

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

Online Version, April 2, 2024

Peter Schwabe<sup>1,2</sup>, Douglas Stebila<sup>3</sup>, and Thom Wiggers<sup>2</sup>

<sup>1</sup> MPI-SP, Bochum, Germany

<sup>2</sup> Radboud University, Nijmegen, The Netherlands

<sup>3</sup> University of Waterloo, Waterloo, Canada

[peter@cryptojedi.org](mailto:peter@cryptojedi.org), [dstebila@uwaterloo.ca](mailto:dstebila@uwaterloo.ca), [thom@thomwiggers.nl](mailto: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

**Updated results:** For updated results, see Thom Wiggers’ Ph.D. thesis [53], available at <https://wggrs.nl/p/thesis/>.

**Erratum (2022-03):** After publication, we observed an error in the implementation that resulted in some executions using an incorrect algorithm for ephemeral key exchange. This revision contains experimental results with the correct algorithms used in all data collection.

**For further changes, see Appendix C.**

## 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

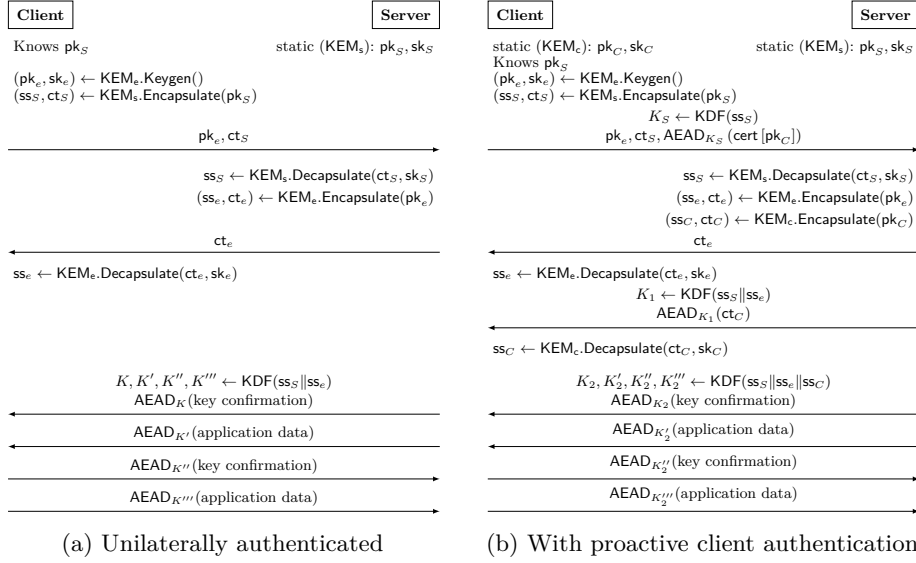


Figure 1: Overview of KEMTLS-PDK variants

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 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.

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
<i>Unilaterally authenticated</i>			
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
<i>Mutually authenticated</i>			
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 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.

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 [31, 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 [ISC:deSSmaWar17, 25, 26]. While [ISC:deSSmaWar17] 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 [30, 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  $(pk, sk)$ ; 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  $\delta$ -correct scheme,  $ss' = ss$  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  $\text{pk}_S$ , obtaining a ciphertext  $\text{ct}_S$  and a shared secret  $\text{ss}_S$ . However, as we assume that the client already has  $\text{pk}_S$ , it can do this *at the start* of the connection and send  $\text{ct}_S$  in a `ClientHello` extension. We plug  $\text{ss}_S$  into the key derivation schedule at the earliest possible stage when deriving the Early Secret ES. Deriving ES from  $\text{ss}_S$  avoids changing the key schedule. It also intuitively makes sense, as data encrypted under traffic keys derived from ES has no forward secrecy or replay protection; just as in TLS 1.3 with PSK and 0-RTT data [43, Sec. 2.3]. The only server who can read a message encrypted under a key derived from ES is the server that has access to  $\text{sk}_S$ ; we consider such keys *implicitly authenticated*. For forward secrecy we also send an ephemeral public key  $\text{pk}_e$  in the `ClientHello` message.

Except for the additional extension transmitting  $\text{ct}_S$ , 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  $\text{ct}_e$  of ephemeral shared secret  $\text{ss}_e$  in the `ServerHello` message. It also indicates in an extension that it has accepted ciphertext  $\text{ct}_S$  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

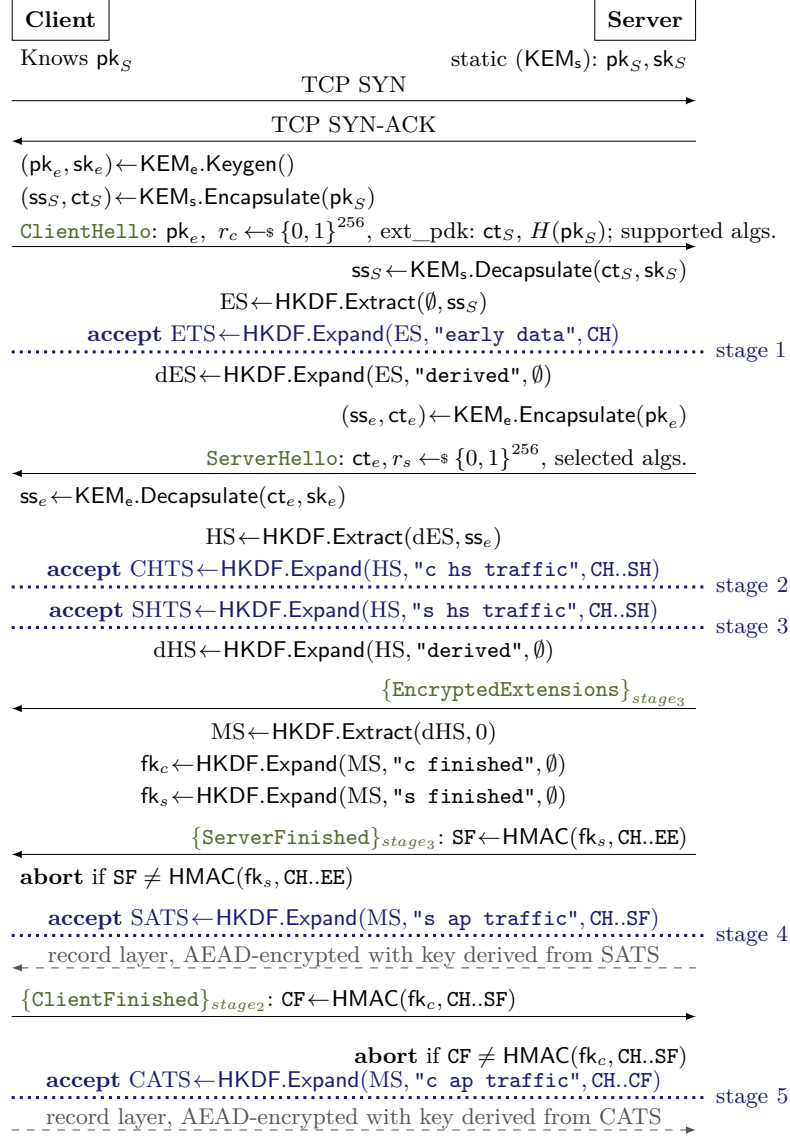


Figure 2: The KEMTLS-PDK handshake for unilateral (server-only) authentication with pre-distributed server public keys

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 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 possesses 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. [22] 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.



