On the Concrete Security of TLS 1.3 PSK Mode*

Hannah Davis[1], Denis Diemert[2], Felix Günther[3], and Tibor Jager[2]

[1] University of California San Diego, La Jolla, CA, USA
h3davis@eng.ucsd.edu
[2] Bergische Universität Wuppertal, Wuppertal, Germany
denis.diemert@uni-wuppertal.de, tibor.jager@uni-wuppertal.de
[3] ETH Zürich, Zürich, Switzerland
mail@felixguenther.info

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.

1 Introduction

The *Transport Layer Security* (TLS) protocol is probably the most widely-used cryptographic protocol. It provides a secure channel between two endpoints

* Some of this work was done while Hannah Davis was visiting ETH Zurich. Felix Günther was supported in part by German Research Foundation (DFG) Research Fellowship grant GU 1859/1-1. Tibor Jager was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement 802823.

(*client* and *server*) for arbitrary higher-layer application protocols. Its most recent version, TLS 1.3 [48], 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* [29], since the latter is not achievable with static symmetric keys.[4] 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

[4] See [2,9] for recent work discussing symmetric key exchange and forward secrecy.

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 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 [42], and several draft revisions as well as the final TLS 1.3 standard in RFC 8446 [48] were analyzed by many different research groups, including computational/reductionist analyses of the full and PSK modes in [19,20,25,21]. 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, [17] 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, [14] 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 [14] and Diemert and Jager [17] 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 [3,28,16,31], 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 [14] and [17] 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. [11], 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 [17,14] to enable the tight proof technique of Cohn-Gordon et al. [11], we support this new abstraction formally. Using the *indifferentiability* framework of Maurer et al. [46] in its recent adaptation by Bellare et al. [5] that treats *multiple* random oracles, in Section 4 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 [17,14].

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 the full version of this paper [13].

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 the full version.

Modularization of record layer encryption. Like most of the prior computational TLS 1.3 analyses [19,25,21,17], we use a *multi-stage key exchange* (MSKE) security model [24] 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 [33] which monolithically treat handshake and record-layer encryption.

As a generic simplification step for MSKE models, we show (in Section 5) 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 [21] over a wide range of fully concrete resource parameters, following the approach of Davis and Günther [14]. 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 [33,40,27,41,44,7], including its PSK-modes [44]; all reductions in these works are highly non-tight.

Brzuska et al. [10] 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 [22,6] 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 [17,14] in that way to future work.

Like many previous cryptographic analyses [33,40,19,20,25,21,17,14] 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 [22] or backwards compatibility issues in settings where multiple TLS versions are used in parallel, such as [34]. 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 [24,19,30,21,17] treating the combined security when subsequent protocols use the session keys established in an MSKE protocol.

Numerous authenticated key exchange protocols [28,11,45,32,31] 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*.[5] 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 [12,39] for analysis rationale behind the binder value).

[5] 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.

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` [47]. In our security analysis, we will model $\mathbf{H}$ as a random oracle.

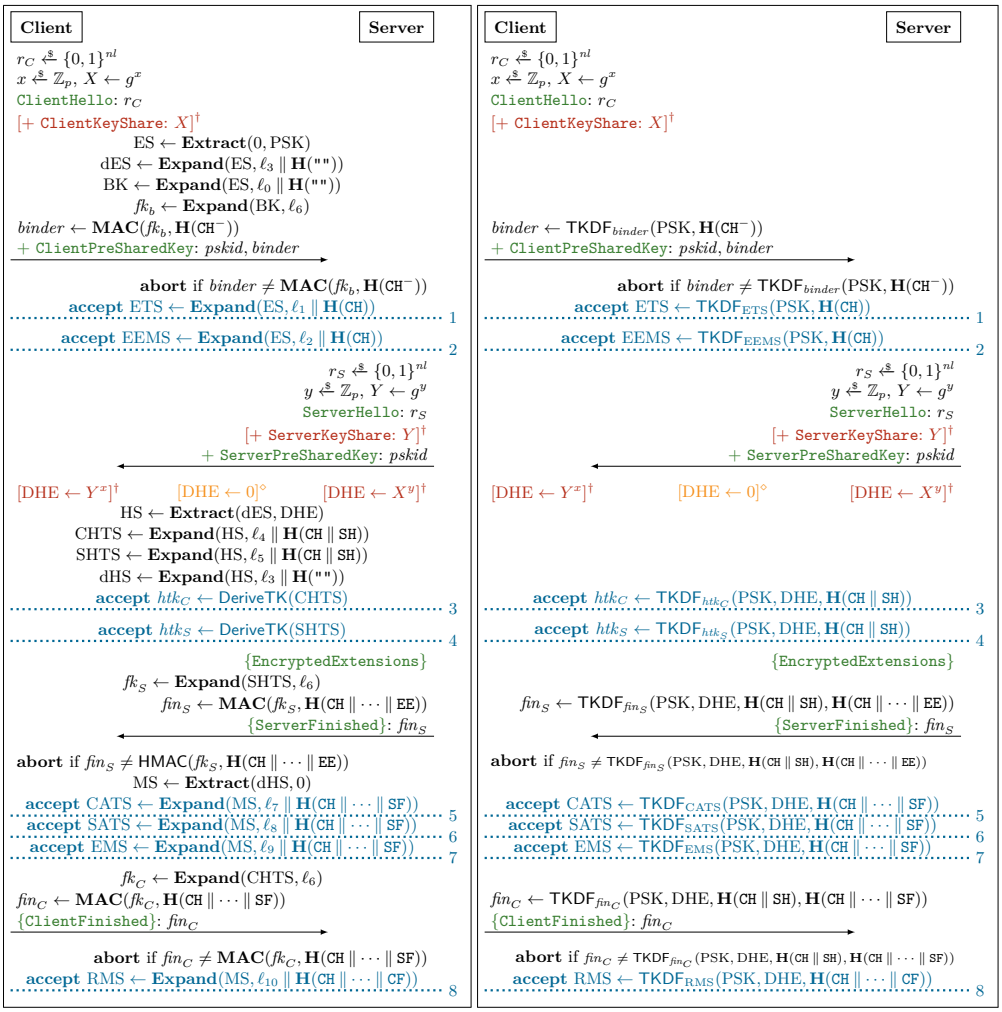
On the left-hand side of Figure 1, we distinguish four named subroutines of TLS 1.3 which use $\mathbf{H}$ for different purposes:

