ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Post-Quantum KEM–based TLS with Pre-Shared Keys

Master Thesis

Radwa Sherif Abdelbar

September 8, 2021

Advisors: Prof. Dr. Kenny Paterson
Dr. Felix Günther
Dr. Patrick Towa

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

For Sherif and Salwa,
my heroes

For my Geddu Farid,
who would have been proud

Abstract

The KEMTLS protocol was introduced in 2020 as a post-quantum, signature-free alternative to TLS 1.3's full, Diffie–Hellman-based handshake. In this thesis, we present KEMTLS-PSK, a resumption handshake for KEMTLS, which aims to replace the PSK-(EC)DHE resumption handshake in TLS 1.3 while providing similar security guarantees and optimizing for the number of round-trip times (RTTs) spent in the handshake. Our design of KEMTLS-PSK depends on a pre-shared key (PSK) established in a previous full handshake or through some out-of-band mechanism. It allows for 0-RTT keys to be established so that the initiator (client) can send encrypted application data with its first flight of messages. It relies on an ephemeral key encapsulation mechanism (KEM) to provide forward secrecy for non-0-RTT stages of the protocol. In addition, we design a multi-stage security model which captures security properties such as implicit/explicit authentication, forward secrecy and replayability for every stage key established in the face of an adversary in full control of communication between parties. We prove our proposed protocol secure and show that it provides security guarantees comparable to that of the TLS 1.3 PSK-(EC)DHE handshake.

Acknowledgments

I thank my advisors, Dr. Felix Günther and Dr. Patrick Towa for their patient guidance, their generosity in sharing their knowledge and insights with me and their empathy towards my struggles throughout the past six months. Felix and Patrick, it has truly been a pleasure working with you.

My gratitude extends to Prof. Kenny Paterson for teaching the Applied Cryptography class and making it so fun and accessible that I thought I could do an entire thesis on it and, of course, for giving me the opportunity to work on this thesis.

It feels almost absurd to try to thank my parents for everything they do for me. I must, however, try. Mummy and Pappy, I am a bundle of nerves most of the time, but I manage to navigate this life powered by your love and your insurmountable faith in my ability to achieve things. I would have quit this degree after my first semester if it were not for your encouragement and unconditional support and I would not have emerged with my sanity intact if it were not for our daily video calls. I love you both dearly and you will always be my heroes.

I thank God every day for my brother, the smarter, kinder and funnier sibling. I am beyond proud of you and will always be your biggest fan.

I am eternally grateful to my grandmothers for constantly mentioning me in their prayers, to Uncle A for being both a cool uncle and a reliable source of wisdom for 25 years and counting and to all the members of my extended family who made time to call/text their niece/cousin doing a master's degree in I-don't-know-what in Switzerland.

I am not the world's most sociable person, but I have somehow been blessed with wonderful friends. Of the friends I met in Switzerland I would like to thank K W for being my Swiss guardian angel and having my back for the past three years, D F and F C and R Z, the best flatmates to go through an apocalypse with, and J D S for listening to me whine about how difficult ETH is for two years.

Of my friends in Egypt, I want to thank the following: N K for metaphorically "holding my hands" through the past six years. M A S for the steady supply of cat memes and for checking on me when I am stuck in my bubble. N K (another one) and S M, my "angry feminist" support system. May we always upset the patriarchy. M A for always finding a way to be there for me. My bookclub friends and our unhealthy obsession with Nasser and the politics of the 1960s for giving me things other than computer science to think about occasionally.

I am somehow grateful to the city of Zurich for all the memories, both good and bad, and for the enlightening irony of being miserable in one of the world's "happiest" cities and to Cairo, the love-hate of my life, for setting the bar so high in some aspects and so low in others.

Last but not least, I am grateful for the bizarre comedic genius of Radio Kafr El Sheikh El Habiba for helping me appreciate the farcical elements of human existence and making me laugh during some really dull times.

Contents

Chapter 1

Introduction

The Transport Layer Security (TLS) protocol is one of the most widely-deployed cryptographic protocols. It is perhaps best known for being the security layer in the HTTPS protocol which is used to secure web communications, though it is also used in other applications such as in securing email communications in the STARTTLS [1] and MTA-STS [2] protocols, or alongside SRTP [3] in securing voice over IP.

The TLS protocol consists of a *handshake protocol*, which negotiates cryptographic algorithms and parameters, establishes symmetric session keys and (mutually) authenticates the communicating parties, and a *record protocol*, which uses the established session keys to provide confidentiality and integrity for application data. TLS 1.3 [4], the latest version of TLS, relies on ephemeral (elliptic-curve) Diffie-Hellman key exchange ((EC)DHE) to establish symmetric session keys and on RSA or elliptic-curve signatures for authentication.

TLS in a post-quantum world. The potential widespread use of quantum computers poses a threat to the TLS protocol. The cryptographic algorithms used in the TLS handshake rely on the intractability of some computational problems, which are not efficiently solvable via conventional computers, but are efficiently solvable by quantum computers using Shor's algorithm [5]. For instance, the Diffie-Hellman key exchange relies on the discrete logarithm problem, while RSA relies on the factorization of large primes.

There has been a concentrated effort in the past five years to design quantum-resistant variants of the TLS protocol. These effort were mainly focused on finding quantum-resistant alternatives to the Diffie-Hellman key exchange. One example is the TLS 1.2 variant designed by Bos et al. [6] which introduces a key exchange mechanism based on the *ring learning with errors (R-LWE)* problem combined with traditional authentication using RSA or el-

liptic curve signatures. Based on this work, Alkim et al. [7] developed New Hope, a R-LWE-based key exchange mechanism.

In 2016, Google ran an experiment [8] on their Canary Chrome browser where they initialized the TLS 1.2 protocol using the Combined Elliptic Curve and Post-Quantum 1 (CECPQ1) cipher suite. New Hope was used as the post-quantum algorithm, while the elliptic curve X25519 key exchange algorithm is used as a fallback in case New Hope turns out to have security faults. This was followed by CECPQ2 [9] in 2018 which still used X25519 as a traditional algorithm but combined it with NTRU-HRSS another lattice-based post-quantum key exchange algorithm.

Another example is the Open Quantum Safe (OQS) initiative [10], which includes contributions research from AWS, Microsoft and IBM. It aims, among other goals, to integrate quantum-resistant cryptography into the TLS ecosystem by modifying the widely-used OpenSSL library.

KEM-based TLS. The aforementioned efforts to transition TLS to post-quantum cryptography were mainly concerned with finding quantum-resistant key exchange mechanisms to replace (EC)DHE, less so with quantum-resistant authentication. This is because the confidentiality of recorded past sessions can be compromised in the future if a powerful enough quantum computer is built, whereas authentication is a time-limited property which cannot be broken retroactively. However, in a post-quantum future, finding quantum-resistant authentication mechanisms for TLS would be imperative.

In 2017, the National Institute of Standards and Technology of the U.S. Department of Commerce (NIST) started a process of collecting and standardizing quantum-resistant cryptographic algorithms. Their goal is to add one or more digital signatures, public-key encryption algorithms or key encapsulation mechanisms to their cryptographic standards. The status report [11] on the algorithms which made the second round of the standardization process shows that 16 out of the 26 algorithms were key encapsulation mechanisms (KEMs).

In addition, analyzing the performance of these algorithms [12] shows that quantum-resistant signatures have significantly larger public key and signature sizes than the public key and ciphertext sizes of key encapsulation mechanisms (KEMs). Therefore, while it might seem intuitive to replace the RSA or elliptic-curve signatures in TLS 1.3 with post-quantum signatures, in practice this would seriously harm the efficiency of the communication.

With this perspective in mind, Schwabe, Stebila and Wiggers introduce a signature-free, KEM-based variant of TLS, which they call KEMTLS [13]. Their design draws from the OPTLS handshake which aims to rid the TLS handshake of signatures. They note, however, that OPTLS relies on non-

interactive key exchange (NIKE) algorithms and that efficient quantum-resistant NIKE's are a rarity while post-quantum KEMs are relatively more efficient. Therefore, they opt for a KEM-based design at the cost of an extra RTT. The KEMTLS handshake relies on KEMs both for authentication and key exchange. Their results show that post-quantum KEMTLS outperforms TLS 1.3 initialized with post-quantum signature algorithms in terms of both bandwidth consumption and computation time.

Resumption and 0-RTT in TLS 1.3. Recent optimizations in the implementation of cryptographic primitives has lead to network latency being the bottleneck of key exchange protocols over the internet. The speed of light will remain a definitive lower bound on the time required for a message to travel back and forth between two parties, the so-called *round-trip time* or *RTT* for short.

Therefore, reducing the number of round trips required before the first application message can be sent has become a major goal in key exchange protocol design in recent years. One notable example is Google's QUIC protocol [14] which allows for the establishment of a zero round-trip time (0-RTT) key. This key is then used to send encrypted "early" application data sent with the very first key exchange message.

This adoption of 0-RTT keys by QUIC encouraged a similar design decision in the TLS 1.3 protocol. The TLS 1.3 protocol has a resumption handshake which assumes that the client and server have established a pre-shared key (PSK) either in a prior full handshake or through an out-of-band mechanism. This PSK is used to derive a 0-RTT key to encrypt early application data. The TLS 1.3 resumption handshake comes in two variants: PSK-only, which relies solely on the PSK to derive session keys, and PSK-(EC)DHE, which uses (EC)DHE key shares to provide forward secrecy for non-0-RTT stages. The introduction of 0-RTT speeds up connections in cases where a client is communicating to a server it has previously connected to, which is the case in a significant percentage of TLS connections. For example, in 2017 Cloudfare estimated that about 40% of their TLS connections are resumed connections and, thus, would benefit significantly from 0-RTT mode [15].

Contribution. The KEMTLS design in [13] focuses on specifying and proving the security of a full 1-RTT handshake without specifying a resumption handshake. It is obvious that, in a post-quantum setting, the TLS 1.3 resumption handshake cannot be combined with KEMTLS as is since it depends on the non-quantum-resistant (EC)DHE for forward secrecy. Therefore, if the ultimate goal is to widely deploy KEMTLS on the Internet to replace TLS 1.3 with minimal disruption to the current TLS ecosystem, and while providing similar security guarantees, a KEMTLS resumption hand-

shake analogous to the PSK/PSK-(EC)DHE handshake must be designed and proven secure.

This thesis aims to fill this gap by providing a resumption handshake for KEMTLS. Our three main contributions are as follows:

- We design a resumption handshake for the KEMTLS protocol which, analogously to the TLS 1.3 resumption handshake, relies on the existence of a pre-shared key established in a previous communication or out of band. Our proposed handshake aims to remain faithful to the full 1-RTT KEMTLS handshake in that it relies solely on KEMs for ephemeral key exchange in order to provide forward secrecy. We also aim to make use of the pre-shared key to derive 0-RTT keys which enable the client to send encrypted early data with the first flight of messages. We call our proposed handshake KEMTLS-PSK and treat it as a separate protocol throughout this text for clarity.
- We propose a multi-stage key exchange security model [16], based on prior models for TLS 1.3 [17] and KEMTLS [13], to analyze the security of KEMTLS-PSK. Our model captures security properties such as key indistinguishability, implicit and explicit authentication, two different levels of forward secrecy and replayability. We define the security game of our model in pseudo-code to avoid ambiguity.
- We analyze the security of KEMTLS-PSK via a game-playing security proof and we show that it provides security guarantees which are comparable to the TLS 1.3 PSK-(EC)DHE handshake. In particular, we show that all non-0-RTT stages enjoy some level of forward secrecy and implicit, as well as retroactive explicit, authentication.

Organization. In the next chapter, we introduce some basic notions which we refer to throughout the text. In Chapter 3, we discuss the TLS 1.3 and KEMTLS protocols in detail. The main contribution of the thesis starts from Chapter 4 where we introduce our KEMTLS-PSK handshake. Then we lay out the security model we use to analyze the protocol in Chapter 5. The security analysis itself is detailed in Chapter 6. We conclude and propose future work in Chapter 7.

Chapter 2

Preliminaries

In this chapter, we present a summary of the main cryptographic primitives used in this thesis as well as the security notions associated with each primitive.

2.1 Key Encapsulation Mechanism (KEM)

A Key Encapsulation Mechanism (KEM) is an asymmetric cryptographic primitive used to derive a shared secret between two parties. It consists of a triplet of algorithms $(\mathsf{KGen}, \mathsf{Encap}, \mathsf{Decap})$, defined as follows:

- $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KGen}()$. A probabilistic algorithm that generates a public key and secret key.
- $(\mathsf{ss}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encap}(\mathsf{pk})$. A probabilistic algorithm that takes as input the public key and outputs and a shared secret ss and a ciphertext ct that encapsulates the shared secret.
- $\mathsf{ss}'/\perp \leftarrow \mathsf{Decap}(\mathsf{ct}, \mathsf{sk})$. An algorithm that takes as input a ciphertext and a secret key and outputs a shared secret or $\perp$.

Correctness. A KEM is δ-correct if for all key pairs $(\mathsf{pk}, \mathsf{sk})$: if $(\mathsf{ss}, \mathsf{ct}) \xleftarrow{\$} \mathsf{Encap}(\mathsf{pk})$ and $\mathsf{ss}' \leftarrow \mathsf{Decap}(\mathsf{ct}, \mathsf{sk})$ then $\mathsf{ss} = \mathsf{ss}'$ with probability 1-δ.

Security. The main security notion for KEMs is indistinguishability, either under chosen plaintext attack (IND-CPA), or under chosen ciphertext attack (IND-CCA), which gives the adversary access to a decapsulation oracle. A special variant of IND-CCA is IND-1CCA in which the adversary is allowed to make only one decapsulation query. Figure 2.1 shows the security experiment in detail.

$G^{\text{IND-CPA/IND-CCA}}_{\text{KEM},\mathcal{A}}$

$(\text{pk}^\star,\ \text{sk}^\star) \xleftarrow{\$} \text{KEM.KGen}()$
$b \xleftarrow{\$} \{0,1\}$
$(\text{ss}_0^\star,\ \text{ct}^\star) \xleftarrow{\$} \text{KEM.Encap}(\text{pk}^\star)$
$\text{ss}_1^\star \xleftarrow{\$} \mathcal{K}$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_b}(\text{pk}^\star, \text{sk}^\star, \text{ss}_b^\star)$
return $b' == b$

Oracle $\mathcal{O}(ct)$ for IND-CPA

return $\perp$

Oracle $\mathcal{O}(ct)$ for IND-CCA

if $\text{ct} \neq \text{ct}^\star$:
    return $\text{KEM.Decap}(\text{sk}^\star,\ ct)$
else:
    return $\perp$

Figure 2.1: Security game for IND-CPA and IND-CCA for KEMs.

