Network Working Group
Internet-Draft
Intended status: Informational
Expires: 17 October 2025

J. Vos
S. Jarecki
University of California, Irvine
C. A. Wood
Apple, Inc.
15 April 2025

Hybrid Post-Quantum Password Authenticated Key Exchange draft-vos-cfrg-pqpake-00

Abstract

This document describes the CPaceOQUAKE+ protocol, a hybrid asymmetric password-authenticated key exchange (aPAKE) that supports mutual authentication in a client-server setting secure against quantum-capable attackers. CPaceOQUAKE+ is the result of a KEM-based transformation from the hybrid symmetric PAKE protocol called CPaceOQUAKE that is also described in this document. This document recommends configurations for CPaceOQUAKE+.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://example.com/LATEST. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-vos-cfrg-pqpake/.

Discussion of this document takes place on the CFRG Crypto Forum Research Group mailing list (mailto:WG@example.com), which is archived at https://example.com/WG.

Source for this draft and an issue tracker can be found at https://github.com/USER/REPO.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

Asymmetric (or Augmented) Password Authenticated Key Exchange (aPAKE) protocols are designed to provide password authentication and mutually authenticated key exchange in a client-server setting without relying on a public key infrastructure (PKI) and without disclosing passwords to servers or other entities other than the client machine. The only stage where PKI is required is during a client's registration.

In the asymmetric PAKE setting, the client first registers a password verifier with the server. A verifier is a value that is derived from the password and which the server will later use to verify the client knowledge of the password. After registration, the client uses its password and the server uses the corresponding verifier to establish an authenticated shared secret such that the server learns nothing of the client's password.

OPAQUE-3DH [OPAQUE] and SPAKE2+ [SPAKE2PLUS] are two examples of specified aPAKE protocols. These protocols provide security in classical threat models. However, in the presence of a quantum-capable attacker, both OPAQUE and SPAKE2+ fail to provide the desired level of security. Both protocols are vulnerable to a Harvest Now, Decrypt Later attack executed by a quantum-capable attacker, in which the attacker learns the shared secret and uses it to compromise application traffic. Upgrading both protocols to provide post-quantum security is non-trivial, especially as there are no known

efficient constructions for certain building blocks used in these protocols (such as the OPRF used in OPAQUE-3DH). As the threat of quantum-capable attackers looms, the viability of existing aPAKE protocols in practice diminishes in time.

This document describes the CPaceOQUAKE+ protocol, an aPAKE that supports mutual authentication in a client-server setting secure against quantum-capable attackers. CPaceOQUAKE+ is the result of a KEM-based transformation from the hybrid symmetric PAKE protocol called CPaceOQUAKE.

This document fully specifies CPaceOQUAKE+ and all dependencies necessary to implement it. Section 7 provides recommended configurations.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.1. Notation and Terminology

The following functions and operators are used throughout the document.

* The function random(n) generates a cryptographically secure pseudorandom byte string of length n bytes.

* The associative binary operator || denotes concatenation of two byte strings.

* The binary function XOR(a, b) denotes an element-wise XOR operation between two byte strings a and b of the same length.

* The functions bytes_to_int and int_to_bytes convert byte strings to and from non-negative integers. bytes_to_int and int_to_bytes are implemented as OS2IP and I2OSP as described in [RFC8017], respectively.

* The function lv_encode encodes a byte string with a two-byte, big-endian length prefix. For example, lv_enode((0x00, 0x01, 0x02)) = (0x00, 0x03, 0x00, 0x01, 0x02). The function lv_decode parses a byte string that is expected to be encoded with a two-byte length preceding the remaining bytes, e.g., lv_decode((0x00, 0x03, 0x00, 0x01, 0x02)) = (0x00, 0x01, 0x02). Note that lv_decode can fail when the length of the actual bytes does not match that encoded in the prefix. For example, lv_decode((0xFF, 0xFF, 0x00)) will fail.

* The notation bytes[l..h] refers to the slice of byte array bytes starting at index l and ending at index h-1. For example, given bytes = (0x00, 0x01, 0x02), then bytes[0..1] = 0x00 and bytes[0..3] = (0x00, 0x01, 0x02). Similarly, the notation bytes[l..] refers to the slice of the byte array bytes starting at l until the end of bytes, i.e.., bytes[l..] = bytes[l..len(bytes)].

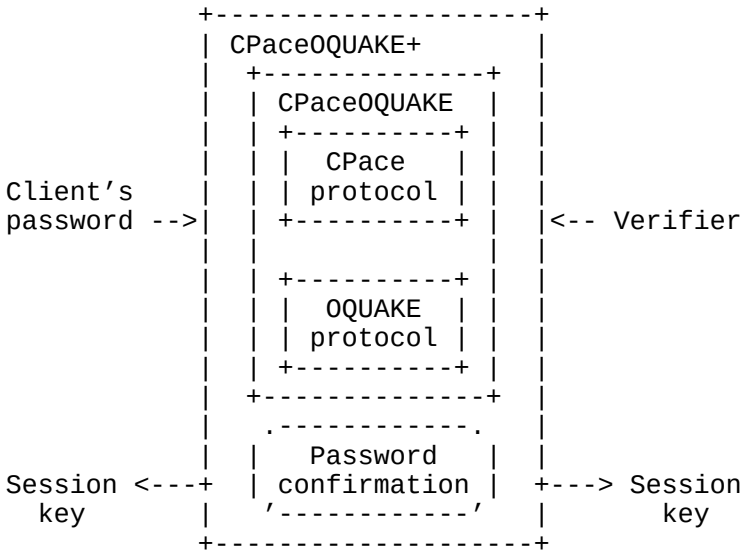
All algorithms and procedures described in this document are laid out in a Python-like pseudocode. Each function takes a set of inputs and parameters and produces a set of output values. Parameters become constant values once the protocol variant and the configuration are fixed.

3. Overview

This document aims to specify two protocols: a symmetric and an asymmetric hybrid PAKE. In the symmetric PAKE setting, the client and server share a password and use it to establish an authenticated shared secret. In the asymmetric PAKE setting, the client first registers a password verifier with the server. A verifier is a value that is derived from the password and which the client will later use to demonstrate knowledge of the password. After registration, the client uses its password and the server uses the corresponding verifier to establish an authenticated shared secret such that the server learns nothing of the client’s password.

The aPAKE specified in this document is composed of multiple smaller protocols, including the hybrid symmetric PAKE protocol called CPaceOQUAKE. CPaceOQUAKE is in turn a composition of two other PAKE protocols: the existing CPace [CPACE] and a new post-quantum PAKE called OQUAKE. To achieve the asymmetric property, the aPAKE also builds upon a password confirmation sub-protocol as specified in Section 6.2.

We refer to the fully composed aPAKE as CPaceOQUAKE+. An abstract overview of the composition of this protocol is shown in the figure below. In the subsequent sections we break down the sub-protocols into even smaller building blocks.



We note that this standard only specifies the composition of CPace and OQUAKE. It is not necessarily true that one can securely compose all PAKEs this way.

The rest of this document specifies CPaceOQUAKE+ and its dependencies. Section 5 specifies the CPaceOQUAKE protocol, and Section 6 specifies the CPaceOQUAKE+ protocol, incorporating the former protocol. Each of these pieces build upon the cryptographic dependencies specified in Section 4.

