More efficient post-quantum KEMTLS with pre-distributed public keys

Online Version

Peter Schwabe[1,2], Douglas Stebila[3], and Thom Wiggers[2]

[1] MPI-SP, Bochum, Germany
[2] Radboud University, Nijmegen, The Netherlands
[3] University of Waterloo, Waterloo, Canada
peter@cryptojedi.org, dstebila@uwaterloo.ca, thom@thomwiggers.nl

Abstract While server-only authentication with certificates is the most widely used mode of operation for the Transport Layer Security (TLS) protocol on the world wide web, there are many applications where TLS is used in a different way or with different constraints. For example, embedded Internet-of-Things clients may have a server certificate pre-programmed and be highly constrained in terms of communication bandwidth or computation power. As post-quantum algorithms have a wider range of performance trade-offs, designs other than traditional "signed-key-exchange" may be worthwhile. The KEMTLS protocol, presented at ACM CCS 2020, uses key encapsulation mechanisms (KEMs) rather than signatures for authentication in the TLS 1.3 handshake, a benefit since most post-quantum KEMs are more efficient than PQ signatures. However, KEMTLS has some drawbacks, especially in the client authentication scenario which requires a full additional roundtrip.

We explore how the situation changes with *pre-distributed public keys*, which may be viable in many scenarios, for example pre-installed public keys in apps, on embedded devices, cached public keys, or keys distributed out of band. Our variant of KEMTLS with pre-distributed keys, called KEMTLS-PDK, is more efficient in terms of both bandwidth and computation compared to post-quantum signed-KEM TLS (even cached public keys), and has a smaller trusted code base. When client authentication is used, KEMTLS-PDK is more bandwidth efficient than KEMTLS yet can complete client authentication in one fewer round trips, and has stronger authentication properties. Interestingly, using pre-distributed keys in KEMTLS-PDK changes the landscape on suitability of PQ algorithms: schemes where public keys are larger than ciphertexts/signatures (such as Classic McEliece and Rainbow) can be viable, and the differences between some lattice-based schemes is reduced. We also discuss how using pre-distributed public keys provides privacy benefits compared to pre-shared symmetric keys in TLS.

Keywords: Post-Quantum Cryptography · Transport Layer Security

1   Introduction

The Transport Layer Protocol (TLS) is among the most-used secure channel protocols. In August 2018, the most recent version was standardized as TLS 1.3 [43]. TLS 1.3 uses an (elliptic curve) Diffie–Hellman key exchange to establish an ephemeral shared secret with forward secrecy. Server (and optionally client) authentication is provided by digital signatures. Long-term signature public keys are exchanged in certificates during the handshake. The most commonly used signature algorithm is RSA, although elliptic curve signatures are also supported.

Migrating to post-quantum TLS. To protect against quantum adversaries, effort has been made to move the TLS handshake towards post-quantum cryptography. The focus has largely been on upgrading the key exchange to post-quantum security. In [9], Bos, Costello, Naehrig and Stebila showed how to replace Diffie–Hellman by lattice-based key agreement in TLS 1.3. The lattice-based scheme was improved in the NewHope proposal [2], which was used in the first real-world post-quantum TLS experiment by Google in 2016 [10]. A second, more wide-scale, post-quantum TLS experiment by Google and Cloudflare has been running since 2019 [34, 36]. Post-quantum authentication in TLS is widely believed to be less urgent, as attacks against authentication cannot be mounted retroactively. However, several works also investigated the use of post-quantum signature schemes and certificates in TLS [6, 7, 50] by dropping in replacements of post-quantum primitives into the existing TLS 1.3 handshake and PKI infrastructure.

KEMTLS [48] is a recent proposal that makes more radical changes to the TLS 1.3 handshake. Instead of Diffie–Hellman and signatures, KEMTLS uses key-encapsulation mechanisms (KEMs) not just for confidentiality, but also for authentication. The main motivation for this design is that most post-quantum KEMs are much more efficient, both computationally and in terms of bandwidth requirements, than post-quantum signature schemes. Additional advantages are a smaller trusted code base and offline deniability. We give a high-level overview of KEMTLS in comparison to the TLS 1.3 handshake in Appendix A.

1.1   Pre-distributed keys

Both TLS 1.3 and KEMTLS assume that the client does not know the server's long-term public-key when sending the `ClientHello` message; the certificate is transmitted as part of the handshake, even if the client already knows the public key. We refer to the scenario when a client already knows the server's public key as the *pre-distributed-key* or *cached-key* scenario. This occurs, for example, when web browsers cache certificates of frequently accessed servers; when mobile apps store certificates of the limited number of servers they connect to; when TLS is used by IoT devices that only ever connect to one or a handful of servers and have those certificates pre-installed; or when certificates have been distributed out of band, for example, through DNS [29].

In fact, the TLS cached information extension [46] allows the client to inform the server that it already knows certain certificates so they need not be

(a) Unilaterally authenticated

Client		Server
Knows pk_S		static (KEM_s): pk_S, sk_S

$(pk_e, sk_e) \leftarrow KEM_e.Keygen()$
$(ss_S, ct_S) \leftarrow KEM_e.Encapsulate(pk_S)$

$\xrightarrow{\quad pk_e, ct_S \quad}$

$ss_S \leftarrow KEM_s.Decapsulate(ct_S, sk_S)$
$(ss_e, ct_e) \leftarrow KEM_e.Encapsulate(pk_e)$

$\xleftarrow{\quad ct_e \quad}$

$ss_e \leftarrow KEM_e.Decapsulate(ct_e, sk_e)$

$K, K', K'', K''' \leftarrow KDF(ss_S \| ss_e)$
$AEAD_K(\text{key confirmation})$

$\xleftarrow{\quad AEAD_{K'}(\text{application data}) \quad}$
$\xrightarrow{\quad AEAD_{K''}(\text{key confirmation}) \quad}$
$\xrightarrow{\quad AEAD_{K'''}(\text{application data}) \quad}$

(b) With proactive client authentication

Client		Server
static (KEM_c): pk_C, sk_C		static (KEM_s): pk_S, sk_S
Knows pk_S		

$(pk_e, sk_e) \leftarrow KEM_e.Keygen()$
$(ss_S, ct_S) \leftarrow KEM_e.Encapsulate(pk_S)$
$K_S \leftarrow KDF(ss_S)$
$pk_e, ct_S, AEAD_{K_S}(\text{cert } [pk_C])$

$ss_S \leftarrow KEM_s.Decapsulate(ct_S, sk_S)$
$(ss_e, ct_e) \leftarrow KEM_e.Encapsulate(pk_e)$
$(ss_C, ct_C) \leftarrow KEM_c.Encapsulate(pk_C)$
$\xleftarrow{\quad ct_e \quad}$

$ss_e \leftarrow KEM_e.Decapsulate(ct_e, sk_e)$
$K_1 \leftarrow KDF(ss_S \| ss_e)$
$AEAD_{K_1}(ct_C)$

$ss_C \leftarrow KEM_c.Decapsulate(ct_C, sk_C)$

$K_2, K_2', K_2'', K_2''' \leftarrow KDF(ss_S \| ss_e \| ss_C)$
$AEAD_{K_2}(\text{key confirmation})$

$\xleftarrow{\quad AEAD_{K_2'}(\text{application data}) \quad}$
$\xrightarrow{\quad AEAD_{K_2''}(\text{key confirmation}) \quad}$
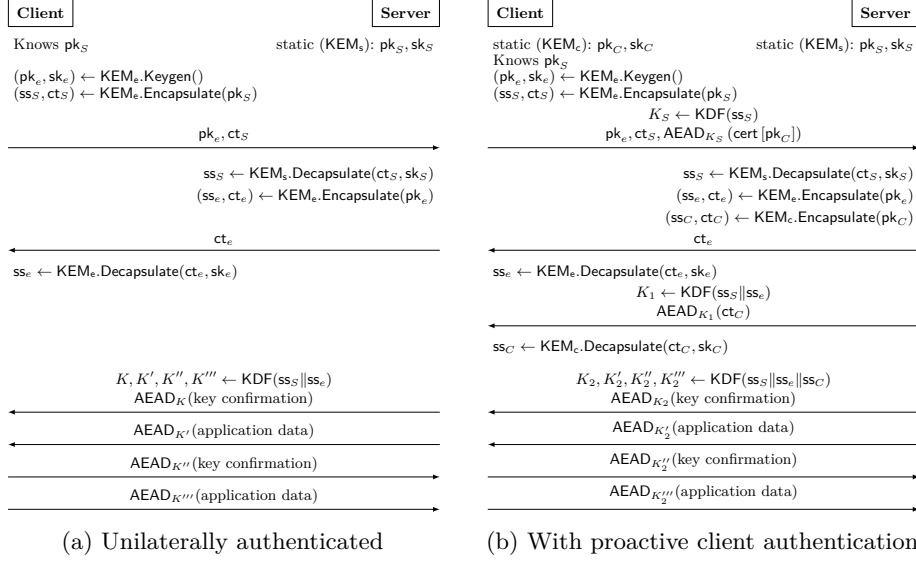$\xrightarrow{\quad AEAD_{K_2'''}(\text{application data}) \quad}$

Figure 1: Overview of KEMTLS variants with pre-distributed public keys

transmitted. However, this RFC is not widely implemented, perhaps because (pre-quantum) certificates are fairly short and thus bandwidth savings are limited.

Contributions of this paper. In this paper we investigate how this situation changes with the move to post-quantum cryptography, both when caching signature-based certificates in the TLS 1.3 handshake, but more importantly when using KEM-based authentication as used in KEMTLS. More specifically,

- we introduce KEMTLS-PDK, a variant of KEMTLS that makes use of pre-distributed keys for earlier authentication in the protocol flow;
- we describe KEMTLS-PDK with proactive client authentication and show that the benefits of earlier authentication are even more significant;
- we analyse the security properties of KEMTLS-PDK (a complete proof in the standard model is in the full version of the paper, available online);
- we implement KEMTLS-PDK in the Rustls library using different instantiations of the KEMs with NIST PQC round-3 candidates; and
- we evaluate the performance of KEMTLS-PDK in comparison to TLS 1.3 instantiated with different post-quantum primitives and pre-distributed (cached) server certificates, and in comparison to KEMTLS.

We give a sketch of KEMTLS-PDK in Fig. 1a. The central property to observe is that, like in TLS 1.3, but unlike in KEMTLS, the first message from the server serves as key confirmation. This means that in this variant on KEMTLS, like in TLS 1.3, the server is explicitly authenticated after a single round trip. Also like TLS 1.3, the server can send data to the client first. The version of KEMTLS-PDK with proactive client authentication is shown in Fig. 1b.

Table 1: Summary of performance characteristics of KEMTLS, signed-KEM TLS 1.3 with cached server certificate, and KEMTLS-PDK

	KEMTLS	Cached TLS	KEMTLS-PDK
Unilaterally authenticated			
Round trips until client receives response data	3	3	3
Size (bytes) of public key crypto objects transmitted:			
• Minimum PQ	932	499	561
• Module-LWE/Module-SIS (Kyber, Dilithium)	5,556	3,988	2,336
• NTRU-based (NTRU, Falcon)	3,486	2,088	2,144
Mutually authenticated			
Round trips until client receives response data	4	3	3
Size (bytes) of public key crypto objects transmitted:			
• Minimum PQ	1,431	2,152	1,060
• MLWE/MSIS	9,554	10,140	6,324
• NTRU	5,574	4,365	4,185

We give a brief summary how this affects performance in Table 1, for a variety of post-quantum algorithm combinations. We see that client authentication in KEMTLS-PDK is just as efficient in terms of round trips as in TLS 1.3. However, in terms of bandwidth requirements, KEMTLS-PDK is *more* efficient than TLS 1.3 with cached certificates (Cached TLS) for most instantiations. We will discuss this in more detail in Section 5, but the unilaterally authenticated "Minimum PQ" instantiation actually shows another interesting effect of considering TLS with cached keys: KEMs and signature schemes with small ciphertexts/signatures, such as Classic McEliece [1] or Rainbow [19], not only become viable but are the most efficient instantiation.

Related work. The KEMTLS and KEMTLS-PDK proposals follow a long line of work on authenticated key exchange (AKE) that started from early works by Bellare and Rogaway [4] and Canetti and Krawczyk [13]. Many earlier works already considered AKE protocols that do not use signatures for authentication; often authentication is then obtained from long-term Diffie–Hellman (DH) keys. See, for example [30, 35, 38, 41]. The approach of constructing AKE with long-term DH keys for authentication has been considered for TLS in the OPTLS proposal by Krawczyk and Wee in [33], and in a subsequent IETF draft [44]. Unfortunately, there is no efficient post-quantum instantiation for non-interactive key exchange required by those DH-based constructions, which means that the closest proposals to KEMTLS and KEMTLS-PDK are generic AKE constructions using KEMs or public-key encryption schemes, such as the protocols described in [17, 25, 26]. While [17] mentions as an application (post-quantum) TLS, none of these earlier works on KEM-based AKE actually present an integration (or implementation) as part of the TLS handshake. This means that they work in the typical setting for AKE that all long-term keys are distributed beforehand –

not just the server's keys as in KEMTLS-PDK. Also those earlier works do not present concrete TLS handshake performance results.

Availability of software. Our implementation, measurement software, and data are available at https://thomwiggers.nl/publication/kemtlspdk/ under permissive open-source licenses.

2   Preliminaries

Notation. The TLS protocol has named messages, such as ClientHello, which we abbreviate like CH. Encrypted messages are written as $\{\text{Message}\}_{key}$. We write the transcript constructed by concatenating a sequence of TLS messages like CH...SF. $\emptyset$ denotes an empty value.

Symmetric cryptography. We rely on standard definitions such as collision-resistant hash functions, authenticated encryption, and pseudorandom functions. In the proof we use message-authentication codes with existential unforgeability under chosen message attacks (EUF-CMA). The key schedule is based on HKDF [31, 32], which consists of two functions, HKDF.Extract and HKDF.Expand. HKDF.Extract takes a random *salt* and *input keying material*; the output is a *pseudorandom key* that is fed to HKDF.Expand along with *context*, to derive keys of specified length for use in the handshake. The key schedule in TLS sets up a chain of these operations. It passes along the internal state via the salt argument to HKDF.Extract. New shared secrets are incorporated via the other argument. The context given to HKDF.Expand is provided as an operation-specific constant value and the current hash of the transcript. For ease of presentation, we will write these as if they are separate arguments and omit the desired output length. Our security analysis of KEMTLS-PDK relies on HKDF.Expand being a *dual PRF* (a PRF in either of its two arguments, salt and input keying material).

Key encapsulation mechanisms. A key encapsulation mechanism (KEM) is an asymmetric primitive that abstracts a basic key exchange and is a focus of the NIST post-quantum standardization project. A KEM consists of: a key generation algorithm KEM.Keygen() which generates a public and private keypair $(\mathsf{pk}, \mathsf{sk})$; an encapsulation algorithm KEM.Encapsulate(pk) which generates a shared secret ss in a shared secret space $\mathcal{K}$ and ciphertext (encapsulation) ct against a given public key pk; and a decapsulation algorithm KEM.Decapsulate(ct, sk) which decapsulates to obtain the shared secret ss'. Decapsulation might fail; in a δ-correct scheme, $\mathsf{ss}' = \mathsf{ss}$ with probability at least $1 - \delta$. Our security analysis of KEMTLS-PDK relies on standard IND-CCA-security of the KEMs used for client and server authentication, and IND-1CCA-security (i.e., IND-CCA restricted to a single decapsulation query) for the KEM used for ephemeral key exchange.

3   KEMTLS with pre-distributed long-term keys

Even though one of the strengths of the TLS protocol is its ability to establish a secure channel with a previously unknown party, it is very often not the case

that the communicating party is completely unknown. TLS 1.3's pre-shared key mechanism can be used with session tickets to enable fast resumption after an initial full handshake [43, Fig. 3]. These mechanisms rely mostly on symmetric cryptography, although TLS 1.3 allows an optional additional ephemeral key exchange in resumption for forward secrecy. There is nothing precluding the use of these mechanisms, including the "0-RTT" client-to-server data flow in the resumption message in KEMTLS.