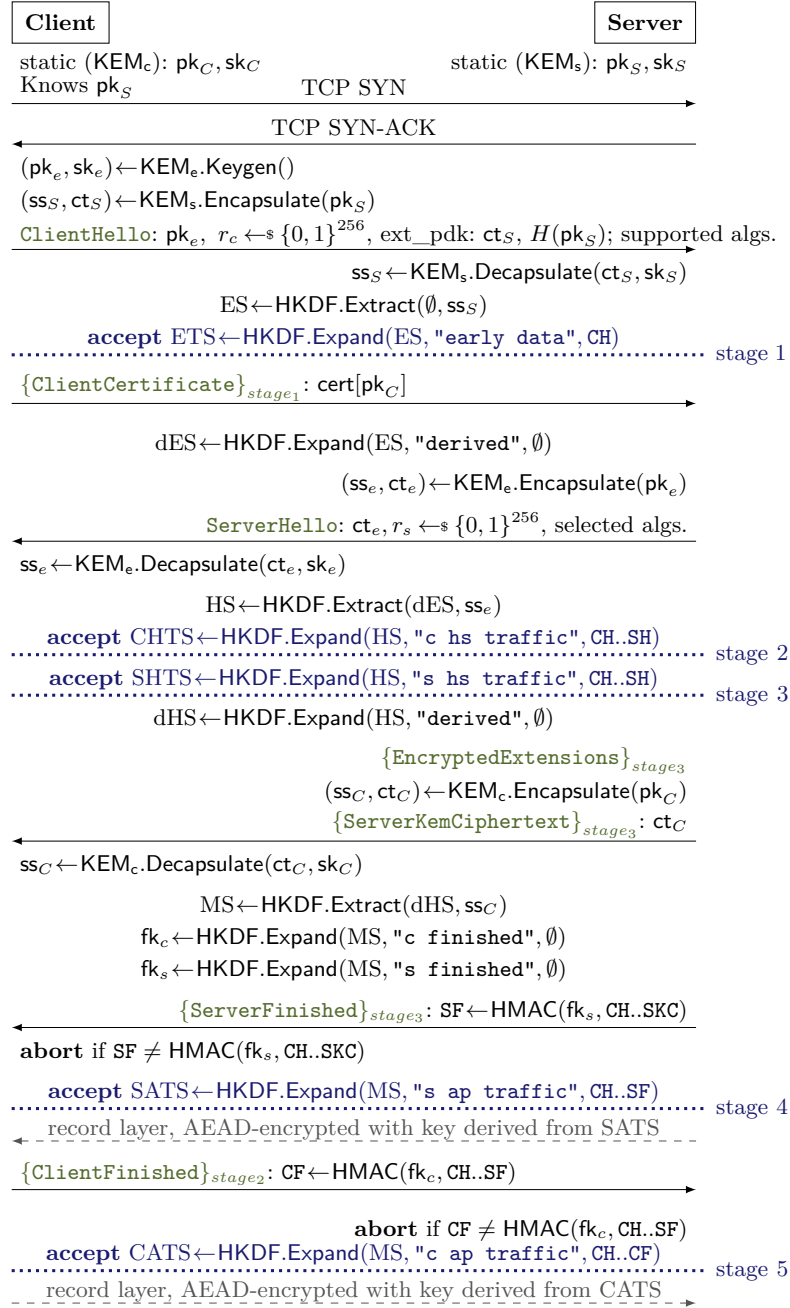


Figure 3: The KEMTLS-PDK handshake for proactive client authentication with pre-distributed server public keys

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. 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 from 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-4 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 trip 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 amounts to a hash of the certificate chain. In TLS 1.3, however, the SCRT message was extended, and may include certificate transparency [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 withhold 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 [17], and NTRU-HPS-2048 [16] 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

Table 2: Sizes (in bytes) of public-key cryptography objects transmitted and cached/pre-distributed in KEMTLS, TLS 1.3 with cached certificates, and KEMTLS-PDK, for server-only and mutual authentication.

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

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, the public keys for Rainbow ( $\approx 160$  KB) and McEliece ( $\approx 260$  KB) are very large, so 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 manageable 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 in turn uses 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 noticeable 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.

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

Unilaterally authenticated		<b>30.9 ms RTT, 1000 Mbps</b>			<b>195.5 ms RTT, 10 Mbps</b>		
		Client sent	Client req. recv.	Server resp. expl. auth.	Client sent	Client req. recv.	Server resp. expl. auth.
KEMTLS	Minimum	117.4	<b>163.4</b>	163.4	448.2	<b>659.1</b>	659.0
	MLWE/MSIS	63.1	<b>94.5</b>	94.4	398.3	<b>595.2</b>	595.2
	NTRU	63.0	<b>94.4</b>	94.4	395.5	<b>592.3</b>	592.3
Cached TLS	TLS 1.3	66.0	<b>97.1</b>	65.9	396.1	<b>592.1</b>	396.0
	Minimum	107.0	<b>138.1</b>	106.9	434.8	<b>630.8</b>	434.7
	MLWE/MSIS	63.6	<b>94.8</b>	63.6	396.7	<b>592.7</b>	396.7
	NTRU	64.5	<b>95.6</b>	64.4	396.3	<b>592.3</b>	396.2
PDK	Minimum	102.5	<b>133.6</b>	102.4	430.8	<b>626.8</b>	430.7
	Kyber	62.8	<b>93.9</b>	62.7	394.7	<b>590.7</b>	394.6
	NTRU	62.8	<b>93.9</b>	62.7	394.7	<b>590.7</b>	394.6
	SABER	62.8	<b>94.0</b>	62.8	394.6	<b>590.6</b>	394.6

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 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 [15].

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

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

Mutually authenticated		<b>30.9 ms RTT, 1000 Mbps</b>			<b>195.5 ms RTT, 10 Mbps</b>		
		Client sent req.	Client recv. resp.	Server expl. auth.	Client sent req.	Client recv. resp.	Server expl. auth.
KEMTLS	Minimum	195.4	<b>226.6</b>	181.8	693.8	<b>890.0</b>	679.8
	MLWE/MSIS	95.0	<b>126.2</b>	94.9	597.7	<b>793.8</b>	597.5
	NTRU	95.0	<b>126.3</b>	94.9	594.7	<b>790.8</b>	594.5
Cached TLS	TLS 1.3	68.8	<b>100.3</b>	66.1	399.1	<b>596.4</b>	396.4
	Minimum	108.7	<b>140.3</b>	107.3	437.0	<b>635.2</b>	435.7
	MLWE/MSIS	64.2	<b>95.8</b>	63.7	400.8	<b>623.6</b>	400.3
	NTRU	66.0	<b>97.8</b>	64.6	397.8	<b>596.7</b>	396.5
PDK	Minimum	129.9	<b>161.0</b>	129.8	464.9	<b>660.9</b>	464.8
	Kyber	63.3	<b>94.4</b>	63.2	399.5	<b>595.5</b>	399.5
	NTRU	63.3	<b>94.5</b>	63.3	396.7	<b>592.7</b>	396.7
	SABER	63.3	<b>94.5</b>	63.3	398.8	<b>594.8</b>	398.7

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 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

The authors gratefully acknowledge insightful discussions with Patrick Towa on the security model, proof, and pre-distributed keys scenario. This work has been supported by the European Research Council through Starting Grant No. 805031 (EPOQUE) and the Natural Sciences and Engineering Research Council of Canada through Discovery grant RGPIN-2016-05146 and a Discovery Accelerator Supplement.



## References

- [1] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. *Classic McEliece*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. “Post-quantum Key Exchange - A New Hope”. In: *USENIX Security 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, Aug. 2016, pp. 327–343.
- [3] Richard L. Barnes, Karthik Bhargavan, Benjamin Lipp, and Christopher A. Wood. *Hybrid Public Key Encryption*. Internet-Draft. <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-08>. Internet Research Task Force, 2021.
- [4] Mihir Bellare and Phillip Rogaway. “Entity Authentication and Key Distribution”. In: *CRYPTO’93*. Ed. by Douglas R. Stinson. Vol. 773. LNCS. Springer, Heidelberg, Aug. 1994, pp. 232–249. DOI: [10.1007/3-540-48329-2\\_21](https://doi.org/10.1007/3-540-48329-2_21).
- [5] Mihir Bellare and Phillip Rogaway. “The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs”. In: *EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. LNCS. Springer, Heidelberg, 2006, pp. 409–426. DOI: [10.1007/11761679\\_25](https://doi.org/10.1007/11761679_25).
- [6] Nina Bindel, Johannes Braun, Luca Gladiator, Tobias Stöckert, and Johannes Wirth. “X.509-Compliant Hybrid Certificates for the Post-Quantum Transition”. In: *Journal of Open Source Software* 4.40 (2019), p. 1606. DOI: [10.21105/joss.01606](https://doi.org/10.21105/joss.01606).
- [7] Nina Bindel, Udyani Herath, Matthew McKague, and Douglas Stebila. “Transitioning to a Quantum-Resistant Public Key Infrastructure”. In: *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*. Ed. by Tanja Lange and Tsuyoshi Takagi. Springer, Heidelberg, 2017, pp. 384–405. DOI: [10.1007/978-3-319-59879-6\\_22](https://doi.org/10.1007/978-3-319-59879-6_22).
- [8] Joseph Birr-Pixton. *A modern TLS library in Rust*. <https://github.com/ctz/rustls> (accessed 2021-04-29).
- [9] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. “Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 553–570. DOI: [10.1109/SP.2015.40](https://doi.org/10.1109/SP.2015.40).
- [10] Matt Braithwaite. *Experimenting with Post-Quantum Cryptography*. Posting on the Google Security Blog. <https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html>. 2016.