4. Cryptographic Dependencies

The protocols in this document have four primary dependencies:

* Key Encapsulation Mechanism (KEM); Section 4.1

* Binary Uniform Key Encapsulation Mechanism (BUKEM); Section 4.2

* Key Derivation Function (KDF); Section 4.3

* Key Stretching Function (KSF); Section 4.4

Section 7 specifies different combinations of each of these dependencies that are suitable for implementation.

4.1. Key Encapsulation Mechanism

A Key Encapsulation Mechanism (KEM) is an algorithm that is used for exchanging a secret from one party to another. We require an IND-CCA-secure KEM with key derivation from a seed. It consists of the following syntax.

* DeriveKeyPair(seed): Deterministic algorithm to derive a key pair (sk, pk) from the byte string seed, where seed SHOULD have Nseed bytes.

* Encaps(pk): Randomized algorithm to generate an ephemeral, fixed-length symmetric key (the KEM shared secret) and a fixed-length encapsulation of that key that can be decapsulated by the holder of the secret key corresponding to pk. This function can raise an EncapsError on encapsulation failure.

* Decaps(ct, skR): Deterministic algorithm using the secret key sk to recover the ephemeral symmetric key (the KEM shared secret) from its encapsulated representation ct. This function can raise a DecapsError on decapsulation failure.

* Nseed: The length in bytes of the seed used to derive a key pair.

* Nct: The length in bytes of an encapsulated key produced by this KEM.

* Npk: The length in bytes of a public key for this KEM.

This specification uses X-Wing [XWING].

4.2. Binary Uniform KEM

A binary uniform KEM supports the same functions as defined above for a KEM, and it must also be IND-CCA secure, but it must also achieve two additional security properties. Namely, in addition to IND-CCA security, a binary uniform KEM requires that:

1. Public keys are indistinguishable from random strings of bytes (of the same length); and

2. Ciphertexts are anonymous in the presence of chosen ciphertext attack (ANO-CCA).

These additional properties are crucial for the security of OQUAKE. In other words, one MUST NOT use a KEM that has no uniform public keys and no anonymous ciphertexts in place of a uniform KEM.

This specification uses a variant of ML-KEM768 [FIPS203], denoted ML-BUKEM768. This is instantiated with "KemeleonNR - ML-KEM768" [KEMELEON]. Note that, while Kemeleon provides uniform encoding for KEM ciphertexts and public keys, we only require uniform enoding for public keys. Future specifications can replace use of Kemeleon with a binary uniform KEM that is more efficient if one becomes available.

4.3. Key Derivation Function

A Key Derivation Function (KDF) is a function that takes some source of initial keying material and uses it to derive one or more cryptographically strong keys. This specification uses a KDF with the following API and parameters:

* Extract(salt, ikm): Extract a pseudorandom key of fixed length Nx bytes from input keying material ikm and an optional byte string salt.

* Expand(prk, info, L): Expand a pseudorandom key prk using the optional string info into L bytes of output keying material.

* Nx: The output size of the Extract() function in bytes.

4.4. Key Stretching Function

This specification makes use of a Key Stretching Function (KSF), which is a slow and expensive cryptographic hash function with the following API:

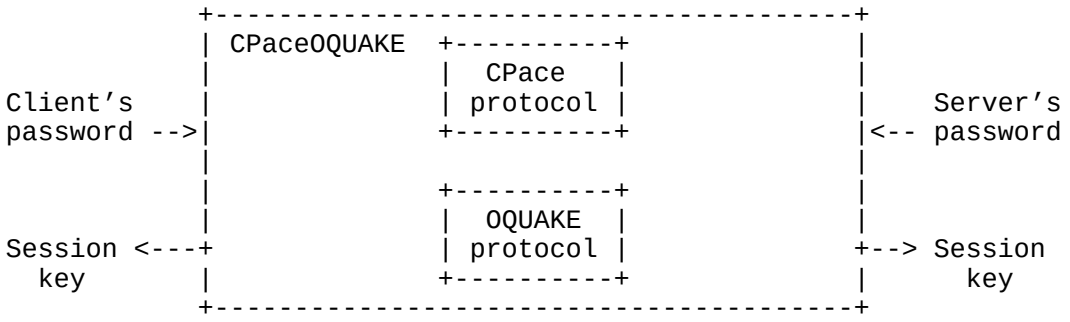
* Stretch(msg, salt, L): Apply a key stretching function to stretch the input msg and salt salt, hardening it against offline dictionary attacks. This function also needs to satisfy collision resistance. The output is a string of L bytes.

5. CPaceOQUAKE Protocol

The hybrid, symmetric PAKE protocol, denoted CPaceOQUAKE consists of CPace [CPACE] combined with OQUAKE [ABJ25]. OQUAKE is a PAKE built from a BUKEM and KDF, using a 2-rounds of Feistel network to password-encrypt the BUKEM public key. The OQUAKE protocol is based on the "NoIC" protocol analyzed in [ABJ25].

The CPaceOQUAKE protocol is based on the ‘Sequential PAKE Combiner’ protocol proposed by [HR24]. A very close variant of this protocol was also analyzed in [LL24].

At a high level, CPaceOQUAKE is a two-round protocol that runs between client and server wherein, upon completion, both parties share the same session key if they agree on the password-related string (PRS). Otherwise, they obtain random session keys. This is summarized in the diagram below.



CPaceOQUAKE composes CPace and OQUAKE by first running CPace between client and server, and then incorporating the CPace session key into the password before running OQUAKE between the server and client. We explain the composition in more detail in Section 5.3.

As describes in Section 5.1 and Section 5.2, both CPace and OQUAKE take as input optional client and server identifiers, denoted U and S, respectively. See Section 9.1 for more discussion about these identities and how they are chosen in practice.

5.1. CPace Specification

CPace is a classical elliptic curve-based PAKE [CPACE]. This section wraps the CPace specification in a consistent interface. We use an interactive version of CPace that takes two rounds, in which there is a designated initiator and responder. In other words, the responder only starts executing the protocol after it received the first message from the initiator.

The flow of the protocol consists of three messages sent between initiator and responder, produced by the functions Init, Respond, and Finish, described below. Both parties take as input a password-related string PRS, an optional unique shared session identifier sid, and an optional client identifier U and server identifier S (e.g., a device identifier, an IP address, or URL pertaining to the client and server). Upon completion, both parties obtain matching session keys if their PRS, sid, key length (specified by N), and client and server identifiers match. Otherwise, they obtain random keys. In exceptional cases, the protocol aborts.