- 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 [38,36]. These functions are defined
  - $\mathbf{Extract}(K, M) := \mathsf{HKDF.Extract}(K, M) = \mathbf{MAC}(K, M)$.
  - $\mathbf{Expand}(K, M) := \mathsf{HKDF.Expand}(K, M) = \mathbf{MAC}(K, M \parallel \texttt{0x01})$.[6]

[6] HKDF.Expand [36] 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.



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		
[...]◇	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)

Fig. 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 the full version [13] for their definition.

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

1 ES $\leftarrow$ **Extract**$(0, \mathrm{PSK})$
2 dES $\leftarrow$ **Expand**$(\mathrm{ES}, \ell_3 \parallel \mathsf{Th}(\texttt{""}))$
3 HS $\leftarrow$ **Extract**$(\mathrm{dES}, \mathrm{DHE})$
4 SHTS $\leftarrow$ **Expand**$(\mathrm{HS}, \ell_5 \parallel \mathsf{h}_1)$
5 $\mathit{fk}_S \leftarrow$ **Expand**(SHTS, ℓ_6)
6 $\mathit{fin}_S \leftarrow$ **MAC**$(\mathit{fk}_S, \mathsf{h}_2)$
7 return fin_S

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

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 [35,36].

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 4. We introduce 11 functions $\mathsf{TKDF}_{\mathit{binder}}$, $\mathsf{TKDF}_{\mathrm{ETS}}$, $\mathsf{TKDF}_{\mathrm{EEMS}}$, $\mathsf{TKDF}_{\mathit{htk}_C}$, $\mathsf{TKDF}_{\mathit{fin}_C}$, $\mathsf{TKDF}_{\mathit{htk}_S}$, $\mathsf{TKDF}_{\mathit{fin}_S}$, $\mathsf{TKDF}_{\mathrm{CATS}}$, $\mathsf{TKDF}_{\mathrm{SATS}}$, $\mathsf{TKDF}_{\mathrm{EMS}}$, 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 $\mathsf{TKDF}_{\mathit{fin}_S}$, 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 `ServerFinished` message authentication code on the hashed transcript and encrypted extensions. Accordingly, we define $\mathsf{TKDF}_{\mathit{fin}_S} \colon \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$ 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 $\mathsf{h}_1 = \mathsf{Th}(\texttt{CH} \parallel \texttt{SH})$ and $\mathsf{h}_2 = \mathsf{Th}(\texttt{CH} \parallel \cdots \parallel \texttt{EE})$, and it outputs a MAC tag for the `ServerFinished` message. The remaining key derivation functions are defined the same way; we give their signatures in the full version [13].

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. [21]. We fully specify our model in pseudocode in the full version [13]. We adopt the explicit authentication property from the model of Davis and Günther [14] and capture forward secrecy by following the model of Schwabe et al. [49].

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.[7] 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 [23]. 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, a formalization of which can be found in the full version [13].

[7] 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.

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.[8] 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 $\mathsf{KE.KS}[s]$ 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, 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 au-

[8] 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 [8]. 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.

thenticated in the sense of Krawczyk [37], 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 $\mathsf{ExplicitAuth}$ 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 [26] 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 [15]. 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 [49]: 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, 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 [21], 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* $\in \{\mathsf{running}_s, \mathsf{accepted}_s, \mathsf{rejected}_s \mid s \in [1, \ldots, \mathsf{STAGES}]\}$, 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.
- 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 ∞.
- revealed$[s]$. A boolean denoting whether the stage s key has been leaked through a RevSessionKey query. Initialized to false.
- tested$[s]$. The time at which the stage s key was tested. Initialized to ∞ before any Test query occurs. (In boolean expressions, we write tested$[s]$ as a shorthand for 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 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 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 above. 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 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* [46] 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 6 significantly simpler and cleaner, but also it puts the entire protocol security analysis on a firmer theoretical ground than previous works. For some background on the indifferentiability framework, see the full version [13].

In their proof of tight security, Diemert and Jager [17] previously modeled the TLS 1.3 key schedule as four independent random oracles. Davis and Günther [14] 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 [17,14] 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 the full version [13] 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 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* $\mathbf{H}$ *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}, ..., RO_RMS *are random oracles with the appropriate signatures (see the full version [13] for the signature details). Then,*

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

We establish this result via three modular steps in the indifferentiability framework introduced by Maurer, Renner, and Holenstein [46]. 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.

4.1 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 [46] given in the full version [13] of this paper 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 the full version [13].

