{50}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-140}$	$\approx 2^{-261}$
2^{60}	2^{35}	2^{35}	2^{50}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-190}$	$\approx 2^{-280}$
2^{60}	2^{35}	2^{45}	2^{50}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-160}$	$\approx 2^{-279}$
2^{60}	2^{35}	2^{55}	2^{50}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-130}$	$\approx 2^{-261}$
2^{80}	2^{25}	2^{35}	2^{70}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-21}$	$\approx 2^{-92}$
2^{80}	2^{25}	2^{45}	2^{70}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-1}$	$\approx 2^{-92}$
2^{80}	2^{25}	2^{55}	2^{70}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{19}$	$\approx 2^{-92}$
2^{80}	2^{35}	2^{35}	2^{70}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-21}$	$\approx 2^{-92}$
2^{80}	2^{35}	2^{45}	2^{70}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-1}$	$\approx 2^{-92}$
2^{80}	2^{35}	2^{55}	2^{70}	secp256r1 ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{20}$	$\approx 2^{-92}$
2^{80}	2^{25}	2^{35}	2^{70}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{-17}$	$\approx 2^{-88}$
2^{80}	2^{25}	2^{45}	2^{70}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{3}$	$\approx 2^{-88}$
2^{80}	2^{25}	2^{55}	2^{70}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{23}$	$\approx 2^{-88}$
2^{80}	2^{35}	2^{35}	2^{70}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{-17}$	$\approx 2^{-88}$
2^{80}	2^{35}	2^{45}	2^{70}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{3}$	$\approx 2^{-88}$
2^{80}	2^{35}	2^{55}	2^{70}	x25519 ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{23}$	$\approx 2^{-88}$
2^{80}	2^{25}	2^{35}	2^{70}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-149}$	$\approx 2^{-219}$
2^{80}	2^{25}	2^{45}	2^{70}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-129}$	$\approx 2^{-219}$
2^{80}	2^{25}	2^{55}	2^{70}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-109}$	$\approx 2^{-219}$
2^{80}	2^{35}	2^{35}	2^{70}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-149}$	$\approx 2^{-219}$
2^{80}	2^{35}	2^{45}	2^{70}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-129}$	$\approx 2^{-219}$
2^{80}	2^{35}	2^{55}	2^{70}	secp384r1 ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-108}$	$\approx 2^{-219}$
2^{80}	2^{25}	2^{35}	2^{70}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-180}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{45}	2^{70}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-150}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{55}	2^{70}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-120}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{35}	2^{70}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-170}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{45}	2^{70}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-140}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{55}	2^{70}	x448 ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-110}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{35}	2^{70}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-180}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{45}	2^{70}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-150}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{55}	2^{70}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-120}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{35}	2^{70}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-170}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{45}	2^{70}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-140}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{55}	2^{70}	secp521r1 ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-110}$	$\approx 2^{-240}$

Table 3: Concrete advantages of a key exchange adversary with given resources t (running time), $\#N$ (number of pre-shared keys), $\#S$ (number of sessions), and $\#RO$ (number of random oracle queries) in breaking the security of the TLS 1.3 PSK-(EC)DH handshake protocol. Numbers based on the prior bounds by Dowling et al. [25] and our bound for PSK-(EC)DHE in Corollary 7.6. "Target" indicates the maximal advantage $t/2^b$ tolerable for a given bound on t when aiming for the respective curve's bit security level b; entries in green-shaded cells meet that target. See Section 8 and Appendix 8.1 for further details.

[13] K. Cohn-Gordon, C. Cremers, K. Gjøsteen, H. Jacobsen, and T. Jager. Highly efficient key exchange protocols with optimal tightness. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 767–797. Springer, Heidelberg, Aug. 2019. 5, 7, 36

[14] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy*, pages 470–485. IEEE Computer Society Press, May 2016. 7

[15] H. Davis and F. Günther. Tighter proofs for the SIGMA and TLS 1.3 key exchange protocols. In *19th International Conference on Applied Cryptography and Network Security (ACNS 2021)*, 2021. 5, 6, 10, 17, 46, 48, 49

[16] C. de Saint Guilhem, M. Fischlin, and B. Warinschi. Authentication in key-exchange: Definitions, relations and composition. In L. Jia and R. Küsters, editors, *CSF 2020 Computer Security Foundations Symposium*, pages 288–303. IEEE Computer Society Press, 2020. 14

[17] D. Diemert. *On the Tight Security of the Transport Layer Security (TLS) Protocol Version 1.3*. PhD thesis, Bergische Universität Wuppertal, Wuppertal, Germany, 2023. https://doi.org/10.25926/BUW/0-98. 57, 62

[18] D. Diemert, K. Gellert, T. Jager, and L. Lyu. More efficient digital signatures with tight multi-user security. In J. Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 1–31. Springer, Heidelberg, May 2021. 5

[19] D. Diemert and T. Jager. On the tight security of TLS 1.3: Theoretically sound cryptographic parameters for real-world deployments. *Journal of Cryptology*, 34(3):30, July 2021. 5, 6, 7, 17

[20] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. Cryptology ePrint Archive, Report 2013/382, 2013. https://eprint.iacr.org/2013/382. 21

[21] Y. Dodis, T. Ristenpart, J. P. Steinberger, and S. Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 348–366. Springer, Heidelberg, Aug. 2012. 19, 21