2.2 (H)MAC

A message authentication code (MAC) is a symmetric cryptographic primitive that attests the authenticity of a message using a private key. It consists of a triplet of algorithms = $(\text{KGen},\ \text{Tag},\ \text{Vfy})$, defined as follows:

- $k \xleftarrow{\$} \text{KGen}()$: a probabilistic algorithm that generates a cryptographic key.
- $t \leftarrow \text{Tag}(k,\ m)$: a deterministic algorithm that takes as input a cryptographic key and a message of arbitrary length and outputs a fixed-length tag on the message.
- $v \xleftarrow{\$} \text{Vfy}(k,\ m,\ t)$: a deterministic algorithm which takes as input a cryptographic key, a message and a tag outputs a value $v \in \{0,1\}$ indicating whether t is the correct tag for message m under key k.

Correctness. A MAC is correct if for all keys k and all messages m: if $t \leftarrow \text{Tag}(k,\ m)$, then $\text{Vfy}(k,\ m,\ t) = 1$.

Security. The security notion we adopt for MAC schemes in this thesis is that of existential unforgeability under chosen message attack (EUF-CMA). An adversary without access to the secret key cannot forge a MAC tag on a new message of its choosing, even if it is given access to a tag oracle. The detailed security game is presented in Figure 2.2.

$G^{\text{EUF-CMA}}_{\text{MAC},\mathcal{A}}$

$k \xleftarrow{\$} \text{KGen}()$
$M \leftarrow \{\}$
$(m,t) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}$
return $(m \notin M \wedge \text{Vfy}(k,m,t) == 1)$

Oracle $\mathcal{O}(m)$

$M \leftarrow M \cup \{m\}$
return $\text{MAC}(k,m)$

Figure 2.2: EUF-CMA security game for MAC schemes.

In this text, we will concern ourselves with a specific type of MACs, namely HMACs. An HMAC [18] is based on a cryptographic hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ and a cryptographic key $k \in \{0,1\}^\lambda$. The computation is defined as $\mathsf{HMAC}(k,m) := \mathsf{H}((k \oplus opad)||\mathsf{H}((k \oplus ipad)||m))$ where $opad$ and $ipad$ are two λ-bit padding values.

2.3 HKDF

A key derivation function (KDF) is a cryptographic algorithm which derives a pseudo-random secret key from some input keying material. This input keying material usually has high, non-uniformly distributed entropy and is not necessarily secret, meaning the adversary could have partial knowledge about it.

The HKDF scheme [19] is an *extract-then-expand* HMAC-based key derivation function. The $\mathsf{HKDF.Extract}(salt, IKM)$ takes as input a non-secret random $salt$ and some input keying material IKM. It extracts entropy from the IKM and outputs a short cryptographically strong pseudo-random key (PRK). The $\mathsf{HKDF.Expand}(PRK, info, length)$ takes as input the PRK, some optional context $info$ and the $length$ of the output in octets. It expands the PRK to the desired length.

2.4 PRF and Dual PRF Security

Pseudo-random function (PRF) security is a property which indicates that the output of a keyed cryptographic function is indistinguishable from random. In our security analysis we rely on the HKDF.Expand function being PRF-secure.

Let $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ be a cryptographic function with key space $\mathcal{K}$, input space $\mathcal{X}$ and output space $\mathcal{Y}$ and let $Funcs[\mathcal{X},\mathcal{Y}]$ be the set of all truly random functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. Figure 2.3 shows the PRF security game in detail.

$\underline{G^{\mathsf{PRF}}_{F,\mathcal{A}}}$

18 $b \xleftarrow{\$} \{0,1\}$
19 $K \xleftarrow{\$} \mathcal{K}$
20 $f \xleftarrow{\$} Funcs[\mathcal{X},\mathcal{Y}]$
21 $b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_b}$
22 return $b == b'$

$\underline{\text{Oracle } \mathcal{O}_b(x)}$

23 if $b == 0$:
24 return $F(K,x)$
25 else:
26 return $f(x)$

Figure 2.3: PRF security game.

The advantage of the adversary $\mathcal{A}$ against the PRF security game is defined as follows:

$$\mathsf{Adv}^{\mathsf{PRF}}_{F,\mathcal{A}} := \left| \Pr\left[G^{\mathsf{PRF}}_{F,\mathcal{A}} = 1\right] - \frac{1}{2} \right|.$$

Dual PRF security. In our analysis, we rely on the dual-PRF security of the HKDF.Extract function. We define a function $F^{\mathsf{swap}}(K, x) := F(x, K)$. The dual-PRF security of F is the PRF security of F^{swap}.

$$\mathsf{Adv}^{\mathsf{dual\text{-}PRF}}_{F,\mathcal{A}} := \mathsf{Adv}^{\mathsf{PRF}}_{F^{\mathsf{swap}},\mathcal{A}}.$$

Chapter 3

Background

The two main protocols from which we draw in our design of KEMTLS-PSK are the TLS 1.3 protocol as specified in RFC 8446 [4] and the KEMTLS protocol by Schwabe, Stebila and Wiggers [13]. In this chapter, we discuss these two protocols in detail and highlight the aspects which are most relevant to our work.

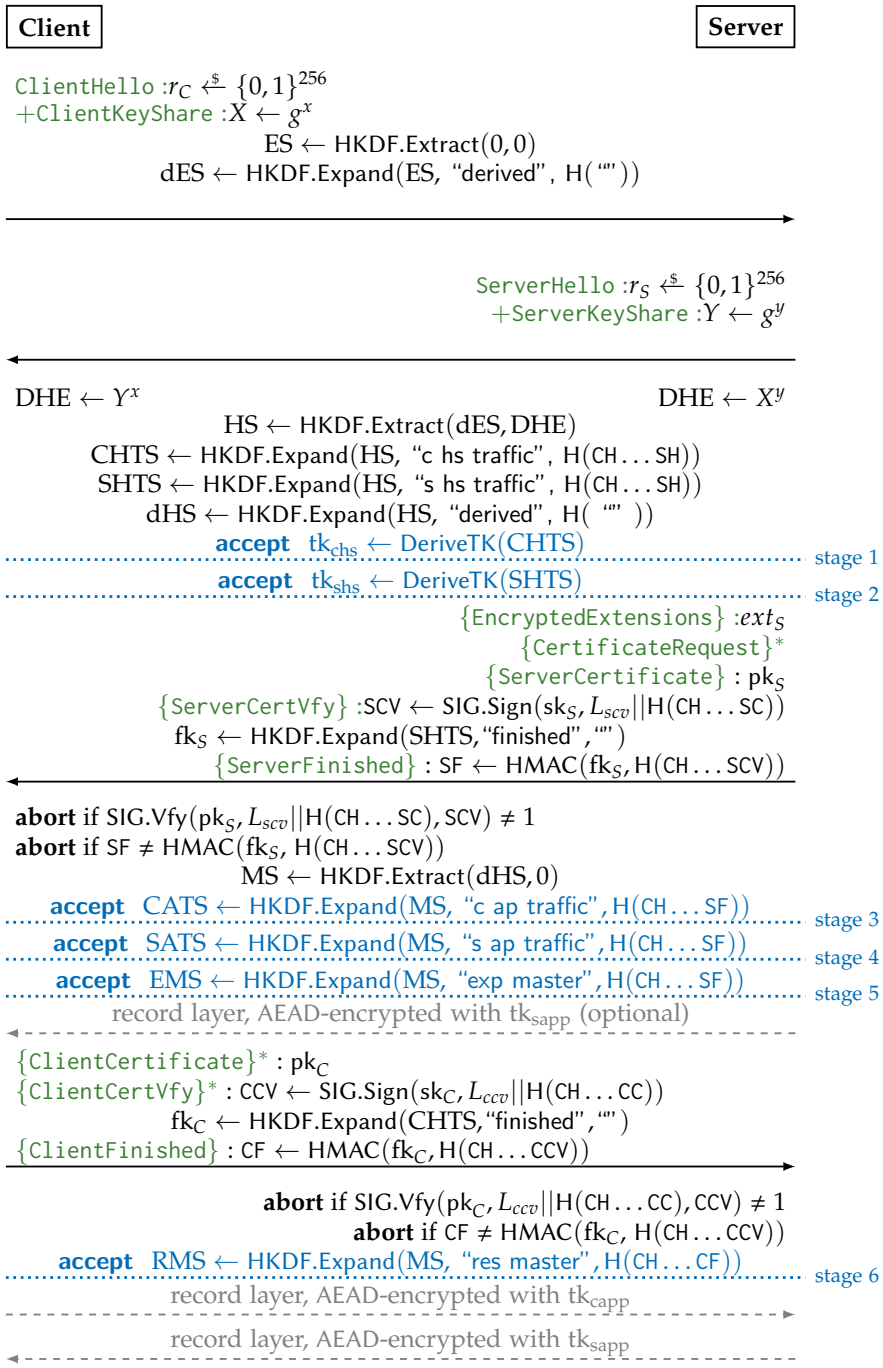
3.1 TLS 1.3

In this section, we lay out the TLS 1.3 handshake in detail. Figure 3.1 shows the full 1-RTT handshake in detail and Figure 3.2 shows the resumption handshake in its two variants: PSK-only, in which all keys are derived solely from the pre-shared key, and PSK-(EC)DHE, in which ephemeral key shares are mixed into the key schedule to provide forward secrecy.

The handshake consists of two main phases: the *key exchange phase* and the *authentication phase*. The *key exchange phase* establishes a number of shared secrets between client and server which are used to encrypt early data (in PSK-only and PSK-(EC)DHE modes), subsequent handshake messages or application data. In the *authentication phase* in the full 1-RTT mode, the server (and optionally the client) authenticates itself to its peer by sending its certificate signed by a trusted certificate authority (CA) and a signature over the transcript using its long-term secret key. In both mdoes, the client and server also exchange MACs over the protocol transcript to ensure they have an identical view of the protocol.

3.1.1 The Key Exchange Phase

This phase of the protocol negotiates cryptographic parameters used by both client and server for the rest of the protocol. It also establishes an ephemeral



Labels:

L_{scv} = "TLS 1.3, server CertificateVerify", L_{ccv} = "TLS 1.3, client CertificateVerify"

DeriveTK:

DeriveTK(Secret) = (HKDF.Expand(Secret, "key", H(""), L_k), HKDF.Expand(Secret, "iv", H(""), L_{iv})) = (key, iv)

Figure 3.1: TLS 1.3 Full 1-RTT Handshake. Legend in Table 3.1

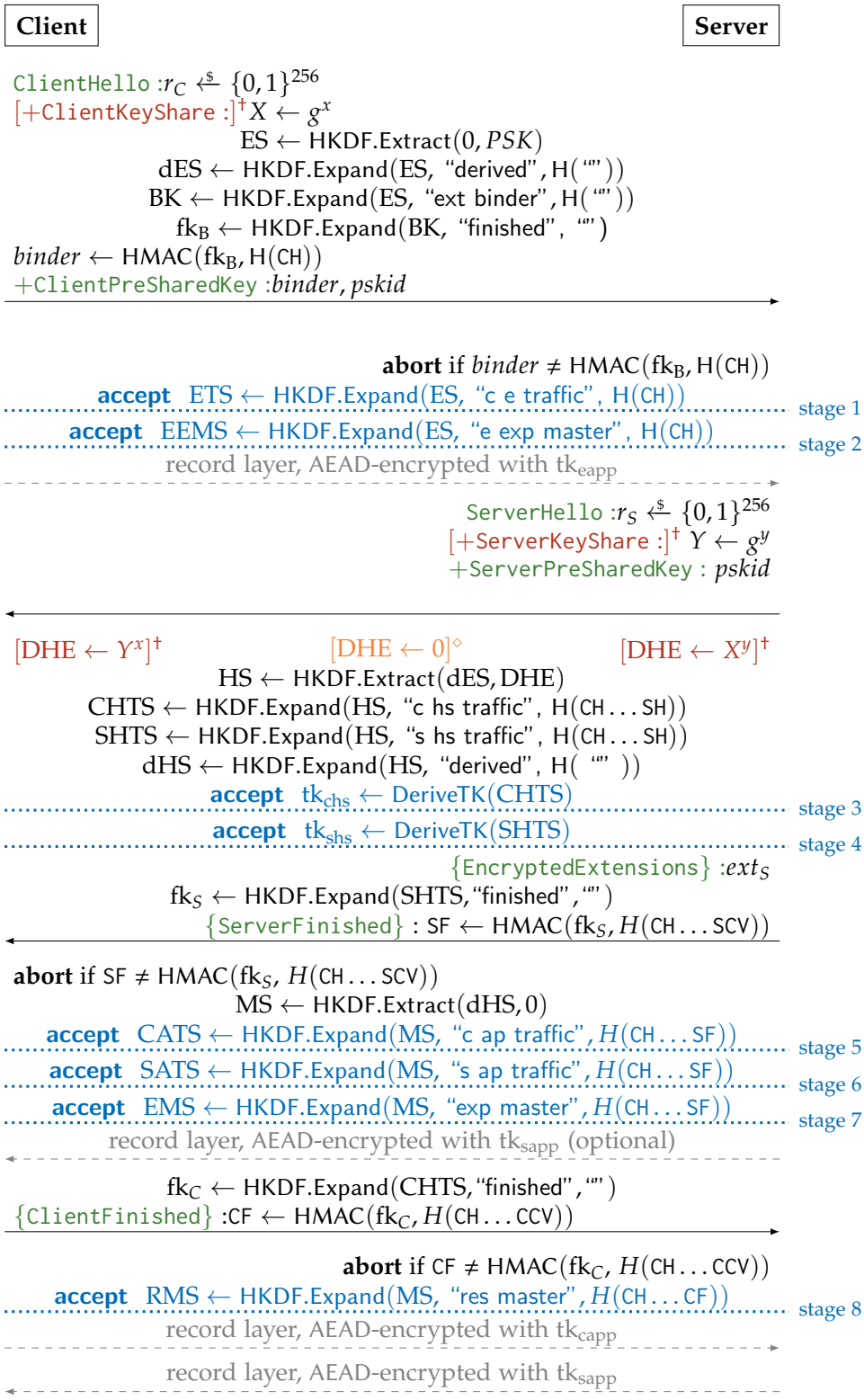


Figure 3.2: TLS 1.3 Pre-Shared Key Mode. Legend in Table 3.1

shared secret called the *handshake secret* (HS), which is used to derive the traffic keys used to encrypt the rest of the handshake.

ClientHello. The key exchange phase starts with the `ClientHello` message which contains a random nonce r_C and a list of cryptographic algorithms and parameters supported by the client. In the full 1-RTT and PSK-(EC)DHE modes, it contains a list of (EC)DHE groups supported by the client and (EC)DHE ephemeral key shares for some or all of them.