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 RO_H to implement Th and Ch, the KE_1 protocol instead uses two independent random oracles RO_Th and $\mathsf{RO}_\mathsf{HMAC}$. 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 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 RO_Ch within the set D_Ch and RO_Th within the set D_Th.
We defer details on the exact domain separation to the full version [13], 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 $\mathsf{HMAC}[\mathsf{RO}_\mathsf{Ch}]$, **MAC** now directly queries a new random oracle $\mathsf{RO}_\mathsf{HMAC}\colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$. Since RO_Ch was only called by **MAC**, we drop it from the protocol, but we do continue to use RO_Th, i.e., KE_2 uses two random oracles: RO_Th and $\mathsf{RO}_\mathsf{HMAC}$. The security of this replacement follows directly from Theorem 4.3 of Dodis et al. [18], which proves the indifferentiability of HMAC with fixed-length keys.[9]

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

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 $\mathsf{RO}_{\mathsf{Th}}$ 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 the full version [13].) For this step, we view TKDF as a construction of 11 random oracles from a single underlying oracle ($\mathsf{RO}_{\mathsf{HMAC}}$). 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 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 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. In the full version [13], we give a theorem that 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 2. *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 Tight Security of the TLS 1.3 PSK Modes

In this section, we apply the insights gained in Sections 4 and 5 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 3 for the TLS 1.3 PSK-(EC)DHE mode and for the TLS 1.3 PSK-only mode in the full version [13].

6.1 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 $(sid[s])_{s\in\{1,\ldots,8\}}$ are defined as follows:[10]

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

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

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 $(\texttt{CH}, \texttt{CKS}^\dagger, \texttt{CPSK})$ set $cid_{\mathsf{responder}}[3] = (\text{“}htk_C\text{”}, (\texttt{CH}, \texttt{CKS}^\dagger, \texttt{CPSK}))$ and $cid_{\mathsf{responder}}[4] = (\text{“}htk_S\text{”}, (\texttt{CH}, \texttt{CKS}^\dagger, \texttt{CPSK}))$. Further, when the client receives (resp. the server sends) the message $(\texttt{SH}, \texttt{SKS}^\dagger, \texttt{SPSK})$, they set $cid_{\mathsf{initiator}}[3] = sid[3]$ and $cid_{\mathsf{initiator}}[4] = sid[4]$. 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 $s \in \{1, \ldots, 8\}$.

Analogously, responder sessions receive explicit authentication right before accepting stage 8 via the `ClientFinished` message; i.e., $\mathsf{EAUTH}[\mathsf{responder}, s] = 8$ for all stages $s \in \{1, \ldots, 8\}$.

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 $s \in \{3, \ldots, 8\}$ 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 $s \in \{3, \ldots, 8\}$. 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 $s \in \{1, 2\}$.

6.2 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 3. *Let* `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 1. 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 2. 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 3. 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 [48, 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 [43] 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.

6.3 Proof overview

The proof proceeds via a sequence of games in three phases, corresponding to the three ways for an adversary to win the MSKE security game. We begin with Game_0, the original MSKE game for protocol `TLS1.3-PSK-(EC)DHE` described above. In the first phase, we establish that the adversary cannot violate the Sound predicate. In the second phase, we establish the same for the $\mathsf{ExplicitAuth}$ predicate. In the third phase, we ensure that all TEST queries return random keys regardless of the value of the challenge bit b, so long as the Fresh predicate is not violated. After that, the adversary cannot win the game with probability better than guessing, rendering its advantage to be 0. We bound the advantage difference introduced by each game hop; collecting these intermediate bounds yields the overall bound. For space reasons, we only provide a summary of the proof in the following and refer to the full version [13] for the full details.

Phase 1: Ensuring Sound

The Sound predicate checks that no more than two sessions can be partnered in a non-replayable stage, and that any two partnered sessions must agree on

the stage, pre-shared key identifier, the stage-s key, and each others' identities and roles. We defined our session identifiers so that the stage-s session identifier contains (1) a label unique to that stage, (2) a unique `ClientHello` and `ServerHello` message, (3) the *binder* message: a MAC tag authenticating the `ClientHello` and pre-shared key, and (4) sufficient information to fix the stage-s key. (This does not mean the key is computable from the *sid*; it is not.)

We then perform three incremental game hops that cause the FINALIZE oracle to return 0 in the event of a collision between two `Hello` messages, *binder* tags, or pre-shared keys. We bound the difference in advantage in the first two game hops via a birthday bound over the number of potentially colliding values (i.e., pairs of nonces and `KeyShares` in $\mathbb{G}$ for `Hello` message collisions, and sampled PSK keys for pre-shared key collisions), and the third hop by a reduction to the collision resistance of the RO_{binder} random oracle whose advantage in turn is upper bounded by a birthday bound $\mathrm{Adv}^{\mathsf{CR}}_{\mathsf{RO}_{binder}}(q_{\mathrm{RO}} + q_{\mathrm{S}}) \leq \frac{(q_{\mathrm{RO}}+q_{\mathrm{S}})^2}{2^{hl}}$. The resulting bounds are, in this order: $\Pr[\mathsf{Game}_0] - \Pr[\mathsf{Game}_3] \leq \frac{2q_{\mathrm{S}}^2}{2^{nl}\cdot p} + \frac{q_{\mathrm{NS}}^2}{2^{hl}} + \frac{(q_{\mathrm{RO}}+q_{\mathrm{S}})^2}{2^{hl}}$. As long as no such collisions occur, each stage-s session identifier uniquely determines one client session, one server session (for non-replayable stages), one pre-shared key (and therefore one peer and identifier owning that key), and one stage-s session key. At this point, the Sound predicate will always be true unless FINALIZE would return 0, so the adversary cannot win by violating Sound.