[22] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 1197–1210. ACM Press, Oct. 2015. 5, 6, 7, 29

[23] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. Cryptology ePrint Archive, Report 2015/914, 2015. https://eprint.iacr.org/2015/914. 29

[24] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081, 2016. https://eprint.iacr.org/2016/081. 5, 6

[25] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 34(4):37, Oct. 2021. 5, 6, 7, 10, 14, 48, 50, 51

[26] B. Dowling and D. Stebila. Modelling ciphersuite and version negotiation in the TLS protocol. In E. Foo and D. Stebila, editors, *ACISP 15*, volume 9144 of *LNCS*, pages 270–288. Springer, Heidelberg, June / July 2015. 6, 7

[27] N. Drucker and S. Gueron. Selfie: reflections on TLS 1.3 with PSK. *Journal of Cryptology*, 34(3):27, July 2021. 11

[28] M. Fischlin and F. Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 2014*, pages 1193–1204. ACM Press, Nov. 2014. 6, 7

[29] M. Fischlin and F. 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, EuroS&P 2017*, pages 60–75. IEEE, Apr. 2017. 5, 6

[30] M. Fischlin, F. Günther, B. Schmidt, and B. 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. 14

[31] F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 2013*, pages 387–398. ACM Press, Nov. 2013. 6

[32] D. K. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919, Aug. 2016. 58

[33] K. Gjøsteen and T. Jager. Practical and tightly-secure digital signatures and authenticated key exchange. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 95–125. Springer, Heidelberg, Aug. 2018. 5, 7

[34] C. G. Günther. An identity-based key-exchange protocol. In J.-J. Quisquater and J. Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 29–37. Springer, Heidelberg, Apr. 1990. 4

[35] F. Günther. *Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, 2018. http://tuprints.ulb.tu-darmstadt.de/7162/. 7, 29

[36] S. Han, T. Jager, E. Kiltz, S. Liu, J. Pan, D. Riepel, and S. Schäge. Authenticated key exchange and signatures with tight security in the standard model. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 670–700, Virtual Event, Aug. 2021. Springer, Heidelberg. 5, 7

[37] T. Jager, E. Kiltz, D. Riepel, and S. Schäge. Tightly-secure authenticated key exchange, revisited. In A. Canteaut and F.-X. Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 117–146. Springer, Heidelberg, Oct. 2021. 7

[38] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, Heidelberg, Aug. 2012. 6

[39] T. Jager, J. Schwenk, and J. Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 1185–1196. ACM Press, Oct. 2015. 7

[40] H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. Cryptology ePrint Archive, Report 2005/176, 2005. https://eprint.iacr.org/2005/176. 14

[41] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, Aug. 2010. 9

[42] H. Krawczyk. A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1438–1450. ACM Press, Oct. 2016. 7

[43] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997. Updated by RFC 6151. 9

[44] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010. 9

[45] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 429–448. Springer, Heidelberg, Aug. 2013. 6

[46] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. Cryptology ePrint Archive, Report 2013/339, 2013. https://eprint.iacr.org/2013/339. 6

[47] H. Krawczyk and H. Wee. The OPTLS protocol and TLS 1.3. In *2016 IEEE European Symposium on Security and Privacy*, pages 81–96. IEEE, Mar. 2016. 5

[48] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748 (Informational), Jan. 2016. 32, 58

[49] Y. Li, S. Schäge, Z. Yang, F. Kohlar, and J. Schwenk. On the security of the pre-shared key ciphersuites of TLS. In H. Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 669–684. Springer, Heidelberg, Mar. 2014. 6

[50] X. Liu, S. Liu, D. Gu, and J. Weng. Two-pass authenticated key exchange with explicit authentication and tight security. In S. Moriai and H. Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 785–814. Springer, Heidelberg, Dec. 2020. 7

[51] U. M. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In M. Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, Heidelberg, Feb. 2004. 5, 16, 17, 18, 20

[52] National Institute of Standards and Technology. FIPS PUB 180-4: Secure Hash Standard (SHS), 2012. 7

[53] National Institute of Standards and Technology. FIPS PUB 186-4: Digital Signature Standard (DSS), 2013. 58

[54] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), Aug. 2018. 4, 5, 32, 56, 57, 59, 60, 61, 62

[55] T. Ristenpart, H. Shacham, and T. Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 487–506. Springer, Heidelberg, May 2011. 16, 17

[56] P. Schwabe, D. Stebila, and T. Wiggers. Post-quantum TLS without handshake signatures. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *ACM CCS 2020*, pages 1461–1480. ACM Press, Nov. 2020. 10, 14

Appendix

A Collision Resistance of Random Oracles

Theorem A.1. *Let* RO *be a random oracle with output length* hl *and let* $\mathcal{A}$ *be any adversary issuing at most* q_{RO} *many queries to* RO*. Then,*

$$\mathrm{Adv}^{\mathsf{CR}}_{\mathsf{RO}}(q_{\mathrm{RO}}) \leq \frac{q_{\mathrm{RO}}^2}{2^{hl}}.$$

Proof. The above theorem follows from the birthday bound. Since there are at most q_{RO} queries issued to RO, in the worst case each of these queries is distinct. That is, the random oracle RO has to sample q_{RO} many uniform and independent bit strings from $\{0,1\}^{hl}$. The probability that two of these bit strings collide can be limited using the birthday bound by $q_{\mathrm{RO}}^2 \cdot 2^{-hl}$. □

B A Careful Discussion of Domain Separation

In our indifferentiability treatment of the TLS 1.3 key schedule (cf. Section 5), we change what we capture as random oracles in the key exchange model. We start with one random oracle, RO_H, used wherever the hash function $\mathbf{H}$ would be called in the protocol. We change this to classify queries to RO_H into two types:

Type 1 queries: *component hashes* (via function Ch) used within **Extract**, **Expand**, and **MAC** to compute HKDF.Extract, HKDF.Expand, resp. HMAC.