- [11] Christina Brzuska. “On the foundations of key exchange”. <https://tuprints.ulb.tu-darmstadt.de/3414/>. PhD thesis. Darmstadt, Germany: Technische Universität Darmstadt, 2013.
- [12] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. “Composability of Bellare-Rogaway key exchange protocols”. In: *ACM CCS 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM Press, Oct. 2011, pp. 51–62. DOI: [10.1145/2046707.2046716](https://doi.org/10.1145/2046707.2046716).
- [13] Ran Canetti and Hugo Krawczyk. “Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels”. In: *EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. LNCS. Springer, Heidelberg, May 2001, pp. 453–474. DOI: [10.1007/3-540-44987-6\\_28](https://doi.org/10.1007/3-540-44987-6_28).
- [14] Sofia Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. “A Tale of Two Models: Formal Verification of KEMTLS via Tamarin”. In: *ESORICS 2022, Part III*. Ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng. Vol. 13556. LNCS. Springer, Heidelberg, Sept. 2022, pp. 63–83. DOI: [10.1007/978-3-031-17143-7\\_4](https://doi.org/10.1007/978-3-031-17143-7_4).
- [15] Sofia Celi and Thom Wiggers. *KEMTLS: Post-quantum TLS without signatures*. Posting on the Cloudflare Blog. <https://blog.cloudflare.com/kemtls-post-quantum-tls-without-signatures/>. 2021.
- [16] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. *NTRU*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [17] Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. *SABER*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [18] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. “Deniable authentication and key exchange”. In: *ACM CCS 2006*. Ed. by Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati. ACM Press, 2006, pp. 400–409. DOI: [10.1145/1180405.1180454](https://doi.org/10.1145/1180405.1180454).
- [19] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias J. Kannwischer, and Jacques Patarin. *Rainbow*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [20] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. “Composability and On-Line Deniability of Authentication”. In: *TCC 2009*. Ed. by Omer Reingold. Vol. 5444. LNCS. Springer, Heidelberg, Mar. 2009, pp. 146–162. DOI: [10.1007/978-3-642-00457-5\\_10](https://doi.org/10.1007/978-3-642-00457-5_10).

- [21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. “A Cryptographic Analysis of the TLS 1.3 Handshake Protocol”. In: *Journal of Cryptology* 34.4 (Oct. 2021), p. 37. DOI: [10.1007/s00145-021-09384-1](https://doi.org/10.1007/s00145-021-09384-1).
- [22] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. “A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates”. In: *ACM CCS 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, Oct. 2015, pp. 1197–1210. DOI: [10.1145/2810103.2813653](https://doi.org/10.1145/2810103.2813653).
- [23] Nir Drucker and Shay Gueron. *Selfie: reflections on TLS 1.3 with PSK*. Cryptology ePrint Archive, Report 2019/347. <https://eprint.iacr.org/2019/347>. 2019.
- [24] Marc Fischlin and Felix Günther. “Multi-Stage Key Exchange and the Case of Google’s QUIC Protocol”. In: *ACM CCS 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM Press, Nov. 2014, pp. 1193–1204. DOI: [10.1145/2660267.2660308](https://doi.org/10.1145/2660267.2660308).
- [25] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. “Practical and post-quantum authenticated key exchange from one-way secure key encapsulation mechanism”. In: *ASIACCS 13*. Ed. by Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng. ACM Press, May 2013, pp. 83–94.
- [26] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. “Strongly Secure Authenticated Key Exchange from Factoring, Codes, and Lattices”. In: *PKC 2012*. Ed. by Marc Fischlin, Johannes Buchmann, and Mark Manulis. Vol. 7293. LNCS. Springer, Heidelberg, May 2012, pp. 467–484. DOI: [10.1007/978-3-642-30057-8\\_28](https://doi.org/10.1007/978-3-642-30057-8_28).
- [27] Felix Günther. “Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols”. PhD thesis. Darmstadt, Germany: Technische Universität Darmstadt, 2018. URL: <https://tuprints.ulb.tu-darmstadt.de/7162>.
- [28] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. *SIKE*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [29] Simon Josefsson. *Storing Certificates in the Domain Name System (DNS)*. RFC 4398. Mar. 2006. DOI: [10.17487/RFC4398](https://doi.org/10.17487/RFC4398).
- [30] Hugo Krawczyk. “Cryptographic Extraction and Key Derivation: The HKDF Scheme”. In: *CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. LNCS. Springer, Heidelberg, Aug. 2010, pp. 631–648. DOI: [10.1007/978-3-642-14623-7\\_34](https://doi.org/10.1007/978-3-642-14623-7_34).
- [31] Hugo Krawczyk. “HMQV: A High-Performance Secure Diffie-Hellman Protocol”. In: *CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. LNCS. Springer, Heidelberg, Aug. 2005, pp. 546–566. DOI: [10.1007/11535218\\_33](https://doi.org/10.1007/11535218_33).

- [32] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: [10.17487/RFC5869](https://doi.org/10.17487/RFC5869).
- [33] Hugo Krawczyk and Hoeteck Wee. “The OPTLS Protocol and TLS 1.3”. In: *EuroS&P 2016*. <https://eprint.iacr.org/2015/978.pdf>. IEEE, 2017.
- [34] Kris Kwiatkowski. *Towards Post-Quantum Cryptography in TLS*. Posting on the Cloudflare Blog. <https://blog.cloudflare.com/towards-post-quantum-cryptography-in-tls/>. 2019.
- [35] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. “Stronger Security of Authenticated Key Exchange”. In: *ProvSec 2007*. Ed. by Willy Susilo, Joseph K. Liu, and Yi Mu. Vol. 4784. LNCS. Springer, Heidelberg, Nov. 2007, pp. 1–16.
- [36] Adam Langley. *CECPQ2*. Posting on the ImperialViolet Blog. <https://www.imperialviolet.org/2018/12/12/cecpq2.html>. 2018.
- [37] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. RFC 6962. June 2013. DOI: [10.17487/RFC6962](https://doi.org/10.17487/RFC6962).
- [38] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. “An Efficient Protocol for Authenticated Key Agreement”. In: *Designs, Codes and Cryptography* 28.2 (2003), pp. 119–134.
- [39] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [40] Christian Paquin, Douglas Stebila, and Goutam Tamvada. “Benchmarking Post-quantum Cryptography in TLS”. In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Heidelberg, 2020, pp. 72–91. DOI: [10.1007/978-3-030-44223-1\\_5](https://doi.org/10.1007/978-3-030-44223-1_5).
- [41] Trevor Perrin. *Noise Protocol Framework*. <https://noiseprotocol.org/noise.html> (accessed 2021-04-29). July 2018.
- [42] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *FALCON*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [43] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: [10.17487/RFC8446](https://doi.org/10.17487/RFC8446).
- [44] Eric Rescorla, Nick Sullivan, and Christopher A. Wood. *Semi-Static Diffie-Hellman Key Establishment for TLS 1.3*. Internet-Draft. <https://tools.ietf.org/html/draft-rescorla-tls-semistatic-dh-02>. Internet Engineering Task Force, 2020.