Phase 2: Ensuring ExplicitAuth

In the second phase of the proof, we change the key-derivation process to avoid sampling pre-shared keys wherever possible, instead replacing keys and MAC tags derived from those pre-shared key by uniformly random strings. We then make the adversary lose if it makes queries that would allow him to detect these changes and bound that probability; in particular we ensure that the adversary does not correctly guess a now-random `ClientFinished` or `ServerFinished` MAC tag. Sessions achieve explicit authentication just after verifying their received `Finished` message; eliminating possible forgeries hence ensures that the $\mathsf{ExplicitAuth}$ predicate cannot be false without FINALIZE returning 0. All changes in this phase apply only to sessions whose pre-shared key has not been corrupted.

Game 4. Our first of six game hops eliminates collisions in the "transcript hash" function $\mathsf{RO}_{\mathsf{Th}}$. We reduce to the collision resistance of $\mathsf{RO}_{\mathsf{Th}}$ and bound this advantage with a birthday bound: $\Pr[\mathsf{Game}_3] - \Pr[\mathsf{Game}_4] \leq \frac{q_{\mathrm{RO}}+6q_{\mathrm{S}}}{2^{hl}}$. (The factor 6 comes from the up to 6 transcript hashes computed in any SEND query.)

Game 5. Our next game forces FINALIZE to return 0 if the adversary guesses any uncorrupted pre-shared key in any random oracle query. Since we assume pre-shared keys are uniformly random, $\Pr[\mathsf{Game}_4] - \Pr[\mathsf{Game}_5] \leq \frac{q_{\mathrm{RO}}\cdot q_{\mathrm{NS}}}{2^{hl}}$.

Games 6 and 7. In our third game hop, we ask log the stage s key computed in any session in a look-up table SKEYS under its session identifier. Sessions whose partners have logged a key can then, in a fourth game hop, copy the key from SKEYS instead of deriving it. Partnered sessions will always derive the same

key as guaranteed by the Sound predicate, so the adversary cannot detect the copying and its advantage does not change. In addition to logging and copying keys, we also log and copy the three MAC tags: *binder*, fin_S, and fin_C using another look-up table TAGS. Since MAC tags do not have associated session identifiers, they are logged under the inputs to RO_{binder}, $\mathsf{RO}_{\mathtt{SF}}$, resp. $\mathsf{RO}_{\mathtt{CF}}$. This technique is inspired by the work of Cohn-Gordon et al. [11].

Game 8. In preparation for the final step in this phase, our fifth game hop eliminates uncorrupted pre-shared keys altogether. We postpone the sampling of the pre-shared key to the RevLongTermKey oracle so 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. Sessions will instead sample these uniformly at random from their respective range. In another look-up table, they log the RO queries they would have made so that these queries can be programmed later if the pre-shared key gets corrupted. Queries to RO before corruption cannot contain the pre-shared key thanks to the previous game, so we do not have to worry about consistency with past queries. We also cannot implement the previous games' check for guessed pre-shared keys in RO queries until these keys are sampled, so we sample new pre-shared keys for all uncorrupted identifiers at the end of the game in the Finalize oracle, then perform the check. The programming of the random oracles is perfectly consistent with their responses in earlier games, so the adversary cannot detect when pre-shared keys are chosen in the game and its advantage does not change.

Game 9. The final game in this phase ensures that either ExplicitAuth = true or Finalize returns 0. In this game, we return 0 from Finalize if any honest session would accept the first explicitly-authenticated stage (stage 5 (CATS) for initiators and stage 8 (RMS) for responders) with an uncorrupted pre-shared key and no honest partner. By the previous game, we established that sessions with uncorrupted pre-shared keys randomly sample their MAC tags, unless they copy a cached result in which case the same computation was made by another session. Thanks to the way we defined our session identifiers, no unpartnered session will copy their MAC tags: the computation of the `ServerFinished` MAC tag contains the hash of the stage-5 *sid* (excluding fin_S); likewise the `ClientFinished` tag contains the hash of the stage-8 *sid*. Since we ruled out hash collisions in the first game of the phase, any two sessions computing the same `ServerFinished` message are stage-5 partners and any two sessions computing the same `ClientFinished` message are stage-8 partners. So any unpartnered session with an uncorrupted pre-shared key has a random MAC tag, and the odds of the adversary guessing such a tag is bounded by $\frac{q_S}{2^h}$. With the prior two games not changing the adversary's advantange, we have $\Pr[\mathsf{Game}_5] - \Pr[\mathsf{Game}_9] \leq \frac{q_S}{2^h}$.

We are now guaranteed that any session accepting the stage that achieves explicit authentication without a corrupted pre-shared key has a partner in that stage. The Sound predicate guarantees that the partner agrees on the peer and pre-shared key identities, which is sufficient to guarantee explicit authentication for all responder sessions. For initiator sessions, we must also note that a partner in stage 5 will become, upon their acceptance, a partner in stages 6 (SATS) and 7

(EMS), whose *sid*s are identical to that of stage 5 apart from their labels. An initiator's stage-5 partner will only accept a `ClientFinished` message identical to the one sent by the initiator, at which point they will become a partner also in stage 8. This ensures that the ExplicitAuth predicate can never be false unless one of the flags introduced in this phase causes Finalize to return 0.