5.1.1. Initiation

The initiator starts the protocol using its password-related string PRS. Additionally, it may bind the session to an existing shared session identifier sid. CPace also allows to bind the session to an existing channel identifier. To remain consistent with the other PAKEs in this specification, the channel identifier is the concatenation of optional client and server identifiers.

```
CPace.Init

Input:
- PRS, password-related string, a byte string
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- ya, discrete logarithm intended to be stored in secret until the protocol finishes
- Ya, public point, intended to be sent to the responder

Parameters:
- G, a group environment as specified in CPace

def Init(PRS, sid, U, S):
  g = G.calculate_generator(H, PRS, U || S, sid)
  ya = G.sample_scalar()
  Ya = G.scalar_mult(ya, g)
  return ya, Ya
```

5.1.2. Response

The responder performs the same actions as the initiator. Since it already received the initiator’s message, it can immediately finish its execution of the protocol. It outputs the shared secret and a message Yb intended to be sent to the initiator.

```
  CPace.Respond

  Input:
  - PRS, password-related string, a byte string
  - Ya, public point, received from the initiator
  - sid, session identifier, a byte string
  - U and S, client and server identifiers

  Output:
  - ISK, the established shared secret
  - Yb, public point, intended to be sent to the initiator

  Parameters:
  - G, a group environment as specified in CPace
  - H, a hash function as specified in CPace

  Exceptions:
  - CPaceError, raised when an invalid value was encountered in CPace

  def Respond(PRS, Ya, sid, U, S):
    g = G.calculate_generator(H, PRS, U || S, sid)
    yb = G.sample_scalar()
    Yb = G.scalar_mult(yb, g)

    K = G.scalar_mult_vfy(yb, Ya)
    If K = G.I, raise CPaceError

    ISK = H.hash(lv_cat(G.DSI || b"_ISK", sid, K) || transcript(Ya, Yb))

    return ISK, Yb
```

The functions lv_cat and transcript are defined in [CPACE].

5.1.3. Finish

The initiator finishes the protocol by combining the discrete logarithm ya generated by CPace.Init and the message Yb received from the responder.

```
  CPace.Finish

  Input:
  - ya, discrete logarithm that was generated using CPace.Init
  - Yb, public point, received from the responder
  - sid, session identifier, a byte string

  Output:
  - ISK, the established shared secret

  Parameters:
  - G, a group environment as specified in CPace
  - H, a hash function as specified in CPace

  Exceptions:
  - CPaceError, raised when an invalid value was encountered in CPace

  def Finish(ya, Yb, sid):
    K = G.scalar_mult_vfy(ya, Yb)
    If K = G.I, raise CPaceError

    ISK = H.hash(lv_cat(G.DSI || b"_ISK", sid, K) || transcript(Ya, Yb))

    return ISK
```

5.2. OQUAKE Specification

OQUAKE is a PAKE built on a BUKEM and KDF. If the BUKEM provides security against quantum-enabled attacks, then so does OQUAKE. It consists of three messages sent between initiator and responder, produced by the functions Init, Respond, and Finish, described below. Both parties take as input a password-related string PRS, an optional session identifier sid, and an optional client identifier U and server identifier S. Upon completion, both parties obtain matching session keys if their PRS, sid, key length (specified by N), and client and server identifiers match. Otherwise, they obtain random session keys.

The shared session identifier has the following requirements. If a client and server identifier are provided:

* The session identifier must match between the client and server

* This session identifier has not been used before in a session between the client and server

If no client and server identifiers are provided:

* The session identifier must match between the client and server

* This session identifier has not been used before by the client or server in any session with any other party

These requirements originate from the security proof for OQUAKE. If these requirements are not met, the proof does not apply, but this does not mean that the protocol becomes vulnerable.

5.2.1. Initiation

Init takes as input the initiator’s PRS, an optional session identifier sid, and optional client and server identifiers U and S. It produces a context for the initiator to store, as well as a protocol message that is sent to the responder. Its implementation is as follows.

OQUAKE.Init

Input:
- PRS, password-related string, a byte string
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- context, opaque state for the initiator to store
- msg, an encoded protocol message for the initiator to send to the responder

Parameters:
- BUKEM, a BUKEM instance
- KDF, a KDF instance
- DST, domain separation tag, a byte string

```
def Init(PRS, sid, U, S):
  seed = random(BUKEM.Nseed)
  (pk, sk) = BUKEM.DeriveKeyPair(seed)

  r = random(3 * Nsec)

  fullsid = encode_sid(sid, U, S)

  // T = XOR(pk, H(fullsid, PRS, r))
  prk_T_pad = KDF.Extract(PRS, DST || "OQUAKE" || fullsid || r)
  T_pad = KDF.Expand(prk_T_pad, DST || "T_pad", Npk)
  T = XOR(pk, T_pad)

  // s = XOR(r, H(fullsid, PRS, T))
  prk_s_pad = KDF.Extract(PRS, DST || "OQUAKE" || fullsid || T)
  s_pad = KDF.Expand(prk_s_pad, DST || "s_pad", 3 * Nsec)
  s = XOR(r, s_pad)

  init_msg = s || T

  return Context(PRS, sk, s, T, fullsid), init_msg
```

The encode_sid function is defined below.

```
encode_sid

Input:
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- fullsid, a byte string

Parameters:
- BUKEM, a BUKEM instance
- KDF, a KDF instance

def encode_sid(sid, U, U):
  fullsid =
    bytes_to_int(len(sid), 4) || sid ||
    bytes_to_int(len(U), 4) || U ||
    bytes_to_int(len(S), 4) || S
  return fullsid
```

5.2.2. Response

Respond takes as input the PRS, the initiator’s protocol message, an optional session identifier, and optional client and server identifiers. It produces a 32-byte symmetric key and a protocol message intended to be sent to the initiator. Its implementation is as follows.

OQUAKE.Respond

Input:
- PRS, password-related string, a byte string
- init_msg, encoded protocol message, a byte string
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- ss, output shared secret, a byte string of 32 bytes
- resp_msg, encoded protocol message, a byte string

Parameters:
- BUKEM, a BUKEM instance
- KDF, a KDF instance
- DST, domain separation tag, a byte string

```
def Respond(PRS, init_msg, sid, U, S):
  (s, T) = init_msg[0..(3 * Nsec)], init_msg[(3 * Nsec)..]

  fullsid = encode_sid(sid, U, S)
  prk_s_pad = KDF.Extract(PRS, DST || "OQUAKE" || fullsid || T)
  s_pad = KDF.Expand(prk_s_pad, DST || "s_pad", 3 * Nsec)
  r = XOR(s, s_pad)

  prk_T_pad = KDF.Extract(PRS, DST || "OQUAKE" || fullsid || r)
  T_pad = KDF.Expand(prk_T_pad, DST || "T_pad", Npk)
  pk = XOR(T, T_pad)

  (ct, k) = BUKEM.Encaps(pk)

  prk_sk = KDF.Extract(PRS, DST || "OQUAKE" || fullsid || k)
  key = KDF.Expand(prk_sk, DST || "sk", Nkey)

  h = KDF.Expand(prk_sk, DST || "confirm", Nkc)

  resp_msg = ct || h

  return resp_msg, key
```

5.2.3. Finish

Finish takes as input the initiator-created context that is output from Init as well as the responder’s reply message resp_msg. It produces a symmetric key that is output to the initiator. Its implementation is as follows.