However, because TLS 1.3 resumption relies on symmetric cryptography, it is not very flexible. The security properties of a resumed session are tied to the previous session. This includes, e.g., if the session was mutually authenticated. For these reasons session tickets expire quickly, after at most 7 days [43, Sec. 4.6.1]. There are also privacy issues, as the tickets, which are opaque to the client, might contain tracking information. To prevent such tracking, Sy et al. [52] even suggested limiting session lifetime to only 10 *minutes*.

Because externally distributed pre-shared keys are symmetric, we quickly run into concerns there as well. If clients have a common installation profile and share keys, when a single client is compromised there will be no security for any client anymore. A client that also acts as a server also needs to use different keys in each role to prevent the Selfie attack [23]. This means we need a key for each client and server pair, quickly turning this into a key-management nightmare.

In our proposed KEMTLS-PDK, we employ a more flexible approach by distributing a server's long-term KEM public key instead of a symmetric key. A detailed protocol flow diagram of KEMTLS-PDK is given in Fig. 2.

Like in KEMTLS, the client encapsulates to the server's long-term KEM public key pk_S, obtaining a ciphertext ct_S and a shared secret ss_S. However, as we assume that the client already has pk_S, it can do this *at the start* of the connection and send ct_S in a `ClientHello` extension. We plug ss_S into the key derivation schedule at the earliest possible stage when deriving the Early Secret ES. Immediately, the only server who can read a message encrypted under a key derived from ES is the server that has access to sk_S; this means such keys are *implicitly authenticated*. For forward secrecy we also send an ephemeral public key pk_e in the `ClientHello` message.

Except for the additional extension transmitting ct_S, the CH message is the same as in KEMTLS. This allows a handshake to fall back to the regular KEMTLS handshake protocol, e.g., if the client has an out-of-date certificate.

The server replies with the encapsulation ct_e of ephemeral shared secret ss_e in the `ServerHello` message. It also indicates in an extension that it has accepted ciphertext ct_S and is proceeding with KEMTLS-PDK. Then it proceeds in a similar fashion as the original TLS 1.3 handshake. The server derives HS from ES and ss_e and sends the `EncryptedExtensions` message encrypted under a key derived from HS. It then immediately finishes its part the handshake by sending a MAC over the message transcript in the `ServerFinished` message. This confirms the server's view of the handshake to the client and explicitly authenticates the server. The server can now start sending application data. The client follows up
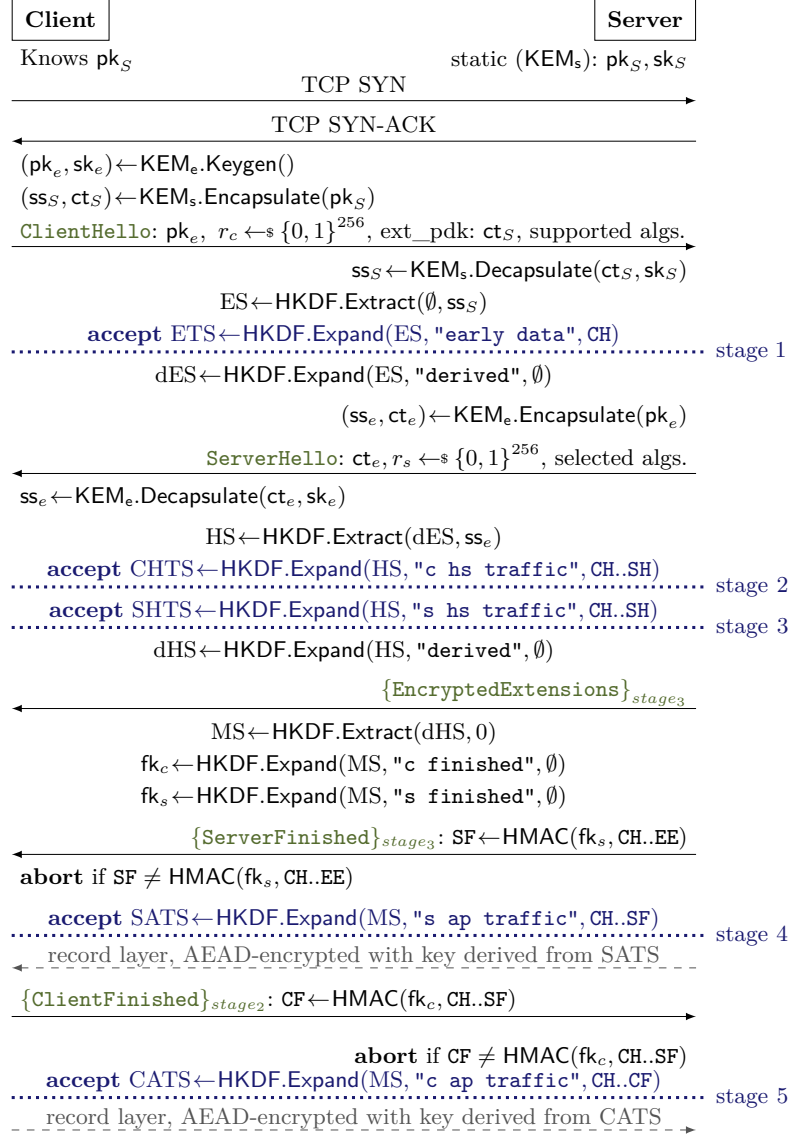
Client	**Server**
Knows pk_S	static ($\mathsf{KEM_s}$): $\mathsf{pk}_S, \mathsf{sk}_S$

$$\xrightarrow{\hspace{3cm}\text{TCP SYN}\hspace{3cm}}$$

$$\xleftarrow{\hspace{3cm}\text{TCP SYN-ACK}\hspace{3cm}}$$

$(\mathsf{pk}_e, \mathsf{sk}_e) \leftarrow \mathsf{KEM_e.Keygen}()$

$(\mathsf{ss}_S, \mathsf{ct}_S) \leftarrow \mathsf{KEM_s.Encapsulate}(\mathsf{pk}_S)$

$$\xrightarrow{\texttt{ClientHello}: \mathsf{pk}_e,\ r_c \leftarrow\!\!{}_\$ \{0,1\}^{256},\ \texttt{ext\_pdk}: \mathsf{ct}_S,\ \text{supported algs.}}$$

$\mathsf{ss}_S \leftarrow \mathsf{KEM_s.Decapsulate}(\mathsf{ct}_S, \mathsf{sk}_S)$

$\mathrm{ES} \leftarrow \mathsf{HKDF.Extract}(\emptyset, \mathsf{ss}_S)$

accept $\mathrm{ETS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{ES}, \texttt{"early data"}, \mathrm{CH})$ $\cdots$ stage 1

$\mathrm{dES} \leftarrow \mathsf{HKDF.Expand}(\mathrm{ES}, \texttt{"derived"}, \emptyset)$

$(\mathsf{ss}_e, \mathsf{ct}_e) \leftarrow \mathsf{KEM_e.Encapsulate}(\mathsf{pk}_e)$

$$\xleftarrow{\texttt{ServerHello}: \mathsf{ct}_e,\ r_s \leftarrow\!\!{}_\$ \{0,1\}^{256},\ \text{selected algs.}}$$

$\mathsf{ss}_e \leftarrow \mathsf{KEM_e.Decapsulate}(\mathsf{ct}_e, \mathsf{sk}_e)$

$\mathrm{HS} \leftarrow \mathsf{HKDF.Extract}(\mathrm{dES}, \mathsf{ss}_e)$

accept $\mathrm{CHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \texttt{"c hs traffic"}, \mathrm{CH..SH})$ $\cdots$ stage 2

accept $\mathrm{SHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \texttt{"s hs traffic"}, \mathrm{CH..SH})$ $\cdots$ stage 3

$\mathrm{dHS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \texttt{"derived"}, \emptyset)$

$$\xleftarrow{\{\texttt{EncryptedExtensions}\}_{stage_3}}$$

$\mathrm{MS} \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}, 0)$

$\mathsf{fk}_c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"c finished"}, \emptyset)$

$\mathsf{fk}_s \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"s finished"}, \emptyset)$

$$\xleftarrow{\{\texttt{ServerFinished}\}_{stage_3}: \mathsf{SF} \leftarrow \mathsf{HMAC}(\mathsf{fk}_s, \mathrm{CH..EE})}$$

abort if $\mathsf{SF} \neq \mathsf{HMAC}(\mathsf{fk}_s, \mathrm{CH..EE})$

accept $\mathrm{SATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"s ap traffic"}, \mathrm{CH..SF})$ $\cdots$ stage 4

$\xleftarrow{\hspace{0.5cm}}$ record layer, AEAD-encrypted with key derived from SATS

$$\xrightarrow{\{\texttt{ClientFinished}\}_{stage_2}: \mathsf{CF} \leftarrow \mathsf{HMAC}(\mathsf{fk}_c, \mathrm{CH..SF})}$$

abort if $\mathsf{CF} \neq \mathsf{HMAC}(\mathsf{fk}_c, \mathrm{CH..SF})$

accept $\mathrm{CATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"c ap traffic"}, \mathrm{CH..CF})$ $\cdots$ stage 5

record layer, AEAD-encrypted with key derived from CATS $\xrightarrow{\hspace{0.5cm}}$

Figure 2: The KEMTLS handshake with pre-distributed keys including key derivation schedule

by also confirming its view of the handshake in a `ClientFinished` message. This means the client is now ready to communicate as well.

3.1   Proactive client authentication

In some applications, such as in a VPN, the client already knows that the server will require mutual authentication. This means that a client can *proactively* authenticate by sending its certificate as early in the handshake as possible, and in particular before the server requests the certificate. For privacy reasons, client authentication in TLS requires that we verify the identity of the server and send the certificate encrypted [43, Sec. E.1.2]. Performing client authentication in KEMTLS thus requires a full additional round-trip: we can only send the client certificate after authenticating the server and the server cannot send the ciphertext before it receives pk_C.

In KEMTLS-PDK the client already posesses the server's long-term public key. We can use the shared secret obtained from encapsulating to the corresponding long-term key to send a client certificate along in the `ClientHello` message. This gives us mutual authentication within a single round-trip. The server supplies the challenge ciphertext ct_C to the client and derives the confirmation and traffic keys from ss_e, ss_S, and ss_C. At this point the server can start sending application data to the client. The client is implicitly authenticated, as they have not yet confirmed that they derived the same keys. As the keys are derived from ss_C, only the client who possesses sk_C can read these messages. To finish the handshake the client sends its own key confirmation message before proceeding to the application traffic. KEMTLS-PDK with mutual authentication is shown in Fig. 3.

4   Security analysis

As KEMTLS-PDK is an authenticated-key-exchange protocol, the main security property it aims for is that keys established should be indistinguishable from random keys, however there are many subtleties that arise in KEMTLS-PDK. In this section we describe in greater detail the specific security properties that KEMTLS-PDK achieves.

Security model. Following the approach of Dowling et al. [21] for the analysis of TLS 1.3, we model KEMTLS-PDK as a multi-stage key-agreement protocol [24], where each session has several *stages* in each of which a shared secret key is established. This model is an adaptation of the Bellare–Rogaway security model for authenticated key exchange [4]. The formal model appears in Appendix B.1; we describe it briefly here.

Each party (client or server) has a long-term public-key/secret-key pair, and we assume there exists a public-key infrastructure for certifying these public keys. Each party can run multiple instances of the protocol simultaneously or in parallel, each of which is called a *session*. During each party's execution of a session, it maintains a variety of state variables, tracking the configuration of the session, status of execution, and intermediate values of the protocol itself.
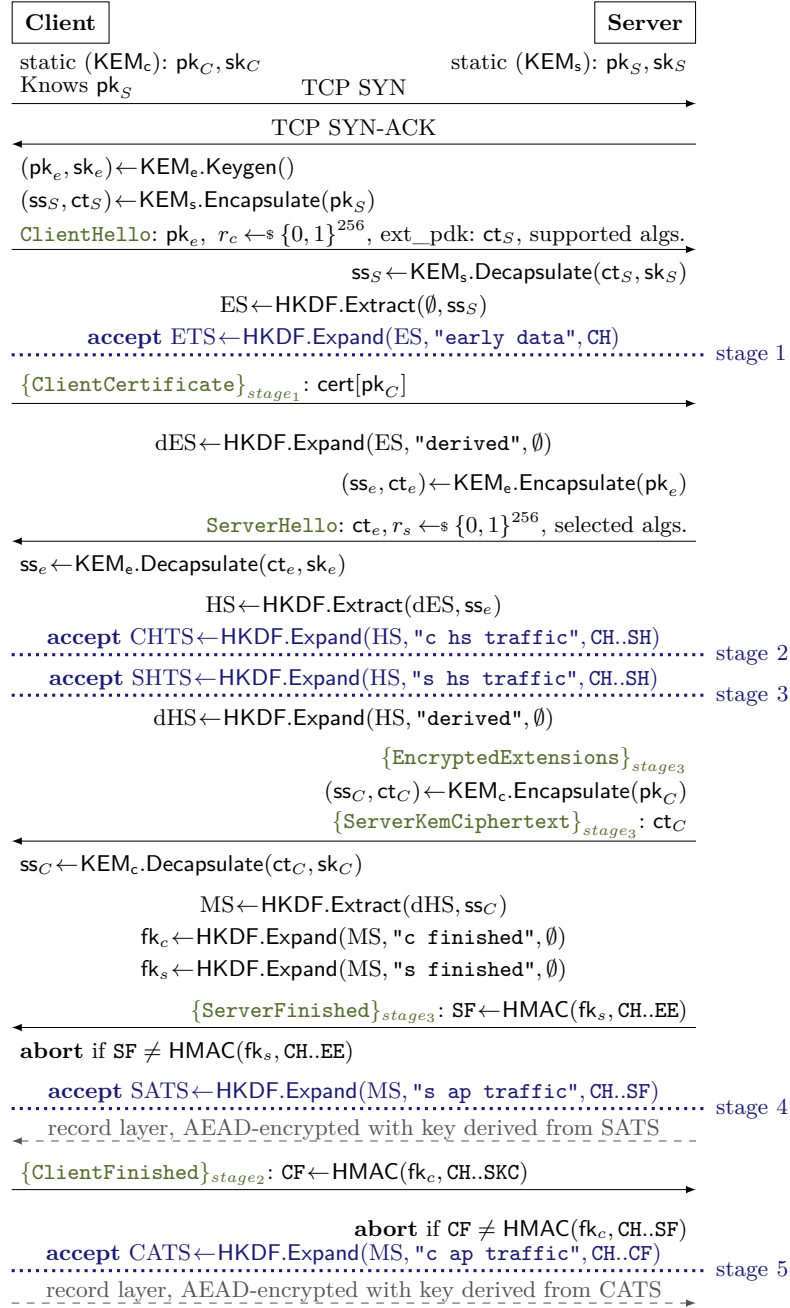
Client   **Server**

static $(\mathsf{KEM_c})$: $\mathsf{pk}_C, \mathsf{sk}_C$ static $(\mathsf{KEM_s})$: $\mathsf{pk}_S, \mathsf{sk}_S$

Knows pk_S

$\qquad\qquad\qquad\qquad\qquad$ TCP SYN $\longrightarrow$

$\longleftarrow\qquad$ TCP SYN-ACK $\qquad$

$(\mathsf{pk}_e, \mathsf{sk}_e) \leftarrow \mathsf{KEM_e.Keygen}()$

$(\mathsf{ss}_S, \mathsf{ct}_S) \leftarrow \mathsf{KEM_s.Encapsulate}(\mathsf{pk}_S)$

$\underline{\texttt{ClientHello}: \mathsf{pk}_e,\ r_c \leftarrow_\$ \{0,1\}^{256},\ \texttt{ext\_pdk}: \mathsf{ct}_S,\ \text{supported algs.}} \longrightarrow$

$\mathsf{ss}_S \leftarrow \mathsf{KEM_s.Decapsulate}(\mathsf{ct}_S, \mathsf{sk}_S)$