Phase 3: Ensuring the Challenge Bit is Random and Independent

Our goal in the third and last phase is to ensure that all session keys targeted by a Test query are uniformly random and independent of the challenge bit b whenever the Fresh predicate is true. Freshness ensures that no session key can be tested twice or tested and revealed in the same stage either by targeting the same session twice or two partnered sessions. It also handles our three levels of forward secrecy.

We can already establish this for Test queries to sessions in non-forward secret stages 1 (ETS) and 2 (EEMS). These queries violate Fresh unless the sessions' pre-shared keys are never corrupted. Since Game_8, all sessions with uncorrupted pre-shared keys either randomly sample their session keys, or copy random keys from a partner session. If one of these session keys is tested, it cannot have been output by another Test or RevSessionKey query without violating Fresh. Therefore the response to the Test query is a uniformly random string, independent of all other oracle responses and the challenge bit b.

The remaining stages (3–8) have weak forward secrecy 2 until explicit authentication is achieved, then they have full forward secrecy. These stages' keys may be tested even if the session's pre-shared key has been corrupted, so long as there is a contributive partner (or, in the case of full forward secrecy, that the corruption occurred after forward secrecy was achieved). We use one last game hop to ensure these keys are uniformly random when they are tested.

Game 10. In Game_{10}, we cause the Finalize oracle to return 0 if the adversary should ever make a random oracle query containing the Diffie–Hellman secret DHE of an honest partnered session whose pre-shared key was corrupted. Without such a query, all keys derived from a Diffie–Hellman secret sampled uniformly at random by the random oracles.

We bound the probability of this event via a reduction $\mathcal{B}_{\mathrm{DHE}}$ to the strong Diffie–Hellman problem in group $\mathbb{G}$. (Recall that $\mathbb{G}$ has order p and generator g.) In this problem, the adversary $\mathcal{B}_{\mathrm{DHE}}$ gets as input a strong DH challenge $(A = g^a, B = g^b)$ as well as access to an oracle stDH_a for the decisional Diffie–Hellman (DDH) problem with the first argument fixed. Given inputs $C \leftarrow g^c$ and W for any $c \in \mathbb{Z}_p$, $\mathsf{stDH}_a(C, W)$ returns true if and only if $W = g^{ac} = C^a$. The goal of $\mathcal{B}_{\mathrm{DHE}}$ is to submit Z to its Finalize oracle such that $Z = g^{ab}$.

The reduction $\mathcal{B}_{\mathrm{DHE}}$ simulates Game_{10} for the MSKE adversary $\mathcal{A}$. At a high level, it uses rerandomization to embed its strong DH challenge A, resp. B, into the key shares of every initiator session, resp. every partnered responder session. To embed a challenge A in its key share, a session samples a "randomizer" $\tau \xleftarrow{\$} \mathbb{Z}_p$, and sets its key share to $X \leftarrow A \cdot g^{\tau}$. If $\mathcal{A}$ should make an RO query

containing the Diffie–Hellman secret associated with two embedded key shares, the reduction can detect this query with its DDH oracle. It then extracts the solution to its strong DH challenge from the query's DH secret, calls the FINALIZE oracle, and wins its own game.

There are a few subtleties to the reduction, which requires us to extend the technique of CCGJJ [11]. Unlike honest executions of the protocol, the reduction's simulated sessions with embedded key shares do not know their own secret Diffie–Hellman exponents. If their pre-shared keys are never corrupted, this does not matter because session keys and MAC tags are randomly sampled. Corrupted sessions, however, cannot use the random oracles to compute these values as they would in Game_{10}. Instead, $\mathcal{B}_{\mathrm{DHE}}$ samples session keys and MAC tags uniformly at random and uses several look-up tables to program random oracle queries and maintain consistency between sessions.

With this infrastructure in place, the reduction proceeds in the following way. Whenever a partnered session with embedded key share would need its Diffie–Hellman secret, it searches all past RO queries for this secret. It looks up the initiator's stored randomizer τ and the responder's randomizer τ'. Then for each guess $\mathbb{Z}$ in a past RO query, the reduction queries the strong Diffie–Hellman oracle on the responder's key share $\mathtt{SKS}$ and $C \leftarrow \mathbb{Z} \cdot g^{-\tau}$. This query will return true if the adversary correctly guessed the Diffie–Hellman secret; in this case the reduction calls $\textsc{Finalize}(Z \cdot g^{-\tau} \cdot g^{-\tau'})$ and solves its strong DH challenge. Unpartnered sessions do the same thing, except that the responder has no randomizer; in response to the strong DH oracle answering true they hence merely program their session keys instead of calling FINALIZE. We emphasize that for tightness, it is crucial to maintain efficiency during this process. We do so by only checking RO queries whose context matches the hashed protocol transcript; this ensures $\mathcal{B}_{\mathrm{DHE}}$ makes at most 2 stDH_a queries for each RO query.

After a session chooses its session key or MAC tag, it stores the chosen value, its transcript, and all known randomizers in a table $\mathsf{RndList}$. When the reduction answers future RO queries, it will use this table to check if a query contains the Diffie–Hellman secret of an accepted session using the strong DH oracle as above; if so, they program or call FINALIZE in the same way.