- [45] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Dr. Carlisle Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960. June 2013. DOI: [10.17487/RFC6960](https://doi.org/10.17487/RFC6960).
- [46] Stefan Santesson and Hannes Tschofenig. *Transport Layer Security (TLS) Cached Information Extension*. RFC 7924. July 2016. DOI: [10.17487/RFC7924](https://doi.org/10.17487/RFC7924).
- [47] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. *CRYSTALS-KYBER*. Tech. rep. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>. National Institute of Standards and Technology, 2020.
- [48] Peter Schwabe, Douglas Stebila, and Thom Wiggers. “Post-Quantum TLS Without Handshake Signatures”. In: *ACM CCS 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 1461–1480. DOI: [10.1145/3372297.3423350](https://doi.org/10.1145/3372297.3423350).
- [49] Peter Schwabe, Douglas Stebila, and Thom Wiggers. *Post-quantum TLS without handshake signatures*. Cryptology ePrint Archive, Report 2020/534. Full version of [48]. Apr. 2021. URL: <https://eprint.iacr.org/2020/534>.
- [50] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. “Post-Quantum Authentication in TLS 1.3: A Performance Study”. In: *NDSS 2020*. The Internet Society, Feb. 2020.
- [51] Douglas Stebila and Michele Mosca. “Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project”. In: *SAC 2016*. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. LNCS. Springer, Heidelberg, Aug. 2016, pp. 14–37. DOI: [10.1007/978-3-319-69453-5\\_2](https://doi.org/10.1007/978-3-319-69453-5_2).
- [52] Erik Sy, Christian Burkert, Hannes Federrath, and Mathias Fischer. “Tracking Users across the Web via TLS Session Resumption”. In: *ACM ACSAC 18*. San Juan, PR, USA: ACM Press, 2018, 289–299. ISBN: 9781450365697. DOI: [10.1145/3274694.3274708](https://doi.org/10.1145/3274694.3274708).
- [53] Thom Wiggers. “Post-Quantum TLS”. PhD thesis. Nijmegen, The Netherlands: Radboud University, Jan. 9, 2024. URL: <https://thomwiggers.nl/publication/thesis/>.

## 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  $\text{pk}_e$  (KEMTLS) to the server in the `ClientHello` message. The server then replies with its key share  $g^y$  (TLS 1.3) or ciphertext  $\text{ct}_e$  (KEMTLS). At this point, the server has the information to derive the handshake shared secret. This shared secret  $\text{ss}_e$  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  $\text{pk}_S$ . 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 possesses 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  $\text{ss}_S$  to the server’s long-term public key  $\text{pk}_S$ . It then sends over the corresponding ciphertext  $\text{ct}_S$  to the server. Only if the server can decapsulate the shared secret from the ciphertext, can it prove possession 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 possession 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  $\text{ss}_S$ . 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.

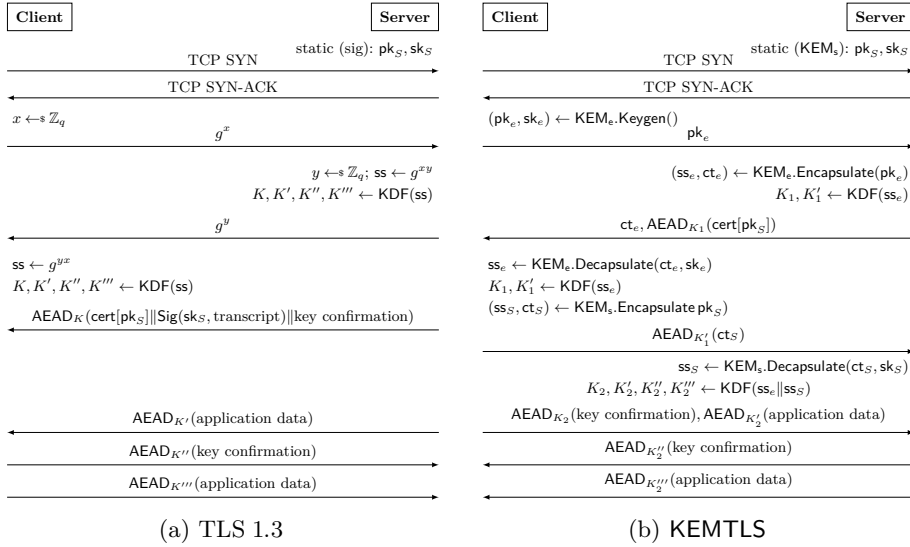


Figure 4: Overview of TLS 1.3 and KEMTLS

## 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  $\mathbf{pk}_U$  and the corresponding private key  $\mathbf{sk}_U$ . Sessions of our protocols are uniquely identified by a label  $\pi \in \mathcal{U} \times \mathbb{N}$ . These are pairs  $(U, n)$  identifying the  $n$ th 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.

- $\text{id} \in \mathcal{U}$ : The identity of the participant that owns this session.
- $\text{pid} \in \mathcal{U} \cup \{*\}$ : The *intended* partner in this session. The identity of the partner may initially be unknown, indicated by  $*$ .
- $\text{role} \in \{\text{initiator}, \text{responder}\}$ .
- $\text{status} \in \{\perp, \text{running}, \text{accepted}, \text{rejected}\}^M$ : The status of each stage.  $\text{status}_i \leftarrow \text{accepted}$  is set when stage  $i$  accepts the  $i$ th stage key. When a key is rejected,  $\text{status}_i \leftarrow \text{rejected}$  and the protocol does not continue.  $\text{status}$  is initialised as  $(\text{running}, \perp^{\times 5})$ .
- $\text{stage} \in \{0, \dots, M\}$ : The last accepted stage. We set  $\text{stage} \leftarrow i$  when  $\text{status}_i$  is set to  $\text{accepted}$ . Initially set to 0.
- $\text{sid} \in (\{0, 1\}^* \cup \{\perp\})^M$ : The session identifier in stage  $i$ . There are initially set to  $\perp$ , and updated when reaching acceptance in that stage.
- $\text{cid} \in (\{0, 1\}^* \cup \{\perp\})^M$ : The stage- $i$  contributive identifier. Initially all  $\perp$ , and updated until reaching acceptance in that stage.
- $\text{key} \in (\mathcal{K} \cup \{\perp\})^M$ : Key established in stage  $i$ .
- $\text{revealed} \in \{\text{true}, \text{false}\}^M$ : Records if  $\text{key}_i$  has been revealed by the adversary. Initially all false.
- $\text{tested} \in \{\text{true}, \text{false}\}^M$ : Records if  $\text{key}_i$  has been tested by the adversary. Initially all false.
- $\text{mutualauth} \in \{\text{true}, \text{false}\}$ : Indicates if mutual authentication will be required.
- $\text{auth} \in \{(u, m) \in \{1, \dots, M, \infty\} \times \{1, \dots, M, \infty\}\}^M$ : Indicates at which stage a stage key is considered to be unilaterally ( $u$ ) or mutually ( $m$ ) authenticated; some may never be authenticated ( $\infty$ ).
- $\text{FS} \in \{0, \text{wfs1}, \text{wfs2}, \text{fs}\}^{M \times M}$ : For  $j \geq i$ ,  $\text{FS}_{i,j}$  indicates the type of forward secrecy expected of stage key  $i$ , assuming stage  $j$  has accepted.
- $\text{use} \in \{\text{internal}, \text{external}\}^M$ : Indicates if a stage- $i$  key is used internally in the key exchange protocol.
- $\text{replay} \in \{\text{nonreplayable}, \text{replayable}\}^M$ : 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  $\pi$ 's element  $X$ .

The *partner* of a session  $\pi$  at stage  $i$  is defined to be the  $\pi'$  where  $\pi.\text{sid}_i = \pi'.\text{sid}_i \neq \perp$  and  $\pi.\text{role} \neq \pi'.\text{role}$ . We define the *contributive partner* similarly, using the contributive identifier  $\text{cid}$ . 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 **Multi-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:

- **NewSession**( $U, V, \text{role}, \text{mutualauth}$ ) creates a new session  $\pi$ , where  $\pi.\text{id} \leftarrow U$ ,  $\pi.\text{pid} \leftarrow V$ ,  $\pi.\text{role} \leftarrow \text{role}$  and  $\pi.\text{mutualauth} \leftarrow \text{mutualauth}$ . Note that possibly  $V = *$ , when  $V$  is left unspecified.
- **Send**( $\pi, m$ ) sends message  $m$  to session  $\pi$ . If sent to a session that has not been created by **NewSession**, this returns  $\perp$ . Additionally, if  $\pi.\text{mutualauth} = \text{false}$ , and  $m$  is **ClientCertificate**, this also returns  $\perp$ . Otherwise, **Send** will operate the protocol on behalf of  $\pi.\text{id}$ , record the updated state and return both the response message and  $\pi.\text{status}_{\pi.\text{stage}}$ .