$\mathrm{ES} \leftarrow \mathsf{HKDF.Extract}(\emptyset, \mathsf{ss}_S)$

accept $\mathrm{ETS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{ES}, \texttt{"early data"}, \mathrm{CH})$ $\cdots\cdots$ stage 1

$\underline{\{\texttt{ClientCertificate}\}_{stage_1}: \mathsf{cert}[\mathsf{pk}_C]} \longrightarrow$

$\mathrm{dES} \leftarrow \mathsf{HKDF.Expand}(\mathrm{ES}, \texttt{"derived"}, \emptyset)$

$(\mathsf{ss}_e, \mathsf{ct}_e) \leftarrow \mathsf{KEM_e.Encapsulate}(\mathsf{pk}_e)$

$\longleftarrow \underline{\texttt{ServerHello}: \mathsf{ct}_e,\ r_s \leftarrow_\$ \{0,1\}^{256},\ \text{selected algs.}}$

$\mathsf{ss}_e \leftarrow \mathsf{KEM_e.Decapsulate}(\mathsf{ct}_e, \mathsf{sk}_e)$

$\mathrm{HS} \leftarrow \mathsf{HKDF.Extract}(\mathrm{dES}, \mathsf{ss}_e)$

accept $\mathrm{CHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \texttt{"c hs traffic"}, \mathrm{CH..SH})$ $\cdots\cdots$ stage 2

accept $\mathrm{SHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \texttt{"s hs traffic"}, \mathrm{CH..SH})$ $\cdots\cdots$ stage 3

$\mathrm{dHS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \texttt{"derived"}, \emptyset)$

$\{\texttt{EncryptedExtensions}\}_{stage_3}$

$(\mathsf{ss}_C, \mathsf{ct}_C) \leftarrow \mathsf{KEM_c.Encapsulate}(\mathsf{pk}_C)$

$\longleftarrow \{\texttt{ServerKemCiphertext}\}_{stage_3}: \mathsf{ct}_C$

$\mathsf{ss}_C \leftarrow \mathsf{KEM_c.Decapsulate}(\mathsf{ct}_C, \mathsf{sk}_C)$

$\mathrm{MS} \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}, \mathsf{ss}_C)$

$\mathsf{fk}_c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"c finished"}, \emptyset)$

$\mathsf{fk}_s \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"s finished"}, \emptyset)$

$\longleftarrow \{\texttt{ServerFinished}\}_{stage_3}: \mathrm{SF} \leftarrow \mathsf{HMAC}(\mathsf{fk}_s, \mathrm{CH..EE})$

abort if $\mathtt{SF} \neq \mathsf{HMAC}(\mathsf{fk}_s, \mathtt{CH..EE})$

accept $\mathrm{SATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"s ap traffic"}, \mathrm{CH..SF})$ $\cdots\cdots$ stage 4

$\longleftarrow$ record layer, AEAD-encrypted with key derived from SATS

$\{\texttt{ClientFinished}\}_{stage_2}: \mathrm{CF} \leftarrow \mathsf{HMAC}(\mathsf{fk}_c, \mathrm{CH..SKC}) \longrightarrow$

abort if $\mathtt{CF} \neq \mathsf{HMAC}(\mathsf{fk}_c, \mathtt{CH..SF})$

accept $\mathrm{CATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \texttt{"c ap traffic"}, \mathrm{CH..CF})$ $\cdots\cdots$ stage 5

record layer, AEAD-encrypted with key derived from CATS $\longrightarrow$

Figure 3: The KEMTLS-PDK handshake including key derivation schedule, with proactive client authentication

One important session variable is the *session identifier*, which will be used in the model to identify pairs of sessions that are partnered to each other. In general the session identifier consists of all of the protocols transmitted up until that point; as KEMTLS-PDK permits pre-distributed public keys, those pre-distributed values will be included in the session identifier.

The adversary controls all communications between parties, and can arbitrarily relay, change, drop, reorder, or insert messages. The adversary activates all sessions (via a NewSession query) and delivers all messages (via a Send query). It may compromise keys established during a stage (Reveal query) and parties' long-term keys (Corrupt query). The security model tracks the Reveal and Corrupt queries, and marks some sessions *unfresh* if too much information has been revealed and security can no longer be expected. The adversary may issue a Test query to a particular session and stage, and obtain either (a) the real key established for that stage, or (b) a uniformly random key; the choice of (a) or (b) depends on a hidden (uniform) bit chosen at the start of the experiment. There are some additional details in the full model in how queries are processed. Stage keys are marked as for *internal* or *external* use; internal-use keys may be used in subsequent parts of the protocol (e.g., to encrypt handshake messages), so the adversary must decide whether to Test an internal-use key when it is accepted.

Key indistinguishability. The Test query captures that keys established should be indistinguishable from random: the goal of the adversary is to guess the hidden bit in the Test query, thereby distinguishing real keys form random. The experiment restricts the adversary from issuing the Test query to stages where the key has been exposed via a Reveal query, including partnered sessions.

Implicit authentication and forward secrecy. Following [48], the security model captures three levels of forward secrecy for stage keys, which simultaneously incorporate notions of implicit authentication, meaning that only the intended party *could* know the shared secret.

- *Weak forward secrecy level 1* (wfs1): The stage key is indistinguishable to adversaries who were passive in the test stage, even if the adversary obtains the peer's long-term key at any point in time. These keys offer no implicit authentication. KEMTLS-PDK achieves wfs1 for server stages 2–5.
- *Weak forward secrecy level 2* (wfs2): The stage key is indistinguishable to adversaries who were passive in the test stage (wfs1) or who never corrupted the peer's long-term key; in the latter case this yields implicit authentication. KEMTLS-PDK achieves wfs2 for client stages 2–3.
- *Forward secrecy* (fs): The stage key is indistinguishable to adversaries who were passive in the test stage (wfs1) or who did not corrupt the peer's long-term key before the stage accepted; in the latter case this yields implicit authentication. KEMTLS-PDK achieves fs for client stages 4 and 5, and for server stages 4 and 5 in the mutually authenticated version.

The model captures *retroactive* revision of forward secrecy: a stage-i key may have a weaker form of forward secrecy at the time it is accepted, but may have a stronger form of forward secrecy after some subsequent stage accepts. In

KEMTLS-PDK, earlier stage keys are upgraded to the level of forward secrecy achieved by later stages once the later stage accepts.

In the unilaterally authenticated version of KEMTLS-PDK (Fig. 2), the client gets full fs one round trip earlier than in KEMTLS.

Explicit authentication. The security model also tracks at what point in time each party receives explicit authentication of its peer; explicit authentication goes a step further than implicit authentication in that the party has received explicit evidence (e.g., a MAC tag) that the intended peer actually is live. The model also includes retroactive authentication, in which an earlier stage may be regarded as explicitly authenticated once a later stage accepts. Client sessions in KEMTLS-PDK receive explicit authentication right before the stage-3 key is accepted, one round trip earlier than in KEMTLS. Server sessions, in mutually authenticated KEMTLS-PDK, receive explicit authentication right before the stage-5 key is accepted, again one round tip earlier than KEMTLS. In particular, this means that KEMTLS-PDK gives explicit authentication for all client application data (although only implicit authentication for the client certificate).

Downgrade resilience. [48] observed that implicit authentication characteristics of early stages of keys in KEMTLS meant that some application data would be transmitted prior to the client having received explicit authentication from the server of the symmetric encryption algorithms negotiated during the handshake. This meant that it would be possible for an adversary to cause a downgrade to a suboptimal (from the server's perspective) algorithm—although still only to an algorithm that the client offered to use. In KEMTLS-PDK, explicit server authentication happens one round trip earlier, and in particular prior to client transmission of application data, so KEMTLS-PDK offers full downgrade resilience.

Replayability. The model tracks that some stages are not protected against replays: in particular, stage-1 keys are not guaranteed to be unique at server instances since an adversary can replay the same `ClientHello` message multiple times to induce the same stage-1 key; all subsequent stages are replay-protected.

Anonymity. Like TLS 1.3 and KEMTLS, KEMTLS-PDK does not offer full anonymity, in particular due to the presence of the `ServerNameIndicator` extension in the `ClientHello` message. Our implementation also identifies the server certificate that was encapsulated to. (This identifier could be omitted by using trial-decryption at the server, though if the server has many public keys, this could be prohibitive.) The TLS working group is considering an "Encrypted `ClientHello`" mechanism that relies on the client obtaining a server public key out-of-band to enable identity protection for the server. KEMTLS-PDK's use of a pre-distributed key for encryption of part of the initial client message may be compatible with a variant of encrypted `ClientHello`, which we leave as future work since encrypted `ClientHello` has not yet been finalized even for TLS 1.3.

Deniability. As KEMTLS-PDK avoids the use of signatures for authentication, like KEMTLS and unlike TLS 1.3, KEMTLS-PDK offers *offline deniability* [18], meaning that a judge, when given a transcript of a protocol execution and

all of the keys involved, cannot tell whether the transcript is genuine or forged. KEMTLS-PDK does not have the harder-to-achieve online deniability property [20] when one party tries to frame the other or collaborates with the judge.

5   Instantiation and Evaluation

In this section we describe how we instantiate and implemented KEMTLS-PDK and compare the performance with KEMTLS and cached TLS.

For a fair comparison with cached TLS we proceed as follows. RFC 7924 [46] proposes a caching mechanism for certificates, which lets the client indicate that it already knows the server's certificate by including the hash of the `ServerCertificate` (SCRT) message in the `ClientHello` message. In TLS 1.2 this exactly amounts to a hash of the certificate chain. In TLS 1.3, however, the SCRT message was extended. It may, for example, include certificate transparancy [37] or OCSP [45] status information. This means that the hash value for the message is no longer stable. For our experiments, we adapt the mechanism of RFC 7924 to TLS 1.3 by using hashes for each individual certificate, instead. The server does not omit the SCRT message, but the certificates are replaced by their hashes. This allows the client to use those hashes to look up and replace the certificates by the originals for validation purposes. When instantiating TLS 1.3 with post-quantum primitives, we replace ephemeral DH by a KEM as described in [9] (for TLS 1.2) and implemented in post-quantum TLS experiments [10, 34, 36].

5.1   Choice of primitives

Table 2 shows the scenarios and primitives considered in our evaluation. We also show the sizes of the cryptographic objects that need to be transmitted, such as public keys, ciphertexts and signatures. All experiments require a KEM public key and ciphertext for ephemeral key exchange. In KEMTLS, we need a full certificate (signed KEM public key) and a ciphertext for authentication. For the TLS 1.3 with caching experiments, we only transmit a signature for authentication; certificates are withheld. Finally, for the KEMTLS-PDK instantiations, we withold the certificate and only transmit the ciphertext for authentication.

To instantiate KEMs and signatures we choose NIST PQC round-3 candidates at "level 1 security" (equivalent to AES-128). Kyber-512 [47], LightSABER [16], and NTRU-HPS-2048 [15] are all finalists, efficient KEMs and suitable for both ephemeral key exchange and authentication. Classic McEliece 348864 [1] is the remaining Finalist KEM, but its large public key makes it only suitable for authentication in KEMTLS-PDK. We include alternate candidate SIKEp434-compressed [28] as it is the KEM with the smallest sum of public key and ciphertext. For the signature schemes we consider the finalists Dilithium II [39], Falcon-512 [42], and Rainbow I Classic [19]. We align the chosen instantiations based on similar assumptions.

These scenarios immediately expose trade-offs that may or may not be feasible in real-world implementations. For example, with $\approx$260 KB for Rainbow and

Table 2: Instantiations of KEMTLS, TLS 1.3 with cached certificates and KEMTLS-PDK with sizes in bytes of transmitted public-key cryptography objects.

| | | Transmitted | | Sum | Client Auth | | Sum | Cached | |
		Ephem. (pk+ct)	Auth		Cert. (pk+ct/sig)	CA (sig)	(total)	Leaf pk	Cl. Auth CA (pk)
KEMTLS	Minimum	SIKE 197 236	SIKE/Rai. crt+ct 499	**932**	SIKE 433	Rainbow 66	**1,431**	N/A	Rainbow 161,600
KEMTLS	Assumption: MLWE/MSIS	Kyber 800 768	Kyber/Dil. crt+ct 3,988	**5,556**	Kyber 1,568	Dilithium 2,420	**9,554**	N/A	Dilithium 1,312
KEMTLS	Assumption: NTRU	NTRU 699 699	NTRU/Fal. crt+ct 2,088	**3,486**	NTRU 1,398	Falcon 690	**5,574**	N/A	Falcon 897
Cached TLS	TLS 1.3	X25519 32 32	RSA-2048 sig 256	**320**	RSA-2048 528	RSA-2048 256	**1,104**	RSA-2048 272	RSA-2048 272
Cached TLS	Minimum	SIKE 197 236	Rainbow sig 66	**499**	Falcon 1,587	Rainbow 66	**2,152**	Rainbow 161,600	Rainbow 161,600
Cached TLS	Assumption: MLWE/MSIS	Kyber 800 768	Dilithium sig 2,420	**3,988**	Dilithium 3,732	Dilithium 2,420	**10,140**	Dilithium 1,312	Dilithium 1,312
Cached TLS	Assumption: NTRU	NTRU 699 699	Falcon sig 690	**2,088**	Falcon 1,587	Falcon 690	**4,365**	Falcon 897	Falcon 897
KEMTLS-PDK	Minimum	SIKE 197 236	McEliece ct 128	**561**	SIKE 433	Rainbow 66	**1,060**	McEliece 261,120	Rainbow 161,600
KEMTLS-PDK	Finalist: Kyber	Kyber 800 768	Kyber ct 768	**2,336**	Kyber 1,568	Dilithium 2,420	**6,324**	Kyber 800	Dilithium 1,312
KEMTLS-PDK	Finalist: NTRU	NTRU 699 699	NTRU ct 699	**2,097**	NTRU 1,398	Falcon 690	**4,185**	NTRU 699	Falcon 897
KEMTLS-PDK	Finalist: SABER	SABER 672 736	SABER ct 736	**2,144**	SABER 1,408	Dilithium 2,420	**5,972**	SABER 672	Dilithium 1,312

$\approx$192 KB for McEliece, these schemes' public keys are very large. This means that they are probably not suitable for scenarios where the public keys are cached for shorter amounts of time, such as TLS resumption. If the cached public key needs to be updated, for example by having a KEMTLS-PDK handshake fall back to the regular KEMTLS, sending a certificate with a McEliece or Rainbow public key could be prohibitive.

5.2 Implementation

For our experiments, we implemented KEMTLS-PDK by extending the Rustls TLS library [8]. It is based on our prior implementation of KEMTLS. We also extend the TLS 1.3 protocol with the certificate caching for server certificates, as described in the previous paragraph. The measured post-quantum KEMs and signature algorithms are provided by the Open Quantum Safe project's `liboqs` library [51]. This library includes optimized, AVX2-accelerated implementations for every primitive measured, except for Rainbow. For a fair comparison, we ad-hoc integrated the AVX2 implementation from the Rainbow submitters into the version of `liboqs` used.

5.3   Handshake sizes

Table 2 shows the sizes of the public-key cryptographic objects transmitted in KEMTLS, TLS 1.3 with certificate caching, and KEMTLS-PDK. For KEMTLS, a full leaf certificate is transmitted (but no intermediate or root certificate); for TLS 1.3 with caching and KEMTLS-PDK we assume server certificates are cached. For client authentication, we have not included caching for the end-point certificate, as servers would presumably talk to many clients.

In scenarios where we aim to minimize communication, TLS 1.3 with caching transmits 433 fewer bytes of public-key cryptography objects than KEMTLS. It needs 66 fewer bytes on the wire than KEMTLS-PDK for unilaterally authenticated handshakes. When client authentication is included, KEMTLS-PDK reduces the number of bytes transmitted in the handshake by 51% (1092 B) compared to TLS 1.3 with caching, however. Compared to KEMTLS it saves 25% (371 B). The minimum scenarios for TLS 1.3 with caching and KEMTLS-PDK both heavily rely on the fact that the (very large) Rainbow and McEliece public keys do not need to be transmitted. This probably makes this scenario only practical for those cases where these keys can be used for very long times, such as when the keys are embedded in IoT devices.