This reduction solves the strong Diffie–Hellman problem whenever the adversary makes an RO query containing a partnered session's Diffie–Hellman secret, so for reduction $\mathcal{B}_{\mathrm{DHE}}$ with runtime $t_{\mathcal{B}_{\mathrm{DHE}}}$, we have $\Pr[\mathsf{Game}_9] - \Pr[\mathsf{Game}_{10}] \leq \mathrm{Adv}^{\mathsf{stDH}}_{\mathbb{G}}(t_{\mathcal{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}})$. Davis and Günther gave a bound in the generic group model for the strong DH problem; applying their Theorem 3.3 [14] results in $\Pr[\mathsf{Game}_9] - \Pr[\mathsf{Game}_{10}] \leq \frac{t^2_{\mathcal{B}_{\mathrm{DHE}}}}{p}$.

At this point in the proof, the adversary $\mathcal{A}$ cannot possibly make a RO query that outputs any tested session key of a forward secret (full or wfs2) stage s. If the tested session's pre-shared key is uncorrupted, $\mathcal{A}$ cannot make the query because of Game_5. If the session has a contributive partner in stage s, then from Game_{10}, $\mathcal{A}$ cannot make the query because it contains the Diffie–Hellman secret of a partnered session. If it has accepted with no contributive partner and a corrupted pre-shared key, then by the guarantees we established in Phase 2, the

corruption must have occurred before forward secrecy and explicit authentication were achieved.

As a result, the output of any TEST query (that does not violate Fresh) is a random string, sampled by either a session or the RO oracle independently of all other game variables including the challenge bit b. The adversary therefore has a probability no greater than $\frac{1}{2}$ of winning Game_{10}. Collecting this probability with the other bounds between games in our sequence gives the proof. □

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

We can finally combine the results of Sections 4, 5, 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 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 1 with handshake traffic encryption and 12 random oracles RO_Th, RO_{binder}, …, RO_RMS.
- KE_2: as defined in Theorem 2, with no handshake traffic encryption and 12 random oracles RO_Th, RO_{binder}, …, RO_RMS.

Theorem 1 grants that $\mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_0}(t, q_\mathrm{NS}, q_\mathrm{S}, q_\mathrm{RS}, q_\mathrm{RL}, q_\mathrm{T}, q_\mathrm{RO}) \leq \mathrm{Adv}^{\mathsf{MSKE}}_{\mathsf{KE}_1}(t, q_\mathrm{NS}, q_\mathrm{S}, q_\mathrm{RS}, q_\mathrm{RL}, q_\mathrm{T}, q_\mathrm{RO}) + \frac{2(12q_\mathrm{S}+q_\mathrm{RO})^2}{2^{hl}} + \frac{2q_\mathrm{RO}^2}{2^{hl}} + \frac{8(q_\mathrm{RO}+36q_\mathrm{S})^2}{2^{hl}}$.

Next, we apply Theorem 2, 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 3 then finally and entirely bounds the advantage against the MSKE security of KE_2. Collecting these bounds yields the following overall result for the MSKE security of the TLS 1.3 PSK-(EC)DHE handshake protocol.

Corollary 1. *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* RO_H*. Then,*

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

	Adversary resources						Security bound	
b	t	$\#N$	$\#S$	$\#RO$	Target	Mode	DFGS [21]	Us (Cor. 1, 2)
128	2^{60}	2^{25}	2^{35}	2^{50}	2^{-68}	PSK-only	$\approx 2^{-119}$	$\approx 2^{-152}$
128	2^{80}	2^{35}	2^{55}	2^{70}	2^{-48}	PSK-only	$\approx 2^{-59}$	$\approx 2^{-112}$
128	2^{60}	2^{25}	2^{35}	2^{50}	2^{-68}	`secp256r1`	$\approx 2^{-61}$	$\approx 2^{-132}$
128	2^{80}	2^{35}	2^{55}	2^{70}	2^{-48}	`secp256r1`	1	$\approx 2^{-92}$
128	2^{60}	2^{25}	2^{35}	2^{50}	2^{-68}	`x25519`	$\approx 2^{-57}$	$\approx 2^{-128}$
128	2^{80}	2^{35}	2^{55}	2^{70}	2^{-48}	`x25519`	1	$\approx 2^{-88}$
192	2^{60}	2^{25}	2^{35}	2^{50}	2^{-132}	`secp384r1`	$\approx 2^{-189}$	$\approx 2^{-259}$
192	2^{80}	2^{35}	2^{55}	2^{70}	2^{-112}	`secp384r1`	$\approx 2^{-108}$	$\approx 2^{-219}$
224	2^{60}	2^{25}	2^{35}	2^{50}	2^{-164}	`x448`	$\approx 2^{-200}$	$\approx 2^{-280}$
224	2^{80}	2^{35}	2^{55}	2^{70}	2^{-144}	`x448`	$\approx 2^{-110}$	$\approx 2^{-240}$
256	2^{60}	2^{25}	2^{35}	2^{50}	2^{-196}	`secp521r1`	$\approx 2^{-200}$	$\approx 2^{-280}$
256	2^{80}	2^{35}	2^{55}	2^{70}	2^{-176}	`secp521r1`	$\approx 2^{-110}$	$\approx 2^{-240}$

Table 1. Exemplary 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 handshake protocols. Numbers based on the prior bounds by Dowling et al. [21] and our bounds for PSK-(EC)DHE and PSK-only (in Corollaries 1 resp. 2). "Target" indicates the maximal advantage $t/2^b$ tolerable for a given bound on t when aiming for the respective curve's (or hash function's, in case of PSK-only mode) bit security level b; entries in green-shaded cells meet that target. Mode indicates PSK-only mode (with `SHA384`) or otherwise PSK-(EC)DHE mode with the given curve `secp256r1`, `x25519` (with `SHA256`), or `secp384r1`, `x448`, `secp521r1` (with `SHA384`).

$$+ \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2 + 2q_{\mathrm{RO}}^2 + 8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.$$

For the PSK-only mode, we obtain a similar bound, naturally omitting the strong Diffie–Hellman and group-element collision terms. Due to space restrictions, we only state the final PSK-only bound here and defer further details to the full version [13].