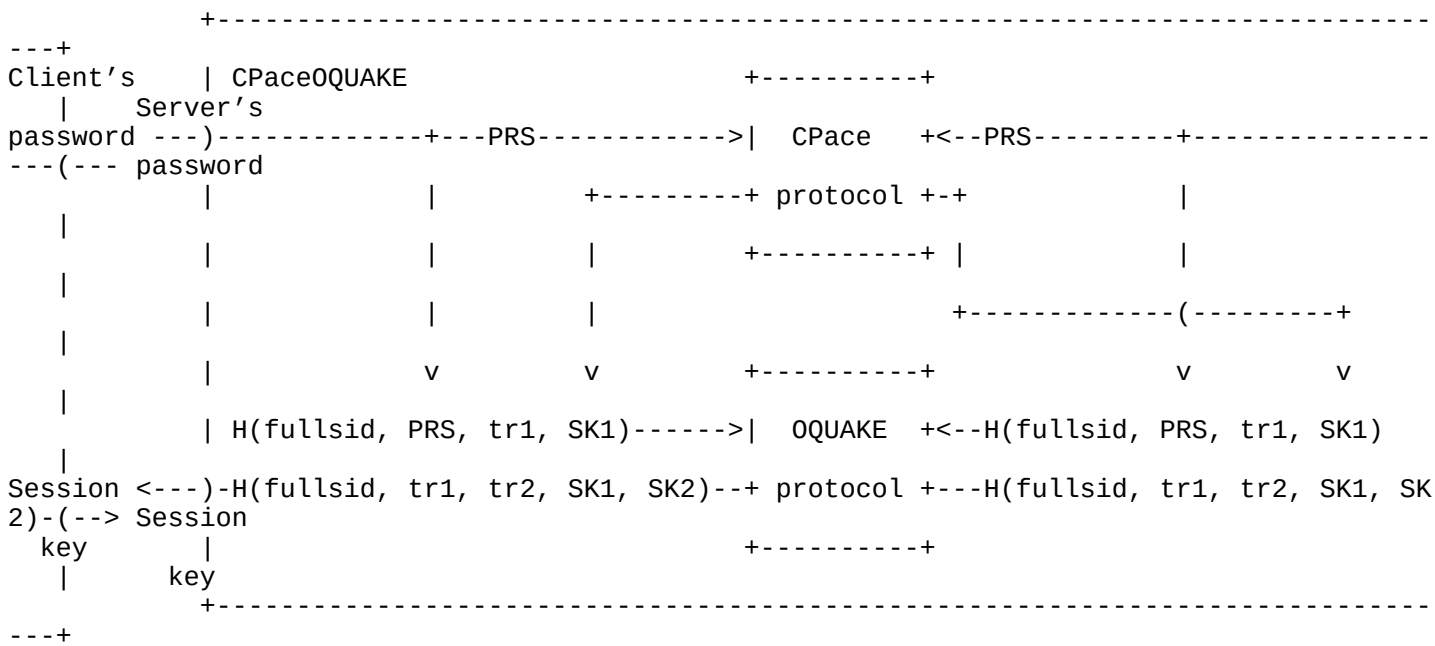
```
OQUAKE.Finish

Input:
- context, opaque state for the initiator to store TODO
- resp_msg, encoded protocol message, a byte string

Output:
- ss, output shared secret, a byte string of 32 bytes

Parameters:
- BUKEM, a BUKEM instance
- KDF, a KDF instance
- DST, domain separation tag, a byte string

Exceptions:
- AuthenticationError, raised when the key confirmation fails

def Finish(context, resp_msg):
  (PRS, sk, s, T, fullsid) = context
  ct, h = resp_msg[0..Npk], resp_msg[Npk..]

  try:
    k = BUKEM.Decaps(sk, ct)
    prk_sk = KDF.Extract(PRS, DST || "OQUAKE" || fullsid || k)
    key = KDF.Expand(prk_sk, DST || "sk", Nkey)

    h_expected = KDF.Expand(prk_sk, DST || "confirm", Nkc)
    if h != h_expected:
      return random(Nkey)

    return key
  catch DecapsError:
    return random(Nkey)
```

5.3. Composition of CPace & OQUAKE

CPaceOQUAKE is a sequential composition of CPace (see Section 5.1) and OQUAKE (see Section 5.2). Whereas running CPace and OQUAKE in parallel realizes a worst-of-both worlds PAKE, this sequential composition realizes a best-of-both worlds PAKE. In other words, CPaceOQUAKE remains as secure as the strongest PAKE, resisting attacks that break the classical CPace (e.g. by a quantum-capable attacker) or attacks that break the quantum-resistant OQUAKE (e.g. by a flaw in the BUKEM). This assumes that OQUAKE is instantiated with a quantum-resistant BUKEM.

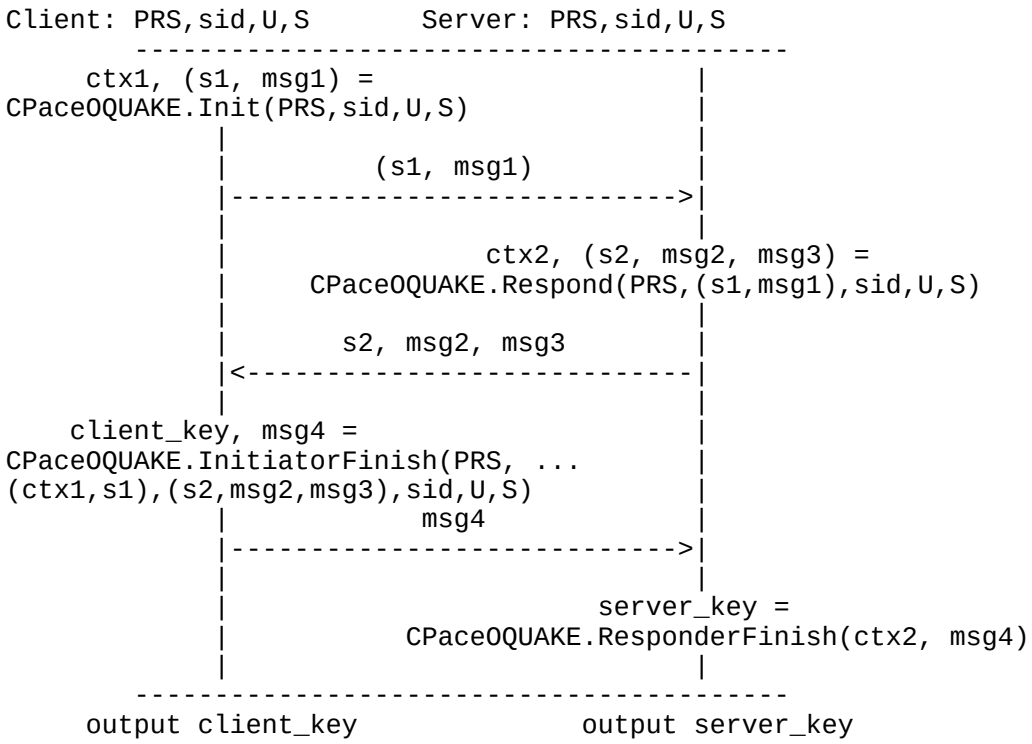
To be precise, CPaceOQUAKE first runs CPace using password-related string PRS, establishing a session key SK1 with the associated transcript tr1. It then initiates OQUAKE using the password-related string H(fullsid, PRS, tr1, SK1); a secret derived from the the original password-related string and the outputs from the CPace instance. Here, fullsid is the output of encode_sid(sid, U, S). The final session key is then a hash of fullsid, the original password-related string, both CPace and OQUAKE transcripts (tr1 and tr2, respectively), and both session keys output from CPace and OQUAKE (SK1 and SK2, respectively), i.e., H(fullsid, tr1, tr2, SK1, SK2).

This is outlined in the diagram below. In CPaceOQUAKE, CPace is initiated by the first party, while OQUAKE is initiated by the other party. This results in a protocol that requires three messages.



Unlike OQUAKE, CPaceOQUAKE does not require a shared session identifier sid, although this is strongly recommended. If no sid is provided, CPace will run without an sid, and OQUAKE will use a random string generated with random material provided by both parties. If an sid is provided, both CPace and OQUAKE will use this sid.