Optionally, if both parties have previously established pre-shared keys, the `ClientHello` also contains a `PreSharedKey` extension with a list of pre-shared key identities that the client is willing to negotiate with the server and a list of *binder* values which authenticate every PSK to the server. To compute each *binder* value the key schedule derives a binder key BK from each PSK and another key fk_B from BK. Then the *binder* value is computed as an `HMAC` using fk_B over the `ClientHello` message up until the `PreSharedKey` extension pre-shared key identities field. Along with this extension the client must provide a `PreSharedKeyExchangeModes` extension indicating the PSK handshake modes it supports: PSK-only or PSK-(EC)DHE.

In a PSK handshake the client is also allowed to send early application data with the `ClientHello` message. For that is must add the `EarlyDataIndication` extension. In order to encrypt the early data, it derives an early secret ES from the PSK corresponding to the first identity in the list of PSK identities and from that an early traffic secret ETS from which the encryption keys are derived.

ServerHello. When the server receives a `ClientHello` message, it responds with its `ServerHello` message, which contains a nonce r_S and the server's choice among the cryptographic algorithms and parameters offered by the client.

If the client offers a PSK handshake, the server could choose to accept it and must then choose among the offered PSK identities and modes and send its choice with the `ServerHello`. In case it declines the PSK handshake or it chooses the PSK-(EC)DHE mode, it must also send an ephemeral (EC)DHE key share for the (EC)DHE group it selected.

At the end of this phase, both parties can compute handshake secret HS which is derived via a series of `HKDF` computations from the ephemeral (EC)DHE shared secret (in full 1-RTT mode), the PSK (in PSK-only mode) or both (in PSK-(EC)DHE) mode. From HS we get the client and server handshake traffic secrets, CHTS and SHTS respectively, and then the client and server handshake traffic keys, tk_{chs} and tk_{shs} respectively , which are used to encrypt the rest of the handshake.

3.1.2 Authentication Phase

This phase authenticates the server to the client and (optionally) also authenticates the client to the server. All client and server messages of this phase are encrypted under tk_{chs} and tk_{shs} respectively.

ServerCertificate. In the full 1-RTT handshake, the server authenticates itself to the client by sending the `ServerCertificate` containing its public key in the same flight as the `ServerHello`. To demonstrate knowledge of its secret key, it signs a hash of the protocol transcript up to this point and sends this signature as the `ServerCertVfy` message. The server could also ask the client to authenticate itself by sending its own certificate via the `CertificateRequest` message.

ServerFinished. Regardless of the handshake mode, the server sends a `ServerFinished` message to the client. It derives the server finished key fk_S from SHTS and uses it to compute the `ServerFinished` message as a MAC tag over the transcript of the protocol. Thus the server explicitly demonstrates to the client its knowledge of the shared secrets established in the key exchange phase and ensures that both parties have an identical view of the protocol so far.

The Master Secret. Now the server can compute the master secret MS and, from that, the client and server application traffic secrets, CATS and SATS respectively. The server can already derive the server application traffic key tk_{sapp} and send encrypted application data with its first flight of messages, thus achieving a 0.5-RTT handshake.

ClientCertificate and ClientFinished. When the client receives the server's first flight of messages, it verifies the signature and MAC tag, then it mirrors the server's behavior. It sends its `ClientCertificate` if the server had requested it and a `ClientCertVfy` which is a signature over a hash of the transcript. It derives the client finished key fk_C from CHTS and uses it to compute a MAC tag over the transcript hash which it sends to the server as the `ClientFinished` message.

After the server verifies the signature and MAC, both parties can compute a resumption master secret RMS which they use to derive a new PSK.

3.1.3 The New Session Ticket Message

After the handshake, the server can send one or more `NewSessionTicket` messages. Each message contains a unique `ticket_nonce` which can be used to derive a new PSK from the resumption master secret as follows:

MSG: Y	message MSG containing Y
+MSG	message sent as an extension to the previous message
{MSG}	message sent encrypted with tk_{chs}/tk_{shs}
MSG*	message sent only in the case of client authentication
[...]†	in PSK-(EC)DHE mode
[...]◇	in PSK-only mode

Table 3.1: Legend for TLS 1.3 diagrams

$PSK \leftarrow$ HKDF.Expand(RMS, "resumption" ||ticket_nonce). It also contains an opaque ticket value to be used as a PSK identity.

3.2 KEMTLS

In this section, we discuss the KEMTLS handshake in detail and, when appropriate, note analogies to the TLS 1.3 handshake. The handshake consists of three main phases: the *ephemeral key exchange* phase, the *implicitly authenticated key exchange* phase and the *key confirmation/explicit authentication* phase.

3.2.1 The Ephemeral Key Exchange Phase

This phase is analogous to the key exchange phase in TLS 1.3. The client and server negotiate cryptographic algorithms and parameters to be used for the rest of the handshake and establish a shared *handshake secret* HS, which is used to encrypt subsequent handshake messages. The main difference from the TLS 1.3 handshake is that, instead of (EC)DHE, the KEMTLS handshake uses a KEM encapsulation-decapsulation sequence to establish the shared secret.

ClientHello. The handshake starts with the ClientHello message which contains a random nonce r_C and a list of cryptographic algorithms and parameters supported by the client. It also sends one or more ephemeral KEM public keys pk_e.

ServerHello. The server responds with the ServerHello message containing its own random nonce r_S and its choice among the algorithms and parameters offered by the client. It performs an encapsulation against pk_e which outputs an ephemeral shared secret ss_e and a ciphertext ct_e. It sends ct_e as part of the ServerHello.

Once the client decapsulates ct_e, both parties have now established an unauthenticated ephemeral shared secret ss_e analogous to the ephemeral (EC)DHE secret in the TLS 1.3 handshake. Both parties can now derive the handshake

secret HS via a series of HKDF computations from ss_e. Then both parties derive the client and server handshake traffic secrets, CHTS and SHTS respectively, from HS and, from those, the client and server handshake traffic keys, tk_{chs} and tk_{shs} respectively, which are used to encrypt subsequent handshake messages.

3.2.2 The Implicitly Authenticated Key Exchange Phase

Similar to the TLS 1.3 handshake, the goal of this phase is for the server to authenticate itself to the client (and possibly the reverse) by sending a certificate containing its public key. However, instead of sending a signature over the transcript to prove knowledge of the secret key, the server decapsulates an encapsulation made against its public key. Hence, the authentication in this phase is *implicit*, which means that the client is certain that only the intended server knows the derived keys, but the server has not actively demonstrated knowledge of these keys.

ServerCertificate and ClientKemCiphertext. The server sends the `ServerCertificate` containing its long-term KEM public key pk_S in the same flight as the `ServerHello`. The client encapsulates against pk_S outputting a shared secret ss_S and a ciphertext ct_S. It sends the ciphertext in the `ClientKemCiphertext`.

The server uses its secret key sk_S to decapsulate ct_S and retrieve ss_S. Now both parties have established an implicitly authenticated shared secret ss_S. Both parties can now derive from ss_S the authenticated handshake secret AHS and from that the client and server authenticated handshake traffic secrets, CAHTS and SAHTS respectively, which are used to encrypt subsequent handshake messages.

ClientFinished. The client derives the client finished key fk_C from CHTS and uses it to compute the `ClientFinished` message as a MAC tag over the transcript so far for key confirmation. It sends it to the server in the same flight as the `ClientKemCiphertext`.

The Master Secret. Through a series of HKDF derivations from AHS, both parties derive the master secret MS. At this point, the client can derive the client application traffic secret CATS and, from that, the client application traffic key tk_{capp} and use it to encrypt application data and send it to the server. Thus, the client achieves a 1-RTT handshake.

3.2.3 The Key Confirmation/Explicit Authentication Phase

So far the server is implicitly authenticated to the client, meaning that the client is certain that only the intended peer could know the established

shared secret. The aim of this phase, however, is for the server to actively demonstrate that it is live and participating in the handshake.

ServerFinished. The server derives the server finished key fk_S from SHTS and uses it to compute a MAC tag over the hash of the transcript so far. The client verifies that MAC to explicitly authenticate the server. The server can also derive the server application traffic secret SATS and, from that, a server application traffic key tk_{sapp} and use it to encrypt and send application data in the same message flight, thus, achieving a 1.5-RTT handshake for the server.

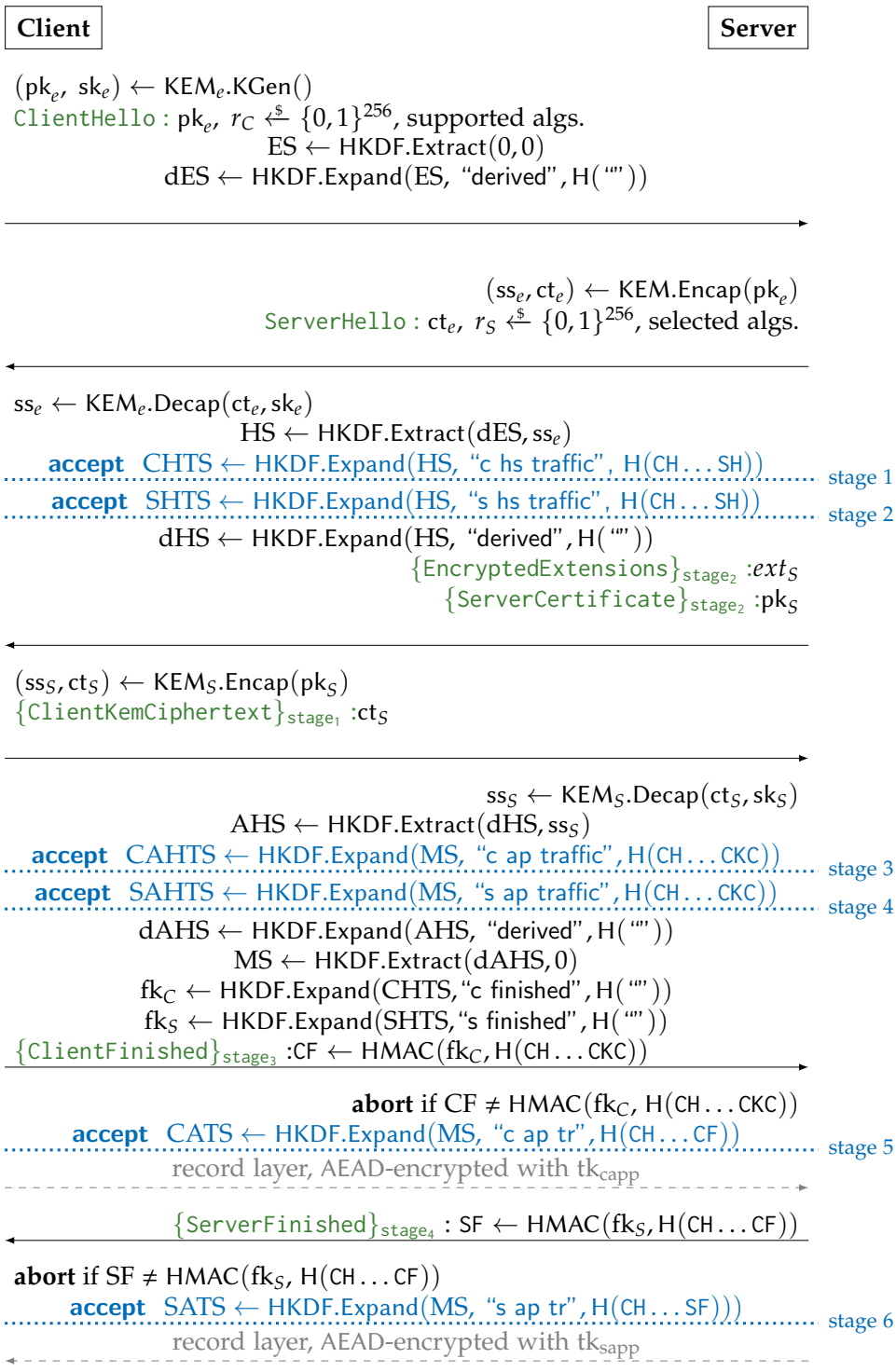


Figure 3.3: KEMTLS Full Handshake

Chapter 4

KEMTLS-PSK

From the previous chapter we note that KEMTLS does not specify a resumption handshake. Therefore, in this chapter, we introduce a resumption handshake for the KEMTLS protocol which depends on a PSK established in a previous handshake or using some out-of-band mechanism.

Our goals is to design a handshake which is analogous to the TLS 1.3 handshake in PSK-(EC)DHE mode and offers similar features such as forward secrecy, authentication and the possibility to send early data encrypted with 0-RTT keys. We depart from the TLS 1.3 key schedule, also used by KEMTLS in its full handshake mode, in favor or a more key compact schedule. To avoid confusion, we therefore treat our handshake as a separate protocol which we call KEMTLS-PSK throughout this text. Figure 4.1 shows the KEMTLS-PSK handshake in detail.

The handshake has two phases: the *implicitly authenticated ephemeral key exchange* phase and the *key confirmation/explicit authentication* phase. We detail both of them below.

4.1 The Implicitly Authenticated Ephemeral Key Exchange Phase

The aim of this phase is to establish a (implicitly authenticated) shared secret between the communicating parties while also including an ephemeral value in the exchange to ensure forward secrecy.

ClientHello. The handshake starts with the `ClientHello` message which includes a nonce r_C, a list of cryptographic algorithms and parameters supported by the client and a list of ephemeral KEM public keys pk_e. The PSK negotiation is similar to that of the TLS 1.3 PSK/PSK-(EC)DHE handshake. The client sends a list of pre-shared key identities that it is willing

to negotiate with the server in the PreSharedKey extension. It derives, via a series of HKDF computations, a binder key BK from each PSK and, from that, a finished key fk_B. It computes a *binder* value as a MAC tag over the ClientHello up until the identities list in the PreSharedKey extension. We omit the PreSharedKeyExchangeModes extension and, therefore, offer no PSK-only mode in our handshake.

The client could send an EarlyDataIndication extension and send 0-RTT data with the ClientHello. This data is encrypted using the early application traffic key tk_{eapp}, which is derived via a series of HKDF computations from the PSK corresponding to the first identity in the PreSharedKey extension. tk_{eapp} is implicitly authenticated but offers no forward secrecy since its derivation involves PSK and no ephemeral value.

ServerHello. The server verifies the *binder* value then responds with the ServerHello message which contains a nonce r_S, its choice among the cryptographic algorithms and parameters offered by the client and its choice of PSK. It computes an encapsulation against pk_e which outputs the ephemeral shared secret ss_e and a ciphertext ct_e. It sends ct_e with the ServerHello. From the server's perspective, ss_e is implicitly authenticated since the *binder* value ensures that pk_e originated from the peer which knows PSK. From the client's perspective, however, ss_e is unauthenticated until it receives the ServerFinished since, until then, it has no guarantees about the identity of the party which computed ct_e.