The other instantiations are based on much faster lattice-based cryptography. These also have more managable public keys. This allows applications where the handshake is initially done without caching, after which the client remembers the certificate. In these instantiations, the Kyber instantiation of KEMTLS-PDK reduces transmission by 58% (3220 B) over KEMTLS for unilaterally authenticated handshakes and by 33% (3230 B) in mutually authenticated handshakes. For NTRU and Falcon—the two lattice-based schemes based on NTRU lattices—we see that KEMTLS-PDK performs better than cached TLS in terms of bandwidth requirements only when used with client authentication. This is expected as NTRU ciphertexts have essentially the same size (699 bytes) as Falcon signatures in the worst case (690 bytes). Note the average signature size of Falcon is advertised as 666 bytes [42], but our API does not use variable-size signatures.

5.4   Handshake times

We measured the times to complete the handshakes for each of our considered scenarios over both a low-latency, high-bandwidth connection and a high-latency, low-bandwidth connection. We follow the same methodology as [48], which is in turn using the methodology from [40]. Table 3 shows the timings of unilaterally authenticated handshakes. We see there is little difference between the three scenarios, although compressed SIKE has noticable performance overhead. The lattice schemes perform very similarly.

For mutually authenticated handshakes, as shown in Table 4, things change. The extra round-trip for client authentication in KEMTLS is clearly visible.

For a comparison of CPU cycles spent on the asymmetric cryptography, results for KEMTLS-PDK are similar to those of KEMTLS [48, Table 2 (left)]. The comparison made there also holds for KEMTLS-PDK and TLS 1.3 with

Table 3: Average time in ms for unilaterally authenticated handshakes of TLS 1.3 with cached leaf certificates and of KEMTLS-PDK.

	Unilaterally authenticated	31.1 ms RTT, 1000 Mbps			195.6 ms RTT, 10 Mbps		
		Client sent req.	Client recv. resp.	Server expl. auth.	Client sent req.	Client recv. resp.	Server expl. auth.
KEMTLS	Minimum	75.2	**115.9**	115.9	408.2	**616.3**	616.2
	MLWE/MSIS	63.2	**94.7**	94.6	397.2	**594.3**	594.3
	NTRU	63.1	**94.7**	94.6	395.6	**592.6**	592.6
Cached TLS	TLS 1.3	66.3	**97.5**	66.2	396.6	**592.8**	396.5
	Minimum	70.0	**101.2**	69.9	402.0	**598.2**	401.9
	MLWE/MSIS	63.9	**95.1**	63.8	396.8	**593.0**	396.7
	NTRU	64.8	**96.1**	64.7	396.7	**592.9**	396.7
PDK	Minimum	66.1	**97.3**	66.1	397.8	**594.0**	397.7
	Kyber	63.0	**94.3**	63.0	395.1	**591.3**	395.0
	NTRU	63.0	**94.3**	62.9	395.0	**591.2**	394.9
	SABER	63.1	**94.3**	63.0	395.1	**591.2**	395.0

caching, as the same cryptographic operations still need to be done and there have not been significant changes since round 2 of the NIST competition. In other words, KEMTLS-PDK offers the same massive savings compared to TLS 1.3 with pre-distributed certificates as KEMTLS offers in comparison to plain TLS 1.3.

6 Discussion

In this paper we presented the first investigation of post-quantum TLS with pre-distributed server public keys, both with traditional signature-based authentication and in the KEM-based authentication setting of KEMTLS. We believe that the results show that a combination of KEMTLS and KEMTLS-PDK may be the more efficient option for the post-quantum future of TLS. However, this will need to be confirmed (or refuted) through real-world experiments; one such experiment is currently underway in collaboration with Cloudflare; see [14].

There are some aspects of KEMTLS-PDK that we have not discussed so far but that deserve being mentioned.

We did not compare KEMTLS-PDK to TLS session resumption using a symmetric pre-shared key (for authentication) and DH (or KEMs) for forward secrecy. The reason is that in this scenario clients need to keep a sensitive secret key that is shared with the server; see the related discussion in Section 3.

For proactive client authentication, the client needs to pick some symmetric ciphersuite to use for encrypting and authenticating its certificate; this decision must be made before any ciphersuite negotiation has taken place. In our implementation we use a "default" ciphersuite; for example, `TLS_AES_128_GCM_SHA256` must be implemented by any TLS 1.3-compliant application [43, Sec. 9.1]. Another option would be to store information about the ciphersuite to use alongside

Table 4: Average time in ms for mutually authenticated handshakes of TLS 1.3 with cached leaf certificates and of KEMTLS-PDK.

Mutually authenticated	31.1 ms RTT, 1000 Mbps			195.6 ms RTT, 10 Mbps		
	Client sent req.	Client recv. resp.	Server expl. auth.	Client sent req.	Client recv. resp.	Server expl. auth.
KEMTLS Minimum	129.6	**160.8**	122.9	630.4	**826.8**	621.9
MLWE/MSIS	95.1	**126.5**	95.0	598.0	**794.3**	597.8
NTRU	95.0	**126.4**	94.8	595.7	**792.1**	595.5
Cached TLS TLS 1.3	68.6	**100.1**	66.0	399.4	**597.0**	396.7
Minimum	71.0	**102.6**	69.9	403.4	**601.9**	402.1
MLWE/MSIS	64.4	**96.1**	63.8	399.7	**616.1**	399.2
NTRU	66.1	**98.0**	64.7	398.3	**597.4**	396.9
PDK Minimum	84.9	**116.0**	84.9	420.4	**616.7**	420.4
Kyber	63.5	**94.8**	63.4	399.4	**595.6**	399.3
NTRU	63.6	**94.9**	63.5	397.2	**593.4**	397.1
SABER	63.6	**94.9**	63.5	400.6	**597.1**	400.5

the certificate – one could even consider integrating ciphersuite information inside certificates. This would clearly constitute a bigger change to the TLS infrastructure, but is not unprecedented (c.f. HPKE [3]).

Interestingly, if KEMTLS uses KEMs with different cryptographic assumptions for the ephemeral and the long-term KEM, we obtain hybrid confidentiality in the following sense: even if the assumption used for the ephemeral KEM is cryptographically broken at some point in the future, handshakes retain confidentiality as long as the long-term KEM keys are not compromised. With KEMTLS-PDK some interesting combinations of ephemeral and long-term KEMs— e.g., McEliece and Saber, Kyber, or NTRU—may become feasible. We leave it to future work to investigate this further and to formalize the notion of hybrid confidentiality sketched here.

Acknowledgements

References

1. Albrecht, M.R., Bernstein, D.J., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., von Maurich, I., Misoczki, R., Niederhagen, R., Paterson, K.G., Per-

sichetti, E., Peters, C., Schwabe, P., Sendrier, N., Szefer, J., Tjhai, C.J., Tomlinson, M., Wang, W.: Classic McEliece. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

2. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: Holz, T., Savage, S. (eds.) USENIX Security 2016. pp. 327–343. USENIX Association (Aug 2016)

3. Barnes, R.L., Bhargavan, K., Lipp, B., Wood, C.A.: Hybrid public key encryption. Internet-draft, Internet Research Task Force (2021), https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-08

4. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO'93. LNCS, vol. 773, pp. 232–249. Springer (Aug 1994). https://doi.org/10.1007/3-540-48329-2_21

5. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer (May / Jun 2006). https://doi.org/10.1007/11761679_25

6. Bindel, N., Braun, J., Gladiator, L., Stöckert, T., Wirth, J.: X.509-compliant hybrid certificates for the post-quantum transition. Journal of Open Source Software **4**(40), 1606 (2019). https://doi.org/10.21105/joss.01606

7. Bindel, N., Herath, U., McKague, M., Stebila, D.: Transitioning to a quantum-resistant public key infrastructure. In: Lange, T., Takagi, T. (eds.) Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017. pp. 384–405. Springer (2017). https://doi.org/10.1007/978-3-319-59879-6_22

8. Birr-Pixton, J.: A modern TLS library in Rust, https://github.com/ctz/rustls (accessed 2021-04-29)

9. Bos, J.W., Costello, C., Naehrig, M., Stebila, D.: Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In: 2015 IEEE Symposium on Security and Privacy. pp. 553–570. IEEE Computer Society Press (May 2015). https://doi.org/10.1109/SP.2015.40

10. Braithwaite, M.: Experimenting with post-quantum cryptography. Posting on the Google Security Blog (2016), https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html

11. Brzuska, C.: On the foundations of key exchange. Ph.D. thesis, Technische Universität Darmstadt, Darmstadt, Germany (2013), https://tuprints.ulb.tu-darmstadt.de/3414/

12. Brzuska, C., Fischlin, M., Warinschi, B., Williams, S.C.: Composability of Bellare-Rogaway key exchange protocols. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM CCS 2011. pp. 51–62. ACM Press (Oct 2011). https://doi.org/10.1145/2046707.2046716

13. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 453–474. Springer (May 2001). https://doi.org/10.1007/3-540-44987-6_28

14. Celi, S., Wiggers, T.: KEMTLS: Post-quantum TLS without signatures. Posting on the Cloudflare Blog (2021), https://blog.cloudflare.com/kemtls-post-quantum-tls-without-signatures/

15. Chen, C., Danba, O., Hoffstein, J., Hulsing, A., Rijneveld, J., Schanck, J.M., Schwabe, P., Whyte, W., Zhang, Z., Saito, T., Yamakawa, T., Xagawa, K.: NTRU. Tech. rep., National Institute of Standards and Technology (2020),

available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

16. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F., Mera, J.M.B., Beirendonck, M.V., Basso, A.: SABER. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

17. de Saint Guilhem, C.D., Smart, N.P., Warinschi, B.: Generic forward-secure key agreement without signatures. In: Nguyen, P.Q., Zhou, J. (eds.) ISC 2017. LNCS, vol. 10599, pp. 114–133. Springer (Nov 2017)

18. Di Raimondo, M., Gennaro, R., Krawczyk, H.: Deniable authentication and key exchange. In: Juels, A., Wright, R.N., De Capitani di Vimercati, S. (eds.) ACM CCS 2006. pp. 400–409. ACM Press (Oct / Nov 2006). https://doi.org/10.1145/1180405.1180454

19. Ding, J., Chen, M.S., Petzoldt, A., Schmidt, D., Yang, B.Y., Kannwischer, M., Patarin, J.: Rainbow. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

20. Dodis, Y., Katz, J., Smith, A., Walfish, S.: Composability and on-line deniability of authentication. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 146–162. Springer (Mar 2009). https://doi.org/10.1007/978-3-642-00457-5_10

21. 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

22. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol. Journal of Cryptology (2021), to appear. Cryptology ePrint Archive, Report 2020/1044, https://eprint.iacr.org/2020/1044

23. Drucker, N., Gueron, S.: Selfie: reflections on TLS 1.3 with PSK. Cryptology ePrint Archive, Report 2019/347 (2019), https://eprint.iacr.org/2019/347

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

25. Fujioka, A., Suzuki, K., Xagawa, K., Yoneyama, K.: Strongly secure authenticated key exchange from factoring, codes, and lattices. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) PKC 2012. LNCS, vol. 7293, pp. 467–484. Springer (May 2012). https://doi.org/10.1007/978-3-642-30057-8_28

26. Fujioka, A., Suzuki, K., Xagawa, K., Yoneyama, K.: Practical and post-quantum authenticated key exchange from one-way secure key encapsulation mechanism. In: Chen, K., Xie, Q., Qiu, W., Li, N., Tzeng, W.G. (eds.) ASIACCS 13. pp. 83–94. ACM Press (May 2013)

27. Günther, F.: Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols. Ph.D. thesis, Technische Universität Darmstadt, Darmstadt, Germany (2018), https://tuprints.ulb.tu-darmstadt.de/7162

28. Jao, D., Azarderakhsh, R., Campagna, M., Costello, C., De Feo, L., Hess, B., Jalali, A., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Renes, J., Soukharev, V., Urbanik, D., Pereira, G., Karabina, K., Hutchinson, A.: SIKE. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

29. Josefsson, S.: Storing Certificates in the Domain Name System (DNS). RFC 4398 (Mar 2006). https://doi.org/10.17487/RFC4398

30. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 546–566. Springer (Aug 2005). https://doi.org/10.1007/11535218_33

31. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_34

32. Krawczyk, H., Eronen, P.: HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (May 2010). https://doi.org/10.17487/RFC5869

33. Krawczyk, H., Wee, H.: The OPTLS protocol and TLS 1.3. In: EuroS&P 2016. IEEE (2017), https://eprint.iacr.org/2015/978.pdf

34. Kwiatkowski, K.: Towards post-quantum cryptography in TLS. Posting on the Cloudflare Blog (2019), https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/

35. LaMacchia, B.A., Lauter, K., Mityagin, A.: Stronger security of authenticated key exchange. In: Susilo, W., Liu, J.K., Mu, Y. (eds.) ProvSec 2007. LNCS, vol. 4784, pp. 1–16. Springer (Nov 2007)

36. Langley, A.: Cecpq2. Posting on the ImperialViolet Blog (2018), https://www.imperialviolet.org/2018/12/12/cecpq2.html

37. Laurie, B., Langley, A., Kasper, E.: Certificate transparency. RFC 6962 (Jun 2013). https://doi.org/10.17487/RFC6962

38. Law, L., Menezes, A., Qu, M., Solinas, J., Vanstone, S.: An efficient protocol for authenticated key agreement. Designs, Codes and Cryptography **28**(2), 119–134 (2003)

39. Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D., Bai, S.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

40. Paquin, C., Stebila, D., Tamvada, G.: Benchmarking post-quantum cryptography in TLS. In: Ding, J., Tillich, J.P. (eds.) Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020. pp. 72–91. Springer (2020). https://doi.org/10.1007/978-3-030-44223-1_5

41. Perrin, T.: Noise protocol framework (Jul 2018), https://noiseprotocol.org/noise.html (accessed 2021-04-29)

42. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

43. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Aug 2018). https://doi.org/10.17487/RFC8446

44. Rescorla, E., Sullivan, N., Wood, C.A.: Semi-static Diffie-Hellman key establishment for TLS 1.3. Internet-draft, Internet Engineering Task Force (2020), https://tools.ietf.org/html/draft-rescorla-tls-semistatic-dh-02

45. Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, D.C.: X.509 internet public key infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Jun 2013). https://doi.org/10.17487/RFC6960

46. Santesson, S., Tschofenig, H.: Transport Layer Security (TLS) Cached Information Extension. RFC 7924 (Jul 2016). https://doi.org/10.17487/RFC7924

47. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

48. Schwabe, P., Stebila, D., Wiggers, T.: Post-quantum TLS without handshake signatures. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 20. pp. 1461–1480. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3423350
49. Schwabe, P., Stebila, D., Wiggers, T.: Post-quantum TLS without handshake signatures. Cryptology ePrint Archive, Report 2020/534 (Apr 2021), https://eprint.iacr.org/2020/534, full version of [48]
50. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: A performance study. In: NDSS 2020. The Internet Society (Feb 2020)
51. Stebila, D., Mosca, M.: Post-quantum key exchange for the internet and the open quantum safe project. In: Avanzi, R., Heys, H.M. (eds.) SAC 2016. LNCS, vol. 10532, pp. 14–37. Springer (Aug 2016). https://doi.org/10.1007/978-3-319-69453-5_2
52. Sy, E., Burkert, C., Federrath, H., Fischer, M.: Tracking users across the web via TLS session resumption. In: ACM ACSAC 18. p. 289–299. ACM Press (2018). https://doi.org/10.1145/3274694.3274708

A   KEMTLS

Fig. 4 presents KEMTLS [48] side-by-side with TLS 1.3. Establishing an ephemeral shared secret happens in a similar way in TLS 1.3 and KEMTLS. The client submits a public key g^x (TLS 1.3) or pk_e (KEMTLS) to the server in the ClientHello message. The server then replies with its key share g^y (TLS 1.3) or ciphertext ct_e (KEMTLS). At this point, the server has the information to derive the handshake shared secret. This shared secret ss_e is used to encrypt the server's certificate before transmitting it to the client.