An overview of the protocol flow is shown below. The protocol has four functions. Init and InitiatorFinish are intended to be called by the initiator, and Respond and ResponderFinish are intended to be called by the responder. The following subsections specify these functions.



5.3.1. Client Initiation

The client initiates a CPace exchange with the server using input PRS, an optional session identifier sid, and optional client and server identifiers U and S. The output of this process is some context for completing the protocol and a protocol message. The client sends this message to the server.

```
CPaceOQUAKE.Init

Input:
- PRS, password-related string, a byte string
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- context, opaque state for the initiator to store
- msg, an encoded protocol message for the initiator to send to the responder

Parameters:
- CPace, parameterized instance of CPace

def Init(PRS, sid, U, S):
  ctx1, msg1 = CPace.Init(PRS, sid, U, S)
  s1 = random(32)
  init_msg = s1 || lv_encode(msg1)

  return (ctx1, s1), init_msg
```

5.3.2. Server Response

The server processes the client message using its input PRS, an optional session identifier sid, and optional client and server identifiers U and S. The first output of this function is a context that is used to finish the protocol later. The second output is a protocol message intended for the client.

The server responds to the CPace session that the client initiated, and it initiates a new OQUAKE session using both the PRS and the key established by CPace.

The server MUST ensure that exactly one of s1 and sid exists. It MUST abort if the message does not have the correct length.

```
CPaceOQUAKE.Respond

Input:
- PRS, password-related string, a byte string
- init_msg, the message received from the client
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- context, opaque state for the responder to store
- msg, an encoded protocol message for the responder to send to the initiator

Parameters:
- CPace, parameterized instance of CPace
- OQUAKE, parameterized instance of OQUAKE
- DST, domain separation tag, a byte string

def Respond(PRS, init_msg, sid, U, S):
  s1, msg1 = init_msg[0..32], lv_decode(init_msg[32..])

  key1, msg2 = CPace.Respond(PRS, msg1, sid, U, S)
  key1A = KDF.Expand(key1, DST || "prskey", Nkey)
  key1B = KDF.Expand(key1, DST || "outputkey", Nkey)

  s2 = random(32)
  prk_extended_sid = KDF.Extract(s1 || s2, DST || "CPaceOQUAKE")
  extended_sid = KDF.Expand(prk_extended_sid, DST || "SID", 32)

  fullsid = encode_sid(extended_sid, U, S)

  prk_PRS2 = KDF.Extract(PRS, DST || "CPaceOQUAKE" || fullsid || msg1 || msg2 || key1A)
  PRS2 = KDF.Expand(prk_PRS2, DST || "PRS2", Nkey)

  ctx2, msg3 = OQUAKE.Init(PRS2, extended_sid, U, S)

  resp_msg = s2 || lv_encode(msg2) || lv_encode(msg3)

  return Context(fullsid, PRS, msg1, msg2, msg3, key1B, ctx2), resp_msg
```

5.3.3. Client Finish

The client finishes the protocol by processing the server response. The client obtains a shared secret and a final message intended for the server. It does so by finishing the CPace session and responding to the OQUAKE session.

The client must ensure that exactly one of (s1, s2) and sid exists. The client should abort when the message does not have the correct length.

CPaceOQUAKE.InitiatorFinish

Input:
- PRS, password-related string, a byte string
- (ctx1, s1), the context generated by CPaceOQUAKE.Init
- resp_msg, the message received from the server
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- key, an N-byte shared secret
- msg, an encoded protocol message for the initiator to send to the responder

Parameters:
- CPace, parameterized instance of CPace
- OQUAKE, parameterized instance of OQUAKE
- DST, domain separation tag, a byte string

```
def InitiatorFinish(PRS, (ctx1, s1), resp_msg, sid, U, S):
  s2 = resp_msg[0..32]
  msg2 = lv_decode(resp_msg[32..])
  msg3 = lv_decode(resp_msg[32+len(msg2)..])

  key1 = CPace.Finish(ctx1, msg2, sid)
  key1A = KDF.Expand(key1, DST || "prskey", Nkey)
  key1B = KDF.Expand(key1, DST || "outputkey", Nkey)

  prk_extended_sid = KDF.Extract(s1 || s2, DST || "CPaceOQUAKE")
  extended_sid = KDF.Expand(prk_extended_sid, DST || "SID", 32)

  fullsid = encode_sid(extended_sid, U, S)
  prk_PRS2 = KDF.Extract(PRS, DST || "CPaceOQUAKE" || fullsid || msg1 || msg2 || key1A)
  PRS2 = KDF.Expand(prk_PRS2, DST || "PRS2", Nkey)

  key2, msg4 = OQUAKE.Respond(PRS2, msg3, extended_sid, U, S)

  prk_sessionkey = KDF.Extract(PRS, DST || "CPaceOQUAKE" || fullsid || msg1 || msg2 || ms
g3 || msg4 || key1B || key2)
  client_key = KDF.Expand(prk_sessionkey, DST || "sessionkey", Nkey)

  return client_key, msg4
```

5.3.4. Server Finish

The server finishes the protocol by finising OQUAKE using the client's response, outputting a shared secret of N bytes. It should abort when the message does not have the correct length.

```
CPaceOQUAKE.ResponderFinish

Input:
- ctx, context from the server's Response
- msg4, the message received from the server, a byte string

Output:
- key, an N-byte shared secret

Parameters:
- OQUAKE, parameterized instance of OQUAKE
- DST, domain separation tag, a byte string

def ResponderFinish(ctx, msg4):
  (fullsid, PRS, msg1, msg2, msg3, key1B, ctx2) = ctx

  key2 = OQUAKE.Finish(ctx2, msg4)

  prk_sessionkey = KDF.Extract(PRS, DST || "CPaceOQUAKE" || fullsid || msg1 || msg2 || ms
g3 || msg4 || key1B || key2)
  server_key = KDF.Expand(prk_sessionkey, DST || "sessionkey", Nkey)

  return server_key
```

6. CPaceOQUAKE+ Protocol

CPaceOQUAKE+ is the 5 message aPAKE resulting from applying a KEM-based PAKE-to-aPAKE transformation to CPaceOQUAKE. At a high level, this involves running CPaceOQUAKE on a verifier of the client's password. To ensure that the client does indeed know the password pertaining to that verifier, there is an additional password confirmation stage that uses seed derived from the password. Both the verifier and the seed are derived from the password using a key stretching function. The seed is later used to derive a KEM public key. We refer to the collection of the verifier and this public key as 'the verifiers'.

The CPaceOQUAKE+ protocol can be seen as a close variant (and a specific instance) of the 'augmented PAKE' construction presented in [LLH24] and in [Gu24].

6.1. Registering Clients

This subsection specifies functions for generating the verifiers and a protocol for registering clients.