The Master Secret. Both parties can now compute the master secret MS. From MS they derive the client and server handshake traffic secrets, CHTS and SHTS respectively, which are used to derive encryption keys for subsequent handshake messages. Note that all subsequent secrets in the protocol are also derived via HKDF computations from MS. Since PSK and ss_e are the only two sources of randomness in the protocol, we do not derive the handshake secret HS or the authenticated handshake secret AHS present in the full KEMTLS handshake. Since it is derived from PSK and the ephemeral shared secret ss_e, MS and all subsequent secrets derived from it are at least implicitly authenticated and offer some level of forward secrecy (we detail level of forward secrecy in Chapter 5), which is analogous to TLS 1.3 PSK-(EC)DHE where non-0-RTT stages are forward secret.

4.2 Key Confirmation/Explicit Authentication Phase

The aim of this phase is to *explicitly* authenticate each party to its peer, meaning that both client and server would be certain that they are communicating with an honest and live partner. This is achieved by exchanging MAC tags

over the protocol transcript computed using keys derived from their shared master secret MS.

ServerFinished. From MS the server derives the server finished key fk_S which it uses to compute the `ServerFinished` as a MAC tag over the transcript of the protocol so far. It sends it in the same flight as the `ServerHello`. Once the client verifies the `ServerHello`, the server, which had been unauthenticated in stages 3 and 4, is explicitly authenticated.

The server can now derive the server application traffic secret SATS from MS and, from that, the server application traffic key tk_{sapp}. It can now send application data encrypted with tk_{sapp} in the same flight.

ClientFinished. The client computes the `ClientFinished` in the same manner that the server computed the `ServerFinished`. It derives the client finished key fk_C from MS and uses it to compute a MAC tag over the protocol transcript. Once the server verified the `ClientFinished`, the client, which had previously been only implicitly authenticated, is explicitly authenticated. The client can now send application data by deriving the client application traffic secret CATS then the client application traffic key tk_{capp} which it uses to encrypt application data.

Both parties then compute the resumption master secret RMS which can be used to derive pre-shared keys associated with this handshake in a manner similar to the TLS 1.3 handshake. We refer the reader to Section 3.1.3 for more details on how this derivation process takes place.

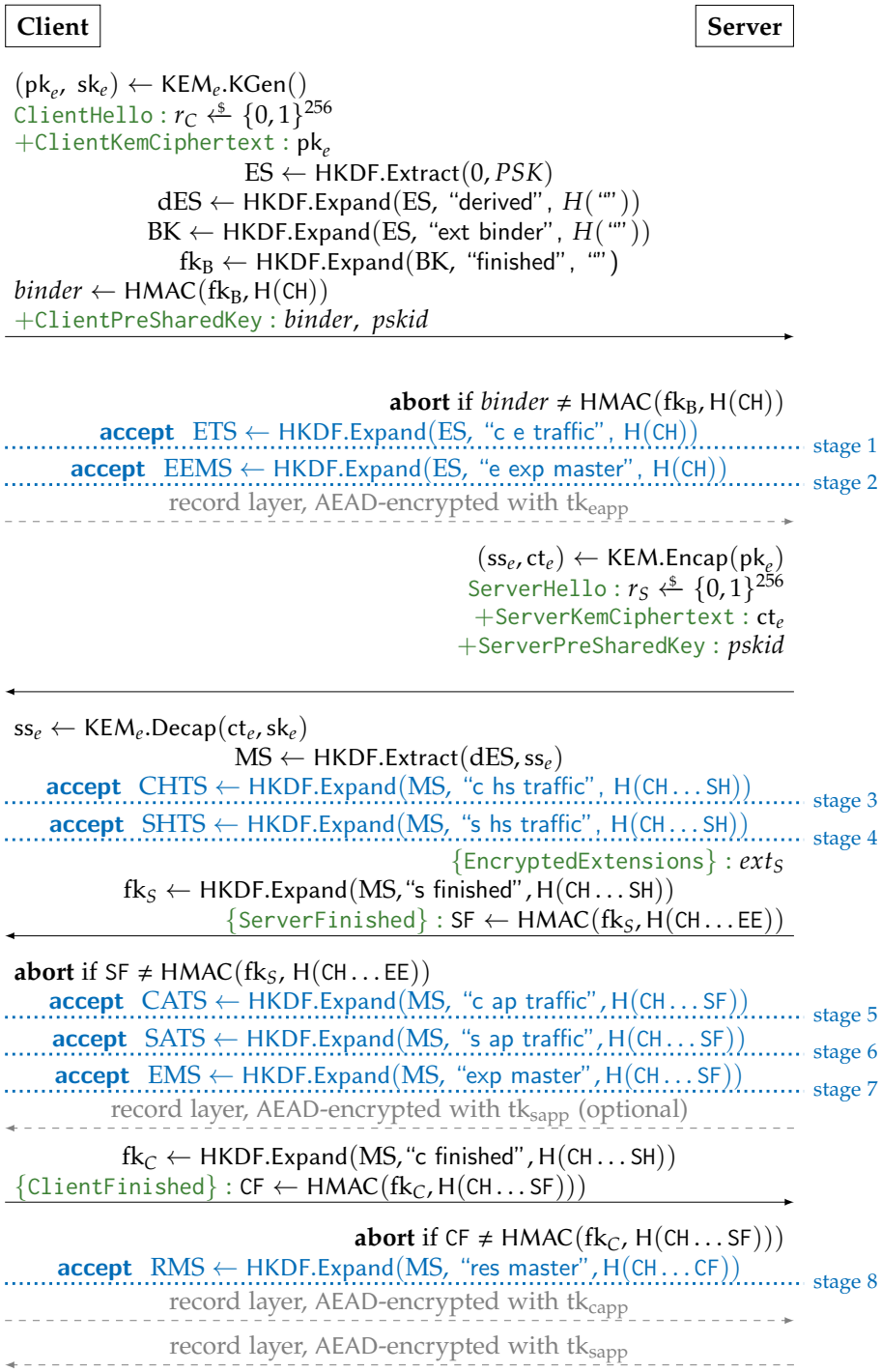


Figure 4.1: KEMTLS in Pre-Shared Key Mode

Chapter 5

Security Model

We have now designed a pre-shared key variant of the KEMTLS protocol (KEMTLS-PSK). In this chapter, we aim to analyze the security of this protocol. In order to achieve this goal, we must, intuitively, know which type of adversary we would like to defend against.

Bellare and Rogaway [20] introduce a model to analyze the security of key exchange protocols. They describe a powerful adversary that is in complete control of the communication between the communicating parties. It can modify protocol messages, learn the long-term keys of the communicating parties, as well as intermediate keys derived at various stages of the key exchange protocol.

The Bellare–Rogaway model assumes that the key exchange protocol is executed once to establish a shared symmetric key, which is then used to secure the actual communication. It analyzes the security of this final key while overlooking the possibility of intermediate keys being derived then used immediately before the key exchange is complete, for example, to secure subsequent protocol messages or encrypt early application data. This alternation between key derivation and key usage steps is addressed by Fischlin and Günther in [16]. They introduce a Bellare–Rogaway-style multi-stage security model which they use to analyze the security of Google's QUIC protocol. This model analyzes the security properties of, not just the final key, but of individual stage keys.

The security model we define for KEMTLS-PSK in this section draws mainly from the work of Dowling et al. [17], which adapts the Fischlin–Günther multi-stage security model for the TLS 1.3 protocol, and from the work of Schwabe, Stebila and Wiggers [13] which does the same for the full KEMTLS handshake.

5.1 Multi-Stage Security Model

In this section, we introduce the multi-stage model we use to analyze the security of the KEMTLS-PSK protocol. We assume a powerful adversary in the Bellare–Rogaway sense. This adversary has full control over the communication between parties. It can initialize new sessions using the NewSession oracle and establish new pre-shared secrets between parties using the NewSecret oracle. The adversary can also use the Reveal oracle to learn a certain stage key or the Corrupt oracle to learn the pre-shared secret between two communicating parties. Through the Send oracle, the adversary can choose to passively pass the message to the intended party or arbitrarily modify it.

The Test oracle gives the adversary a value and challenges it to guess whether this value is the real stage key or a random value. Our security goals, intuitively speaking, is to ensure that the adversary cannot guess this correctly. That is, it cannot distinguish stage keys from random values.

5.1.1 Security Notions for Stage Keys

Our model addresses the following security notions for every derived stage key. These notions are used to define properties such as "freshness", which we later use to impose limits on adversarial behavior.

Key indistinguishability. The key derived in every stage should be indistinguishable from random to the adversary. More formally, an adversary can issue a Test query to the model. Depending on a randomly sampled bit b, the model gives the adversary either the real stage key or a randomly sampled key. Key indistinguishabilty is achieved if the adversary's ability to guess the bit b is negligible.

Forward secrecy. The stage key should remain indistinguishable from random even if the long-term secret of the communicating party is compromised. We distinguish two levels of forward secrecy:

- Weak forward secrecy level 2 ($wfs2$): The stage key is indistinguishable from random if the adversary is passive or the adversary never corrupted the pre-shared key.
- Forward secrecy (fs): The stage key is indistinguishable from random if the adversary is passive or the adversary never corrupted the pre-shared key before the tested stages achieved forward secrecy.

Note that [13] defines weak forward secrecy level 1 ($wfs1$), which we exclude here since it is not needed in our analysis.

Implicit authentication. The stage key could only be known by the intended peer. This is achieved if the adversary did not corrupt the pre-shared key. Note that our formalization forward secrecy implies implicit authentication.

Explicit authentication. The stage key could only be known by the intended peer (implicit authentication) *and* the intended peer actively demonstrates knowledge of the key. In our formalization, we capture this property using the notion of malicious acceptance. If an adversary succeeds in causing a party to *maliciously accept*, that is accept without the presence of an honest partner which knows the secret key, then it has successfully violated explicit authentication.

Replayability. A stage is replayable if the adversary can replay the same client messages to multiple servers causing them to have identical session identifiers and keys at this stage. Both TLS 1.3 in resumption mode and our proposed KEMTLS-PSK do not offer replay protection for their 0-RTT stages.

5.1.2 Key exchange protocol

We would like to model the ability of the security model to run the key exchange protocol on behalf of honest parties. Therefore we define the key exchange protocol as consisting of two algorithms Activate and Run which we describe below:

- $\mathsf{Activate}(u, pssid, v, \mathsf{pss}, role) \xrightarrow{\$} (st', m')$.
  Initializes a session owned by u with v as an intended partner, u acting as *role*, pss as the chosen pre-shared key and *pssid* as the pre-shared key identifier. It outputs an initial state st' and an initial response message m'.
- $\mathsf{Run}(u, pssid, st, \mathsf{pss}, m) \xrightarrow{\$} (st', m')$.
  Takes as input the current state st, an input message m, a pre-shared key identifier *pssid* and pre-shared key pss and runs the protocol on m in the session owned by u. It outputs an updated state st' and a response message m'.

5.1.3 Model Syntax

We define the syntax of our security model. We distinguish protocol-specific properties and session states used to track information about individual sessions.

Protocol-specific variables. We define object P which represents protocol-specific values as follows:

- $P.\mathsf{M} \in \mathbb{N}$: the number of stages in the protocol.
- $P.\mathsf{iauth}[i] \in \{0,\ 1, \ldots, \mathsf{M},\ \infty\}$: the stage upon whose acceptance stage key i is implicitly authenticated.
- $P.\mathsf{eauth\_initiator}[i] \in \{0,\ 1, \ldots, \mathsf{M},\ \infty\}$: the stage upon whose acceptance stage key i in an initiator session is explicitly authenticated.
- $P.\mathsf{eauth\_responder}[i] \in \{0,\ 1, \ldots, \mathsf{M},\ \infty\}$: the stage upon whose acceptance stage key i in a responder session is explicitly authenticated.
- $P.\mathsf{FS\_initiator}[i][j] \in \{\perp,\ \mathit{wfs}2,\ \mathit{fs}\}$: the level of forward secrecy expected for stage key i in an initiator session assuming stage j has accepted.
- $P.\mathsf{FS\_responder}[i][j] \in \{\perp,\ \mathit{wfs}2,\ \mathit{fs}\}$: the level of forward secrecy expected for stage key i in a responder session assuming stage j has accepted.
- $P.\mathsf{use}[i] \in \{\mathsf{internal},\ \mathsf{external}\}$: whether the key derived in stage i is used internally (e.g. to encrypt subsequent handshake messages or as a MAC key) or externally (e.g. to encrypt application data).
- $P.\mathsf{replay}[i] \in \{\mathsf{replayable},\ \mathsf{non\text{-}replayable}\}$: whether stage i is replayable or not.

Session state. We denote a session by π. Assuming $\mathcal{K}$ is the set of all possible session keys. define a session state as containing at least the following information:

- $\pi.\mathsf{id} \in \mathbb{N}$: identity of session owner.
- $\pi.\mathsf{pid} \in \mathbb{N}$: identity of intended peer session.
- $\pi.\mathsf{pssid} \in \{0,1\}^*$: the identifier of the pre-shared key used in this session.
- $\pi.\mathsf{role} \in \{\mathsf{initiator},\ \mathsf{responder}\}$: whether this session is an initiator (client) or responder (server) session.
- $\pi.\mathsf{status}[i] \in \{\perp,\ \mathsf{running},\ \mathsf{accepted},\ \mathsf{rejected}\}$. Initially set to $\perp$.
- $\pi.\mathsf{stage} \in \{1, \ldots, \mathsf{M}\}$: the current stage at which the session is executing.
- $\pi.\mathsf{sid}[i]$: the session identifier at stage i. Initially set to $\perp$. Set once upon stage acceptance.

- π.cid_initiator$[i]$: the contributive session identifier checked by an initiator session for contributive partnering at stage i. Initially set to $\perp$. Set once before or upon stage acceptance.
- π.cid_responder$[i]$: the contributive session identifier checked by a responder session for contributive partnering at stage i. Initially set to $\perp$. Set once before or upon stage acceptance.
- π.key$[i] \in \mathcal{K}$: key derived at stage i. Initially set to $\perp$.
- π.revealed$[i] \in \{\text{true, false}\}$. Whether π.key$[i]$ has been revealed via a call to the Reveal oracle. Initially set to false.
- π.tested$[i] \in \{\text{true, false}\}$. Whether π.key$[i]$ has been tested via a call to the Test oracle. Initially set to false.

5.1.4 Security Game

Our security mode consists of an adversary $\mathcal{A}$ playing the Multi-Stage security game defined in Figure 5.0. We define the security game in pseudo-code in a manner similar to [21] for brevity and clarity. We denote session s of user u by π_u^s throughout.

Game-specific variables. These are variable that the game keeps track of to maintain consistency among its oracles.