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

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 **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 **Send**( $\pi, \text{continue}$ ). They will then receive the next protocol message and the status of the key in the current stage. On this call, we also set  $\pi.\text{status}_{\pi.\text{stage}+1} \leftarrow \text{running}$ , except if the current stage is the last one.

If there exists a partner  $\pi'$  of  $\pi$  at stage  $i$  who has been tested ( $\pi'.\text{tested}_i = \text{true}$ ), we set  $\pi.\text{tested} \leftarrow \text{true}$ . If this stage is an internal stage, we also set  $\pi.\text{key} \leftarrow \pi'.\text{key}$ . 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:

- **Reveal**( $\pi, i$ ) gives the adversary session key  $\pi.\text{key}_i$ . It also sets  $\pi.\text{revealed}_i \leftarrow \text{true}$ . For non-existing sessions or stages that have not accepted, returns  $\perp$ .



- **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:

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

### B.3 Specifics of KEMTLS-PDK

In our protocol, the number of states is  $M = 5$ . 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:

```

sid1 = (“ETS”, ServerCertificate, ClientHello),
sid2 = (“CHTS”, ServerCertificate, ClientHello...ServerHello),
sid3 = (“SHTS”, ServerCertificate, ClientHello...ServerHello),
sid4 = (“SATS”, ServerCertificate, ClientHello...ServerFinished),
sid5 = (“CATS”, ServerCertificate, ClientHello...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  $\text{cid}_i$  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  $\text{cid}_1 = (\text{“ETS”}, \text{SCRT}, \emptyset)$  initially. When the client sends or the server receives the **ClientHello** message, we update it to  $\text{cid}_1 = (\text{“ETS”}, \text{SCRT}, \text{CH})$ .

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  $\text{cid}_2 = (\text{“CHTS”}, \text{SCRT}, \text{CH})$  when sending or receiving the **ClientHello** message. At that time we also set  $\text{cid}_3 = (\text{“SHTS”}, \text{SCRT}, \text{CH})$ . If  $\pi.\text{mutualauth} = \text{true}$ , when the **CCRT** message is sent or received we update  $\text{cid}_2$  and  $\text{cid}_3$  by updating the transcripts to  $\text{CH} \dots \text{CCRT}$ . When the **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  $\text{sid}_i$  is set.

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

We set  $\text{auth} = ((4, m), (4, m), (4, m), (4, m), (\infty, m))$ . In both modes of the KEMTLS-PDK protocol, the server is explicitly authenticated with the client accepts stage 4. (Client stage 5 does not achieve explicit authentication because the client-to-server flow is the last flow in the handshake and there is no assurance it has been delivered to the server.) In unilaterally authenticated KEMTLS-PDK, only the server is authenticated, so  $m = \infty$ . When using mutual authentication, i.e.  $\pi.\text{mutualauth} = \text{true}$ , the client is explicitly authenticated when stage  $m = 5$  is accepted.

In client sessions, we set

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

For server sessions, we set

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

where  $f = \text{fs}$  if  $\pi.\text{mutualauth} = \text{true}$  and  $f = \text{wfs1}$  otherwise, and  $f' = \text{wfs2}$  if  $\pi.\text{mutualauth} = \text{true}$  and  $f' = \text{wfs1}$  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,  $\pi.\text{sid}$  and  $\pi'.\text{sid}$  correctly match, where  $\pi$  and  $\pi'$  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  $G_{\text{KE}, \mathcal{A}}^{\text{Match}}$ :*

**Setup** *The challenger generates long-term keypairs  $(\text{pk}_U, \text{sk}_U)$  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  $\pi, \pi'$  be distinct, partnered sessions for which  $\pi.\text{sid}_i = \pi'.\text{sid}_i \neq \perp$  at some stage  $i \in \{1, \dots, M\}$ . The adversary  $\mathcal{A}$  wins the Match security game, denoted  $G_{\text{KE}, \mathcal{A}}^{\text{Match}} = 1$ , if it can falsify any of the following conditions:*

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

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

is negligible in the security parameter.

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

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

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

*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  $\text{KEM}_s$  shared secret and the  $\text{ClientHello}$  message. For stages 2 and 3, the input to the agreed keys are the previous key, messages up to and including  $\text{ServerHello}$  and the ephemeral  $\text{KEM}_e$  shared secret. For final stages 4 and 5, the inputs are the previous keys, messages up to  $\text{ServerFinished}$  and  $\text{ClientFinished}$ , respectively, and, when using mutual authentication, the  $\text{KEM}_c$  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  $\text{KEM}_e$  may fail with probability  $\delta_e$ , the static KEM  $\text{KEM}_s$  may fail with probability  $\delta_s$ . When using mutual authentication, the static KEM  $\text{KEM}_c$  may fail with probability  $\delta_c$ .

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  $\text{sid}_i$  (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,  $\text{cid}_i$  is final and equal to  $\text{sid}_i$  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  $\text{sid}_5$  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  $\text{sid}$  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  $n_s^2 \cdot 2^{-|\text{nonce}|}$ . Here,  $n_s$  is the maximum number of sessions and  $|\text{nonce}| = 256$  is the nonces’ length in bits.  $\square$

### 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  $\pi$ , stage  $i$  is said to be fresh if conditions 1, 2, 3, and at least one of 4, 5, or 6 hold:*

1.  $\pi.\text{revealed}_i = \text{false}$ : the session key has not been revealed.
2. In any partnered sessions  $\pi'$  where  $\pi'.\text{sid}_i = \pi.\text{sid}_i$ ,  $\pi'.\text{revealed} = \text{false}$ : no partnered session at the same stage has been revealed.

3. If stage  $i$  is replayable,  $\text{Corrupt}(\pi.\text{pid})$  is never called.
4. (Weak forward secrecy 1) There exists  $j \geq i$  such that  $\pi.\text{FS}_{i,j} = \text{wfs1}$ ,  $\pi.\text{status}_j = \text{accepted}$ , and there exists a contributive partner at stage  $i$ .
5. (Weak forward secrecy 2) There exists  $j \geq i$  such that  $\pi.\text{FS}_{i,j} = \text{wfs2}$ ,  $\pi.\text{status}_j = \text{accepted}$ , and either (a) there exists a contributive partner at stage  $i$  or (b)  $\text{Corrupt}(\pi.\text{pid})$  was never called.
6. (Forward secrecy) There exists  $j \geq i$  such that  $\pi.\text{FS}_{i,j} = \text{fs}$ ,  $\pi.\text{status}_j = \text{accepted}$ , and either (a) there exists a contributive partner at stage  $i$  or (b)  $\text{Corrupt}(\pi.\text{pid})$  was not called before  $\pi$  accepted stage  $j$ .

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

1. if  $\pi$  is the initiator then  $\pi.\text{status}_u = \text{accepted}$  or if  $\pi$  is a responder then  $\pi.\text{status}_m = \text{accepted}$ ;
2. if stage  $i$  is not replayable, there does not exist a unique partner of  $\pi$  at stage  $i$ ; and
3.  $\text{Corrupt}(\pi.\text{pid})$  was not called before the last stage accepted by  $\pi$  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  $\text{KE}$ , let  $\mathcal{A}$  be a probabilistic adversary.  $\mathcal{A}$  interacts with  $\text{KE}$  via the queries defined in Appendix B.2. The adversary tries to win the following game  $G_{\text{KE}, \mathcal{A}}^{\text{Multi-Stage}}$ :

**Setup** The challenger chooses the test bit  $b \leftarrow_{\$} \{0, 1\}$ . It also generates long-term keys  $(\text{pk}_U, \text{sk}_U)$  for all identities  $U \in \mathcal{U}$ . The public keys  $\text{pk}_U$  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  $G_{\text{KE}, \mathcal{A}}^{\text{Multi-Stage}}$  outputs 1. Otherwise the adversary has lost the game, in which  $G_{\text{KE}, \mathcal{A}}^{\text{Multi-Stage}}$  outputs a uniformly random bit.

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