In TLS 1.3, this certificate contains a long-term public key for a digital signature algorithm pk_S. The server signs the transcript of all the transmitted messages so far and submits this signature in the ServerCertificateVerify message. This allows the client to immediately verify the server's identity: the signature proves the server posesses the private key corresponding to the certificate. The server then finishes the handshake by sending the key confirmation message. The client replies with its own key confirmation message.

When using long-term public keys for KEMs in certificates, signing the transcript is not possible. So, in KEMTLS, the client encapsulates a new shared secret ss_S to the server's long-term public key pk_S. It then sends over the corresponding ciphertext ct_S to the server. Only if the server can decapsulate the shared secret from the ciphertext, can it prove posession of the private key corresponding to the certificate. The server mixes in the new shared secret with the ephemeral secret key and derives new handshake keys. The server's key confirmation message then proves posesion of the long-term key.

If the client were to wait for the server to prove that it has decapsulated the ciphertext, KEMTLS would need a full extra round-trip over TLS 1.3. However, before the confirmation message, the client already knows that only the intended server would be able to read any data that is encrypted with keys derived from ss_S. KEMTLS uses this to allow the client to send data to the *implicitly authenticated* server before it has received the server's key confirmation message. Once the key confirmation is received, the server is *explicitly authenticated* and the client then knows the server is present.
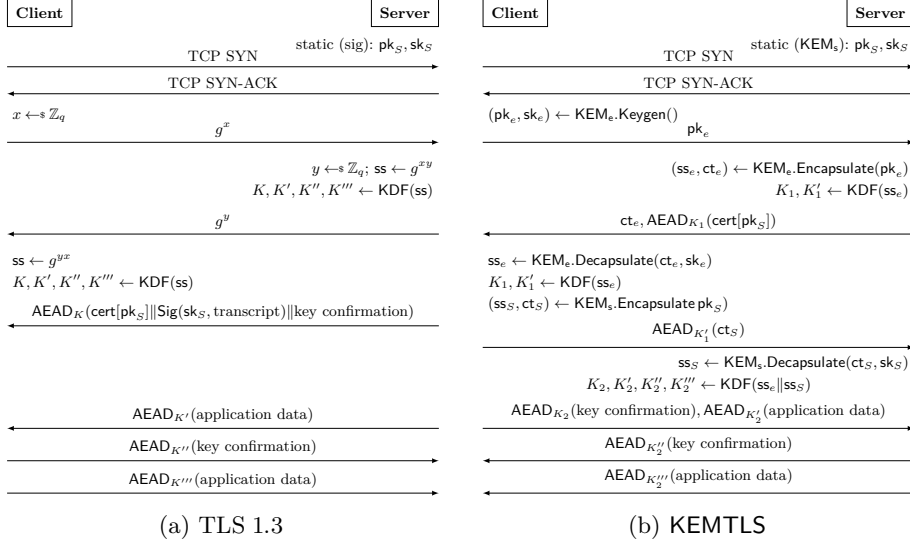
(a) TLS 1.3   (b) KEMTLS

Figure 4: Overview of TLS 1.3 and KEMTLS

B   Security proof

B.1   Model Syntax

The set of identities of honest participants in the system is denoted as $\mathcal{U}$. Each identity $U \in \mathcal{U}$ is associated with a certified long-term public key pk_U and the corresponding private key sk_U. Sessions of our protocols are uniquely identified by a label $\pi \in \mathcal{U} \times \mathbb{N}$. These are pairs (U, n) identifying the nth local session of U. In this model, each session consists of multiple stages, 1 through M. The session is run sequentially through each stage with shared state. Each stage aims to establish a key.

Each participant in each session maintains the following session-specific state information. The values that are M-length vectors specify values for each stage.

- $\mathsf{id} \in \mathcal{U}$: The identity of the participant that owns this session.
- $\mathsf{pid} \in \mathcal{U} \cup \{*\}$: The *intended* partner in this session. The identity of the partner may initially be unknown, indicated by $*$.
- $\mathsf{role} \in \{\mathsf{initiator}, \mathsf{responder}\}$.
- $\mathsf{status} \in \{\bot, \mathsf{running}, \mathsf{accepted}, \mathsf{rejected}\}^{\mathsf{M}}$: The status of each stage. $\mathsf{status}_i \leftarrow \mathsf{accepted}$ is set when stage i accepts the ith stage key. When a key is rejected, $\mathsf{status}_i \leftarrow \mathsf{rejected}$ and the protocol does not continue. status is initialised as $\left(\mathsf{running}, \bot^{\times 5}\right)$.
- $\mathsf{stage} \in \{0, \ldots, \mathsf{M}\}$: The last accepted stage. We set $\mathsf{stage} \leftarrow i$ when status_i is set to $\mathsf{accepted}$. Initially set to 0.
- $\mathsf{sid} \in \left(\{0,1\}^* \cup \{\bot\}\right)^{\mathsf{M}}$: The session identifier in stage i. There are initially set to $\bot$, and updated when reaching acceptance in that stage.

- $\mathsf{cid} \in \left(\{0,1\}^* \cup \{\bot\}\right)^{\mathsf{M}}$: The stage-$i$ contributive identifier. Initially all , and updated until reaching acceptance in that stage.
- $\mathsf{key} \in (\mathcal{K} \cup \{\bot\})^{\mathsf{M}}$: Key established in stage i.
- $\mathsf{revealed} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{M}}$: Records if key_i has been revealed by the adversary. Initially all false.
- $\mathsf{tested} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{M}}$: Records if key_i has been tested by the adversary. Initially all false.
- $\mathsf{mutualauth} \in \{\mathsf{true}, \mathsf{false}\}$: Indicates if mutual authentication will be required.
- $\mathsf{auth} \in \{(u,m) \in \{1, \ldots, \mathsf{M}, \infty\} \times \{1, \ldots, \mathsf{M}, \infty\}\}^{\mathsf{M}}$: Indicates at which stage a stage key is considered to be unilaterally (u) or mutually (m) accepted; some may never be authenticated (∞).
- $\mathsf{FS} \in \{0, \mathsf{wfs1}, \mathsf{wfs2}, \mathsf{fs}\}^{\mathsf{M} \times \mathsf{M}}$: For $j \geq i$, $\mathsf{FS}_{i,j}$ indicates the type of forward secrecy expected of stage key i, assuming stage j has accepted.
- $\mathsf{use} \in \{\mathsf{internal}, \mathsf{external}\}^{\mathsf{M}}$: Indicates if a stage-$i$ key is used internally in the key exchange protocol.
- $\mathsf{replay} \in \{\mathsf{nonreplayable}, \mathsf{replayable}\}^{\mathsf{M}}$: Indicates whether a stage is expected to be unique against replay attacks or not. The adversary should still not be able to distinguish a replayed accepted key and a random one.

We may write $\pi.X$ as shorthand to refer to π's element X.

The *partner* of a session π at stage i is defined to be the π' where $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_i \neq \bot$ and $\pi.\mathsf{role} \neq \pi'.\mathsf{role}$. We define the *contributive partner* similarly, using the contributive identifier cid. Correctness requires this equality holds for all stages on acceptance, for honest joint executions of the protocol.

B.2 Interacting with the adversary

As in our previous work [48, 49] and following [21, 22], both Match security and $\mathsf{Multi\text{-}Stage}$ security exist within the same adversary interaction model. The adversary $\mathcal{A}$ is a probabilistic algorithm. It controls all communication between all parties. This means it can intercept, inject or drop any message. In this model, for two honest parties to establish a connection they need $\mathcal{A}$ to facilitate their communication.

We will now give a number of queries that enable the interaction of the adversary with the model and the participants. We will not allow all combinations of queries, as otherwise the adversary might trivially win the test challenge. For example, allowing $\mathcal{A}$ to both reveal and test a particular session key would not model security appropriately.

The following two queries model honest protocol functionality:

- $\mathsf{NewSession}(U, V, role, mutualauth)$ creates a new session π, where $\pi.\mathsf{id} \leftarrow U$, $\pi.\mathsf{pid} \leftarrow V$, $\pi.\mathsf{role} \leftarrow role$ and $\pi.\mathsf{mutualauth} = mutualauth$. Note that possibly $V = *$, when V is left unspecified.
- $\mathsf{Send}(\pi, m)$ sends message m to session π. If sent to a session that has not been created by $\mathsf{NewSession}$, this returns $\bot$. Additionally, if $\pi.\mathsf{mutualauth} = \mathsf{false}$, and m is $\texttt{ClientCertificate}$, this also returns $\bot$. Otherwise, Send will

operate the protocol on behalf of $\pi.\mathsf{id}$, record the updated state and return both the response message and $\pi.\mathsf{status}_{\pi.\mathsf{stage}}$.

To let π initiate the protocol when $\pi.\mathsf{role} = \mathsf{initiator}$, the adversary may submit the special symbol $m = \mathsf{init}$.

Any keys that have already been used may not be tested. Because internal keys may be used immediately, we let Send pause execution whenever an internal key is accepted. It will then return $\mathsf{accepted}$ to the adversary. This allows the adversary to choose to test the session (or do anything in another session). Whenever the adversary decides to continue the session, they may call $\mathsf{Send}(\pi, \mathsf{continue})$. They will then receive the next protocol message and the status of the key in the current stage. On this call, we also set $\pi.\mathsf{status}_{\pi.\mathsf{stage}+1} \leftarrow \mathsf{running}$, except if the current stage is the last one.

If there exists a partner π' of π at stage i who has been tested ($\pi'.\mathsf{tested}_i = \mathsf{true}$), we set $\pi.\mathsf{tested} \leftarrow \mathsf{true}$. If this stage is an internal stage, we also set $\pi.\mathsf{key} \leftarrow \pi'.\mathsf{key}$. This ensures the adversary can not test keys twice and that session keys are consistently used.

The adversary can compromise participants and learn secret information through the following two queries:

- $\mathsf{Reveal}(\pi, i)$ gives the adversary session key $\pi.\mathsf{key}_i$. It also sets $\pi.\mathsf{revealed}_i \leftarrow \mathsf{true}$. For non-existing sessions or stages that have not accepted, returns $\bot$.
- $\mathsf{Corrupt}(U)$ provides the adversary with the long-term secret key sk_U. The time of corruption of U is recorded.

The final query models the challenge to the adversary of breaking a key established by two honest participants:

- $\mathsf{Test}(\pi, i)$ If the $\pi.\mathsf{stage}_i \neq \mathsf{accepted}$ or the key $\pi.\mathsf{key}_i$ has already been tested, or the session's partner has been tested at stage i, return $\bot$.
  We do not allow testing any keys that partnered sessions may have used. If $\pi.\mathsf{use}_i = \mathsf{internal}$, we require a partner π' to π to exist. This partner must not have yet used the key, i.e. $\pi'.\mathsf{status}_{i+1} = \bot$. Otherwise, we return $\bot$.
  We now set $\pi.\mathsf{tested}_i \leftarrow \mathsf{true}$. The Test oracle has a uniformly random bit b. This bit is fixed throughout the game. If $b = 0$, we sample a uniformly random key $K \leftarrow_\$ \mathcal{K}$. If $b = 1$, we set $K \leftarrow \pi.\mathsf{key}_i$. To ensure consistency with any later-used keys, we set $\pi.\mathsf{key}_i = K$ if $\pi.\mathsf{use}_i = \mathsf{internal}$. We then ensure consistency with partnered sessions. In sessions π' that are partner to π at stage i for which $\pi'.\mathsf{status}_i = \mathsf{accepted}$, set $\pi'.\mathsf{tested}_i \leftarrow \mathsf{true}$ and, if $\pi'.\mathsf{use}_i = \mathsf{internal}$, also set $\pi'.\mathsf{key}_i \leftarrow K$. We return K to the adversary.

B.3   Specifics of KEMTLS-PDK

In our protocol, the number of states is $\mathsf{M} = 5$. KEMTLS-PDK without client authentication is shown in Fig. 2 and with client authentication in Fig. 3 The

session identifiers are set up as follows in unilaterally authenticated sessions:

$\text{sid}_1 = (\text{``ETS''}, \texttt{ServerCertificate}, \texttt{ClientHello}),$

$\text{sid}_2 = (\text{``CHTS''}, \texttt{ServerCertificate}, \texttt{ClientHello} \ldots \texttt{ServerHello}),$

$\text{sid}_3 = (\text{``SHTS''}, \texttt{ServerCertificate}, \texttt{ClientHello} \ldots \texttt{ServerHello}),$

$\text{sid}_4 = (\text{``SATS''}, \texttt{ServerCertificate}, \texttt{ClientHello} \ldots \texttt{ServerFinished}),$

$\text{sid}_5 = (\text{``CATS''}, \texttt{ServerCertificate}, \texttt{ClientHello} \ldots \texttt{ClientFinished}).$

Each identifier is made up of a label, the server's certificate, and all the unencrypted handshake messages up to that point.

For the contributive identifiers cid_i we take some special care. For stage 1, we want to ensure client sessions can be tested, even if the $\mathcal{A}$ drops the client's message to the server. We set $\text{cid}_1 = (\text{``ETS''}, \texttt{SCRT}, \emptyset)$ initially. When the client sends or the server receives the $\texttt{ClientHello}$ message, we update it to $\text{cid}_1 = (\text{``ETS''}, \texttt{SCRT}, \texttt{CH})$.

In Case A of our Multi-Stage proof we need to identify the unique pair of honest contributive server and client sessions, even if $\mathcal{A}$ drops the server's response to the client. This requires us to set $\text{cid}_2 = (\text{``CHTS''}, \texttt{SCRT}, \texttt{CH})$ when sending or receiving the $\texttt{ClientHello}$ message. At that time we also set $\text{cid}_3 = (\text{``SHTS''}, \texttt{SCRT}, \texttt{CH})$. If $\pi.\mathsf{mutualauth} = \mathsf{true}$, when the $\texttt{CCRT}$ message is sent or received we update cid_2 and cid_3 by updating the transcripts to $\texttt{CH} \ldots \texttt{CCRT}$. When the $\texttt{ServerHello}$ message is received or sent, they update this to $\text{cid}_i = \text{sid}_i$ for $i = 2, 3$. All other contributive identifiers $(i = 4, 5)$ are set when the corresponding sid_i is set.

The value of $\mathsf{replay} = (\mathsf{replayable}, \mathsf{nonreplayable}^{\times 4})$ for both protocols.

We set $\mathsf{auth} = ((4, m), (4, m), (4, m), (4, m), (5, m))$. In unilaterally authenticated KEMTLS-PDK, only the server is authenticated, so $m = \infty$. When using mutual authentication, i.e. $\pi.\mathsf{mutualauth} = \mathsf{true}$, the client is explicitly authenticated when stage $m = 5$ is accepted.

In client sessions, we set

$$\mathsf{FS} = \begin{pmatrix} 0 & \mathsf{wfs2} & \mathsf{wfs2} & \mathsf{fs} & \mathsf{fs} \\ & \mathsf{wfs2} & \mathsf{wfs2} & \mathsf{fs} & \mathsf{fs} \\ & & \mathsf{wfs2} & \mathsf{fs} & \mathsf{fs} \\ & & & \mathsf{fs} & \mathsf{fs} \\ & & & & \mathsf{fs} \end{pmatrix}.$$

For server sessions, we set

$$\mathsf{FS} = \begin{pmatrix} 0 & \mathsf{wfs1} & \mathsf{wfs1} & f & f \\ & \mathsf{wfs1} & \mathsf{wfs1} & f & f \\ & & \mathsf{wfs1} & f & f \\ & & & f & f \\ & & & & f \end{pmatrix},$$

where $f = \mathsf{fs}$ if $\pi.\mathsf{mutualauth} = \mathsf{true}$ and $f = \mathsf{wfs1}$ otherwise.

B.4   Match security

As in prior work [11, 12, 21, 22, 24, 48] need to show that our model has sound behaviour of session matching. Match security ensures that in honest sessions, $\pi.\mathsf{sid}$ and $\pi'.\mathsf{sid}$ correctly match, where π and π' are partnered.