- $\text{pss}_{u,v,pssid} \in \mathcal{P}$: the pre-shared key used by u in initiator role and v in responder role, identified by $pssid$. We assume $\mathcal{P}$ is the set of all possible pre-shared keys. The global list of all pre-shared keys is denoted by pss.
- $\text{revltp}_{u,v,pssid}$: the time at which the pre-shared key $\text{pss}_{u,v,pssid}$ is revealed to the adversary.
- $T \in \mathbb{N} \times \mathbb{N} \times \{0, 1, \ldots, M\}$: A set of user-session-stage tuples. It stores all the stages that have been tested.
- $\pi_u^s.\text{t}_{\text{acc}}[i] \in \mathbb{N}$: the time at which stage i of session s of user u accepted. Initially set to ∞.

The Multi-Stage game starts when adversary $\mathcal{A}$ calls Initialize which samples the random bit b. $\mathcal{A}$ then has access to different oracles: NewUser creates a new user, NewSecret establishes a new pre-shared secret between two users of the adversary's choosing and NewSession starts a new session between two users (by calling Activate) and using a pre-shared key identifier of the adversary's choosing.

Furthermore, the Send oracle allows the adversary to send arbitrary messages of its choosing to any session. It checks if the session exists and runs

$G^{\text{Multi-Stage}}_{\mathsf{KE},\mathcal{A}}$

Initialize

time $\leftarrow 0$; users $\leftarrow 0$
$b \xleftarrow{\$} \{0,1\}$
$T \leftarrow \{\}$ // set of tested sessions

NewUser

users $\leftarrow$ users $+ 1$

NewSecret$(u, v, pssid)$

// check pre-shared secret is not already defined for u, v and *pssid*...
if $\mathsf{pss}_{u,v,pssid} \neq \perp$:
  return $\perp$
$\mathsf{pss}_{u,v,pssid} \xleftarrow{\$} \mathcal{P}$ // .....otherwise sample new pre-shared secret and store it in the corresponding entry in the global pre-shared secret list...
$\mathsf{revltp}_{u,v,pssid} \leftarrow \infty$ // set the corruption time to infinity, i.e. this pss has not been corrupted

NewSession$(u, v, role, pssid)$

// initialize session with owner u, intended peer v and PSK identified by pssid
$(\pi^s_u, m') \xleftarrow{\$}$ Activate$(u, pssid, v, \mathsf{pss}, role)$
return m'

Send(u, s, m)

if $\pi^s_u = \perp$:
  return $\perp$
$(\pi^s_u, m') \xleftarrow{\$}$ Run$(u, \pi^s_u.\mathsf{pssid}, \pi^s_u, \mathsf{pss}, \pi^s_u.\mathsf{role})$
if $\pi^s_u.\mathsf{status}[i] == \mathsf{accepted}$: // special handling of acceptance
  $\pi^s_u.\mathsf{t_{acc}}[i] \leftarrow \mathsf{time}$ // record acceptance time
  time $\leftarrow$ time $+ 1$
  if $\exists_{(v,l)\neq(u,s)}(\pi^l_v.\mathsf{sid}[i] = \pi^s_u.\mathsf{sid}[i])$: // if partnered session...
    if $\pi^l_v.\mathsf{tested}[i] = \mathsf{true}$: // ..is tested..
      $\pi^s_u[i].\mathsf{tested} \leftarrow \mathsf{true}$ // ..set current session to tested
      if $P.\mathsf{use}[i] = \mathsf{internal}$: // .. and if accepted key used internally..
        $\pi^s_u[i].\mathsf{key} \leftarrow \pi^l_v[i].\mathsf{key}$ // ..set accepted key to partnered session's key for consistency
return m'

Reveal(u, s, i)

// check that session exists and desired stage has accepted
if $\pi^s_u = \perp$ or $\pi^s_u.\mathsf{status}[i] \neq \mathsf{accepted}$:
  return $\perp$
$\pi^s_u.\mathsf{revealed}[i] \leftarrow true$ // set revealed to true
return $\pi^s_u.\mathsf{key}[i]$ // return stage key to the adversary

Corrupt$(u, v, pssid)$

// arguments given in the following order (initiator, responder, pssid)
$\mathsf{revltp}_{u,v,pssid} \leftarrow \mathsf{time}$ // record corruption time
time $\leftarrow$ time $+ 1$ // increment time
return $\mathsf{pss}_{u,v,pssid}$ // return PSK to the adversary

Test(u, s, i)

if $\pi^s_u.status[i] \neq accepted$ or $\pi^s_u.tested[i] =$ true:
  return $\perp$
if $\exists_{(v,l)\neq(u,s)}(\pi^l_v.\mathsf{sid}[i] = \pi^s_u.\mathsf{sid}[i])$
  and $\pi^l_v.status[i+1] \neq \perp$
  and $P.use[i] =$ internal: // if the stage key is internal and the key has been used in a partnered session, i.e. next stage has started executing
  return $\perp$

π_u^s.tested$[i] \leftarrow$ true // set tested to true
$T \leftarrow T \cup \{(u,\ s,\ i)\}$ // record the tested stage
$k_0 \xleftarrow{\$} \mathcal{K}$ // sample random key
$k_1 \leftarrow \pi_u^s$.key$[i]$
if $P.use[i] =$ internal: // if key is internal..
  π_u^s.key$[i] \leftarrow k_b$ // ..set current key to the key returned to the adversary, to avoid trivial distinguisability
return k_b

FINALIZE

if $\exists_{(u,s,i)\in T}(\neg$Fresh$(u,s,i))$: // if a tested stage is not fresh..
  $b' \xleftarrow{\$} \{0,1\}$ // ..adversary loses..
if $\exists_{(u,s,i)}$(MalAccept(u,s,i)): // ..otherwise if there is a maliciously accepted stage
  $b' \leftarrow b$ // adversary wins
return $b == b'$ // did adversary guess correctly?

Fresh(u,s,i)

if π_u^s.revealed$[i] =$ true // if stage key is revealed
or $\exists_{(v,l)\neq(u,s)}(\pi_v^l$.sid$[i] = \pi_u^s$.sid$[i]$ // ..or stage key of partnered session is revealed
  and π_v^l.revealed$[i] =$ true)
  return false // then stage is not fresh
cont_partner = $\exists_{(v,l)\neq(u,s)}(\pi_v^l$.cid$[i]_\langle\pi_u^s$.role$\rangle = \pi_u^s$.cid$[i]_\langle\pi_u^s$.role$\rangle)$ // is there a contributive partner?
revlt $\leftarrow$ getRevlt(u,s)
// The definition of freshness:
if $\exists_{i,j}(j \geq i$ and π_u^s.FS$_\langle\pi_u^s$.role$\rangle[i][j] = \perp$
  and *revlt* $= \infty$): // no forward secrecy and PSK never corrupted
  return true
if $\exists_{i,j}(j \geq i$ and π_u^s.FS$_\langle\pi_u^s$.role$\rangle[i][j] =$ *wfs*2
  and π_u^s.status$[j] =$ accepted
  and (*cont_partner* or *revlt* $= \infty$)) : // wfs2 and either there is a contributive partner or PSK never corrupted
  return true
if $\exists_{i,j}(j \geq i$ and π_u^s.FS$_\langle\pi_u^s$.role$\rangle[i][j] =$ *fs*
  and π_u^s.status$[j] =$ accepted
  and (*cont_partner* or *revlt* $> \pi_u^s$.t$_{\text{acc}}[j]$)) : // full forward secrecy and either there is a contributive partner or PSK was not corrupted before stage acceptance
  return true
return false

MalAccept(u,s,i)

revlt $\leftarrow$ getRevlt(u,s)
auth_stage $\leftarrow$ P.eauth$_\langle\pi_u^s$.role$\rangle[i]$ // the stage at which stage i of π_u^s becomes explicitly authenticated
return π_u^s.status$[$*auth_stage*$]$ == accepted // the stage is explicitly authenticated...
  and $\neg\exists_{(v,l)\neq(u,s)}(\pi_v^l$.sid$[i] = \pi_u^s$.sid$[i])$ // ..without a partnered session..
  and *revlt* $> \pi_u^s$.t$_{\text{acc}}[$*auth_stage*$]$ // ..and PSK was not corrupted before explicit authentication happened

getRevlt(u, s)

// helper function, not part of protocol. Returns the revelation time of the PSK

$v \leftarrow \pi_u^s$.pid // ID of partner session
revlt $\leftarrow \perp$
pssid $\leftarrow \pi_u^s$.pssid
if π_u^s.role $=$ initiator
  revlt $\leftarrow$ revltp$_{u,v,pssid}$
else:
  revlt $\leftarrow$ revltp$_{v,u,pssid}$
return *revlt*

Figure 5.0: Key exchange security game.

the protocol on the input message using Run and returns the output message and the updated session state. If this execution causes the session to accept a new stage i, special care must be taken to ensure consistency and prevent the adversary from trivially winning the game. First, if stage i in a partnered session has been tested, the Send oracle marks the newly accepted stage as tested and, in case key i is used internally, sets the newly accepted key to the partnered session's key i. This procedure is shown in our pseudo-code in Figure 5.0. Second, the game should pause its execution to allow the adversary to test the newly accepted key before it is used in the protocol. We do not explicitly show this in the pseudo-code and assume that it is handled by the Run algorithm using special messages agreed upon with the adversary.

The Test oracle provides the adversary with a stage key of its choosing (if $b = 1$) or a random value (if $b = 0$). If the tested key is used internally in the protocol, the oracle sets this key to whichever value given to the adversary. That is, if b is 0, the key is set to the random value provided to the adversary, otherwise it remains the same. This is to prevent the adversary from trivially winning the game.

Finally, Reveal gives the adversary any stage key of its choosing and Corrupt gives the adversary any pre-shared secret of its choosing.

When the adversary calls Finalize it outputs its guessed bit b'. The adversary wins if it guessed b correctly and has not tested an unfresh stage or if it has caused a stage to maliciously accept. This is defined through the Fresh and MalAccept predicates:

- Fresh: A stage which has been revealed or whose partner stage has been revealed is, by definition, unfresh. Otherwise, freshness relies on the forward secrecy definitions of Section 5.1.1. A stage is deemed fresh at some point in time if it satisfies the conditions of the forward secrecy level it has (possibly retroactively) reached.
- MalAccept: A stage has maliciously accepted if it has (possibly retroactively) achieved explicit authentication (as defined in Section 5.1.1) without an honest partner and without the pre-shared key having been corrupted by the adversary prior to acceptance.

5.2 Security Notions for Key Exchange Protocols

There are two main notions of security which we analyze (in accordance with the distinction introduced by Brzuska et al. [22]) regarding key exchange protocols. The first is match security which ensures proper session partnering with respect to the session identifiers (sid's) and the second is

the classical notion of key indistinguishability which we augment with the multi-stage features of the Fischlin-Günther model.

5.2.1 Match Security

Definition 5.1 *(Match security) Let* KE *be a key exchange protocol and* $\mathcal{A}$ *be a probabilistic adversary interacting with* KE. $\mathcal{A}$ *aims to win the following game* $G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}}$*:*

Setup. The adversary starts the game.

Query. The adversary $\mathcal{A}$ has access to the queries NewSecret, NewSecret, Reveal, Corrupt and Test as described in Figure 5.0.

Stop. At some point, the adversary stop with no output.

Let π, π' be distinct partnered sessions with $\pi.\mathsf{sid}[i] = \pi'.\mathsf{sid}[i] \neq \perp$ for some stage $i \in \{1, \ldots, \mathsf{M}\}$. The adversary $\mathcal{A}$ is said to win the game, denoted by $G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} = 1$, if it succeeds in falsifying at least one of the following conditions:

1. $\forall_{j \leq i} \left(\pi.\mathsf{key}_j = \pi'.\mathsf{key}_j\right)$: π, π' agree on the established key for every stage $j \leq i$.
2. $\pi.\mathsf{role} = \pi'.\mathsf{role} \rightarrow P.\mathsf{replay}[i] = \mathsf{replayable} \wedge \pi.\mathsf{role} = \mathsf{responder}$: π, π' have opposite roles, except for replayable stages where two responder sessions could be partnered.
3. $\pi.\mathsf{cid\_initiator}[i] = \pi'.\mathsf{cid\_initiator}[i] \neq \perp$ and $\pi.\mathsf{cid\_responder}[i] = \pi'.\mathsf{cid\_responder}[i] \neq \perp$: π, π' have set their contributive identifiers for each role to the same value.
4. $\forall_{j \leq i}(\pi.\mathsf{status}[i] = \mathsf{accepted} \wedge i \geq \pi.\mathsf{eauth\_}\langle\mathsf{role}\rangle[j] \rightarrow \pi.\mathsf{pid} = \pi'.\mathsf{id} \wedge \pi.\mathsf{pssid} = \pi'.\mathsf{pssid})$: for every non-replayable stage in π that has been (retroactively) explicitly authenticated, the intended peer identity matches the identity of π' and both sessions agree on the pre-shared key identifier.
5. $\forall_{\pi,\pi'',i,j}\,(\pi.\mathsf{sid}[i] = \pi''.\mathsf{sid}[j] \rightarrow i = j)$: no two distinct stages (including in the same session, i.e. if $\pi = \pi''$) share the same session identifier.
6. $P.\mathsf{replay}[i] = \mathsf{non\text{-}replayable} \wedge \exists_{\pi''}\,(\pi.\mathsf{sid}[i] = \pi'.\mathsf{sid}[i] = \pi''.\mathsf{sid}[i]) \rightarrow (\pi' = \pi'' \vee \pi = \pi'')$: if i is non-replayable π, π' do not have a third partnered session.

The advantage of the adversary $\mathcal{A}$ against the match security game is defined as follows:

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} := \mathsf{Pr}\left[G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} = 1\right].$$

5.2.2 Multi-stage Security

Definition 5.2 *(Multi-stage security) Let* KE *be a key exchange protocol and* $\mathcal{A}$ *be a probabilistic adversary interacting with* KE. $\mathcal{A}$ *aims to win the game* $G_{\mathsf{KE},\mathcal{A}}^{\text{Multi-Stage}}$ *(described in more detail in Figure 5.0) as follows:*

Setup. The game starts when $\mathcal{A}$ calls Initialize.

Query. $\mathcal{A}$ has access to the query oracles defined in Figure 5.0.

Stop. The game stops when $\mathcal{A}$ calls Finalize and $\mathcal{A}$ outputs its guess bit b'. The Finalize oracle determines the output of the game: the wins if it correctly guesses b without testing an unfresh stage or if it causes a stage to malicious accept.

The advantage of the adversary $\mathcal{A}$ against the multi-stage security game is defined as follows:

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

Chapter 6

Security Analysis

In this chapter, we aim to analyze how the KEMTLS-PSK protocol fares with respect to the key exchange security model defined in Chapter 5. In Section 6.1, we instantiate the model for KEMTLS-PSK, including defining session and contributive identifiers. In Section 6.2, we analyze the match security of the protocol and in Section 6.3, we analyze its multi-stage security. In Section 6.4, we give a brief comparison of our analysis of KEMTLS-PSK to that of TLS 1.3 PSK-(EC)DHE in [17].