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

**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*

$$\text{Adv}_{\text{KEMTLS-PDK}, \mathcal{A}}^{\text{Multi-Stage}} \leq \frac{n_s^2}{2^{|\text{nonce}|}} + \text{Adv}_{\text{H}, \mathcal{B}_1}^{\text{COLL}} + n_s \cdot \left( \begin{array}{l} \left( \text{Adv}_{\text{KEM}, \mathcal{B}_2}^{\text{IND-CCA}} + \text{Adv}_{\text{HKDF.Extract}, \mathcal{B}_3}^{\text{PRF-sec}} \right. \\ \left. + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_4}^{\text{PRF-sec}} + \text{Adv}_{\text{KEM}, \mathcal{B}_5}^{\text{IND-ICCA}} \right. \\ \left. + \text{Adv}_{\text{HKDF.Extract}, \mathcal{B}_6}^{\text{PRF-sec}} + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_7}^{\text{PRF-sec}} \right. \\ \left. + \text{Adv}_{\text{dual-PRF-sec}, \mathcal{B}_8}^{\text{HKDF.Extract}} + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_9}^{\text{PRF-sec}} \right. \\ \left. + 2 \text{Adv}_{\text{HMAC}, \mathcal{B}_{10}}^{\text{EUF-CMA}} \right) \\ + 5n_s \cdot \left( \begin{array}{l} \left( \text{Adv}_{\text{KEM}, \mathcal{B}_{11}}^{\text{IND-CCA}} + \text{Adv}_{\text{HKDF.Extract}, \mathcal{B}_{12}}^{\text{dual-PRF-sec}} \right. \\ \left. + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_{13}}^{\text{PRF-sec}} + \text{Adv}_{\text{HKDF.Extract}, \mathcal{B}_{14}}^{\text{dual-PRF-sec}} \right. \\ \left. + \text{Adv}_{\text{KEM}, \mathcal{B}_6}^{\text{IND-CCA}} + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_{15}}^{\text{PRF-sec}} \right. \\ \left. + \text{Adv}_{\text{HKDF.Extract}, \mathcal{B}_{17}}^{\text{dual-PRF-sec}} + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_{18}}^{\text{PRF-sec}} \right. \\ \left. + 2 \text{Adv}_{\text{HMAC}, \mathcal{B}_{19}}^{\text{EUF-CMA}} \right) \end{array} \right).$$

*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:

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

**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  $|\text{nonce}| = 256$ -bit nonces:

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

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:

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

**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  $\text{sid}$ . Only the first one is unencrypted, but the later  $\text{sid}$  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:

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

This restriction of  $\mathcal{A}$  to  $\mathcal{A}_1$  means we can now refer to *the* session  $\pi$  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: **wfs1** and **wfs2**. By rejecting malicious acceptance, we finally show **fs**.

- A. In these games, denoted  $G_A$ , the tested session  $\pi$  has a unique contributive partner in stage 2. This means there exists a session  $\pi'$  such that  $\pi.\text{cid}_2 = \pi'.\text{cid}_2$ .
- B. In these games, denoted  $G_B$ , the tested session  $\pi$  does *not* have a contributive partner in stage 2. In addition,  $\text{Corrupt}(\pi.\text{pid})$  was not called *before* stage  $i$  of  $\pi$  accepted.

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

$$\begin{aligned} \text{Adv}_{\mathcal{A}_1}^{G_3} &\leq \max \left\{ \text{Adv}_{\mathcal{A}_1}^{G_A}, \text{Adv}_{\mathcal{A}_1}^{G_B} \right\} \\ &\leq \text{Adv}_{\mathcal{A}_1}^{G_A} + \text{Adv}_{\mathcal{A}_1}^{G_B}. \end{aligned}$$

### Case A: session $\pi$ has a unique contributive partner in stage 2

In this case, we assume that  $\pi$  has a  $\pi'$  with whom they share  $\pi.\text{cid}_2 = \pi'.\text{cid}_2$ . If the tested session  $\pi$  is a client (initiator) session, then  $\pi.\text{cid}_2 = \pi.\text{sid}_2$  and a partner session at  $\pi'$  also exists.  $\text{sid}_2$  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  $\pi$  has  $\text{role} = \text{responder}$ , 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:  $\text{cid}_2$  includes the client nonce (unique by game 1) and contributive partnering includes roles.

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

**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  $\pi$  at stage 2. As

the challenger guesses correctly with probability  $\frac{1}{n_s}$ , this reduces the advantage of  $\mathcal{A}_1$  as:

$$\text{Adv}_{\mathcal{A}_1}^{G_A} \leq n_s \cdot \text{Adv}_{\mathcal{A}_1}^{G_{A_1}}.$$

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

**Game A2 (Static KEM).** In this game we replace the shared secret  $\text{ss}_S$  encapsulated to  $\text{pk}_S$  by a uniformly random  $\widetilde{\text{ss}}_S$ . We make this replacement in  $\pi_c$  and  $\pi_s$ , and, as this stage is replayable, in any other sessions  $\pi''$  of  $S$  that received  $\text{ct}_S$ .

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  $\text{KEM}_s$ .  $\mathcal{B}_2$  obtains the IND-CCA challenge  $\text{pk}^*$ ,  $\text{ct}^*$  and challenge shared secret  $\text{ss}^*$  and uses  $\text{pk}^*$  as the long-term key  $\text{pk}_S$  of party  $S$ . In  $\pi_c$ ,  $\mathcal{B}_2$  sends  $\text{ct}^*$  in the `ClientHello` message.  $\mathcal{B}_2$  uses  $\text{ss}^*$  for  $\text{ss}_S$  in both  $\pi_c$  and  $\pi_s$ . If  $\mathcal{A}_1$  delivers  $\text{ct}^*$  to some other session  $\pi''$  of party  $S$ ,  $\mathcal{B}_2$  uses  $\text{ss}^*$  as value for  $\text{ss}_S$  in  $\pi''$ . If  $\mathcal{A}_1$  delivers a different  $\text{ct}' \neq \text{ct}^*$  to some other session  $\pi''$  of party  $S$ ,  $\mathcal{B}_2$  queries its IND-CCA decapsulation oracle with  $\text{ct}'$  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  $\text{ss}^*$  was the real shared secret,  $\mathcal{B}_2$  has exactly simulated  $G_{A_1}$  to  $\mathcal{A}_1$ . If it was a random value,  $\mathcal{B}_2$  has exactly simulated  $G_{A_2}$  to  $\mathcal{A}_1$ . We obtain:

$$\text{Adv}_{\mathcal{A}_1}^{G_{A_1}} \leq \text{Adv}_{\mathcal{A}_1}^{G_{A_2}} + \text{Adv}_{\text{KEM}_s, \mathcal{B}_2}^{\text{IND-CCA}}.$$

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

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  $\text{ES}$  in  $\pi$  or  $\pi'$  it queries its `HKDF.Extract` challenge oracle on  $(\emptyset, \widetilde{\text{ss}}_S)$  and uses the response as  $\text{ES}$ . If the response was the real shared secret,  $\mathcal{B}_3$  has exactly simulated  $G_{A_2}$  to  $\mathcal{A}_1$ . If it was a random value,  $\mathcal{B}_3$  has exactly simulated  $G_{A_3}$  to  $\mathcal{A}_1$ . We obtain:

$$\text{Adv}_{\mathcal{A}_1}^{G_{A_2}} \leq \text{Adv}_{\mathcal{A}_1}^{G_{A_3}} + \text{Adv}_{\text{HKDF.Extract}, \mathcal{B}_3}^{\text{PRF-sec}}.$$

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

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  $\text{ETS}$  or  $\text{dES}$  in  $\pi_c$  or  $\pi_s$  it queries its `HKDF.Expand` challenge oracle with  $\widetilde{\text{ES}}$  and the corresponding label and transcript, and uses the responses. If the response was the real output,  $\mathcal{B}_4$  has exactly simulated  $G_{A_3}$  to  $\mathcal{A}_1$ . If it was a random value,  $\mathcal{B}_4$  has exactly simulated  $G_{A_4}$  to  $\mathcal{A}_1$ .

We obtain:

$$\text{Adv}_{\mathcal{A}_1}^{G_{A_3}} \leq \text{Adv}_{\mathcal{A}_1}^{G_{A_4}} + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_4}^{\text{PRF-sec}}.$$



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  $\pi_s$  we replace the ephemeral shared secret  $ss_e$  with a uniformly random  $\widetilde{ss}_e$ . In  $\pi_c$  we replace  $ss_e$  with the same  $\widetilde{ss}_e$ , but only if it received the same  $ct_e$  that  $\pi_s$  sent. If  $ss_e$  was replaced in a session by  $\widetilde{ss}_e$ , that session will now derive anything originally derived from  $ss_e$  from  $\widetilde{ss}_e$  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  $\text{KEM}_e$ .  $\mathcal{B}_5$  obtains the IND-1CCA challenge  $pk^*$ ,  $ct^*$  and the challenge ciphertext  $ss^*$ . In  $\pi_c$ , it uses  $pk^*$  in the `ClientHello` message. In the server session  $\pi_s$   $\mathcal{B}_5$  sends  $ct^*$  in the `ServerHello` reply. It also sets  $ss^*$  as the shared secret  $ss_e$  in  $\pi_s$ . If  $\mathcal{A}_1$  sends  $ct^*$  to  $\pi_c$ ,  $\mathcal{B}_5$  also sets  $ss_e$  to  $ss^*$  in  $\pi_c$ . But if  $\mathcal{A}_1$  sends any other  $ct' \neq ct^*$  to  $\pi_c$ ,  $\mathcal{B}_5$  uses its single query to the IND-1CCA decapsulation oracle to obtain  $\pi_c$ 's shared secret.