Type 2 queries: *transcript hashes* (via function Th) computing hash values of protocol transcripts (or empty strings).

We wish to model Ch and Th now as *two* independent random oracles: RO_Ch resp. RO_Th.

To change the model, we can just change the pseudocode of the protocol to replace RO_H with whichever of RO_Ch and RO_Th seems more appropriate. However, we must define an explicit construction that performs this substitution in a systematic way in order to give a formal proof of security. This construction needs a Boolean condition to determine which of RO_Ch and RO_Th should be queried, and this condition cannot be dependent on the higher-level context of the protocol's usage. Instead, we must define two disjoint sets D_Ch and D_Th such that honest executions of TLS 1.3 only query RO_H on inputs in D_Ch when computing HKDF.Extract, HKDF.Expand, or HMAC, and it otherwise only queries RO_H on inputs in D_Th.

This separation must hold even when an honest session is responding to adversarially-chosen messages. We do make some assumptions about the way that honest sessions process incoming messages. We assume that a server receiving a first `ClientHello` message from a client will not respond or execute the protocol unless the message contains correct encodings of all of the mandatory parameters for TLS 1.3. If the client fails to specify a valid group and key share in PSK-(EC)DHE mode, or version number, mode, and pre-shared key in any mode, the server should abort. Of course, the `ClientHello` message may also contain invalid encodings of these values or even arbitrary data; we do not exclude this possibility. Note that our conditions apply only to random-oracle queries made by honest executions of the protocol. An adversary may of course call RO_H on any input it chooses in either D_Ch or D_Th.

The TLS 1.3 handshake protocol does not provide any intentional domain separation between Type 1 and Type 2 queries. We therefore turn to the formatting of queries to RO_H in the hopes of finding some unintentional separation. We identify seven subtypes of query: five subtypes of Type 1 and two subtypes of Type 2. Queries of each subtype have some unique formatting: a fixed length, a byte with a particular value, an encoded label. These attributes are heavily dependent on the specific configuration of the TLS 1.3 protocol; we therefore analyze four separate cases: two modes of operation (PSK-(EC)DHE and PSK-only mode) and two ciphersuites defining RO_H as `SHA256` and `SHA384` respectively. Throughout, we will assume that any pre-shared-keys are the same length as the output length of RO_H, i.e., hl bits. This is true of resumption keys, but may not be true in general for pre-shared keys negotiated out-of-band. As TLS 1.3 fields length are given in (full) *bytes*, we will be talking about *byte lengths* if not otherwise stated in the following and use the shorthand $Hl := hl/8$ for the output length of RO_H in *bytes*. We also assume that if a Diffie–Hellman group is used, it is one of the standardized elliptic curve or finite field groups.

All Type 1 queries to $\mathsf{RO_H}$ are intermediate steps in the computation of HMAC, HKDF.Extract, and HKDF.Expand. They consequently share some formatting which we discuss here before addressing each subtype individually. HKDF.Extract and HMAC are two names for the same function. Given a key K and input s, $\mathsf{HKDF.Expand}(K, s)$ pads s with a single trailing counter byte with value `0x01`, then returns $\mathsf{HMAC}(K, s \parallel \mathtt{0x01})$. Therefore all Type 1 queries to $\mathsf{RO_H}$ arise in the computation of HMAC. $\mathsf{HMAC}[\mathsf{RO_H}](K, s)$ takes a key K of length Hl bytes. It then pads this key with zeroes up to the block length Bl of its hash function. The block lengths of `SHA256` and `SHA384` are 64 and 128 bytes respectively. We call the padded key K'. Then $\mathsf{HMAC}[\mathsf{RO_H}]$ makes two queries to $\mathsf{RO_H}$:

1. $d \leftarrow \mathsf{RO_H}(K' \oplus \mathsf{ipad} \parallel s)$,
2. $\mathsf{RO_H}(K' \oplus \mathsf{opad} \parallel d)$.

The values ipad and opad are strings of Bl bytes. Each byte in ipad is fixed to `0x36`, and each byte in opad is fixed to `0x5c`. The padded key K' is Bl long, longer than K, so every Type 1 query has a segment of length $Bl - Hl$ bytes in which each byte equals one of `0x36` and `0x5c`. We refer to this segment as the "fixed region". When the hash function is `SHA256`, resp. `SHA384`, the fixed region is 32, resp 80 bytes long.

Now we can present the seven subtypes of queries made by TLS 1.3. The first five types are Type 1 queries, and the last two (Empty and Transcript) are Type 2 queries.

The seven subtypes of queries are:

1. **Outer HMAC queries.** These queries are the second query made in the computation of HMAC. Its key has length Hl, and the digest d also has length Hl. In between these is the fixed region, in which every byte contains `0x5c`. The total query is 96, resp. 176 bytes long.