6.1 Instantiating the Security Model for KEMTLS-PSK

Before we proceed to prove the security of the protocol, we define the protocol-specific values for the KEMTLS-PSK protocol as follows:

- $P.\mathsf{M} = 8$: KEMTLS in pre-shared key mode has 8 stages.
- $P.\mathsf{iauth} = (1, 2, 3, 4, 5, 6, 7, 8)$: since PSK is mixed into all key derivations, all stage keys are immediately implicitly authenticated.
- $P.\mathsf{eauth\_initiator} = (5, 5, 5, 5, 5, 6, 7, 8)$: in an initiator session, explicit authentication is (retroactively) achieved when stage 5 is accepted. That is, when the client verifies the `ServerFinished` by which the server actively demonstrates its knowledge of the derived shared secrets.
- $P.\mathsf{eauth\_responder} = (8, 8, 8, 8, 8, 8, 8, 8)$: in a responder session, explicit authentication is (retroactively) achieved when stage 8 accepts. We note here that, due to the presence of the *binder* value in the `ClientHello` message, stages 1 and 2 could be considered explicitly authenticated. In this case, we would require our analysis to handle a non-monotonous explicit authentication property, where stages 1 and 2 would be explicitly authenticated, while stages 3 through 7 would

only be implicitly authenticated until stage 8 accepts. To avoid overcomplicating the analysis, we assume that explicit authentication is monotonous and that stages 1 and 2 are only implicitly authenticated.

- In an initiator session, stages 1 and 2 have no forward secrecy, stages 3 and 4 have weak forward secrecy level 2 upon acceptance and retroactive full forward secrecy upon acceptance of stage 5 and stages 5-8 have full forward secrecy on acceptance.

$$P.\mathsf{FS\_initiator} = \begin{bmatrix} \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot \\ & \bot & \bot & \bot & \bot & \bot & \bot & \bot \\ & & \mathit{wfs}2 & \mathit{wfs}2 & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & \mathit{wfs}2 & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & & & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & & & & \mathit{fs} & \mathit{fs} \\ & & & & & & & \mathit{fs} \end{bmatrix}$$

- In a responder session, stages 1 and 2 have no forward secrecy, stages 3-8 have full forward secrecy upon acceptance.

$$P.\mathsf{FS\_responder} = \begin{bmatrix} \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot \\ & \bot & \bot & \bot & \bot & \bot & \bot & \bot \\ & & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & & \mathit{fs} & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & & & \mathit{fs} & \mathit{fs} & \mathit{fs} \\ & & & & & & \mathit{fs} & \mathit{fs} \\ & & & & & & & \mathit{fs} \end{bmatrix}$$

  We note that, unlike in an initiator session where stages 3 and 4 get *wfs*2 on acceptance, they get full forward secrecy *fs* in a responder session. This is due to the presence of the *binder* value in the `ClientHello` message. An adversary that attempts to learn the keys of stages 3 and 4 in a tested responder session would have to send its own ephemeral public key with the `ClientHello` and corrupt *PSK* before stage 1 accepts in order to forge the *binder* value.

- $P.\mathsf{use} = (\mathsf{internal} : \{3,\ 4\},\ \mathsf{external} : \{1,\ 2,\ 5,\ 6,\ 7,\ 8\})$

Session identifiers. The session identifier of every stage consists of a label and the transcript of messages up to this stage. We define session identifiers

in more detail below:

$$\begin{aligned}
\mathsf{sid}[1] &= (\text{“ETS”}, \texttt{ClientHello}) \\
\mathsf{sid}[2] &= (\text{“EEMS”}, \texttt{ClientHello}) \\
\mathsf{sid}[3] &= (\text{“CHTS”}, \texttt{ClientHello} \ldots \texttt{ServerHello}) \\
\mathsf{sid}[4] &= (\text{“SHTS”}, \texttt{ClientHello} \ldots \texttt{ServerHello}) \\
\mathsf{sid}[5] &= (\text{“CATS”}, \texttt{ClientHello} \ldots \texttt{ServerFinished}) \\
\mathsf{sid}[6] &= (\text{“SATS”}, \texttt{ClientHello} \ldots \texttt{ServerFinished}) \\
\mathsf{sid}[7] &= (\text{“EMS”}, \texttt{ClientHello} \ldots \texttt{ServerFinished}) \\
\mathsf{sid}[8] &= (\text{“RMS”}, \texttt{ClientHello} \ldots \texttt{ClientFinished})
\end{aligned}$$

Contributive idenifiers. Every stage has two contributive identifiers: $\mathsf{cid\_initiator}[i]$ is the identifier checked by the initiator (i.e. client) for contributive partnering at stage i and $\mathsf{cid\_responder}[i]$ is the identifier checked by the responder (i.e. server) for contributive partnering at stage i.

For stages 1 and 2, both contributive identifiers are set to the values of the session identifiers upon acceptance.

Upon sending (resp. receiving) the `ClientHello` message, the client (resp. server) sets $\mathsf{cid\_responder}[3]$ to (“ETS”, `ClientHello`). For all stages $i > 3$, $\mathsf{cid\_responder}[i] = \mathsf{cid\_responder}[3]$.

Upon sending (resp. receiving) the `ServerHello` message, the server (resp. client) sets $\mathsf{cid\_initiator}[3]$ to (“CHTS”, `ClientHello` ... `ServerHello`). For all stages $i > 3$, $\mathsf{cid\_initiator}[i] = \mathsf{cid\_initiator}[3]$.

6.2 Match Security Analysis for KEMTLS-PSK

Theorem 6.1 *Let $\mathcal{A}$ be an adversary and let n_S be the number of sessions and n_{psk} be the number of pre-shared keys and $\mathcal{P}$ be the pre-shared key space. The advantage $\mathsf{Adv}^{\mathsf{Match}}_{\text{KEMTLS-PSK},\mathcal{A}}$ as defined in Definition 5.1 is bound as follows:*

$$\mathsf{Adv}^{\mathsf{Match}}_{\text{KEMTLS-PSK},\mathcal{A}} \leq n_S \cdot \delta_e + \frac{n_{psk}^2}{|\mathcal{P}|} + \mathsf{Adv}^{COLL}_{\mathsf{HMAC},\mathcal{B}} + \frac{n_S^2}{|nonce|}$$

.

Proof Let π, π' be two distinct partnered sessions with $\pi.\mathsf{sid}[i] = \pi'.\mathsf{sid}[i] \neq \perp$ for some stage $i \in \{1, \ldots, M\}$. We show that KEMTLS-PSK satisfies every property of match security as described in Definition 5.1:

1. *π, π' agree on the established key for every stage $j \leq i$.*
   The session identifier $\mathsf{sid}[i]$ contains all the messages of the handshake

up to stage i. The key schedule takes the hash of these messages as input. For stages 1 and 2, the input to the key schedule is the hash of the `ClientHello` message which contains the ephemeral public key of KEM_e, the pre-shared key identifier *pskid* and the *binder* value. For all subsequent stages, the input to the key schedule is the ephemeral shared secret ss_e established by KEM_e and the hash of messages up to that stage: for stages 3 and 4, up to `ServerHello`, for stages 5 through 7, up to `ServerFinished` and for stage 8 up to `ClientFinished`. The only case in which the keys of π and π' can differ is if the KEM_e correctness condition fails. Since KEM_e is δ_e-correct and we have n_S session, this failure event can happen with a probability of $n_S \cdot \delta_e$.

2. *π, π' have opposite roles, except for replayable stages where two responder sessions could be partnered.*
   By definition of the KEMTLS-PSK handshake, no initiator or responder session will accept a wrong-role incoming message. Therefore, assuming that at most two sessions share the same session identifier at non-replayable stages (which we show below), π and π' must have opposite roles. For replayable stages 1 and 2, two initiator sessions can only have matching session identifiers on nonce repetition (which we rule out below), while multiple responder sessions might be partnered if the `ClientHello` is forwarded to multiple servers.

3. *π, π' have set their contributive identifiers for each role to the same value.*
   By definition of contributive identifiers, $\mathsf{cid}[i]$ is set to $\mathsf{sid}[i]$ when stage i has accepted.

4. *For every non-replayable stage in π that has been (retroactively) explicitly authenticated, the intended peer identity matches the identity of π' and both sessions agree on the pre-shared key identifier.*
   The tuple $(\mathit{session\_owner}, \mathit{peer}, \mathsf{pssid})$ uniquely determines a PSK, unless there is a collision in the NewSecret oracle, which can happen with a birthday-bound probability of $n_{psk}^2/|\mathcal{P}|$. It remains to show that by agreeing on the *binder* value and pssid, which are included in the session identifiers, π and π' agree on the PSK and $\pi.\mathsf{pid} = \pi'.\mathsf{id}$ and $\pi'.\mathsf{pid} = \pi.\mathsf{id}$. The *binder* value is computed from the PSK using a series of HKDF/HMAC computations. After ruling out PSK collisions, the only way that π and π' can agree on the *binder* value without agreeing on PSK is if there is an HMAC collision, which can be bound by the advantage of an adversary $\mathcal{B}$ against the collision-resistence of HMAC, $\mathsf{Adv}^{COLL}_{\mathsf{HMAC},\mathcal{B}}$.

5. *No two distinct stages (including in the same session, i.e. if $\pi = \pi''$) share the same session identifier.*
   Each stage has a unique label, therefore it trivially holds that no two stages have the same session identifier.

6. *If i is non-replayable π, π' do not have a third partnered session.*
   The session identifiers include the nonce values. In non-replayable stages, in order to have three-way partnering, there needs to be a collision in any of these values. We can bound the probability of this event by a birthday bound over the nonces: $n_S^2/2^{|nonce|}$. This argument need only hold for stage 3 and above since stages 1 and 2 are replayable and, hence, allow for three-way partnering. □

6.3 Multi-Stage Security Analysis for KEMTLS-PSK

The aim of this analysis is to prove that any adversary acting within the bounds of the freshness conditions defined in Figure 5.0 can win the security game with a negligible probability. This is done by deriving an upper bound on the advantage of the adversary against the multi-stage security game $\mathsf{Adv}^{\text{Multi-Stage}}_{\text{KEMTLS-PSK},\mathcal{A}}$. More formally, we aim to prove the following theorem:

Theorem 6.2 *Let $\mathcal{A}$ be an adversary, and let n_S be the number of sessions and n_{psk} be the number of pre-shared keys used in the game. There exist algorithms $\mathcal{B}_1, \ldots, \mathcal{B}_9$, as described in the proof, such that $\mathsf{Adv}^{\text{Multi-Stage}}_{\text{KEMTLS-PSK},\mathcal{A}}$ as defined in Definition 5.2 is bound as follows:*

$$\begin{aligned}\mathsf{Adv}^{\text{Multi-Stage}}_{\text{KEMTLS-PSK},\mathcal{A}} \leq\ & \frac{n_S^2}{2^{|nonce|}} + \mathsf{Adv}^{\text{COLL}}_{\text{H},\,\mathcal{B}_1} \\ & + 8n_S \left(\begin{array}{l} n_{psk} \left(\begin{array}{l} 3 \cdot \mathsf{Adv}^{\text{dual-PRF}}_{\text{HKDF.Extract},\mathcal{B}_2} + 3 \cdot \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\mathcal{B}_3} \\ +\ 3 \cdot \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Extract},\,\mathcal{B}_4} + 3 \cdot \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\,\mathcal{B}_5} \\ +\ \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\mathcal{B}_7} + \mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\,\mathcal{B}_8} \end{array} \right) \\ +\ n_S \left(\begin{array}{l} \mathsf{Adv}^{\text{IND-1CCA}}_{\text{KEM}_e,\,\mathcal{B}_6} + \mathsf{Adv}^{\text{dual-PRF}}_{\text{HKDF.Extract},\mathcal{B}_4} \\ \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\,\mathcal{B}_5} \end{array} \right) \end{array} \right).\end{aligned}$$

By examining the FINALIZE oracle in Figure 5.0, we observe that the adversary can win either by winning an indistinguishability game against a fresh stage or by causing any stage to maliciously accept. Therefore, we can compute an upper bound on the advantage of the adversary against the game won by breaking indistinguishability $\mathsf{Adv}^{\text{Multi-Stage.IND}}_{\text{KEMTLS-PSK},\mathcal{A}}$ and the game won by malicious acceptance $\mathsf{Adv}^{\text{Multi-Stage.MAL}}_{\text{KEMTLS-PSK},\mathcal{A}}$ and bound the overall advantage by the maximum of the two as follows:

$$\mathsf{Adv}^{\text{Multi-Stage}}_{\text{KEMTLS-PSK},\mathcal{A}} \leq \max\left\{ \mathsf{Adv}^{\text{Multi-Stage.IND}}_{\text{KEMTLS-PSK},\mathcal{A}}, \mathsf{Adv}^{\text{Multi-Stage.MAL}}_{\text{KEMTLS-PSK},\mathcal{A}} \right\}.$$

In the following subsections, we first proceed to first prove key indistinguishability then the absence of malicious acceptance.

6.3.1 Proving Key Indistinguishability

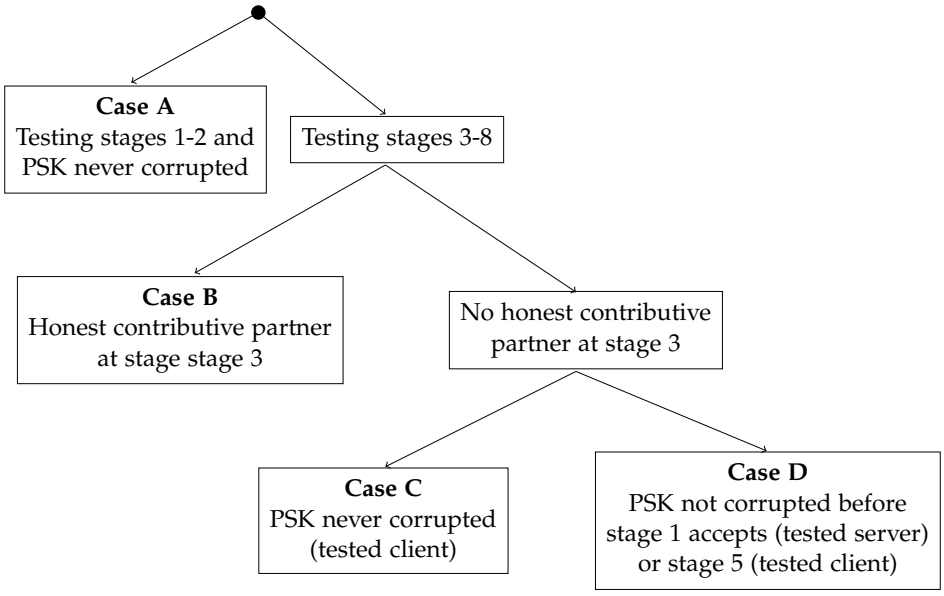


Figure 6.1: Division of proof cases

In Figure 6.1, we show the different proof cases as a tree. The conjunction of a leaf node and all its ancestors up to the root represents one proof case. It remains to explain how we arrived at these conditions.