In the end,  $\mathcal{A}_1$  terminates it outputs its guess of the uniform bit  $b$ . If  $ss^*$  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:

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

**Game A6 (Replacing HS).** In this game we replace the handshake secret HS by a uniformly random  $\widetilde{HS}$  in  $\pi_s$ . If  $\pi_c$  received the same  $ct_e$  that  $\pi_s$  sent, we also make this replacement there. If HS was replaced in a session by  $\widetilde{HS}$ , that session will now derive anything originally derived from HS from  $\widetilde{HS}$  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  $\pi_s$  (or  $\pi_c$  if it received the correct  $ct_e$ ) it queries its `HKDF.Extract` challenge oracle on  $(\text{dES}, \widetilde{ss}_s)$  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:

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

**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  $\pi_s$ . If  $\pi_c$  received the same `ServerHello` and  $ct_e$  that  $\pi_s$  sent, we also make these replacements there. If  $\pi_c$  received the same  $ct_e$  but not the same SH, we replace its dHS by the same  $\widetilde{dHS}$  as set in  $\pi_s$ , but  $\pi_c$ 's CHTS and SHTS are set to independent uniformly random values. If dHS was replaced in a session by  $\widetilde{dHS}$ , that session will now derive anything originally derived from dHS from  $\widetilde{dHS}$  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  $\pi_s$  (or  $\pi_c$ , if it received the same  $ct_e$  that  $\pi_s$

sent) it queries its `HKDF.Expand` challenge oracle with  $\widetilde{\text{HS}}$  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  $\pi_c$  received the same  $\text{ct}_e$  that  $\pi_s$  sent, but other parts of the `ServerHello` were changed such that  $\pi_s$  and  $\pi_c$  are no longer partnered at stage 2 or 3, the adversary may issue `Reveal`( $\pi_c, 2$ ) or `Reveal`( $\pi_c, 3$ ). But since any changes to `SH` make the transcript in  $\pi_s$  and  $\pi_c$  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:

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

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 keys have been shown to have `wfs1` security.

**Game A8 (Replacing MS).** In this game we replace main secret `MS` by a uniformly random value  $\widetilde{\text{MS}}$  in  $\pi_s$ . If  $\pi_c$  is a partner to  $\pi_s$  at stage 3 and, if  $\pi.\text{mutualauth} = \text{true}$ , received the same  $\text{ct}_C$  as  $\pi_s$  sent, we replace its `MS` with the same value. Otherwise, we set `MS` in  $\pi_c$  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  $\pi_s$  (or in  $\pi_c$ , if it received the same  $\text{ct}_C$  that  $\pi_s$  sent, if any, and it is partnered with  $\pi_s$  in stage 3) it queries its `HKDF.Extract` challenge oracle with  $\text{ss}_C$  or  $\emptyset$  if  $\pi.\text{mutualauth} = \text{false}$ . It uses the response as `MS`. If the response 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:

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

**Game A9 (Replacing SATS,  $\text{fk}_c$ ,  $\text{fk}_s$ , and CATS).** In this game we replace the values `SATS`,  $\text{fk}_c$ ,  $\text{fk}_s$  and `CATS` with uniformly random values in  $\pi_s$ . If  $\pi_c$  is a partner of  $\pi_s$  at stage 4, we also make these replacements there. If it is not, but it received the same  $\text{ct}_C$  that  $\pi_s$  sent (or none, if  $\pi.\text{mutualauth} = \text{false}$ ), we replace  $\pi_c$ 's `SATS`,  $\text{fk}_c$ ,  $\text{fk}_s$  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`,  $\text{fk}_c$ ,  $\text{fk}_s$  or `CATS` in  $\pi_s$ , it queries its `HKDF.Expand` oracle on the corresponding labels and transcripts. If  $\pi_c$  received the same  $\text{ct}_C$  that server session sent (or none, if  $\pi.\text{mutualauth} = \text{false}$ ) and they are partnered in stage 4,  $\mathcal{B}_9$  does the same in  $\pi_c$ . 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  $\pi_c$  received the same  $\text{ct}_C$  (if any was sent) that  $\pi_s$  sent, but other parts of the transcript were

changed such that  $\pi_s$  and  $\pi_c$  are no longer partnered at stage 4, the adversary may issue  $\text{Reveal}(\pi_c, 4)$ . But since any such changes make the transcript in  $\pi_s$  and  $\pi_c$  different, the label input to  $\text{HKDF.Expand}$  is now different for SATS. Similarly, if  $\pi_c$  received the same  $\text{ct}_S$  (if any was sent) that  $\pi_s$  sent, but other parts of the transcript were changed such that  $\pi_s$  and  $\pi_c$  are no longer partnered at stage 5, the adversary may issue  $\text{Reveal}(\pi_c, 5)$ . But since any such changes make the transcript in  $\pi_s$  and  $\pi_c$  different, the label input to  $\text{HKDF.Expand}$  is now different for CATS. This means the simulation in  $\mathcal{B}_9$  remains good. We obtain:

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

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  $\pi.\text{mutualauth} = \text{false}$ ,  $j = 4$ . 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:

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

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

**Game A11 (HMAC forgery)**. In this game,  $\pi_c$ , if it does not have a session partner in stage 4, rejects upon receiving the **ServerFinished** message. If  $\pi_s.\text{mutualauth} = \text{true}$  and  $\pi_s$  does not have a session partner in stage 5,  $\pi_s$  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  $\pi_c$  at stage 4 exists, no honest  $\pi_s$  exists with the same session identifier and thus transcript. This means no honest  $\pi_s$  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  $\pi$  at stage 5 exists, no honest  $\pi_c$  exists with the same session identifier and thus transcript. This means no honest  $\pi_c$  ever created a MAC tag for the transcript that the server verified, and thus it must be a forgery. Concluding:

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

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:  $\Pr[G_{A11} \text{ reaches bad}] = 0$ .

This concludes case A, yielding:

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

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

In this case, the tested session  $\pi$  does not have a contributive partner in stage 2. This means that stages aiming for `wfs1` are out of scope of this case. If  $\pi.\text{mutualauth} = \text{false}$ , the tested  $\pi$  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  $\pi$  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 `ssC` or `ssS` is 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  $n_u$ :

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

**Game B2 (Static KEM).** In this game we replace the shared secret `ssS` in  $\pi$ , a client session, with a uniformly random  $\widetilde{\text{ss}}_S$ . In any (server) sessions  $\pi'$  of  $V$  that received the same `ctS` as was sent by  $\pi$  in the `ClientHello` message, we

replace the value of  $ss_S$  with the same  $\widetilde{ss}_S$ . All values derived from  $ss_S$  in  $\pi$  or the sessions of  $V$  that received the same  $ct_S$  use the new value  $\widetilde{ss}_S$ .

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  $\text{KEM}_s$ .  $\mathcal{B}_{11}$  obtains the IND-CCA challenge  $pk^*$ ,  $ct^*$  and challenge shared secret  $ss^*$  and gives  $pk^*$  to  $\mathcal{A}_1$ . In  $\pi$ ,  $\mathcal{B}_{11}$  sends  $ct^*$  in the `ClientHello` message and uses  $ss^*$  for  $ss_S$ . If  $\mathcal{A}_1$  delivers  $ct^*$  to any  $\pi'$  of  $V$ ,  $\mathcal{B}_{11}$  uses  $ss^*$  as value for  $ss_S$  in  $\pi'$ . If  $\mathcal{A}_1$  delivers some other  $ct' \neq ct^*$ ,  $\mathcal{B}_{11}$  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 `Corrupt(V)` queries.