6.1.1. Generating Verifiers

Verifiers are random-looking value derived from password-related strings from which it is computionally impractical to derive the password-related string. To make verifiers unique between different users with the same password or servers that they interact with, we employ a salt, a user account identifier, and an optional server identifier. The material required for the verifiers is generated as follows:

```
GenVerifierMaterial

Input:
- PRS, password-related string, a byte string
- salt, client-specific salt, a byte string
- U and S, client and server identifiers

Output:
- ss, output shared secret, a byte string of 32 bytes
- resp_msg, encoded protocol message, a byte string

Parameters:
- KEM, a KEM instance
- KSF, a parameterized KSF instance
- DST, domain separation tag, a byte string

def GenVerifierMaterial(PRS, salt, U, S):
  verifier_seed = KSF.Stretch(DST || PRS || U || S, salt, Nverifier + KEM.Nseed)
  verifier = verifier_seed[0:Nverifier]
  seed = verifier_seed[Nverifier:Nverifier + KEM.Nseed]
  return verifier, seed
```

To derive an actual public key from the verifier material, we use the following function:

GenVerifiers

Input:
- PRS, password-related string, a byte string
- salt, client-specific salt, a byte string
- U and S, client and server identifiers

Output:
- ss, output shared secret, a byte string of 32 bytes
- resp_msg, encoded protocol message, a byte string

Parameters:
- KEM, a KEM instance

```
def GenVerifiers(PRS, salt, U, S):
  verifier, seed = GenVerifierMaterial(PRS, salt, U, S)
  (pk, sk) = KEM.DeriveKeyPair(seed)
  return verifier, pk
```

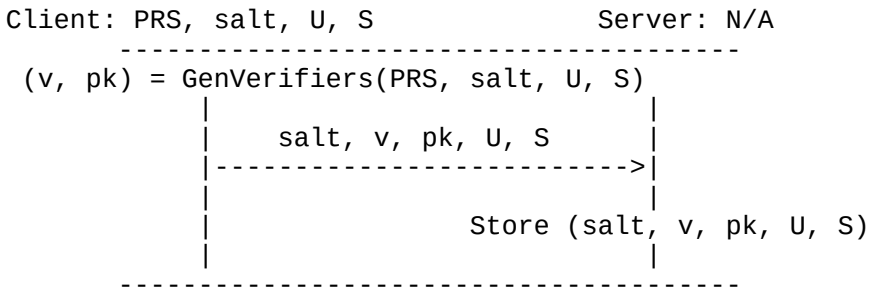
The server MUST store pk; it MUST NOT store seed.

6.1.2. Registration

The registration phase consists of one message sent from the client to the server. This message contains the verifier, a public key, and 32-byte salt. The server stores this information corresponding to the client for future use in the verification flow. This phase requires a secure channel from client to server in order to transfer the password verifier and public key. The salt can be sent in plain text.

We recommend that the salt is a random byte string: salt = random(32). However, in practice this may require an additional communication flow, used by the server to send the salt to the client before protocol CPaceOQUAKE+ starts. Instead, one may consider deriving the salt from some client-specific value that it knows and can retain locally.

A high level flow overview of the registration flow is below.

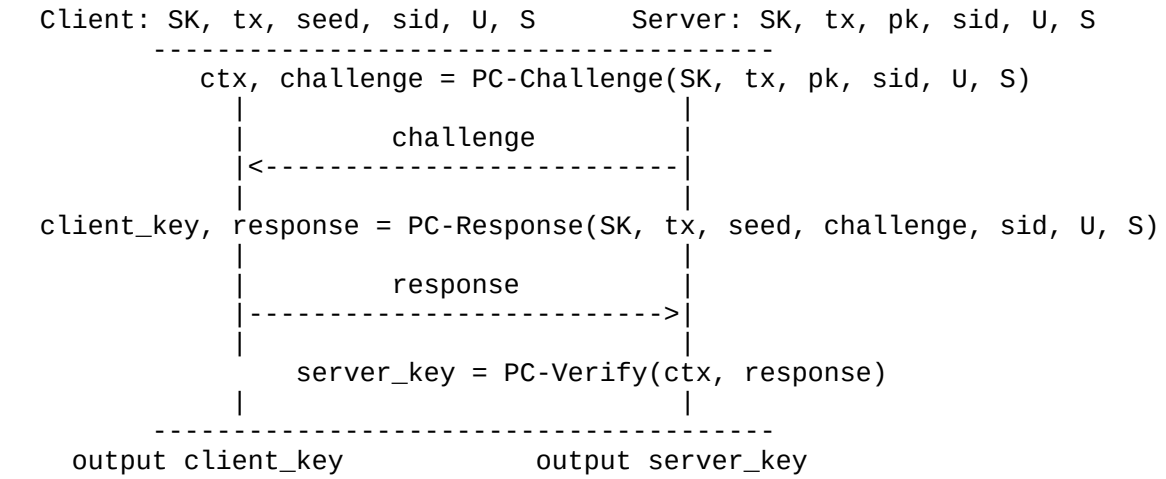


6.2. The Password Confirmation Stage

In the password confirmation (PC) stage, the client proves knowledge of its password without revealing it. It uses the registered verifiers from the previous subsection. To do so securely, it uses the key established by CPaceOQUAKE, which allows it to realize a confidential but unauthenticated channel. In other words, this password confirmation stage cannot be used by itself. This PC stage is parameterized by a KEM, KDF, KSF, and is additionally bound to the preceding protocol via an agreed-upon transcript (tx); see Section 7 for specific parameter configurations.

The password confirmation is a two-round challenge-response flow between the server and client. In particular, the server challenges the client to prove knowledge of its password. More precisely, it challenges the client to prove knowledge of a seed, derived from the GenVerifierMaterial function (and in turn derived from the password using a key stretching function). Both client and server share a symmetric key as input. Additionally, the server has the client's public key and salt stored from the previous registration flow.

A high level overview of this flow is below.



6.2.1. Server Challenge

To construct the challenge, the server encapsulates to the client's public key. From the resulting shared secret, it then derives password confirmation values and a new shared secret. The challenge message is the ciphertext encrypted using a one-time pad derived from the shared secret. The password confirmation values are byte strings of length Nkc.

The implementation MUST NOT reveal server_key from the context.

PC-Challenge

```
Input:
- SK, 32-byte symmetric key, a byte string
- transcript, the transcript from previously executed protocols to which this protocol is
 bound, a byte string
- pk, client-registered public key, a KEM public key
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- context, opaque state for the server to store values to complete the protocol
- challenge, an encoded protocol message for the server to send to the client

Parameters:
- KEM, a KEM instance
- KDF, a KDF instance
- DST, domain separation tag, a byte string

def PC-Challenge(SK, transcript, pk, sid, U, S):
  (c, k) = KEM.Encaps(pk)
  r = KDF.Expand(SK, DST || "OTP", Nct)
  enc_c = XOR(c, r)

  confirm_input = encode_sid(sid, U, S) || enc_c || transcript

  prk_k_h1 = KDF.Extract(SK, DST || "h1" || confirm_input)
  prk_k_h2 = KDF.Extract(SK, DST || "h2" || confirm_input || k)

  // Derive h1 from the full transcript excluding k
  client_confirm = KDF.Expand(prk_k_h1, DST || "client_confirm", Nkc)

  // Derive h2 || SK from the full transcript including k
  server_confirm = KDF.Expand(prk_k_h2, DST || "server_confirm", Nkc)
  server_key = KDF.Expand(prk_k_h2, DST || "key", Nkey)

  challenge = (enc_c, client_confirm)

  return Context(server_confirm, server_key), challenge
```

6.2.2. Client Response

Upon receipt of the challenge, the client recovers the KEM ciphertext by decrypting the one-time pad ciphertext included in the challenge, using the key derived from the shared secret. It then uses the seed to re-derive the KEM key pair, using the same procedure followed during the registration flow. The client then decapsulates the KEM ciphertext to recover the shared secret and derive the same password confirmation values and new shared secret as the server.