The first criteria for dividing proof cases is whether the adversary tested a stage that possesses some level of forward secrecy or not. Since stages 1 and 2 have no forward secrecy, we will rely only on the security of the PSK to prove key indistinguishabilty in case the adversary tests one of these two stages. This yield **Case A**.

In case the adversary tests one of stages 3 through 8, we need another criteria of division. This is because the definitions of both *wfs*2 and *fs* include two conditions. One is the absence of an active adversary and the other restricts the corruption of the pre-shared key. The second criteria for dividing proof cases is, therefore, whether there is an honest contributive partner (i.e. the adversary is passive) at stage 3. If there is an honest contributive partner at stage 3, we can rely on the security of the ephemeral shared secret ss_e in our proof. This yields **Case B**.

In case there is no honest contributive partner, we must make one last division based on when the adversary is allowed to corrupt the pre-shared

key. In **Case C**, the adversary is never allowed to corrupt the pre-shared key. This case only applies to when the tested session is a client session due to the fact that stages 3 and 4 are *wfs*2 in the tested client case whereas in the tested server case full *fs* is activated immediately on acceptance of stage 3. The proof for **Case C** employs a proof strategy similar to **Case A** where we rely on the security of pre-shared key. In **Case D**, the adversary is forbidden from corrupting the pre-shared key only before stage 1 in case of a tested server session or stage 5 in case of a tested client session.

Proof The proof proceeds via a series of games. Games 0 through 3 are common to all the cases. After Game 3, we branch into the four cases discussed above.

Game 0. The original Multi-Stage.IND game. That is, the multi-stage game that can only be won by breaking indistinguishability:

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

Game 1 (no nonce repetition). We abort if any two sessions sample the same nonce. The probability of this event is computed via a birthday bound. Since there are n_S session and $2^{|nonce|}$ possible nonce values, this gives the following bound:

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

Game 2 (no hash collisions). We abort if any two sessions compute the same hash value on two different input values. We can construct a reduction adversary $\mathcal{B}_1$ against the hash collision resistance of the hash function H which will output the two input values as its collision. This yields the following bound:

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

Game 3 (single Test query). We restrict $\mathcal{A}$ to make a single Test query. Using the hybrid argument in [17], this reduces the advantage of the adversary by $1/8n_S$ where n_S is the number of sessions. We, thus, get the following bound:

$$\mathsf{Adv}^{G_2}_{\mathcal{A}} \leq 8n_S \cdot \mathsf{Adv}^{G_3}_{\mathcal{A}}.$$

We then branch into the four cases discussed above and bound the advantage $\mathsf{Adv}^{G_3}_{\mathcal{A}}$ as follows:

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

Case A: Testing stages 1-2 and PSK is never corrupted. This case depends on the indistinguishability of PSK which is never corrupted by the adversary.

Game A.1 (guess PSK). We guess the pre-shared key that will be used in the tested session. Assuming that n_{psk} is the number of pre-shared key used in the entire experiment, we get the following bound:

$$\mathsf{Adv}_{\mathcal{A}}^{G_A} \leq n_{psk} \cdot \mathsf{Adv}_{\mathcal{A}}^{G_{A.1}}.$$

Game A.2 (random ES). We replace ES with a uniformly random value $\overline{\text{ES}}$ in the tested session and all other sessions which use the same PSK guessed in Game A.1. An adversary $\mathcal{A}$ that can distinguish these two values can be used to construct an adversary $\mathcal{B}_2$ against the dual-PRF security of the HKDF.Extract function.

The dual-PRF adversary $\mathcal{B}_2$ simulates the Multi-Stage challenger to adversary $\mathcal{A}$. Whenever $\mathcal{B}_2$ needs to compute ES, it queries the dual-PRF challenger with PSK and forwards the response to $\mathcal{A}$. If the dual-PRF game is in the real case, then ES $=$ HKDF.Extract$(0, PSK)$ and $\mathcal{B}_2$ is simulating Game A.1 to $\mathcal{A}$. If it is in the random case, then $\mathcal{B}_2$ is simulating Game A.2 to $\mathcal{A}$. This yields the following bound:

$$\mathsf{Adv}_{\mathcal{A}}^{G_{A.1}} \leq \mathsf{Adv}_{\mathcal{A}}^{G_{A.2}} + \mathsf{Adv}_{\mathsf{HKDF.Extract},\mathcal{B}_2}^{\text{dual-PRF}}.$$

Game A.3 (random dES, BK, ETS, EEMS). We replace dES, BK, ETS and EEMS in the tested session and all other session using the guessed PSK with uniformly random values $\overline{\text{dES}}$, $\overline{\text{BK}}$, $\overline{\text{ETS}}$ and $\overline{\text{EEMS}}$. An adversary $\mathcal{A}$ that can distinguish the real and random value can be used to construct an adversary $\mathcal{B}_3$ against the PRF security of the HKDF.Expand function.

The PRF adversary $\mathcal{B}_3$ simulates the Multi-Stage challenger to $\mathcal{A}$. Whenever $\mathcal{B}_3$ needs to compute any of these values, it queries the PRF challenger on $\overline{\text{ES}}$ and forwards the response to $\mathcal{A}$. If the PRF game is in the real case, then response is HKDF.Expand$(\overline{\text{ES}}, \dots)$ and $\mathcal{B}_3$ is simulating Game A.2 to $\mathcal{A}$. If it is in the random case, then $\mathcal{B}_3$ is simulating Game A.3 to $\mathcal{A}$. This yields the following bound:

$$\mathsf{Adv}_{\mathcal{A}}^{G_{A.2}} \leq \mathsf{Adv}_{\mathcal{A}}^{G_{A.3}} + \mathsf{Adv}_{\mathsf{HKDF.Expand},\mathcal{B}_3}^{\text{PRF}}.$$

We also argue that the keys ETS and EEMS are independent of their counterparts in any non-partnered session. This is because any non-partnered session will have a different transcript, since we excluded nonce collisions in Game 1 and because the HKDF uses the hash of the transcript to compute stage keys and we excluded hash collisions in Game 2.

Game A.4 (random MS). We replace MS with a random value $\overline{\text{MS}}$ in the tested session and all other sessions using the PSK guessed in Game A.1. Distinguishing this game from the previous game is reducible to the PRF security of the HKDF.Extract function using a reduction adversary $\mathcal{B}_4$. Similar to Game A.3, this yield the following bound:

$$\text{Adv}_{\mathcal{A}}^{G_{A.3}} \leq \text{Adv}_{\mathcal{A}}^{G_{A.4}} + \text{Adv}_{\text{HKDF.Extract},\ \mathcal{B}_4}^{\text{PRF}}.$$

Game A.5 (random CHTS, SHTS, CATS, SATS, EMS, RMS, $\mathbf{fk_C}$, $\mathbf{fk_S}$). We replace CHTS, SHTS, CATS, SATS, EMS, RMS, fk_C and fk_S with random values in the tested session and all other sessions using the same PSK guessed in Game A.1. Similar to previous game hops, we get the following bound:

$$\text{Adv}_{\mathcal{A}}^{G_{A.4}} \leq \text{Adv}_{\mathcal{A}}^{G_{A.5}} + \text{Adv}_{\text{HKDF.Expand},\ \mathcal{B}_5}^{\text{PRF}}.$$

Case B: Testing stages 3-8 and there is an honest contributive partner at stage 3 . Since there is an honest contributive partner at stage 3 which computes the ephemeral shared secret ss_e, we depend on the indistinguishability of ss_e to bound the advantage of the adversary in this case.

Game B.1 (guess contributive session). We guess the identity of the honest contributive partner session. This reduces the advantage of the adversary by a factor of the number of sessions n_S.

$$\text{Adv}_{\mathcal{A}}^{G_B} \leq n_S \cdot \text{Adv}_{\mathcal{A}}^{G_{B.1}}.$$

Game B.2 ($\mathbf{KEM_e}$ security). We replace the ephemeral secret value ss_e with a uniformly random value $\overline{\text{ss}_e}$ in the tested session and any session that received the same ct_e that the tested session sent.

An adversary that can detect this replacement of the real shared secret with a random value can be used to construct an adversary $\mathcal{B}_6$ against the IND-1CCA security of KEM_eas follows:

$\mathcal{B}_6$ receives the challenge tuple $(\text{pk}^*, \text{ct}^*, \text{ss}^*)$ from the IND-1CCA challenger. It uses pk^* as the ephemeral public key in the `ClientHello` message, ct^* as the ephemeral ciphertext in `ServerHello` and ss^* as the ephemeral shared secret.

In case the tested session is a client session, since the cid_initiator[3]'s match, then the client received the same ct^* from the server and both parties share the same ephemeral shared secret ss^*. If ss^* is the real value, then $\mathcal{B}_6$ has

simulated Game B.1 to $\mathcal{A}$. If it is the random value, then $\mathcal{B}_6$ has simulated Game B.2 to $\mathcal{A}$. Therefore, strictly speaking, in this case the reduction can play against an IND-CPA challenger since we do not issue decapsulation queries.

However, in case the tested session is a server session, matching cid_responder[3]'s do not guarantee that the client received the ServerHello message unmodified. The adversary could send it's own $\mathrm{ct}' \neq \mathrm{ct}^*$ to the client and hence the client would compute $\mathrm{ss}' \neq \mathrm{ss}^*$. To stop the adversary from trivially winning the game by revealing a client stage key derived from ss', we need the reduction to be able to compute ss^* to use it as the clients ephemeral shared secret. Therefore, we need to allow the reduction to issue one decapsulation query to the IND-1CCA challenger.

This yields the following bound:

$$\mathrm{Adv}_{\mathcal{A}}^{G_{B.1}} \leq \mathrm{Adv}_{\mathcal{A}}^{G_{B.2}} + \mathrm{Adv}^{\text{IND-1CCA}}_{\mathsf{KEM}_e,\, \mathcal{B}_6}.$$

Game B.3 (random MS). We replace the master secret MS with a random value $\overline{\mathrm{MS}}$ in the tested session and all the sessions that received the same ct_e that it sent. Distinguishing this game from the previous one is reducible to the dual-PRF security of the HKDF.Expand function via a reduction $\mathcal{B}_4$ (see Game A.2). Thus, we get the following bound:

$$\mathrm{Adv}_{\mathcal{A}}^{G_{B.2}} \leq \mathrm{Adv}_{\mathcal{A}}^{G_{B.3}} + \mathrm{Adv}^{\text{dual-PRF}}_{\mathsf{HKDF.Extract},\mathcal{B}_4}.$$

Game B.4 (random CHTS, SHTS, CATS, SATS, EMS, RMS, $\mathbf{fk_C}$, $\mathbf{fk_S}$). Similar to Game A.5. We get the following bound:

$$\mathrm{Adv}_{\mathcal{A}}^{G_{B.3}} \leq \mathrm{Adv}_{\mathcal{A}}^{G_{B.4}} + \mathrm{Adv}^{\text{PRF}}_{\mathsf{HKDF.Expand},\, \mathcal{B}_5}.$$

Case C: Testing stages 3-8, there is no honest contributive partner and PSK is never corrupted (tested client). This case roughly corresponds to weak forward secrecy level 2 and is, therefore, relevant only for tested client sessions. Similar to **Case A**, it depends on the fact that PSK is never corrupted to prove the indistinguishability of stage keys.

Games C.1-C.5. Similar to Games A.1-A.5.

Case D: Testing stages 3-8, there is no honest contributive partner and PSK is not corrupted before stage 1 (tested server) or stage 5 (tested client). In this case, as in **Case A**, we would like to rely on the indistinguishability of the PSK for security. We

cannot, however, use exactly the same proof steps since in the current case the adversary is allowed to corrupt PSK after a certain stage accepts. The reduction algorithms in **Case A**, especially the reduction for the hop to Game A.2, is unable to respond to CORRUPT queries from the adversary.

Since the game in this case proceeds in a similar manner to **Case A** unless the tested stage accepts without an honest contributive partner while PSK is not corrupted before stage 1 (in the tested server case) or stage 5 (in the tested client case), we designate this event as a bad event. Then we use the *identical-until-bad* formulation introduced by Bellare and Rogaway (see Section 3 in [23]) to bound the probability of the bad event.

Game D.1 (guess PSK). Similar to Game A.1 where we guess the identity of the PSK used in the tested session. We get the following bound:

$$\mathsf{Adv}_{\mathcal{A}}^{G_D} \leq n_{psk} \cdot \mathsf{Adv}_{\mathcal{A}}^{G_{D.1}}.$$

Game D.2 (identical-until-bad). This game is identical to the previous one, except that it aborts when the bad event occurs. Using the fundamental lemma of game-playing (see Section 3 in [23]), we can bound the advantages as follows:

$$\mathsf{Adv}_{\mathcal{A}}^{G_{D.1}} - \mathsf{Adv}_{\mathcal{A}}^{G_{D.2}} \leq \Pr[\text{D.2 reaches bad}].$$

Trivially, the adversary cannot win in Game D.2. In order to win, the adversary must cause the tested stage to accept without an honest contributive partner while its PSK is not corrupted (before stage 1 in the tested server case or stage 5 in the tested server case). This, however, is the bad event upon which Game D.2 aborts. Therefore, we get:

$$\mathsf{Adv}_{\mathcal{A}}^{G_{D.2}} = 0.$$

What remains is to bound the probability Pr[D.2 reaches bad], which we do via the subsequent game hops.

Game D.3 (random ES). This game is similar to Game A.2 where we replace ES with a random value, except that it aborts when bad is reached. This yields the following bound:

$$\Pr[\text{D.2 reaches bad}] \leq \Pr[\text{D.3 reaches bad}] + \mathsf{Adv}^{\mathsf{dual\text{-}PRF}}_{\mathsf{HKDF.Extract},\mathcal{B}_2}.$$

Game D.4 (random dES, BK, ETS, EEMS). This game is similar to Game A.3 where dES, BK, ETS and EEMS with the additional condition of aborting when bad occurs. This yields the following bound:

$$\Pr[\text{D.3 reaches bad}] \leq \Pr[\text{D.4 reaches bad}] + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF.Expand},\mathcal{B}_3}.$$

Game D.5 (random MS). This game is similar to Game A.4 where MS is replaced with a random value, with the additional condition that is aborts when bad occurs. This yields the following bound:

$$\Pr[\text{D.4 reaches bad}] \leq \Pr[\text{D.5 reaches bad}] + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF.Extract},\ \mathcal{B}_4}.$$

Game D.6 (random CHTS, SHTS, CATS, SATS, EMS, RMS, $\mathbf{fk_C}$, $\mathbf{fk_S}$). This game is similar to Game A.5 where we replace CHTS, SHTS, CATS, SATS, EMS, RMS, fk_C and fk_S with random values. Similar to previous game hops, we get the following bound:

$$\Pr[\text{D.5 reaches bad}] \leq \Pr[\text{D.6 reaches bad}] + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF.Expand},\ \mathcal{B}_5}.$$

Game D.7 (random $\mathbf{fk_B}$). We replace fk_B with a random value $\overline{\mathrm{fk}_B}$ in the tested session. Distinguishing this game from the previous one is reducible, via a reduction $\mathcal{B}_7$ to the PRF security of the HKDF.Expand function. Thus, we get the following bound:

$$\Pr[\text{D.6 reaches bad}] \leq \Pr[\text{D.7 reaches bad}] + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HKDF.Expand},\mathcal{B}_7}.$$

Note that a similar game hope does not exist in **Case A** since we do not depend on the *binder* value for proving security.