In the end,  $\mathcal{A}_1$  terminates it outputs its guess of the uniform bit  $b$ . If  $ss^*$  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:

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

**Game B3 (Replacing ES).** In this game, we replace the early handshake secret ES by a uniformly random value  $\widetilde{\text{ES}}$ . Additionally, in any sessions  $\pi'$  of  $V$  which either sent or received the same  $ct_S$  that was sent or received in  $\pi$ , 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  $\pi$  or any of the sessions of  $V$  that received or sent the same  $ct_S$  that was sent or received by  $\pi$ ,  $\mathcal{B}_{12}$  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:

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

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

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  $\pi$  or any of the sessions of  $V$  that received or sent the same  $ct_S$  that was sent or received by  $\pi$ , 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:

$$\text{Adv}_{\mathcal{A}_1}^{G_{B3}} \leq \text{Adv}_{\mathcal{A}_1}^{G_{B4}} + \text{Adv}_{\text{HKDF.Expand}, \mathcal{B}_{13}}^{\text{PRF-sec}}.$$

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

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  $\text{HKDF.Extract}$  in its second argument. When  $\mathcal{B}_{14}$  needs to compute HS in  $\pi$  or any of the sessions of  $V$  that received or sent the same  $\text{ct}_S$  that was sent or received by  $\pi$ , it queries its  $\text{HKDF.Extract}$  challenge oracle with  $\text{dES}$  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:

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

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

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  $\text{HKDF.Expand}$ . When  $\mathcal{B}_{15}$  needs to compute any of CHTS, SHTS or dHS in  $\pi$  or any of the sessions of  $V$  that received or sent the same  $\text{ct}_S$  that was sent or received by  $\pi$ , it queries its  $\text{HKDF.Expand}$  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:

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

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

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

We replace the client authentication shared secret  $\text{ss}_C$  in  $\pi$ , a server, by a uniformly random value  $\widetilde{\text{ss}}_C$ . If any of  $V$ 's client sessions  $\pi'$  received the same  $\text{ct}_C$  as  $\pi$  sent in  $\text{ServerKemCiphertext}$ , we make the same replacement in those  $\pi'$  if they have  $\pi'.\text{mutualauth} = \text{true}$ . Any value derived from  $\text{ss}_C$  in a session where it was replaced will now use the replacement value  $\widetilde{\text{ss}}_C$ . Sending any

`ServerKemCiphertext` to  $\pi'$  with  $\pi'.\text{mutualauth} = \text{false}$  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  $\text{KEM}_c$ .  $\mathcal{B}_{16}$  obtains the IND-CCA challenge  $\text{pk}^*$ ,  $\text{ct}^*$  and the challenge ciphertext  $\text{ss}^*$ .  $\mathcal{B}_{16}$  uses  $\text{pk}^*$  in the `ClientCertificate` message sent in  $V$ 's client sessions. In  $\pi$ ,  $\mathcal{B}_{16}$  uses  $\text{ct}^*$  in the `ServerKemCiphertext` message and sets  $\text{ss}^*$  as the shared secret  $\text{ss}_C$ . If  $\mathcal{A}_1$  sends  $\text{ct}^*$  to any of  $V$ 's  $\pi'$ ,  $\mathcal{B}_{16}$  also sets  $\text{ss}_C$  to  $\text{ss}^*$  in those  $\pi'$ . But if  $\mathcal{A}_1$  sends any other  $\text{ct}' \neq \text{ct}^*$  to any of  $V$ 's  $\pi'$ ,  $\mathcal{B}_{16}$  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  $\text{ss}^*$  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:

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

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

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  $\pi$  or any of the sessions of  $V$  that received or sent the same  $\text{ct}_S$  that was sent or received by  $\pi$ , it queries its `HKDF.Extract` challenge oracle and uses the response. In each of these sessions  $\pi'$ , if  $\pi'.\text{mutualauth} = \text{false}$ ,  $\mathcal{B}_{17}$  calls `HKDF.Extract` with  $\emptyset$ . If  $\pi'.\text{mutualauth} = \text{true}$ ,  $\mathcal{B}_{17}$  calls `HKDF.Extract` with  $\text{ss}_C$ . 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:

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

**Game B9 (Replacing SATS,  $\text{fk}_s$ ,  $\text{fk}_c$  and CATS).** In this game we replace the application traffic secrets SATS and CATS, and finished keys  $\text{fk}_s$  and  $\text{fk}_c$  by uniformly random values in  $\pi$ . Additionally, in any sessions  $\pi'$  of  $V$  which either sent or received the same  $\text{ct}_S$  and  $\text{ct}_C$  (or no  $\text{ct}_C$ , if  $\pi.\text{mutualauth} = \text{false}$ ) that were sent or received in  $\pi$  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,  $\text{fk}_s$  or  $\text{fk}_c$  in  $\pi$  or any of the sessions of  $V$  that received or sent the same  $\text{ct}_S$  and  $\text{ct}_C$  (or no  $\text{ct}_C$ , if  $\pi.\text{mutualauth} = \text{false}$ ) that were sent or received by  $\pi$ , 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:

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

The stage-4 key SATS and stage-5 key CATS in the tested session  $\pi$  are now uniformly random independent from anything else in the game. Thus, they have been shown to have wfs2 security. If  $\pi$  is a server session, and  $\pi.\text{mutualauth} = \text{false}$ , wfs2 security of SATS is out of scope.

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  $\pi.\text{mutualauth} = \text{false}$ ,  $j = 4$ . 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,

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

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:

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

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

**Game B11 (HMAC forgery).** In this game,  $\pi$ , if it is a client, rejects upon receiving the **ServerFinished** message. If  $\pi$  is a server and  $\pi.\text{mutualauth} = \text{true}$ ,  $\pi$  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  $\pi$  at stage 4 exists, so no honest server session  $\pi'$  exists with the same session identifier and thus transcript. No honest  $\pi'$  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  $\pi$  at stage 5 exists, so no honest client session  $\pi'$  exists with the same session identifier and thus transcript. Concluding:

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

Since this game rejects all **SF** messages, the event **bad** is never reached in client sessions. If  $\pi.\text{mutualauth} = \text{true}$ , this game also rejects all **CF** messages. If  $\pi.\text{mutualauth} = \text{false}$ , 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:  $\Pr[G_{B11} \text{ reaches bad}] = 0$ .



This concludes case  $G_B$ , yielding:

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

## C Changelog

Since the published version, the following things have been changed:

**2022-03** : After publication, we observed an error in the implementation that resulted in some executions using an incorrect algorithm for ephemeral key exchange. This revision contains experimental results with the correct algorithms used in all data collection. Finally, this revision contains corrected properties for the security of KEMTLS-PDK, based on “A Tale of Two Models: Formal Verification of KEMTLS via Tamarin” [14].

**2024-03** : We fixed several typos in our paper and the proof. It also fixes the input to the computation of the `ServerFinished` HMAC; thanks to Noemi Terzo for pointing out this bug.

**Ph.D. thesis** : Thom Wiggers’ Ph.D. thesis [53] contains greatly extended and updated benchmark results using NIST round 3 submissions and security levels I, III, and V, as well as a proof that was integrated with the proof of KEMTLS.