The client then checks that the server-provided confirmation value matches its own and aborts if not. Otherwise, it returns its own password confirmation value. The client outputs the new shared secret as its output.

PC-Response

Input:
- SK, 32-byte symmetric key, a byte string
- transcript, the transcript from previously executed protocols to which this protocol is bound, a byte string
- seed, seed used to derive KEM public key
- challenge, an encoded protocol message for the server to send to the client
- sid, session identifier, a byte string
- U and S, client and server identifiers

Output:
- client_key, a 32-byte string
- response, an encoded protocol message for the client to send to the server

Exceptions:
- AuthenticationError, raised when the password confirmation values do not match

Parameters:
- KEM, a KEM instance
- KDF, a KDF instance
- DST, domain separation tag, a byte string

```
def PC-Response(SK, transcript, seed, challenge, sid, U, S):
  (enc_c, client_confirm_target) = challenge
  r = KDF.Expand(SK, DST || "OTP", Nct)
  c = XOR(enc_c, r)

  (pk, sk) = KEM.DeriveKeyPair(seed)

  try:
    k = KEM.Decaps(sk, c)

    confirm_input = encode_sid(sid, U, S) || enc_c || transcript

    prk_k_h1 = KDF.Extract(SK, DST || "h1" || confirm_input)
    prk_k_h2 = KDF.Extract(SK, DST || "h2" || confirm_input || k)

    // Derive h1 from the full transcript excluding k
    client_confirm = KDF.Expand(prk_k_h1, DST || "client_confirm", Nkc)

    // Derive h2 || SK from the full transcript including k
    server_confirm = KDF.Expand(prk_k_h2, DST || "server_confirm", Nkc)
    client_key = KDF.Expand(prk_k_h2, DST || "key", Nkey)
```

```
    if client_confirm != client_confirm_target:
      raise AuthenticationError

    return client_key, server_confirm
  catch DecapsError:
    raise AuthenticationError
```

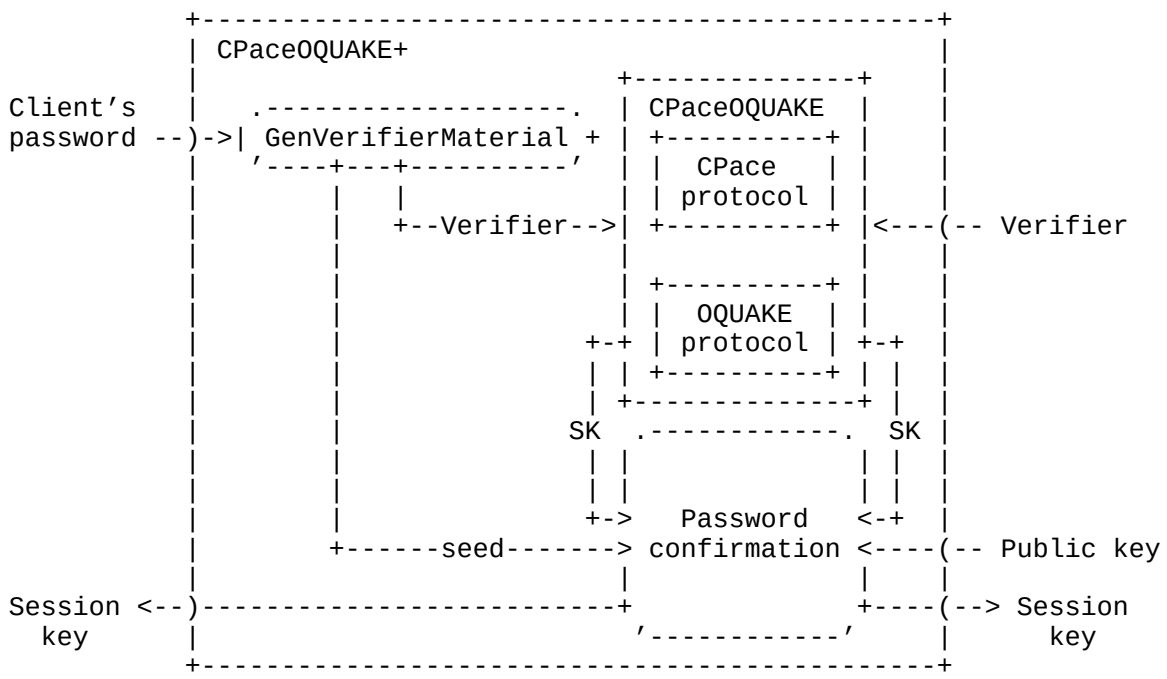
6.2.3. Server Verify

Upon receipt of the response, the server validates that the password confirmation value matches its own value. If the value does not match, the server aborts. Otherwise, the server outputs the new shared secret as its output.

```
PC-Verify

Input:
- context, opaque context produced by Challenge
- server_confirm_target, client’s response message, a byte string

Output:
- server_key, a 32-byte string

Exceptions:
- AuthenticationError, raised when the password confirmation values do not match

Parameters:

def PC-Verify(context, server_confirm_target):
  (server_confirm, server_key) = context
  if server_confirm != server_confirm_target:
    raise AuthenticationError
  return server_key
```

6.3. Composition of CPaceOQUAKE & Password Confirmation

The composition of CPaceOQUAKE and the password confirmation stage is strictly sequential. First, the parties run CPaceOQUAKE using the verifier. The client recovers this verifier using the GenVerifierMaterial function. After that, the parties proceed with password confirmation, which is initiated by the server using the stored public key. The client uses the seed that was also produced by GenVerifierMaterial to prove knowledge of the password. This seed MUST remain secret to prevent impersonation. An overview of the composition is below.



Upon successful completion of the entire protocol, the client and server will share a symmetric key that was authenticated by knowledge of the password. The protocol aborts if the password did not match. The protocol flows are shown below. Note here that if the client does not know the salt, the server must send it to the client before the protocol starts, which it can do in plain text.

7. CPaceOQUAKE+ Configurations

CPaceOQUAKE+ is instantiated by selecting a configuration of a group and hash function for the CPace protocol, a KEM, KDF, KSF, for password confirmation, and a KEM and KDF for CPaceOQUAKE, and a general purpose cryptographic hash function H. The KEM, KDF, are not required to be the same, so they are distinguished by "PC-" and "PAKE-" prefixes, e.g., PC-KDF and PAKE-KDF are the KDFs for the password confirmation stage and the CPaceOQUAKE protocol, respectively.

The RECOMMENDED configuration is below.

* CPace-Group: CPACE-RISTR255-SHA512 Section 4 of [CPACE]

* CPace-Hash: SHA-512

* KEM: X-Wing [XWING], where Nseed = 32, Nct = 1120, and Npk = 1216.

* PC-KDF: HKDF-SHA-256

* PC-KSF: Argon2id(S = zeroes(16), p = 4, T = Nh, m = 2^21, t = 1, v = 0x13, K = nil, X = nil, y = 2) [ARGON2]

* BUKEM: ML-BUKEM768 Section 4.2, where Nseed = 64, Nct = 1514, and Npk = 1172.