Game D.8 (HMAC forgery). In case the tested session is a client session, the client rejects the `ServerFinished` message. In case the tested session is a server session, the server rejects the *binder* value. We argue that distinguishing this game from the previous one is reducible to the EUF-CMA security of the HMAC algorithm. Since there is no honest contributive partner, we know that no honest session output the MAC contained in the `ServerFinished` (resp. *binder* value). The only way that this game can differ from the previous one is if the client (resp. server) was supposed to accept, because the adversary successfully forged a MAC over the transcript, but it actually rejected.

We can build a reduction $\mathcal{B}_8$ against the EUF-CMA of the HMAC. Whenever $\mathcal{B}_8$ wants to compute an HMAC tag, it queries the Tag oracle of the challenger. Upon receiving the `ServerFinished` (resp. *binder* value), since there is no honest contributive partner and this value has never been queried to the Tag oracle before, the reduction outputs it as its forgery.

This gives us the following bound:

$$\Pr[\text{D.7 reaches bad}] \leq \Pr[\text{D.8 reaches bad}] + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC},\ \mathcal{B}_8}.$$

Since Game D.8 always rejects the *binder* value or `ServerFinished`, it never reaches the bad event. Therefore, $\Pr[\text{D.8 reaches bad}] = 0$.

We can now derive the following bound:

$$\begin{aligned}\mathsf{Adv}^{\text{Multi-Stage.IND}}_{\text{KEMTLS-PSK},\mathcal{A}} \leq\ & \frac{n_S^2}{2^{|nonce|}} + \mathsf{Adv}^{\text{COLL}}_{\text{H},\ \mathcal{B}_1} \\ & + 8n_S \left(\begin{array}{l} n_{psk} \left(\begin{array}{l} 3 \cdot \mathsf{Adv}^{\text{dual-PRF}}_{\text{HKDF.Extract},\mathcal{B}_2} + 3 \cdot \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\mathcal{B}_3} \\ +\ 3 \cdot \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Extract},\ \mathcal{B}_4} + 3 \cdot \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\ \mathcal{B}_5} \\ +\ \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\mathcal{B}_7} + \mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\ \mathcal{B}_8} \end{array} \right) \\ +\ n_S \left(\begin{array}{l} \mathsf{Adv}^{\text{IND-1CCA}}_{\text{KEM}_e,\ \mathcal{B}_6} + \mathsf{Adv}^{\text{dual-PRF}}_{\text{HKDF.Extract},\mathcal{B}_4} \\ \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\ \mathcal{B}_5} \end{array} \right) \end{array} \right).\end{aligned}$$

□

6.3.2 Proving the Absence of Malicious Acceptance

We aim to bound the adversary's ability to cause a stage in the multi-stage game in Figure 5.0 to maliciously accept. We start from the assumption that at least one stage has maliciously accepted. According to the definition of malicious acceptance in Figure 5.0, this means that a) the said stage is (possibly retroactively) explicitly authenticated, b) it does not have a partnered session and c) the pre-shared key was not corrupted before the said stage was explicitly authenticated.

Proof We designate the malicious acceptance event as a bad event. The proof proceeds via a series of game hops similar to **Case D** in the key indistinguishability proof where we bound the probability of the bad event.

Game 0. The original Multi-Stage.MAL game. That is, the multi-stage game that can only be won by malicious acceptance:

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

Game 1 (no nonce repetition). We abort if any two honest sessions compute the same nonce. The probability of such a repetition is computed via a birthday bound. Since there are n_S session and $2^{|nonce|}$ possible nonce values, this gives the following bound:

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

Game 2 (no hash collisions). We abort if any two honest sessions compute the same hash value on two different input values. We can construct a reduction adversary $\mathcal{B}_1$ against the hash collision resistance of hash function H which will output the two input values as its collision. This yields the following bound:

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

Game 3 (guessing the maliciously accepted stage). We guess the identity of the maliciously accepted session. Since there are n_S each with 8 stages, the advantage is reduced by a factor of $1/8n_S$. We, thus, get the following bound:

$$\mathsf{Adv}^{G_2}_{\mathcal{A}} \leq 8n_S \cdot \mathsf{Adv}^{G_3}_{\mathcal{A}}.$$

Games 4-10. Similar to Games D.1-D.7 in the key indistinguishability proof.

Game 11 (HMAC forgery). This game is similar to the previous game, except that it rejects the finished message it receives. That is, if the tested session is a client session, it rejects the `ServerFinished` message and if the tested session is a server session, it rejects the `ClientFinished` message. We know that the guessed session has maliciously accepted in the previous game. That is, it has accepted a finished message without a partner session. We show that this is reducible to forging an HMAC tag.

We construct reduction adversary $\mathcal{B}_9$ as a EUF-CMA adversary to the HMAC scheme with key fk_S in case we are rejecting the `ServerFinished` message or fk_C in case we are rejecting the `ClientFinished` message. When the tested client (resp. server) receives a `ServerFinished` (resp. `ClientHello`), we know that no honest contributive partner computed the MAC tag over the transcript contained in this message. Therefore the reduction $\mathcal{B}_9$ can output this MAC tag as its forgery and if it is accepted, then it has won the EUF-CMA game. We note that this reduction is similar to reduction $\mathcal{B}_8$ in Game D.8, except that it rejects the `ClientFinished` when the tested session is a server session, whereas $\mathcal{B}_8$ rejects the *binder* value. Since both reductions are similar and run in essentially the same time, we can say that $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC},\,\mathcal{B}_8} \approx \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC},\,\mathcal{B}_9}$ and we can use the advantage term $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC},\,\mathcal{B}_8}$ in the bound for this game hop:

Thus, we get the following bound:

$$\mathsf{Adv}^{G_8}_{\mathcal{A}} \leq \mathsf{Adv}^{G_9}_{\mathcal{A}} + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HMAC},\,\mathcal{B}_8}.$$

We can now derive the following bound:

$$\begin{aligned}\mathsf{Adv}^{\text{Multi-Stage.MAL}}_{\text{KEMTLS-PSK},\mathcal{A}} &\leq \frac{n_S^2}{2^{|nonce|}} + \mathsf{Adv}^{\text{COLL}}_{\text{H},\,\mathcal{B}_1} \\ &+ 8n_S \cdot n_{psk} \begin{pmatrix} \mathsf{Adv}^{\text{dual-PRF}}_{\text{HKDF.Extract},\mathcal{B}_2} + \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\mathcal{B}_3} \\ + \ \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Extract},\,\mathcal{B}_4} + \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\,\mathcal{B}_5} \\ + \ \mathsf{Adv}^{\text{PRF}}_{\text{HKDF.Expand},\mathcal{B}_7} + \mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\,\mathcal{B}_8} \end{pmatrix}.\end{aligned}$$

We have now proven the statement of Theorem 6.2. □

6.4 Comparison to the Analysis of TLS 1.3 in PSK-(EC)DHE Mode

In this section, we give a brief comparison of our analysis of KEMTLS-PSK and the analysis of the TLS 1.3 PSK-(EC)DHE handshake performed by Dowling et al. in [17].

Security properties. Our model, drawing from the full 1-RTT KEMTLS model in [13], considers both implicit and explicit authentication, while the model of Dowling et al. considers only implicit authentication. They differentiate unilateral (server-only) and mutual authentication, while we define authentication separately for client and server roles. The notion of retroactive authentication exists in both models.

Both protocols provide forward secrecy and non-replayability for non-0-RTT stages. However, our formulation of forward secrecy, which also draws from full 1-RTT KEMTLS model, is slightly more granular with two different levels of forward secrecy depending on when the adversary is allowed to corrupt the PSK.

Cryptographic assumptions. The main difference in the cryptographic assumptions used by both analyses is that KEMTLS-PSK depends on the IND-1CCA security of the ephemeral KEM_e to prove the security of the ephemeral secret ss_e, while TLS 1.3 PSK-(EC)DHE depends on a variant of the pseudorandom-function oracle-Diffie–Hellman (PRF-ODH) security of the HKDF.Extract along with the ephemeral Diffie–Hellman key shares to prove the security of its handshake secret. While PRF-ODH is an assumption tailored for Diffie–Hellman key exchange and, therefore, breaks down in a post-quantum setting, IND-CCA is a generic notion that is independent of the different hardness assumptions used to build KEMs.

Otherwise, both analyses depend on the PRF and dual-PRF security of the HKDF.Extract and HKDF.Expand functions for key indistinguishability as well as the EUF-CMA security of the HMAC scheme for authentication.

Chapter 7

Conclusion

In this thesis, we introduced KEMTLS-PSK, a resumption handshake for the KEMTLS protocol whose full 1-RTT handshake is designed by Schwabe, Stebila and Wiggers [13] as a signature-free, quantum-resistant replacement for TLS 1.3. KEMTLS-PSK relies on an existing pre-shared key established by client and server in a previous full 1-RTT handshake or through an out-of-band mechanism. It also allows for 0-RTT key derivation which enables the client to send early data to the server with the first flight of messages. Thus, we aim to aid future efforts of deploying KEMTLS instead of TLS 1.3 on the Internet with minimal disruption to existing infrastructure and with comparable efficiency and security guarantees.

We then designed a multi-stage security model in order to analyze the security of KEMTLS-PSK. Our security model assigns security properties such as implicit/explicit authentication, forward secrecy and replayability for each stage of the protocol. For our design, we draw from the work of Dowling et al. [17], which adapts the multi-stage security model of Fischlin and Günther [16] to analyze TLS 1.3 in full 1-RTT and resumption modes, and from the multi-stage model introduced by Schwabe, Stebila and Wiggers to analyze the full KEMTLS handshake in [13]. We define our Multi-Stage security game in pseudo-code, instead of the textual format used in these previous works, to avoid ambiguity.

Given the security properties captured by our model, we proceeded to analyze the security of KEMTLS-PSK in detail. We provided a game-playing proof which shows that for every possible scenario of valid adversarial behavior, our protocol preserves the Bellare–Rogaway key indistinguishability property so long as its is instantiated with secure cryptographic primitives. That is, we bound the advantage of an adversary playing against our Multi-Stage security game in terms of the advantages of reduction algorithms playing against adversaries of standard security games of the different cryptographic primitives used in our protocol. We also derive a bound for the

advantage of an adversary attempting to cause a session to *maliciously accept* i.e., complete the handshake without an honest partner.

We, thus, showed that our proposed protocol provides security guarantees which are comparable to those proved by Dowling et al. for TLS 1.3 in PSK-(EC)DHE mode. It provides some level of authentication and forward secrecy to non-0-RTT stages. In addition, we proved the explicit authentication property which is not discussed for TLS 1.3.

Future work. There are various areas of improvement and development that can be addressed in future work. For instance, in some game hops in our proof, we guess the tested session, the used PSK or the identity of the partner session. These "guessing" game hops can be avoided in favor of existing or novel proof techniques that derive tighter security bounds (for example, tighter security bounds are derived for SIGMA and TLS 1.3 in [21]).

Future work can also implement KEMTLS-PSK, perhaps by incorporating it into the full 1-RTT KEMTLS implementation of [13], and benchmark its performance against that of TLS 1.3 in PSK-(EC)DHE mode.

Furthermore, our KEMTLS-PSK design (and also that of the full KEMTLS handshake), can be augmented to a hybrid design which composes multiple KEMs in the same handshake and mixes their share secrets into the key schedule. This way, the handshake could resort to using one or more post-quantum KEMs to defend against quantum-enabled adversaries and one or more pre-quantum KEMs as a fallback in case security faults are later discovered in the post-quantum ones. This hybrid approach has been experimented with for TLS 1.2 in the past, most notably by Google through its CECPQ cipher suites [8, 9] and by Amazon Web Services (AWS) [24] which allowed three KEMs from the second round of the NIST post-quantum standardization process to be combined with traditional Diffie–Hellman.

Bibliography

[1] Paul E. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207, February 2002.

[2] MTA-STS. https://dmarcian.com/mta-sts/.

[3] Karl Norrman, David McGrew, Mats Naslund, Elisabetta Carrara, and Mark Baugher. The Secure Real-time Transport Protocol (SRTP). RFC 3711, March 2004.

[4] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[5] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[6] 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*, pages 553–570, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[7] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 327–343, Austin, TX, USA, August 10–12, 2016. USENIX Association.

[8] Adam Langley. CECPQ1 results, 2016. https://www.imperialviolet.org/2016/11/28/cecpq1.html.

[9] Adam Langley. CECPQ2 results, 2018. https://www.imperialviolet.org/2018/12/12/cecpq2.html.

[10] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 14–37, Cham, 2017. Springer International Publishing.

[11] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the nist post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[12] Dustin Moody. The 2nd Round of the NIST PQC Standardization Process-Opening Remarks at PQC 2019. https://csrc.nist.gov/Presentations/2019/the-2nd-round-of-the-nist-pqc-standardization-proc.

[13] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. Cryptology ePrint Archive, Report 2020/534, 2020. https://ia.cr/2020/534.

[14] Jim Roskind. Quick udp internet connections: Multiplexed stream transport over udp. *Adresse: https://docs. google. com/document/d/1RNHkx_ VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit (besucht am 05. 07. 2017)*, 2012.

[15] Nick Sullivan. Introducing Zero Round Trip Time Resumption (0-RTT), 2017. https://blog.cloudflare.com/introducing-0-rtt/.

[16] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of google's quic protocol. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1193–1204, New York, NY, USA, 2014. Association for Computing Machinery.

[17] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the tls 1.3 handshake protocol. Cryptology ePrint Archive, Report 2020/1044, 2020. https://ia.cr/2020/1044.

[18] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany.

[19] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

[20] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 232–249, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[21] Hannah Davis and Felix Günther. Tighter proofs for the sigma and tls 1.3 key exchange protocols. In *International Conference on Applied Cryptography and Network Security*, pages 448–479. Springer, 2021.

[22] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011: 18th Conference on Computer and Communications Security*, pages 51–62, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

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

[24] Alex Weibel. Round 2 post-quantum TLS is now supported in AWS KMS, 2020. https://aws.amazon.com/blogs/security/round-2-post-quantum-tls-is-now-supported-in-aws-kms/.