Definition 1 (Match security). *Let* KE *be an* M*-stage key-exchange protocol. Probabilistic adversary $\mathcal{A}$ interacts with* KE *via the queries defined in Appendix B.2. $\mathcal{A}$ tries to win the following game $G_{\mathsf{KE},\mathcal{A}}^{\mathsf{Match}}$:*

Setup *The challenger generates long-term keypairs $(\mathsf{pk}_U, \mathsf{sk}_U)$ for each participant $U \in \mathcal{U}$. All keys are provided to $\mathcal{A}$.*

Query *The adversary has access to the queries* NewSession, Send, Reveal, Corrupt *and* Test.

Stop *At some point, the adversary stops with no output.*

Let π, π' be distinct, partnered sessions for which $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_i \neq \bot$ at some stage $i \in \{1, \ldots, \mathsf{M}\}$ The adversary $\mathcal{A}$ wins the Match security game, denoted $G_{\mathsf{KE},\mathcal{A}}^{\mathsf{Match}} = 1$, if it can falsify any of the following conditions:

1. *At every stage $j \leq i$, $\pi.\mathsf{key}_j = \pi'.\mathsf{key}_j$: both sessions agree on the same key at every stage up to and including stage i.*
2. *$\pi.\mathsf{role} \neq \pi'.\mathsf{role}$, except if $\pi.\mathsf{role} = \mathsf{responder}$ and $\pi.\mathsf{replay}_i = \mathsf{replayable}$: both sessions must have opposite roles, except if they are both responders in a replayable stage.*
3. *$\pi.\mathsf{cid}_i = \pi'.\mathsf{cid}_i$: both sessions have the same contributive identifier.*
4. *$\pi.\mathsf{auth}_i = \pi'.\mathsf{auth}_j$: both sessions agree on the explicit authentication level.*
5. *At every stage j, if $\pi.\mathsf{status}_j = \mathsf{accepted}$ and $\pi.\mathsf{stage} \geq \pi.\mathsf{auth}_{j,0}$ or $\pi.\mathsf{stage} \geq \pi.\mathsf{auth}_{j,1}$, $\pi.\mathsf{pid} = \pi'.\mathsf{id}$: sessions are partnered with the intended (explicitly authenticated) participant.*
6. *If $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_j$, then $i = j$: session labels are different in different stages.*
7. *If $\pi.\mathsf{replay}_i = \mathsf{nonreplayable}$, for any three sessions π, π', π'', if $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_i = \pi''.\mathsf{sid}_i \neq \bot$, then $\pi = \pi'$, or $\pi = \pi''$, or $\pi' = \pi''$: at most two sessions share the same session label.*

We say that KE *is* Match*-secure if for all $\mathcal{A}$ that run in polynomial time,*

$$\mathsf{Adv}_{\mathsf{KE},\mathcal{A}}^{\mathsf{Match}} := \left| \Pr\left[G_{\mathsf{KE},\mathcal{A}}^{\mathsf{Match}} \Rightarrow 1 \right] - \frac{1}{2} \right|$$

is negligible in the security parameter.

Theorem 1. KEMTLS-PDK *is* Match*-secure. Any efficient $\mathcal{A}$ has advantage*

$$\mathsf{Adv}_{\mathsf{KEMTLS\text{-}PDK},\mathcal{A}}^{\mathsf{Match}} \leq n_s(\delta_e + \delta_s + \delta_c) + n_s^2/2^{|\mathsf{nonce}|},$$

where n_s is the number of sessions, $|\mathsf{nonce}|$ is the length of the nonces r_c, r_s in bits. δ_e is the correctness of the ephemeral KEM, and δ_s and δ_c are the correctness of the long-term KEMs of the server and the client, respectively. If $\pi.\mathsf{mutualauth} = \mathsf{false}$, $\delta_c = 0$.

Proof. We will show that the properties of Match-security hold.

1. By definition, the session identifiers contain all handshake messages. The KEM keys and the hashes of the handshake messages are the only inputs into the key schedule. At stage 1, the input to the agreed key is the $\mathsf{KEM_s}$ shared secret and the `ClientHello` message. For stages 2 and 3, the input to the agreed keys are the previous key, messages up to and including `ServerHello` and the ephemeral $\mathsf{KEM_e}$ shared secret. For final stages 4 and 5, the inputs are the previous keys, messages up to `ServerFinished` and `ClientFinished`, respectively, and, when using mutual authentication, the $\mathsf{KEM_c}$ shared secret. These inputs are all included in the session identifiers, which means that both parties use the same inputs to key computations. The only way they could arrive at different keys is when any of the KEMs fail. In each of the n_s sessions, the ephemeral KEM $\mathsf{KEM_e}$ may fail with probability δ_e, the static KEM $\mathsf{KEM_s}$ may fail with probability δ_s. When using mutual authentication, the static KEM $\mathsf{KEM_c}$ may fail with probability δ_c.

2. All messages are exclusively sent or received by either role. This means no initiator or responder will accept an incoming message intended for the other role. This implies any pair of two sessions in a non-replay stage must have an initiator and responder. As at most two sessions in a non-replay phase have the same sid_i (shown later), these pairings are unique and opposite. As the only replayable messages are `ClientHello` and `ClientCertificate` messages, not opposite-role pairings can only be between responders.

3. By definition, cid_i is final and equal to sid_i whenever stage i accepts.

4. The keys in stage 1–4 are fixed to unilateral (retroactive) explicit authentication up to stage 4. The presence of `ServerKemCiphertext` in sid_5 unambiguously decides if, at stage 5, keys are mutually or unilaterally authenticated.

5. The partnered sessions have to agree once they reach a retroactively authenticated stage, so at stage 4 for unilateral authentication and stage 5 for mutual authentication. For Match security, we are only concerned with honest client and server sessions. The client already knows the identity of the server through the pre-distributed key, which is included in the session identifiers at stage 1. The server learns the identity of the client through the `ClientCertificate` message, which is included in the session identifiers at stage 5. An honest client will only send its own certificate.

6. As each stage's session identifier has a unique label, this holds trivially.

7. We are only concerned about stages 2 and later. All session identifiers sid contain nonces r_c and r_s embedded in the `ClientHello` and `ServerHello` messages. To get any collision between sessions of honest parties, some session would need to pick the same nonce as another session. If this happens, the parties may then be partnered through a regular protocol run to another one. The probability for such a collision is bounded by the birthday bound $n_s^2 \cdot 2^{-|\mathsf{nonce}|}$. Here, n_s is the maximum number of sessions and $|\mathsf{nonce}| = 256$ is the nonces' length in bits. □

B.5 Multi-Stage security

Like [48] we prove the security of KEMTLS-PDK via Multi-Stage security games as introduced by [24]. The adversary wins, Bellare–Rogaway-style [4], if they correctly distinguish the keys derived in the protocol from random. They also win if they get the protocol to maliciously accept (Definition 3); this models the acceptance of an invalid `ServerFinished` or `ClientFinished` message that was forged by the adversary.

We first define the terms *fresh* and *maliciously accept*, that we will throughout the proof.

Definition 2 (Freshness). *In a session π, stage i is said to be* fresh *if conditions 1, 2, 3, and at least one of 4, 5, or 6 hold:*

1. *$\pi.\mathsf{revealed}_i = \mathsf{false}$: the session key has not been revealed.*
2. *In any partnered sessions π' where $\pi'.\mathsf{sid}_i = \pi.\mathsf{sid}_i$, $\pi'.\mathsf{revealed} = \mathsf{false}$: no partnered session at the same stage has been revealed.*
3. *If stage i is* replayable, *$\mathsf{Corrupt}(\pi.\mathsf{pid})$ is never called.*
4. *(Weak forward secrecy 1) There exists $j \geq i$ such that $\pi.\mathsf{FS}_{i,j} = \mathsf{wfs1}$, $\pi.\mathsf{status}_j = \mathsf{accepted}$, and there exists a contributive partner at stage i.*
5. *(Weak forward secrecy 2) There exists $j \geq i$ such that $\pi.\mathsf{FS}_{i,j} = \mathsf{wfs2}$, $\pi.\mathsf{status}_j = \mathsf{accepted}$, and either (a) there exists a contributive partner at stage i or (b) $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was never called.*
6. *(Forward secrecy) There exists $j \geq i$ such that $\pi.\mathsf{FS}_{i,j} = \mathsf{fs}$, $\pi.\mathsf{status}_j = \mathsf{accepted}$, and either (a) there exists a contributive partner at stage i or (b) $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was not called before π accepted stage j.*

Definition 3 (Malicious acceptance). *Let $(u, m) = \pi.\mathsf{auth}_i$. Here stage u is the stage at which unilateral authentication is reached. Stage m is the stage at which mutual authentication is reached. Stage i of session π is said to have* maliciously accepted *if all of the following hold:*

1. *$\pi.\mathsf{status}_i = \mathsf{accepted}$;*
2. *if π is the* initiator *then $\pi.\mathsf{status}_u = \mathsf{accepted}$ or if π is a* responder *then $\pi.\mathsf{status}_m = \mathsf{accepted}$;*
3. *if stage i is not* replayable, *there does not exist a unique partner of π at stage i; and*
4. *$\mathsf{Corrupt}(\pi.\mathsf{pid})$ was not called before the last stage accepted by π was accepted.*

Next we define our version of the Multi-Stage security game. The proof follows the definition.

Definition 4 (Multi-Stage security). *For a M-stage key exchange protocol KE, let $\mathcal{A}$ be a probabilistic adversary. $\mathcal{A}$ interacts with KE via the queries defined in Appendix B.2. The adversary tries to win the following game $G_{\mathsf{KE},\mathcal{A}}^{\mathsf{Multi\text{-}Stage}}$:*

Setup *The challenger chooses the test bit $b \leftarrow_\$ \{0, 1\}$. It also generates long-term keys $(\mathsf{pk}_U, \mathsf{sk}_U)$ for all identities $U \in \mathcal{U}$. The public keys pk_U are provided to $\mathcal{A}$.*

Query *The adversary has access to the queries* NewSession, Send, Reveal, Corrupt *and* Test.

Stop *At some point,* $\mathcal{A}$ *stops and outputs their guess* b' *of* b.

Finalize *The adversary wins the game if either of the following conditions hold:*

1. *All tested stages are fresh (Definition 2) and* $b' = b$; *or*

2. *There exists a stage that has maliciously accepted (Definition 3).*

In either of these cases the experiment $G_{\mathsf{KE},\mathcal{A}}^{\mathsf{Multi\text{-}Stage}}$ *outputs 1. Otherwise the adversary has lost the game, in which* $G_{\mathsf{KE},\mathcal{A}}^{\mathsf{Multi\text{-}Stage}}$ *outputs a uniformly random bit.*

The Multi-Stage-*advantage of* $\mathcal{A}$ *is defined as*

$$\mathsf{Adv}_{\mathsf{KE},\mathcal{A}}^{\mathsf{Multi\text{-}Stage}} = \left| \Pr\left[G_{\mathsf{KE},\mathcal{A}}^{\mathsf{Multi\text{-}Stage}} \Rightarrow 1 \right] - \frac{1}{2} \right|.$$

Theorem 2. *Let* $\mathcal{A}$ *be an algorithm.* n_s *is the number of sessions and* n_u *is the number of identities. There exist algorithms* $\mathcal{B}_1, \dots, \mathcal{B}_{19}$, *given in the proof, such that*

$$\mathsf{Adv}_{\mathsf{KEMTLS\text{-}PDK},\mathcal{A}}^{\mathsf{Multi\text{-}Stage}} \leq \frac{n_s^2}{2^{|\mathsf{nonce}|}} + \mathsf{Adv}_{\mathsf{H},\mathcal{B}_1}^{\mathsf{COLL}}$$

$$+ 5 n_s \cdot \left(\begin{array}{l} n_s \cdot \left(\begin{array}{l} \mathsf{Adv}_{\mathsf{KEM}_s,\mathcal{B}_2}^{\mathsf{IND\text{-}CCA}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_3}^{\mathsf{PRF\text{-}sec}} \\ + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_4}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{KEM}_e,\mathcal{B}_5}^{\mathsf{IND\text{-}1CCA}} \\ + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_6}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_7}^{\mathsf{PRF\text{-}sec}} \\ + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_8}^{\mathsf{dual\text{-}PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_9}^{\mathsf{PRF\text{-}sec}} \\ + 2\,\mathsf{Adv}_{\mathsf{HMAC},\mathcal{B}_{10}}^{\mathsf{EUF\text{-}CMA}} \end{array} \right) \\ + n_u \cdot \left(\begin{array}{l} \mathsf{Adv}_{\mathsf{KEM}_s,\mathcal{B}_{11}}^{\mathsf{IND\text{-}CCA}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{12}}^{\mathsf{dual\text{-}PRF\text{-}sec}} \\ + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{13}}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{14}}^{\mathsf{dual\text{-}PRF\text{-}sec}} \\ + \mathsf{Adv}_{\mathsf{KEM}_c,\mathcal{B}_{16}}^{\mathsf{IND\text{-}CCA}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{15}}^{\mathsf{PRF\text{-}sec}} \\ + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{17}}^{\mathsf{dual\text{-}PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{18}}^{\mathsf{PRF\text{-}sec}} \\ + 2\,\mathsf{Adv}_{\mathsf{HMAC},\mathcal{B}_{19}}^{\mathsf{EUF\text{-}CMA}} \end{array} \right) \end{array} \right).$$

Proof. We follow the basic structure of the proof of the KEMTLS handshake [48]. This in turn is based on the proofs of the TLS 1.3 handshake by Dowling, Fischlin, Günther and Stebila [21,22]. The proof proceeds by a sequence of games in which we keep reducing the advantage of the adversary. As the adversary otherwise loses the game, we assume that all tested sessions remain fresh throughout the experiment.

Game 0 (Multi-Stage game). We define G_0 to be the original Multi-Stage game:

$$\mathsf{Adv}_{\mathsf{KEMTLS\text{-}PDK},\mathcal{A}}^{\mathsf{Multi\text{-}Stage}} = \mathsf{Adv}_{\mathcal{A}}^{G_0}.$$

Game 1 (Nonce collisions). If any honest session uses the same nonce r_c or r_s as any other session, the challenger aborts. The chance of such a repeat, and

thus the reduction in advantage of $\mathcal{A}$, is given by the birthday bound over n_s sessions using $|\mathsf{nonce}| = 256$-bit nonces:

$$\mathsf{Adv}^{G_0}_{\mathcal{A}} \leq \mathsf{Adv}^{G_1}_{\mathcal{A}} + \frac{n_s^2}{2^{|\mathsf{nonce}|}}.$$

This means we can now rule out nonce collisions in future games.

Game 2 (Hash collisions). If any two honest sessions compute the same hash for different inputs of hash function H, the challenger aborts. If this event occurs, we obtain a reduction $\mathcal{B}_1$ that can break the collision-resistance of H. $\mathcal{B}_1$ outputs the two distinct input values when a collision occurs. This gives us the following:

$$\mathsf{Adv}^{G_1}_{\mathcal{A}} \leq \mathsf{Adv}^{G_2}_{\mathcal{A}} + \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}.$$

Game 3 (Single Test query). By invoking a hybrid argument by Günther [27], we restrict $\mathcal{A}$ to only make a single Test-query. This reduces the advantage at most by $1/5n_s$ for the five stages of KEMTLS-PDK. Any single-query adversary $\mathcal{A}_1$ can emulate the original multi-query adversary $\mathcal{A}$ by guessing the to-be-tested session in advance. Any other Tests that $\mathcal{A}$ may submit, $\mathcal{A}_1$ simulates by carefully selected Reveal queries. $\mathcal{A}_1$ needs to know how sessions are partnered from the session identifiers sid. Only the first one is unencrypted, but the later sid can be obtained by $\mathcal{A}_1$ by revealing handshake traffic secrets.