2. **Inner HMAC queries.** We divide the first $\mathsf{RO_H}$ query made by HMAC into several subtypes; this type includes only those where the input to HMAC is an arbitrary string of length Hl. This subtype is formatted identically to an outer HMAC query, except that the bytes of the fixed region are fixed to the value `0x36` instead of `0x5c`. TLS 1.3 makes inner HMAC queries while computing `Finished` and *binder* messages (where the input is a hashed transcript), the early and master secrets, and in PSK-only mode, also the handshake secret.

3. **Diffie–Hellman HMAC query.** In PSK-(EC)DHE mode, TLS 1.3 computes the handshake secret by calling HMAC on an encoded Diffie–Hellman key share. HMAC's first query is a Diffie–Hellman HMAC query. The formatting is the same as an inner HMAC hash except that the segment following the fixed region has a different length. Namely, the byte length (denoted by $|\mathbb{G}|/8$) of the encoding of an element of a standardized Diffie–Hellman group. The actual byte length for each standardized Diffie–Hellman group can be found in Table 4. The total query length is then $Bl + |\mathbb{G}|/8$ bytes, which is $64 + |\mathbb{G}|/8$ bytes if the hash function is `SHA256` and $128 + |\mathbb{G}|/8$ bytes if the hash function is `SHA384`.

4. **`Derive-Secret` hashes.** The `Derive-Secret` function is a component of the TLS key schedule [54, Section 7.1]. Its inputs are a key of length Hl, a label string of 2 to 12-bytes in length, and an input `Messages` string.

   `Derive-Secret` queries $\mathsf{RO_H}$ three times: once to hash the `Messages` string, and twice as part of HKDF.Expand. The first of these three queries is a transcript query, and the third is an Outer HMAC query. The second query we call a `Derive-Secret` query. The `Derive-Secret` query has the same formatting as Inner HMAC queries and Diffie–Hellman queries, but the segment following the fixed region contains a strictly formatted `HkdfLabel` struct [54, Section 7.1].

   This struct begins with a two-byte field encoding the integer value Hl. This is followed by a variable-length vector with a 1-byte length field containing the string `"tls13 "` followed by a label string with length between 2 and 12 bytes. Lastly comes a vector of length Hl, prefixed with a 1-byte field encoding its length. The last byte in the input contains the `0x01`. This byte is the counter mandated by the definition of HKDF.Expand; however since HKDF.Expand is never called on inputs longer than Hl, the counter never reaches a value higher than 1.

   The total length of the label struct, including the counter byte, is at least $Hl + 13$ bytes and at most $Hl + 23$ bytes. The total query is thus in the range of $Bl + Hl + 13$ and $Bl + Hl + 23$ bytes, which is 109–119 bytes if the hash function is `SHA256` and 189–199 bytes if the hash function is `SHA384`.

5. **Finished hash.** The HKDF-Expand-Label function is a subroutine of the Derive-Secret function, but also called during the computation of Finished messages and the *binder* value [54, Section 4.4.4]. HKDF-Expand-Label makes two calls to $\mathsf{RO_H}$. The second is an Outer HMAC hash; we call the first a Finished hash. A Finished hash is identical to a Derive-Secret hash, except that the label string is fixed to finished and the final vector has length 0. The counter byte is still present. In total, the label struct occupies 19 bytes. The total query is thus $Bl + 19$ bytes, which is 83 bytes if the hash function is SHA256 and 147 bytes if the hash function is SHA384.

6. **Empty hashes.** Occasionally in the key schedule, TLS 1.3 calls $\mathsf{RO_H}$ on the empty string.

7. **Transcript hashes.** The last use of $\mathsf{RO_H}$ is to condense partial transcripts. Each transcript includes at least a partial ClientHello message. We assume calling $\mathsf{RO_H}$ on a transcript which includes at least a partial ClientHello. The minimum length of a partial ClientHello message in PSK-only mode is 73 bytes. This includes the following fields[15] [54, Section 4.1.2]:

   - 1 byte message type fixed to 0x01
   - 3 bytes encoded message length
   - 2 bytes legacy_version fixed to 0x0303
   - 32 bytes random
   - 1 byte legacy_session_id (for an empty vector with 1-byte length field)
   - 4 bytes ciphersuites (must include a 2-byte length field and at least one value)
   - 2 bytes legacy_compression_methods (must include a 1-byte length field and the value 0x00)
   - 2 bytes encoded length of extensions field
   - 7 bytes supported_versions extension extension [54, Section 4.2.1] (must start with 0x002b and include 0x0304)
   - 6 bytes psk_key_exchange_modes extension [54, Section 4.2.9] (must start with 0x002d and include 0x00)
   - 13 bytes pre_shared_key extension [54, Section 4.2.11] (partial: excluding the binder list; must come last, must start with 0x0029)

   The first 47 bytes (through the extensions' length encoding), must appear in the order displayed, although the legacy_session_id, ciphersuites, and legacy_compression_methods fields can be longer than the lengths given above. We will occasionally refer to this segment as the "fixed preface" of a ClientHello because it must appear at the beginning of every well-formed ClientHello message. The extensions can be reordered arbitrarily (except for the pre_shared_key extension) and additional extensions and ciphersuites can be added or repeated, up to a maximum length of $2^{16} - 2$ bytes of ciphersuites and $2^{16} - 1$ bytes for extensions. The vectors legacy_session_id and legacy_compression_methods have a maximum length of 32 bytes and $2^8 - 1$ bytes, respectively. The overall maximum length of a truncated ClientHello is then $2 \cdot 2^{16} + 328$ bytes. A full ClientHello in PSK-only mode, including the binder list, adds at least another $3 + Hl$ bytes for a binders vector with 3 bytes of encoded length. The binders vector has a maximum length of $2^{16} - 1$ bytes with a 2-byte length field. The ClientHello message thus contains a minimum of $76 + Hl$ bytes and a maximum of $3 \cdot 2^{16} + 329$ bytes.