Corollary 2. *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* $\mathsf{RO_H}$*. Then,*

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

7 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 [14]. Table 1 shows exemplary concrete advantages; the full range of evaluated parameters is given in extended tables in the full version [13], 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. [21] 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.

References

1. Arfaoui, G., Bultel, X., Fouque, P.A., Nedelcu, A., Onete, C.: The privacy of the TLS 1.3 protocol. PoPETs **2019**(4), 190–210 (Oct 2019). https://doi.org/10.2478/popets-2019-0065 18
2. Avoine, G., Canard, S., Ferreira, L.: Symmetric-key authenticated key exchange (SAKE) with perfect forward secrecy. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 199–224. Springer, Heidelberg (Feb 2020). https://doi.org/10.1007/978-3-030-40186-3_10 2
3. Bader, C., Hofheinz, D., Jager, T., Kiltz, E., Li, Y.: Tightly-secure authenticated key exchange. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015, Part I. LNCS, vol. 9014, pp. 629–658. Springer, Heidelberg (Mar 2015). https://doi.org/10.1007/978-3-662-46494-6_26 4
4. Bader, C., Jager, T., Li, Y., Schäge, S.: On the impossibility of tight cryptographic reductions. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 273–304. Springer, Heidelberg (May 2016). https://doi.org/10.1007/978-3-662-49896-5_10 4
5. Bellare, M., Davis, H., Günther, F.: Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part II. LNCS, vol. 12106, pp. 3–32. Springer, Heidelberg (May 2020). https://doi.org/10.1007/978-3-030-45724-2_1 4, 14, 15, 16
6. Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., Zanella-Béguelin, S.: Downgrade resilience in key-exchange protocols. In: 2016 IEEE Symposium on Security and Privacy. pp. 506–525. IEEE Computer Society Press (May 2016). https://doi.org/10.1109/SP.2016.37 5
7. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zanella Béguelin, S.: Proving the TLS handshake secure (as it is). In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 235–255. Springer, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-662-44381-1_14 5

8. Boyd, C., Cremers, C., Feltz, M., Paterson, K.G., Poettering, B., Stebila, D.: ASICS: Authenticated key exchange security incorporating certification systems. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 381–399. Springer, Heidelberg (Sep 2013). https://doi.org/10.1007/978-3-642-40203-6_22 11
9. Boyd, C., Davies, G.T., de Kock, B., Gellert, K., Jager, T., Millerjord, L.: 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 2
10. Brzuska, C., Delignat-Lavaud, A., Egger, C., Fournet, C., Kohbrok, K., Kohlweiss, M.: Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Report 2021/467 (2021), https://eprint.iacr.org/2021/467 5
11. Cohn-Gordon, K., Cremers, C., Gjøsteen, K., Jacobsen, H., Jager, T.: Highly efficient key exchange protocols with optimal tightness. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 767–797. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26954-8_25 4, 6, 22, 24
12. Cremers, C., Horvat, M., Scott, S., van der Merwe, T.: Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In: 2016 IEEE Symposium on Security and Privacy. pp. 470–485. IEEE Computer Society Press (May 2016). https://doi.org/10.1109/SP.2016.35 6
13. Davis, H., Diemert, D., Günther, F., Jager, T.: On the Concrete Security of TLS 1.3 PSK Mode. Cryptology ePrint Archive (2022), https://eprint.iacr.org/2022/246 4, 8, 9, 10, 14, 15, 16, 17, 18, 20, 26, 27
14. Davis, H., Günther, F.: 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) 3, 4, 5, 6, 9, 14, 24, 27
15. de Saint Guilhem, C., Fischlin, M., Warinschi, B.: Authentication in key-exchange: Definitions, relations and composition. In: Jia, L., Küsters, R. (eds.) CSF 2020 Computer Security Foundations Symposium. pp. 288–303. IEEE Computer Society Press (2020). https://doi.org/10.1109/CSF49147.2020.00028 12
16. Diemert, D., Gellert, K., Jager, T., Lyu, L.: More efficient digital signatures with tight multi-user security. In: Garay, J. (ed.) PKC 2021, Part II. LNCS, vol. 12711, pp. 1–31. Springer, Heidelberg (May 2021). https://doi.org/10.1007/978-3-030-75248-4_1 4
17. Diemert, D., Jager, T.: On the tight security of TLS 1.3: Theoretically sound cryptographic parameters for real-world deployments. Journal of Cryptology **34**(3), 30 (Jul 2021). https://doi.org/10.1007/s00145-021-09388-x 3, 4, 5, 6, 14
18. Dodis, Y., Ristenpart, T., Steinberger, J.P., Tessaro, S.: To hash or not to hash again? (In)differentiability results for H^2 and HMAC. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 348–366. Springer, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_21 16
19. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 1197–1210. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813653 3, 5, 6
20. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: 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 3, 6

21. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol. Journal of Cryptology **34**(4), 37 (Oct 2021). https://doi.org/10.1007/s00145-021-09384-1 3, 5, 6, 9, 12, 26, 27
22. Dowling, B., Stebila, D.: Modelling ciphersuite and version negotiation in the TLS protocol. In: Foo, E., Stebila, D. (eds.) ACISP 15. LNCS, vol. 9144, pp. 270–288. Springer, Heidelberg (Jun / Jul 2015). https://doi.org/10.1007/978-3-319-19962-7_16 5, 6
23. Drucker, N., Gueron, S.: Selfie: reflections on TLS 1.3 with PSK. Journal of Cryptology **34**(3), 27 (Jul 2021). https://doi.org/10.1007/s00145-021-09387-y 10
24. Fischlin, M., Günther, F.: Multi-stage key exchange and the case of Google's QUIC protocol. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 2014. pp. 1193–1204. ACM Press (Nov 2014). https://doi.org/10.1145/2660267.2660308 5, 6
25. Fischlin, M., Günther, F.: 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. pp. 60–75. IEEE (Apr 2017) 3, 5, 6
26. Fischlin, M., Günther, F., Schmidt, B., Warinschi, B.: Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In: 2016 IEEE Symposium on Security and Privacy. pp. 452–469. IEEE Computer Society Press (May 2016). https://doi.org/10.1109/SP.2016.34 12
27. Giesen, F., Kohlar, F., Stebila, D.: On the security of TLS renegotiation. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 387–398. ACM Press (Nov 2013). https://doi.org/10.1145/2508859.2516694 5
28. Gjøsteen, K., Jager, T.: Practical and tightly-secure digital signatures and authenticated key exchange. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 95–125. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0_4 4, 6
29. Günther, C.G.: An identity-based key-exchange protocol. In: Quisquater, J.J., Vandewalle, J. (eds.) EUROCRYPT'89. LNCS, vol. 434, pp. 29–37. Springer, Heidelberg (Apr 1990). https://doi.org/10.1007/3-540-46885-4_5 2
30. Günther, F.: Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols. Ph.D. thesis, Technische Universität Darmstadt, Darmstadt, Germany (2018), http://tuprints.ulb.tu-darmstadt.de/7162/ 6
31. Han, S., Jager, T., Kiltz, E., Liu, S., Pan, J., Riepel, D., Schäge, S.: Authenticated key exchange and signatures with tight security in the standard model. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part IV. LNCS, vol. 12828, pp. 670–700. Springer, Heidelberg, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84259-8_23 4, 6
32. Jager, T., Kiltz, E., Riepel, D., Schäge, S.: Tightly-secure authenticated key exchange, revisited. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 117–146. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-77870-5_5 6
33. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 273–293. Springer, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_17 5, 6
34. Jager, T., Schwenk, J., Somorovsky, J.: On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 1185–1196. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813657 6

35. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational) (Feb 1997). https://doi.org/10.17487/RFC2104, https://www.rfc-editor.org/rfc/rfc2104.txt, updated by RFC 6151 9
36. Krawczyk, H., Eronen, P.: HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational) (May 2010). https://doi.org/10.17487/RFC5869, https://www.rfc-editor.org/rfc/rfc5869.txt 7, 9
37. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. Cryptology ePrint Archive, Report 2005/176 (2005), https://eprint.iacr.org/2005/176 12
38. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_34 7
39. Krawczyk, H.: A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 1438–1450. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978325 6
40. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: A systematic analysis. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 429–448. Springer, Heidelberg (Aug 2013). https://doi.org/10.1007/978-3-642-40041-4_24 5, 6
41. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: A systematic analysis. Cryptology ePrint Archive, Report 2013/339 (2013), https://eprint.iacr.org/2013/339 5
42. Krawczyk, H., Wee, H.: The OPTLS protocol and TLS 1.3. In: 2016 IEEE European Symposium on Security and Privacy. pp. 81–96. IEEE (Mar 2016). https://doi.org/10.1109/EuroSP.2016.18 3
43. Langley, A., Hamburg, M., Turner, S.: Elliptic Curves for Security. RFC 7748 (Informational) (Jan 2016). https://doi.org/10.17487/RFC7748, https://www.rfc-editor.org/rfc/rfc7748.txt 20
44. Li, Y., Schäge, S., Yang, Z., Kohlar, F., Schwenk, J.: On the security of the pre-shared key ciphersuites of TLS. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 669–684. Springer, Heidelberg (Mar 2014). https://doi.org/10.1007/978-3-642-54631-0_38 5
45. Liu, X., Liu, S., Gu, D., Weng, J.: Two-pass authenticated key exchange with explicit authentication and tight security. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part II. LNCS, vol. 12492, pp. 785–814. Springer, Heidelberg (Dec 2020). https://doi.org/10.1007/978-3-030-64834-3_27 6
46. Maurer, U.M., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 21–39. Springer, Heidelberg (Feb 2004). https://doi.org/10.1007/978-3-540-24638-1_2 4, 14, 15, 16
47. National Institute of Standards and Technology: FIPS PUB 180-4: Secure Hash Standard (SHS) (2012) 7
48. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard) (Aug 2018). https://doi.org/10.17487/RFC8446, https://www.rfc-editor.org/rfc/rfc8446.txt 2, 3, 20
49. Schwabe, P., Stebila, D., Wiggers, T.: Post-quantum TLS without handshake signatures. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1461–1480. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3423350 9, 12