We get the following advantage by letting $\mathcal{A}_1$ guess the right session and stage:

$$\mathsf{Adv}^{G_2}_{\mathcal{A}} \leq 5n_s \cdot \mathsf{Adv}^{G_3}_{\mathcal{A}_1}.$$

This restriction of $\mathcal{A}$ to $\mathcal{A}_1$ means we can now refer to *the* session π at stage i that is tested. We can also assume we know this from the start.

Two separate cases. We now need to consider two separate cases of game 3. These cases, respectively, roughly correspond to the specified properties of weak forward secrecy: $\mathsf{wfs1}$ and $\mathsf{wfs2}$. By rejecting malicious acceptance, we finally show fs.

A. In these games, denoted G_A, the tested session π has a unique contributive partner in stage 2. This means there exists a session π' such that $\pi.\mathsf{cid}_2 = \pi'.\mathsf{cid}_2$.

B. In these games, denoted G_B, the tested session π does *not* have a contributive partner in stage 2. In addition, $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was not called *before* stage i of π accepted.

The advantage of the adversary can be considered separately for these cases as:

$$\mathsf{Adv}^{G_3}_{\mathcal{A}_1} \leq \max\left\{\mathsf{Adv}^{G_A}_{\mathcal{A}_1}, \mathsf{Adv}^{G_B}_{\mathcal{A}_1}\right\}$$
$$\leq \mathsf{Adv}^{G_A}_{\mathcal{A}_1} + \mathsf{Adv}^{G_B}_{\mathcal{A}_1}.$$

Case A: session π has a unique contributive partner in stage 2

In this case, we assume that π has a π' with whom they share $\pi.\mathsf{cid}_2 = \pi'.\mathsf{cid}_2$. If the tested session π is a client (initiator) session, then $\pi.\mathsf{cid}_2 = \pi.\mathsf{sid}_2$ and a partner session at π' also exists. sid_2 includes the client and server nonces, and by game 1 no honest sessions repeat nonces. This means that the contributive partner at stage 2 is unique.

However, if π has role $=$ responder, then it may have received a replayed CH message. This would mean a contributive partner session exists at stage 2, but there is no partnered session. However, there exists only one honest client session that is a contributive partner: cid_2 includes the client nonce (unique by game 1) and contributive partnering includes roles.

This means we can speak of a particular tested session, π. Its unique contributive stage-2 partner we call π'. Of these two, we let π_c be the session that is the client (role $=$ initiator) session. The other session, with role $=$ responder, is the server session π_s.

Game A1 (guess contributive partner session). In this game, the challenger tries to guess the $\pi' \neq \pi$ that is the honest contributive partner to π at stage 2. As the challenger guesses correctly with probability $\frac{1}{n_s}$, this reduces the advantage of $\mathcal{A}_1$ as:

$$\mathsf{Adv}^{G_A}_{\mathcal{A}_1} \leq n_s \cdot \mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1} .$$

In the remainder of case A, we will keep replacing keys in π and π'.

Game A2 (Static KEM). In this game we replace the shared secret ss_S encapsulated to pk_S by a uniformly random $\widetilde{\mathsf{ss}_S}$. We make this replacement in π_c and π_s, and, as this stage is replayable, in any other sessions π'' of S that received ct_S.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_2$ that breaks the IND-CCA security of $\mathsf{KEM_s}$. $\mathcal{B}_2$ obtains the IND-CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and challenge shared secret $\mathsf{ss}^\star$ and uses $\mathsf{pk}^\star$ as the long-term key pk_S of party S. In π_c, $\mathcal{B}_2$ sends $\mathsf{ct}^\star$ in the ClientHello message. $\mathcal{B}_2$ uses $\mathsf{ss}^\star$ for ss_S in both π_c and π_s. If $\mathcal{A}_1$ delivers $\mathsf{ct}^\star$ to some other session π'' of party S, $\mathcal{B}_2$ uses $\mathsf{ss}^\star$ as value for ss_S in π''. If $\mathcal{A}_1$ delivers a different $\mathsf{ct}' \neq \mathsf{ct}^\star$ to some other session π'' of party S, $\mathcal{B}_2$ queries its IND-CCA decapsulation oracle with ct' to obtain the required shared secret.

Stage 1 cannot maliciously accept since it is replayable. By the definition of freshness (Definition 2) we also do not need to answer Corrupt queries.

In the end, $\mathcal{A}_1$ terminates and outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_2$ has exactly simulated G_{A1} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_2$ has exactly simulated G_{A2} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A2}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM_s},\mathcal{B}_2} .$$

Game A3 (Replacing ES). In this game we replace the early handshake secret ES by a uniformly random $\widetilde{\mathrm{ES}}$ in both sessions π, π'.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_3$ that breaks the PRF security of HKDF.Extract in its second

argument. When $\mathcal{B}_3$ needs to compute ES in π or π' it queries its HKDF.Extract challenge oracle on $(\emptyset, \widetilde{\mathsf{ss}}_S)$ and uses the response as ES. If the response was the real shared secret, $\mathcal{B}_3$ has exactly simulated G_{A2} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_3$ has exactly simulated G_{A3} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A2}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{A3}} + \mathsf{Adv}_{\mathsf{HKDF.Extract}, \mathcal{B}_3}^{\mathsf{PRF\text{-}sec}} \,.$$

Game A4 (Replacing ETS and dES). In this game we replace the values ETS and dES by uniformly random values in both sessions π_c, π_s. All values derived from dES in both sessions use the new value $\widetilde{\mathsf{dES}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_4$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_4$ needs to compute ETS or dES in π_c or π_s it queries its HKDF.Expand challenge oracle with $\widetilde{\mathsf{ES}}$ and the corresponding label and transcript, and uses the responses. If the response was the real output, $\mathcal{B}_4$ has exactly simulated G_{A3} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_4$ has exactly simulated G_{A4} to $\mathcal{A}_1$.

We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A3}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{A4}} + \mathsf{Adv}_{\mathsf{HKDF.Expand}, \mathcal{B}_4}^{\mathsf{PRF\text{-}sec}} \,.$$

The stage-1 key ETS is now a uniformly random string independent of anything else in the game. It is, however, not forward-secure.

Game A5 (Ephemeral KEM). In this game, in session π_s we replace the ephemeral shared secret ss_e with a uniformly random $\widetilde{\mathsf{ss}}_e$. In π_c we replace ss_e with the same $\widetilde{\mathsf{ss}}_e$, but only if it received the same ct_e that π_s sent. If ss_e was replaced in a session by $\widetilde{\mathsf{ss}}_e$, that session will now derive anything originally derived from ss_e from $\widetilde{\mathsf{ss}}_e$ instead.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_5$ that breaks the IND-1CCA security of KEM_e. $\mathcal{B}_5$ obtains the IND-1CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and the challenge ciphertext $\mathsf{ss}^\star$. In π_c, it uses $\mathsf{pk}^\star$ in the `ClientHello` message. In the server session π_s $\mathcal{B}_5$ sends $\mathsf{ct}^\star$ in the `ServerHello` reply. It also sets $\mathsf{ss}^\star$ as the shared secret ss_e in π_s. If $\mathcal{A}_1$ sends $\mathsf{ct}^\star$ to π_c, $\mathcal{B}_5$ also sets ss_e to $\mathsf{ss}^\star$ in π_c. But if $\mathcal{A}_1$ sends any other $\mathsf{ct}' \neq \mathsf{ct}^\star$ to π_c, $\mathcal{B}_5$ uses its single query to the IND-1CCA decapsulation oracle to obtain π_c's shared secret.

In the end, $\mathcal{A}_1$ terminates it outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_5$ has exactly simulated G_{A4} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_5$ has exactly simulated G_{A5} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A4}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{A5}} + \mathsf{Adv}_{\mathsf{KEM}_e, \mathcal{B}_5}^{\mathsf{IND\text{-}1CCA}} \,.$$

Game A6 (Replacing HS). In this game we replace the handshake secret HS by a uniformly random $\widehat{\mathsf{HS}}$ in π_s. If π_c received the same ct_e that π_s sent, we also make this replacement there. If HS was replaced in a session by $\widehat{\mathsf{HS}}$, that session will now derive anything originally derived from HS from $\widehat{\mathsf{HS}}$ instead.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_6$ that breaks the PRF security of HKDF.Extract in its second

argument. When $\mathcal{B}_6$ needs to compute HS in π_s (or π_c if it received the correct ct_e) it queries its HKDF.Extract challenge oracle on $(\mathrm{dES}, \widetilde{\mathsf{ss}_S})$ and uses the response as HS. If the response was the real output, $\mathcal{B}_6$ has exactly simulated G_{A5} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_6$ has exactly simulated G_{A6} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A5}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{A6}} + \mathsf{Adv}_{\mathsf{HKDF.Extract}, \mathcal{B}_6}^{\mathsf{PRF\text{-}sec}} \,.$$

Game A7 (Replacing CHTS, SHTS and dHS). In this game we replace the handshake traffic secrets CHTS and SHTS and the value of dHS by uniformly random values in π_s. If π_c received the same `ServerHello` and ct_e that π_s sent, we also make these replacements there. If π_c received the same ct_e but not the same `SH`, we replace its dHS by the same $\widetilde{\mathrm{dHS}}$ as set in π_s, but π_c's CHTS and SHTS are set to independent uniformly random values. If dHS was replaced in a session by $\widetilde{\mathrm{dHS}}$, that session will now derive anything originally derived from dHS from $\widetilde{\mathrm{dHS}}$ instead.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_7$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_7$ needs to compute CHTS, SHTS or dHS in π_s (or π_c, if it received the same ct_e that π_s sent) it queries its HKDF.Expand challenge oracle with $\widetilde{\mathrm{HS}}$ and the corresponding label and transcript and uses the responses. If the responses are real values, $\mathcal{B}_7$ has exactly simulated G_{A6} to $\mathcal{A}_1$. If the responses are random values, $\mathcal{B}_7$ has exactly simulated G_{A7} to $\mathcal{A}_1$. Note that if π_c received the same ct_e that π_s sent, but other parts of the `ServerHello` were changed such that π_s and π_c are no longer partnered at stage 2 or 3, the adversary may issue $\mathsf{Reveal}(\pi_c, 2)$ or $\mathsf{Reveal}(\pi_c, 3)$. But since any changes to `SH` make the transcript in π_s and π_c different, the label input to HKDF.Expand is now different for CHTS and SHTS. This means the simulation in $\mathcal{B}_7$ remains good. We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A6}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{A7}} + \mathsf{Adv}_{\mathsf{HKDF.Expand}, \mathcal{B}_7}^{\mathsf{PRF\text{-}sec}} \,.$$

The stage-2 and stage-3 keys CHTS and SHTS are now uniformly random strings independent of anything else in the game. This means that these key have been shown to have `wfs1` security.

Game A8 (Replacing MS). In this game we replace main secret MS by a uniformly random value $\widetilde{\mathrm{MS}}$ in π_s. If π_c is a partner to π_s at stage 3 and, if $\pi.\mathsf{mutualauth} = \mathsf{true}$, received the same ct_C as π_s sent, we replace its MS with the same value. Otherwise, we set MS in π_c to an independent uniformly random value. All values derived from MS use these newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_8$ that breaks the PRF security of HKDF.Extract in its first argument (which we view as the "dual PRF security"). When $\mathcal{B}_8$ needs to compute MS in π_s (or in π_c, if it received the same ct_C that π_s sent, if any, and it is partnered with π_s in stage 3) it queries its HKDF.Extract challenge oracle with ss_C or $\emptyset$ if $\pi.\mathsf{mutualauth} = \mathsf{false}$. It uses the response as MS. If the responses is the real value, $\mathcal{B}_8$ has exactly simulated G_{A7} to $\mathcal{A}_1$. If it is a random value, $\mathcal{B}_8$ has exactly

simulated G_{A8} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A7}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{A8}} + \mathsf{Adv}_{\mathsf{HKDF.Extract}, \mathcal{B}_8}^{\mathsf{dual\text{-}PRF\text{-}sec}} .$$

Game A9 (Replacing SATS, fk_s, fk_c, and CATS). In this game we replace the values SATS, fk_c, fk_s and CATS with uniformly random values in π_s. If π_c is a partner of π_s at stage 4, we also make these replacements there. If it is not, but it received the same ct_C that π_s sent (or none, if $\pi.\mathsf{mutualauth} = \mathsf{false}$), we replace π_c's SATS, fk_c, fk_s and CATS with independent uniformly random values.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_9$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_9$ needs to compute SATS, fk_c, fk_s or CATS in π_s, it queries its HKDF.Expand oracle on the corresponding labels and transcripts. If π_c received the same ct_C that server session sent (or none, if $\pi.\mathsf{mutualauth} = \mathsf{false}$) and they are partnered in stage 4, $\mathcal{B}_9$ does the same in π_c. It uses the responses in those sessions. If the responses are real values, $\mathcal{B}_9$ has exactly simulated G_{A8} to $\mathcal{A}_1$. If the responses are random values, $\mathcal{B}_9$ has exactly simulated G_{A9} to $\mathcal{A}_1$. Note that if π_c received the same ct_C (if any was sent) that π_s sent, but other parts of the transcript were changed such that π_s and π_c are no longer partnered at stage 4, the adversary may issue $\mathsf{Reveal}(\pi_c, 4)$. But since any such changes make the transcript in π_s and π_c different, the label input to HKDF.Expand is now different for SATS. Similarly, if π_c received the same ct_S (if any was sent) that π_s sent, but other parts of the transcript were changed such that π_s and π_c are no longer partnered at stage 5, the adversary may issue $\mathsf{Reveal}(\pi_c, 5)$. But since any such changes make the transcript in π_s and π_c different, the label input to HKDF.Expand is now different for CATS. This means the simulation in $\mathcal{B}_9$ remains good. We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A8}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{A9}} + \mathsf{Adv}_{\mathsf{HKDF.Expand}, \mathcal{B}_9}^{\mathsf{PRF\text{-}sec}} .$$

The stage-4 key SATS and stage-5 key CATS are now a uniformly random string independent of everything else in the game. This means that the stage-4 and stage-5 keys have been shown to have wfs1 security.

Let bad denote the event that G_{A9} maliciously accepts in stage j in the (fresh) tested session without a session partner in stage j. If $\pi.mutualauth = \mathsf{false}$, $j = 4$. Otherwise, $j = 5$.

Game A10 (Identical-until-bad).

This game is identical to game G_{A9}, except that we abort the game if the event bad occurs. Games G_{A9} and G_{A10} are identical-until-bad [5]. Thus,

$$|\Pr[G_{A9} \Rightarrow 1] - \Pr[G_{A10} \Rightarrow 1]| \leq \Pr[G_{A10} \text{ reaches bad}] .$$

In game G_{A9}, all stage keys in the tested session are uniformly random and independent of all messages in the game. The adversary has no possibility to distinguish stage keys anymore. By this game, it can no longer reach bad. Thus:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{A10}} = 0.$$

It remains to bound $\Pr\left[G_{A10} \text{ reaches bad}\right]$.

Game A11 (HMAC forgery). In this game, π_c, if it does not have a session partner in stage 4, rejects upon receiving the `ServerFinished` message. If $\pi_s.\mathsf{mutualauth} = \mathsf{true}$ and π_s does not have a session partner in stage 5, π_s rejects upon receiving the `ClientFinished` message.

Any adversary that behaves differently in G_{A11} compared to G_{A10} can be used to construct an HMAC forger $\mathcal{B}_{10}$. The only way that G_{A10} and G_{A11} behave differently is if G_{A11} rejects a MAC that should have been accepted as valid. When rejecting SF, if no partner to π_c at stage 4 exists, no honest π_s exists with the same session identifier and thus transcript. This means no honest π_s ever created a MAC tag for the transcript that the client verified, and thus it must be a forgery. When rejecting CF, if no partner to π at stage 5 exists, no honest π_c exists with the same session identifier and thus transcript. This means no honest π_c ever created a MAC tag for the transcript that the server verified, and thus it must be a forgery. Concluding:

$$\Pr\left[G_{A10} \text{ reaches bad}\right] \leq \Pr\left[G_{A11} \text{ reaches bad}\right] + 2\,\mathsf{Adv}_{\mathsf{HMAC},\mathcal{B}_{10}}^{\mathsf{EUF\text{-}CMA}}.$$

By the above, the event bad is never reached.