   In PSK-(EC)DHE mode, two additional extensions are also mandatory: the key_share and supported_groups extensions [54, Section 9.2], so the minimum ClientHello length increases by at least $18+|\mathbb{G}|/8$ bytes, cf. Table 4[16]. This increase occurs for both truncated and full ClientHello messages. In this mode,

[15] An earlier version omitted the leading 1-byte message type and 3-byte message length encoding. We thank Robert Merget for pointing this out, which leads to an accordingly modified domain separation analysis. In brief, domain separation for SHA384 is still lacking, unless one assumes parties only accept defined extensions and ciphersuites; see [17].

[16] This includes 8 bytes for supported_groups and $10 + |\mathbb{G}|/8$ bytes for key_share. An standard-compliant key_share extension may be empty and thus only 6 bytes if the client is requesting a HelloRetryRequest message; however in this case the subsequent transcript hash will contain two ClientHello messages and a HelloRetryRequest;

Group name	`NamedGroup` enum value	Encoding length $|\mathbb{G}|/8$
secp256r1 [53]	0x0017	32
secp384r1 [53]	0x0018	48
secp521r1 [53]	0x0019	66
x25519 [48]	0x001d	32
x448 [48]	0x001E	56
ffdhe2048 [32]	0x0100	128
ffdhe3072 [32]	0x0101	192
ffdhe4096 [32]	0x0102	256
ffdhe6144 [32]	0x0103	384
ffdhe8192 [32]	0x0104	512

Table 4: Table displaying the standardized groups for use with TLS 1.3, their encodings in the `NamedGroup` enum, and the length of an encoded group element in bytes.

Type	Minimum length (bytes)	Maximum length (bytes)
Outer HMAC	96	96
Inner HMAC	96	96
`Derive-Secret`	109	119
`Finished`	83	83
Empty	0	0
Transcript	73	$2 \cdot 2^{16} + 328$

Table 5: Table showing input lengths for hash function calls made by TLS 1.3 in PSK-only mode with `SHA256`.

a truncated `ClientHello` message is at least $91 + |\mathbb{G}|/8$ bytes long, and a full `ClientHello` is at least $94 + Hl + |\mathbb{G}|/8$ bytes long. The maximum lengths are identical to those in PSK-only mode as the two additional mandatory extensions `key_share` and `supported_groups` were only accounted for in the maximum length of the `extensions` field.

B.1 PSK-only mode with `SHA256`

The block length of this hash function is 64 bytes, and the output length is 32 bytes. In Table 5, we give the minimum and maximum input lengths for each of the six call types. (Diffie–Hellman HMAC calls do not occur in this mode.)

In Table 5 we note the minimum and maximum input lengths of each type of message. For those types with overlapping length ranges, we must show they have separate domains by other means. Outer and Inner HMAC hashes have identical lengths; however each of them has a 32-byte fixed region. In outer HMAC hashes, the fixed region contains opad; in inner HMAC hashes, it contains ipad. These are distinct values, so no string can be both an outer and an inner HMAC hash.

Transcript hashes are not domain-separated by length from any hash except the empty hashes. We therefore turn to formatting to separate these from other types. In the following, we visually lay out each byte of potentially overlapping inputs.

For a string to be both a transcript and an HMAC hash (outer or inner), it must be 96 bytes (cf. Table 5)

the second `ClientHello` in the transcript must contain a non-empty `key_share` extension along with the other mandatory extensions; thus the total length of the transcript will increase by more than $10 + |\mathbb{G}|/8$ bytes even if the first `key_share` extension is empty.

Type	Minimum length (bytes)	Maximum length (bytes)
Outer HMAC	96	96
Inner HMAC	96	96
Diffie–Hellman HMAC	$64 + \lvert\mathbb{G}\rvert/8$	$64 + \lvert\mathbb{G}\rvert/8$
Derive-Secret	109	119
Finished	83	83
Empty	0	0
Transcript	$91 + \lvert\mathbb{G}\rvert/8$	$2 \cdot 2^{16} + 328$

Table 6: Table showing input lengths for hash function calls made by TLS 1.3 in PSK-(EC)DHE mode with SHA256. For transcript hashes, the encoding lengths $|\mathbb{G}|/8$ can be found in Table 4.

long. We diagram and compare a transcript hash containing a partial ClientHello[17] and an HMAC hash (outer or inner) in Figure 12.

Fixed preface: 47 B	Extension data: 36 B	End PSK: 13 B

Key: 32 B	Fixed ipad/opad: 32 B	Arbitrary string: 32 B

Figure 12: Domain separation in PSK-only mode with SHA256: Transcript hash containing a partial ClientHello (top) vs. (outer or inner) HMAC hash (bottom). "End PSK" is the end of the pre_shared_key extension.

We can see that the fixed preface of the transcript hash overlaps the fixed region of the HMAC hash that is fixed to either ipad or opad. Consequently, the legacy_session_id vector must begin within the fixed region (at byte 39). This is a variable-length vector preceded by a 1-byte length field, and its maximum length is 32 bytes [54, Section 4.1.2]. Therefore the maximum value of the length field is 0x20 and it cannot contain either byte 0x36 or 0x5c. Any string containing a valid partial ClientHello therefore cannot also be a correctly formatted HMAC hash.

The same argument applies to Finished and Derive-Secret hashes, both of which contain the same fixed region in the same location as inner HMAC hashes.

For this mode, we define the set D_{Th} to include of the empty string and all strings of length greater than or equal to 69 bytes for which the 39th byte is not equal to ipad or opad. We let D_{Ch} contain all other elements of $\{0,1\}^*$.

B.2 Pre-shared key with Diffie–Hellmann mode with SHA256

Again, we present the minimum and maximum lengths of each hash type; see Table 6. We now include Diffie–Hellman HMAC hashes, and transcript hashes include additional mandatory extensions for PSK-(EC)DHE mode.

In this mode, Diffie–Hellman HMAC hashes may collide with Inner HMAC or Derive-Secret hashes for certain choices of $\mathbb{G}$. This is not a failure of domain separation because these inputs to these three types will all belong to D_{Ch}. Transcript hashes now only have length overlaps with Diffie–Hellman HMAC and Derive-Secret hashes. In both cases, however, the same argument about the 39th byte containing the length of legacy_session_id applies, and no string can be two different types.

[17] A full ClientHello contains at least $76 + Hl \geq 108$ bytes, which is too long to be an HMAC hash.

Type	Minimum length (bytes)	Maximum length (bytes)
Outer HMAC	176	176
Inner HMAC	176	176
Diffie–Hellman HMAC	$128 + \lvert\mathbb{G}\rvert/8$	$128 + \lvert\mathbb{G}\rvert/8$
`Derive-Secret`	189	199
`Finished`	147	147
Empty	0	0
Transcript	$91 + \lvert\mathbb{G}\rvert/8$	$2 \cdot 2^{16} + 328$

Table 7: Table showing input lengths for hash function calls made by TLS 1.3 in PSK-(EC)DHE mode with `SHA384`.

For this mode, the set D_{Th} consists of the empty string and all strings of length greater than or equal to $91 + |\mathbb{G}|$ bytes for which the 39th byte is not equal to ipad or opad. D_{Ch} contains all other elements of $\{0,1\}^*$.

B.3 Pre-shared key with Diffie–Hellmann mode with SHA384

Table 7 shows the minimum and maximum lengths of each hash type for this configuration. The hash function `SHA384` has 48-byte output and 128-byte block length, so the fixed region in HMAC, `Finished`, and `Derive-Secret` hashes will be 80 bytes long.

Unlike the PSK modes with `SHA256`, we cannot rely on the distinction between `legacy_session_id` length field and the fixed region for domain separation, because the 48-byte HMAC keys for `SHA384` already reach past the position of the `legacy_session_id` length field at byte 39. Instead, we consider whether a minimum-length `ClientHello` can accommodate the mandatory extensions for this mode.

We worry only about possible collisions between transcript hashes and the other types: `Finished`, (outer and inner) HMAC, and `Derive-Secret`. We diagram a transcript hash of 176 bytes together with an outer HMAC hash as a demonstration of the domain-separation argument in Figure 13, but the same argument applies to all.

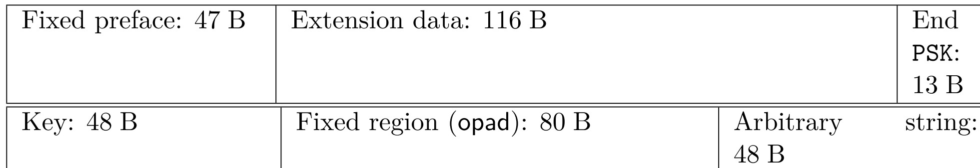


Figure 13: Domain separation in PSK-(EC)DHE mode with `SHA384`: Transcript hash of 176 bytes (top) vs. outer HMAC hash (bottom). "End `PSK`" is the end of the `pre_shared_key` extension.

There are no obvious conflicts here: the fixed preface of a `ClientHello` message is covered by the key section of the HMAC hash, and the `pre_shared_key` extension is covered by the arbitrary string at the end. However, notice that of the 116 bytes available for extension data in the `ClientHello`, 80 of them must be fixed to opad to allow a collision. Even including the 1 bytes immediately after the fixed preface and 13 bytes reserved for the `pre_shared_key` extension, this leaves only 50 bytes. In PSK-(EC)DHE mode, five extensions are mandatory even for truncated `ClientHello` messages. They are `supported_versions` [54, Section 4.2.1] (minimum 7 bytes), `supported_groups` [54, Section 4.2.7] (minimum 8 bytes), `key_share` [54, Section 4.2.8] (minimum $10+|\mathbb{G}|/8$ bytes), `psk_key_exchange_modes` [54, Section 4.2.9] (minimum 6 bytes), and `pre_shared_key` [54, Section 4.2.11] (minimum 13 bytes). Even for the smallest choice of $\mathbb{G}$, at least 76 bytes are required to contain these extensions. At least one of the extensions must overlap with the fixed field, and will differ from opad in at least one byte.