Analysis of game A11. By game G_{A9}, all stage keys are uniformly random and independent of all messages in the game. By game A11, all events bad are rejected. Thus: $\Pr\left[G_{A11} \text{ reaches bad}\right] = 0$.

This concludes case A, yielding:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_A} \leq n_s \left(\mathsf{Adv}_{\mathsf{KEM_s},\mathcal{B}_2}^{\mathsf{IND\text{-}CCA}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_3}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_4}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{KEM_e},\mathcal{B}_5}^{\mathsf{IND\text{-}1CCA}} \right.$$
$$+ \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_6}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_7}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_8}^{\mathsf{dual\text{-}PRF\text{-}sec}}$$
$$\left. + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_9}^{\mathsf{PRF\text{-}sec}} + 2\,\mathsf{Adv}_{\mathsf{HMAC},\mathcal{B}_{10}}^{\mathsf{EUF\text{-}CMA}} \right).$$

Case B: session π has no contributive partner in stage 2, and $\pi.\mathsf{pid}$ is not corrupted before stage i accepted.

In this case, the tested session π does not have a contributive partner in stage 2. This means that stages aiming for wfs1 are out of scope of this case. If $\pi.\mathsf{mutualauth} = \mathsf{false}$, the tested π can be assumed to a client session. Otherwise, it can both be a server or a client session.

We also allow the intended peer V of the tested session π to be corrupted, but not before the tested session accepted. This models forward secrecy: even if the adversary obtains the peer's long-term key, the tested keys should still be indistinguishable.

Allowing Corrupt in this case means that any reductions that replace ss_C or ss_S problematic. However, we show by assumption on the EUF-CMA security of HMAC that no client can be made to maliciously accept at stage 4 and no server session at stage 5. This means that if a client accepts in stage 4, then it has a partner at stage 4, and all prior stages. Similarly, if a server accepts in stage 5, then it has a partner at stage 5, and all prior stages.

This allows us to make the following conclusions. Once stage 4 accepts, all client stages are retroactively authenticated. Once stage 5 accepts, all server stages are retroactively authenticated. By case G_A, all stage keys are indistinguishable, even to an adversary that corrupts any long-term key. This yields retroactive FS security for all stage keys.

Game B1 (Guessing the intended peer). In this game, we attempt to the identity of the peer with which the tested session attempts to connect. If we do not guess this identity V correctly, i.e. this identity $V \neq \pi.\text{pid}$, we abort. This reduces the advantage of $\mathcal{A}_1$ by a factor of the number of users n_u:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_B} \leq n_u \cdot \mathsf{Adv}_{\mathcal{A}_1}^{G_{B1}} .$$

Game B2 (Static KEM). In this game we replace the shared secret ss_S in π, a client session, with a uniformly random $\widetilde{\mathsf{ss}}_S$. In any (server) sessions π' of V that received the same ct_S as was sent by π in the ClientHello message, we replace the value of ss_S with the same $\widetilde{\mathsf{ss}}_S$. All values derived from ss_S in π or the sessions of V that received the same ct_S use the new value $\widetilde{\mathsf{ss}}_S$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{11}$ that breaks the IND-CCA security of KEM_s. $\mathcal{B}_{11}$ obtains the IND-CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and challenge shared secret $\mathsf{ss}^\star$ and gives $\mathsf{pk}^\star$ to $\mathcal{A}_1$. In π, $\mathcal{B}_{11}$ sends $\mathsf{ct}^\star$ in the ClientHello message and uses $\mathsf{ss}^\star$ for ss_S. If $\mathcal{A}_1$ delivers $\mathsf{ct}^\star$ to any π' of V, $\mathcal{B}_{11}$ uses $\mathsf{ss}^\star$ as value for ss_S in π'. If $\mathcal{A}_1$ delivers some other $\mathsf{ct}' \neq \mathsf{ct}^\star$, $\mathcal{B}_{11}$ queries its IND-CCA decapsulation oracle with ct' to obtain the required shared secret. By the definition of case B, we will never need to answer any $\mathsf{Corrupt}(V)$ queries.

In the end, $\mathcal{A}_1$ terminates it outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_{11}$ has exactly simulated G_{B1} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{11}$ has exactly simulated G_{B2} to $\mathcal{A}_1$.

We obtain:
$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{B1}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{B2}} + \mathsf{Adv}_{\mathsf{KEM}_\mathsf{s},\mathcal{B}_{11}}^{\mathsf{IND\text{-}CCA}} .$$

Game B3 (Replacing ES). In this game, we replace the early handshake secret ES by a uniformly random value $\widetilde{\mathsf{ES}}$. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make the same replacement.

Any $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_{12}$ that breaks the PRF security of HKDF.Extract in its first argument as follows. When $\mathcal{B}_{12}$ needs to compute ES in π or any of the sessions of V that received or sent the same ct_S that was sent or received by π, $\mathcal{B}_{12}$ uses its HKDF.Extract challenge oracle on ct_S. It uses the response as ES. If the responses the real values, $\mathcal{B}_{12}$ has exactly simulated G_{B2} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{12}$ has exactly simulated G_{B3} to $\mathcal{A}_1$.

We obtain:
$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{B2}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{B3}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{12}}^{\mathsf{dual\text{-}PRF\text{-}sec}} .$$

Game B4 (Replacing ETS and dES). In this game we replace the values ETS and dES by uniformly random values in π. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make the same replacements. All values derived from dES in π and the π' sessions of V that made the replacements use the new value $\widetilde{\mathrm{dES}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{13}$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_{13}$ needs to compute ETS or dES in π or any of the sessions of V that received or sent the same ct_S that was sent or received by π, it queries its HKDF.Expand challenge oracle with ES and the corresponding label and transcript and uses the responses. If the response was the real shared secret, $\mathcal{B}_{13}$ has exactly simulated G_{B3} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{13}$ has exactly simulated G_{B4} to $\mathcal{A}_1$.

We obtain:

$$\mathsf{Adv}^{G_{B3}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B4}}_{\mathcal{A}_1} +.$$

The stage-1 key ETS is now a uniformly random string independent of anything else in the game. It is, however, not forward-secure.

Game B5 (Replacing HS). In this game we replace the value of HS by a uniformly random value $\widetilde{\mathrm{HS}}$ in π. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make the same replacement. All values derived from HS in π and the π' of V that made the replacement use the new value $\widetilde{\mathrm{HS}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{14}$ that breaks the PRF security of HKDF.Extract in its second argument. When $\mathcal{B}_{14}$ needs to compute HS in π or any of the sessions of V that received or sent the same ct_S that was sent or received by π, it queries its HKDF.Extract challenge oracle with dES and uses the response. If the response was the real shared secret, $\mathcal{B}_{14}$ has exactly simulated G_{B4} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{14}$ has exactly simulated G_{B5} to $\mathcal{A}_1$.

We obtain:

$$\mathsf{Adv}^{G_{B4}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B5}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{dual\text{-}PRF\text{-}sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_{14}} .$$

Game B6 (Replacing CHTS, SHTS and dHS). In this game we replace the values CHTS and SHTS and dHS by uniformly random values in π. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make the same replacements. All values derived from dHS in π and the π' sessions of V that made the replacements use the new value $\widetilde{\mathrm{dHS}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{15}$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_{15}$ needs to compute any of CHTS, SHTS or dHS in π or any of the sessions of V that received or sent the same ct_S that was sent or received by π, it queries its HKDF.Expand challenge oracle on the corresponding label and transcript and uses the responses. If the response was the real shared secret, $\mathcal{B}_{15}$ has exactly simulated G_{B5} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{15}$ has exactly simulated G_{B6} to $\mathcal{A}_1$.

We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{B5}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{B6}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{15}}^{\mathsf{PRF\text{-}sec}}.$$

The stage-2 and stage-3 keys CHTS and SHTS in π are now uniformly random independent from anything else in the game. Thus, they have been shown to have wfs2 security in client sessions. Recall that server sessions aim for wfs1 in this stage, which is out of scope.

Game B7 (Client authentication static KEM). We only play this game if $\pi.\mathsf{mutualauth} = \mathsf{true}$. Otherwise this game is equal to the previous one as there is no reduction in advantage.

We replace the client authentication shared secret ss_C in π, a server, by a uniformly random value $\widetilde{\mathsf{ss}_C}$. If any of V's client sessions π' received the same ct_C as π sent in `ServerKemCiphertext`, we make the same replacement in those π' if they have $\pi'.\mathsf{mutualauth} = \mathsf{true}$. Any value derived from ss_C in a session where it was replaced will now use the replacement value $\widetilde{\mathsf{ss}_C}$. Sending any `ServerKemCiphertext` to π' with $\pi'.\mathsf{mutualauth} = \mathsf{false}$ will simply terminate those sessions at no advantage to the adversary.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{16}$ that breaks the IND-CCA security of $\mathsf{KEM_c}$. $\mathcal{B}_{16}$ obtains the IND-CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and the challenge ciphertext $\mathsf{ss}^\star$. $\mathcal{B}_{16}$ uses $\mathsf{pk}^\star$ in the `ClientCertificate` message sent in V's client sessions. In π, $\mathcal{B}_{16}$ uses $\mathsf{ct}^\star$ in the `ServerKemCiphertext` message and sets $\mathsf{ss}^\star$ as the shared secret ss_C. If $\mathcal{A}_1$ sends $\mathsf{ct}^\star$ to any of V's π', $\mathcal{B}_{16}$ also sets ss_C to $\mathsf{ss}^\star$ in those π'. But if $\mathcal{A}_1$ sends any other $\mathsf{ct}' \neq \mathsf{ct}^\star$ to any of V's π', $\mathcal{B}_{16}$ uses the IND-CCA decapsulation oracle to obtain the appropriate shared secret.

In the end, $\mathcal{A}_1$ terminates it outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_{16}$ has exactly simulated G_{B6} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{16}$ has exactly simulated G_{B7} to $\mathcal{A}_1$.

We obtain:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{B6}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{B7}} + \mathsf{Adv}_{\mathsf{KEM_c},\mathcal{B}_{16}}^{\mathsf{IND\text{-}CCA}}.$$

Game B8 (Replacing MS). In this game we replace the value of main secret MS by a uniformly random value $\widetilde{\mathrm{MS}}$ in π. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make the same replacement. All values derived from MS in π and the π' of V that made the replacement use the new value $\widetilde{\mathrm{MS}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{17}$ that breaks the PRF security of HKDF.Extract in its first argument. When $\mathcal{B}_{17}$ needs to compute MS in π or any of the sessions of V that received or sent the same ct_S that was sent or received by π, it queries its HKDF.Extract challenge oracle and uses the response. In each of these sessions π', if $\pi'.\mathsf{mutualauth} = false$, $\mathcal{B}_{17}$ calls HKDF.Extract with $\emptyset$. If $\pi'.\mathsf{mutualauth} = \mathsf{true}$, $\mathcal{B}_{17}$ calls HKDF.Extract with ss_C. If the response was the real shared secret, $\mathcal{B}_{17}$ has exactly simulated G_{B7} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{17}$ has exactly simulated G_{B8} to $\mathcal{A}_1$.

We obtain:
$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{B7}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{B8}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{17}}^{\mathsf{dual\text{-}PRF\text{-}sec}} .$$

Game B9 (Replacing SATS, fk_s, fk_c and CATS). In this game we replace the application traffic secrets SATS and CATS, and finished keys fk_s and fk_c by uniformly random values in π. Additionally, in any sessions π' of V which either sent or received the same ct_S and ct_C (or no ct_C, if $\pi.\mathsf{mutualauth} = \mathsf{false}$) that were sent or received in π we make the same replacements.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{18}$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_{18}$ needs to compute SATS, CATS, fk_s or fk_c in π or any of the sessions of V that received or sent the same ct_S and ct_C (or no ct_C, if $\pi.\mathsf{mutualauth} = \mathsf{false}$) that were sent or received by π, it queries its HKDF.Expand challenge oracle with MS and the corresponding label and transcript and uses the responses. If the response was the real shared secret, $\mathcal{B}_{18}$ has exactly simulated G_{B8} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{18}$ has exactly simulated G_{B9} to $\mathcal{A}_1$.

We obtain:
$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{B8}} \leq \mathsf{Adv}_{\mathcal{A}_1}^{G_{B9}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{18}}^{\mathsf{PRF\text{-}sec}} .$$

The stage-4 key SATS and stage-5 key CATS in the tested session π are now uniformly random independent from anything else in the game. Thus, they have been shown to have wfs2 security.

Let bad denote the event that G_{B9} maliciously accepts in stage j in the (fresh) tested session without a session partner in stage j. If $\pi.mutualauth = \mathsf{false}$, $j = 4$. Otherwise, $j = 5$.

Game B10 (Identical-until-bad).

This game is identical to game G_{B9}, except that we abort the game if the event bad occurs. Games G_{B9} and G_{B10} are identical-until-bad [5]. Thus,

$$|\Pr\left[G_{B9} \Rightarrow 1\right] - \Pr\left[G_{B10} \Rightarrow 1\right]| \leq \Pr\left[G_{B10} \text{ reaches } \mathsf{bad}\right].$$

In game G_{B9}, all stage keys in the tested session are uniformly random and independent of all messages in the game. The adversary has no possibility to distinguish stage keys anymore. By this game, it can no longer reach bad. Thus:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_{B10}} = 0.$$

It remains to bound $\Pr\left[G_{B10} \text{ reaches } \mathsf{bad}\right]$.

Game B11 (HMAC forgery). In this game, π, if it is a client, rejects upon receiving the `ServerFinished` message. If π is a server and $\pi.\mathsf{mutualauth} = \mathsf{true}$, π rejects upon receiving the `ClientFinished` message.

Any adversary that behaves differently in G_{B11} compared to G_{B10} can be used to construct an HMAC forger $\mathcal{B}_{19}$. The only way that G_{B10} and G_{B11} behave differently is if G_{B11} rejects a MAC that should have been accepted as valid. When rejecting SF, no partner to π at stage 4 exists, so no honest server session π' exists with the same session identifier and thus transcript. No honest π' ever created a MAC tag for the transcript that the client verified, and thus it must be

a forgery. When rejecting CF, no partner to π at stage 5 exists, so no honest client session π' exists with the same session identifier and thus transcript. Concluding:

$$\Pr\left[G_{B10} \text{ reaches bad}\right] \leq \Pr\left[G_{B11} \text{ reaches bad}\right] + 2\,\mathsf{Adv}_{\mathsf{HMAC},\mathcal{B}_{19}}^{\mathsf{EUF\text{-}CMA}}.$$

Since this game rejects all SF messages, the event bad is never reached in client sessions. If $\pi.\mathsf{mutualauth} = \mathsf{true}$, this game also rejects all CF messages. If $\pi.\mathsf{mutualauth} = \mathsf{false}$, rejecting ClientFinished is out of scope as we only aim for wfs1.

Analysis of game B11. By game G_{B9}, all stage keys are uniformly random and independent of all messages in the game. By game B11, all events bad are rejected. Thus: $\Pr\left[G_{B11} \text{ reaches bad}\right] = 0$.

This concludes case G_B, yielding:

$$\mathsf{Adv}_{\mathcal{A}_1}^{G_B} \leq n_u \left(\mathsf{Adv}_{\mathsf{KEM_s},\mathcal{B}_{11}}^{\mathsf{IND\text{-}CCA}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{12}}^{\mathsf{dual\text{-}PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{13}}^{\mathsf{PRF\text{-}sec}} \right.$$
$$+ \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{14}}^{\mathsf{dual\text{-}PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{15}}^{\mathsf{PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{KEM_c},\mathcal{B}_{16}}^{\mathsf{IND\text{-}CCA}}$$
$$\left. + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_{17}}^{\mathsf{dual\text{-}PRF\text{-}sec}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_{18}}^{\mathsf{PRF\text{-}sec}} + 2\,\mathsf{Adv}_{\mathsf{HMAC},\mathcal{B}_{19}}^{\mathsf{EUF\text{-}CMA}} \right).$$