Any valid transcript hash will need at least $91 + |\mathbb{G}|/8$ bytes outside the fixed region: 47 bytes for the

preface and $44 + |\mathbb{G}|/8$ for the mandatory extensions. An outer HMAC hash has only 96 unfixed bytes and cannot meet this threshold. This is true also for inner HMAC hashes (96 unfixed bytes), and Diffie–Hellman HMAC hashes, which have $48 + |\mathbb{G}|/8$ unfixed bytes. It is true for `Finished` hashes, which have 48 unfixed bytes, because of the 80-byte fixed region and the fixed 19-byte label struct for the `finished` label. And it is true for `Derive-Secret` hashes, which have at most 119 unfixed bytes.

Let us be even more clear about why this overlap means no collision is possible. We cannot fit all of the mandatory extensions in the segment after the fixed region. Therefore one of the extensions must start either in the fixed region, or before the fixed region. None of these extensions can start in the fixed region because they all begin with an extension type different from `ipad` or `opad` (cf. [54, Section 4.2]). Therefore one of them must start before the fixed region and continue into the fixed region. We call this the "first extension". The `pre_shared_key` extension must be the last extension, so it cannot be the first extension. Therefore the first extension is one of `key_share`, `supported_groups`, and `psk_key_exchange_modes`, and `supported_versions`. All extensions start with a 4 byte encoding of their type and length. Since the fixed preface is already 47 bytes, the second extension type byte of the first extension would need to be either `0x5c` or `0x36`. However, none of the aforementioned extensions contains these bytes on the second position of its extension type. Consequently, the extensions can neither start before nor in the fixed region. Moreover, we outline above that the space after the fixed region alone is too tight to fit all of the mandatory extensions.

To be precise, the mandatory extensions must occupy no more than 71 bytes after the fixed region (for the longest possible `Derive-Secret` hash) or $|\mathbb{G}|/8$ bytes after (for an inner HMAC hash). But summing their minimum lengths gives $44 + |\mathbb{G}|/8$ bytes. Even for the smallest possible $|\mathbb{G}|/8 = 32$, the extensions just do not fit in the given space. It is therefore impossible to construct a valid `ClientHello` message, truncated or otherwise, that collides with a possible HMAC, `Derive-Secret`, or `Finished` hash.

Consequently we can set D_{Th} to contain the empty string and all strings of at least 86 bytes for which at least one of bytes 49 through 128 does not equal either `ipad` or `opad`. Again, we set D_{Ch} to be all other elements of $\{0,1\}^*$.

B.4 PSK-only mode with SHA384

In this mode/hash function combination, the transcript hash *can* collide with outer HMAC hashes. There are other collisions as well, but one is sufficient to demonstrate the lack of domain separation. We illustrate this via a 176-byte transcript hash (containing a truncated `ClientHello`) and an outer HMAC hash, shown in Figure 14.

Fixed preface: 39 B	`ciphersuites`: `0x0058` ‖ `0x1301` ‖ (`opad`)^{43}: 90 B	`cookie`: 19 B	Mandatory extensions: 26 B
Key: 48 bytes	Fixed region (`opad`): 80 bytes	Arbitrary string: 48 bytes	

Figure 14: Failing domain separation in PSK-only mode with SHA384: Transcript hash of 176 bytes, containing a truncated `ClientHello` (top) vs. outer HMAC hash (bottom).

We construct the following message, which is both a truncated `ClientHello` (and therefore a transcript hash) and an outer HMAC hash. The message starts with the fixed 39 bytes through the `legacy_session_id`. That is, 1 byte message type fixed to `0x01`, 3 bytes encoding the message length 176 (i.e., `0x0000B0`), 2 bytes `legacy_version` fixed to `0x0303`, 32 arbitrary bytes for `random`, and 1 byte encoding the `legacy_session_id` length fixed `0x00` followed by the empty vector. The next segment of the preface is the `ciphersuites`. To construct the collision, we use that the standard RFC8446 mandates that servers must ignore ciphersuite values that it does not recognize (e.g., undefined) and must process only the recognized ones as usual (cf. [54, Section 4.1.2]). This means that a `ciphersuites` vector containing at least one standardized ciphersuite is well-formed. We let the `ciphersuites` contain 44 ciphersuites, where the first ciphersuite is the valid, mandatory `0x1301`, and the remaining (undefined) 43 are `opad` (= `0x5c5c`). Thus, the 39 bytes described above are followed by the length field of the `ciphersuites` vector encoding 88 in 2 bytes, i.e., `0x0058`, and the `ciphersuites` vector `0x1301` ‖ (`opad`)^{43}. This `ciphersuites` field is followed by the `legacy_compression_methods` vector (with 1-byte length field), which must contain a single null byte. We now have fixed 131 bytes of the message, which means that there are still $176 - 131 = 45$ bytes to define.

In particular, we need to define 45 bytes of extensions. Therefore, the `legacy_compression_methods` vector is followed by 2 bytes encoding 45, which is the length of the `extensions` vector. In PSK-only mode, the mandatory extensions are only `supported_versions`, `psk_key_exchange_modes`, and (the truncated) `pre_shared_key`, and they take up 26 bytes. Since `pre_shared_key` always has to be the last extension, we set these 26 bytes at the end of the message. Finally, there are 19 bytes left undefined between the extension length field and the mandatory extensions. We can fill these with a `cookie` [54, Section 4.2.2] extension with arbitrary content. Like every extension this starts with 4 bytes of extension type (`0x0068`) and length (`0x00FF`), and is then followed by 15 bytes of arbitrary content. (We could also fill these bytes without including additional extensions.)

The `0x5c5c` extension values in the constructed message match the HMAC `opad` key padding in the format-restricted portion of the outer HMAC hash, the remainder lies in the unrestricted portion. This type of collision is unavoidable, so there are no disjoint sets D_{Th} and D_{Ch} that capture the way TLS 1.3 calls `SHA384` in pre-shared key only mode. Consequently the indifferentiability step of Section 5.1.1 does not apply to this mode.

We remark that if one is willing to accept the additional assumption (for PSK-only with `SHA384`) that client and servers only consider `Hello` messages valid that solely use standardized ciphersuite values and extension types, meaning that the above collision would be discarded as a malformed `ClientHello`, one can show domain-separation for this mode/hash function combination as well, as demonstrated in [17]. The author of [17] considers this assumption reasonable as honest client implementations would never use undefined values in their `Hello` messages. That is, the presence of undefined values would inherently uncover tampering with a message. Since for domain separation only queries from honest executions are of interest, considering only messages that might be output by honest parties is only a mild assumption even though this is not standard-compliant.

B.5 Repairing Domain Separation for TLS 1.3-like Protocols

The above analysis demonstrates that complete domain separation is nontrivial to achieve for a protocol like TLS 1.3 which uses a hash function for multiple purposes and at multiple levels of abstraction. We would like to present our suggestions for how this could be achieved most simply and efficiently in future iterations of TLS and other schemes. As discussed by Bellare et al. [5], the most well-known method of domain separation is the inclusion of distinct labels into each hash function call; this is precisely the method adopted by TLS 1.3 to distinguish calls to its `Derive-Secret` function. Ideally, a future scheme could specify a unique label string for each purpose: not only the various derived secrets, but also each time the transcript is hashed and each internal call made by HMAC, HKDF.Extract, and HKDF.Expand.

Unfortunately, this ideal method is not compatible with the existing specifications of HMAC and HKDF. Both of these functions make "outer HMAC queries" as discussed above; these calls have a fixed input length of $Bl + Hl$ bytes and this input does not include a label. A protocol could avoid this roadblock by using an implementation of HMAC or HKDF with a custom underlying hash function that prepends an HMAC-specific label to its input. This approach would be both standard-compliant and efficient, but we do not recommend it because existing cryptographic libraries already have trustworthy HMAC and HKDF functionality and encouraging custom implementations for every new protocol increases the probability of accidental errors in these new implementations. Instead, we suggest making no adjustments to the internal execution of HMAC or HKDF and instead altering direct hash function calls (the other six subtypes we discuss) to avoid collisions.

In practice, this means that under our recommendation, all hash function calls which are not outer HMAC queries should obey two simple rules: first, they should end with a unique label and second, that their input must not be $Bl + Hl$ long. To conform with the first rule, TLS 1.3 would need to make the following changes.

1. Add distinct labels to the end of each transcript before hashing; for clarity we suggest using the names of the last message in the transcript; i.e. "`PartialClientHello`", "`ClientHello`", "`ServerHello`", etc. If HKDF is used, we would also recommend that these labels should not end with the byte `0x01`.

2. Add distinct labels to the end of the input each time HMAC is called; this would include inner HMAC queries, Diffie–Hellman HMAC queries, `Finished` queries, and `Derive-Secret` queries. Note that the labels should be postpended to the HMAC payload and not the key. The labels used by `Derive-Secret` could then be omitted, although this is not necessary.

3. Ensure that none of the labels used is a suffix of another; this can introduce collisions even if the labels are distinct.

We encourage using suffixes for domain separation, although prefixes are more commonly-used, because they are easier to use in conjunction with HMAC and HKDF. Although we are not applying labels to outer HMAC queries, we would still like to use them to domain separate inner HMAC queries (and the other subtypes). The inputs to these queries begin with the HMAC key, which undergoes an XOR operation with ipad before it is hashed. So prefixed labels would need to remain unique and prefix-free after this XOR operation; this introduces some confusion that we prefer to avoid. Additionally, the second step of our indifferentiability proof relies crucially on the fact that HMAC uses fixed-length keys shorter than Bl; prefixed labels would therefore need to share a fixed length shorter than $Bl - Hl$ bytes. With suffixes, we still need to contend with the counter byte that HKDF.Expand appends to its input, but in TLS 1.3 where this byte is always `0x01`, this presents less of a restriction.

To conform with the second rule, TLS 1.3 would need to enforce that it never hashes a string of $Bl + Hl$ except as an outer HMAC query. The easiest and least error-prone way to do this would be to pad every non-empty hash function call and input to HMAC and HKDF with exactly $Bl + Hl$ bytes (before the suffixed labels); all calls would then be strictly longer than $Bl + Hl$. This method adds two additional compression function calls to each hash function execution. There are some ways to lessen this requirement without impacting the effectiveness of the length-based domain separation. Calls which already have input longer than $Bl + Hl$ bytes can omit the padding; so can calls which have strictly shorter input. It would also be possible to use only as much padding is needed to make inputs at least $Bl + Hl + 1$ bytes long. However, non-uniform padding should be done carefully so that, for example, two previously distinct `ClientHello` messages do not collide after being padded.