* PAKE-KDF: HKDF-SHA-256

* H: SHA256

* DST: "1b3abc3cd05e8054e8399bc38dfcbc1321d2e1b02da335ed1e8031ef5199f672" (a randomly generated 32-byte string)

The RECOMMENDED parameters are (see Appendix A):

* Nverifier = 32

* Nkc = 64

* Nsec = 32

* Nkey = 32, this is achieved by choosing H in CPace with H.b_in_bytes = 32

Other documents can define configurations as needed for their use case, subject to the following requirements:

1. KEM MUST be a hybrid KEM, i.e., one that achieves both classical and post-quantum security.

2. The parameters must be chosen so they correspond with this KEM. E.g., Nseed must have the correct length.

For instance, one possible additional configuration is as follows.

* CPace-Group: CPACE-P256_XMD:SHA-256_SSWU_NU_-SHA256 Section 4 of [CPACE]

* CPace-Hash: SHA-256

* KEM: X-Wing [XWING], where Nseed = 32, Nct = 1120, and Npk = 1216.

* PC-KDF: HKDF-SHA-256

* PC-KSF: Scrypt(N = 32768, r = 8, p = 1) [SCRYPT]

* BUKEM: ML-BUKEM768 Section 4.2, where Nseed = 64, Nct = 1514, and Npk = 1172.

* PAKE-KDF: HKDF-SHA-256

* H: SHA256

* DST: "b840fa4d4b4caec9e25d13d8c016cfe93e7468d54e936490bd0b0a3ffca1a01b" (a randomly generated 32-byte string)

8. Implementation Considerations

Some functions included in this specification are fallible (as noted by their ability to raise exceptions). The explicit errors generated throughout this specification, along with conditions that lead to each error, are as follows:

* AuthenticationError: The PC protocol fails password confirmation checks at the client or server; Section 6.2

Beyond these explicit errors, CPaceOQUAKE+ implementations can produce implicit errors. For example, if protocol messages sent between client and server do not match their expected size, an implementation should produce an error.

The errors in this document are meant as a guide for implementors. They are not an exhaustive list of all the errors an implementation might emit. For example, an implementation might run out of memory.

9. Security Considerations

This section discusses security considerations for the protocols specified in this document.

9.1. Identities

Client and server identities are essential to authenticated key exchange protocols, and PAKEs are no exception. This section discusses the role and importance of identities in the PAKE protocols specified in this document.

9.1.1. Symmetric PAKE identities

PAKEs are often analyzed in the universal composability (UC) framework, which imposes several requirements on the protocols: (1) the existence of a globally-unique session identifer associated with each protocol invocation, and (2) unique party identifiers. Both are considered as inputs to PAKEs, along with the password itself. In practice, however, computing or agreeing on session and party identifiers is non-trivial and cumbersome. For example, agreeing on a globally unique session identifier requires a protocol to run before the PAKE. Moreover, assigning identifiers to parties -- especially in symmetric PAKE settings -- is problematic as there are rarely pragmatic choices to be made for each party’s identifier. IP addresses are not always unique, PKI or some other registry mechanism for assigning names may not exist, and so on.

Intuitively, in symmetric settings, passwords are the only secret input to the PAKE protocol; party identities are assumed to be public. As such, an adversary is assumed to know these identifiers. Fortunately, there exists a UC model in which symmetric PAKEs such as CPace are proven secure without requiring party or session identifiers -- the bare PAKE model [BARE-PAKE]. The UC bare PAKE model, and proof of security for CPace in this model, demonstrate that PAKEs are universally composable without relying on unique party or session identifiers. We believe that the current proof of security of OQUAKE in [ABJ25] can be extended to show that NoIC, the basis of OQUAKE, realizes the Bare PAKE model as well, although we note that that this proof has not been published yet.

As such, for the PAKEs in Section 5, both the party and session identifier are optional. Applications are free to choose values for these identifiers if applicable, but they are not required for security.

[[OPEN ISSUE: adjust the requirements for the identities in OQUAKE on the basis on the bare PAKE analysis]]

9.1.2. Asymmetric PAKE identities

In contrast to the symmetric PAKE setting, party identities in the asymmetric PAKE setting play a different role. The very nature of the asymmetric PAKE is that one server, with many different registered passwords, can authenticate many different clients. Consequently, when the protocol runs, the server needs some way to determine which password registration to use in the protocol. Beyond ensuring that the server is authenticating the correct client, the client's identity is what helps the server make this selection.

However, the server identifier carries a similar burden. Indeed, the server identifier is used to distinguish distinct server instances from each other so, for example, a client cannot mistakenly authenticate with server A when communicating with server B. This is especially important if the client re-uses their identifier across server instances, since a password registration for server A would then be valid for server B if the server identity were not incorporated into the protocol.

Based on this, client and server identities are RECOMMENDED for the asymmetric PAKEs specified in this document (in Section 6). Both client and server identities can be long-lived, e.g., a client identity could be an email address and a server identity could be a domain name.

Practically, applications should be mindful of what happens when these identities change. Since they are both included in the password verifier (see Section 6.1.1), changing either identifier will require the veirifer to be re-computed and the client to be re-registered. For a single client, this change is minimal, but for a single server, which can have many registered clients, this change can be expensive. Applications therefore ought to consider the longevitiy and uniqueness of their party identifiers when instantiating these protocols.

10. IANA Considerations

This document has no IANA actions.

11. References

11.1. Normative References

[ARGON2] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <https://www.rfc-editor.org/rfc/rfc9106>.

[CPACE] Abdalla, M., Haase, B., and J. Hesse, "CPace, a balanced composable PAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-cpace-13, 14 October 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-cpace-13>.

[FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

[FIPS203] National Institute of Standards and Technology (NIST), "Module-Lattice-Based Key-Encapsulation Mechanism Standard", August 2024, <https://csrc.nist.gov/pubs/fips/203/final>.

[KEMELEON] G端nther, F., Stebila, D., and S. Veitch, "Kemeleon Encodings", Work in Progress, Internet-Draft, draft-veitch-kemeleon-00, 29 November 2024, <https://datatracker.ietf.org/doc/html/draft-veitch-kemeleon-00>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[SCRYPT] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/rfc/rfc7914>.

[XWING] Connolly, D., Schwabe, P., and B. Westerbaan, "X-Wing: general-purpose hybrid post-quantum KEM", Work in Progress, Internet-Draft, draft-connolly-cfrg-xwing-kem-06, 21 October 2024, <https://datatracker.ietf.org/doc/html/draft-connolly-cfrg-xwing-kem-06>.

11.2. Informative References

[ABJ25] Arriaga, A., Barbosa, M., and S. Jarecki, "NoIC: PAKE from KEM without Ideal Ciphers", n.d., <https://eprint.iacr.org/2025/231>.

[BARE-PAKE] Barbosa, M., Gellert, K., Hesse, J., and S. Jarecki, "Bare PAKE: Universally Composable Key Exchange from Just Passwords", Springer Nature Switzerland, Lecture Notes in Computer Science pp. 183-217, DOI 10.1007/978-3-031-68379-4_6, ISBN ["9783031683787", "9783031683794"], 2024, <https://doi.org/10.1007/978-3-031-68379-4_6>.

[Gu24] Gu, Y., "New Paradigms For Efficient Password Authentication Protocols", n.d., <https://www.escholarship.org/uc/item/7qm0220s>.

[HR24] Hesse, J. and M. Rosenberg, "PAKE Combiners and Efficient Post-Quantum Instantiations", n.d., <https://eprint.iacr.org/2024/1621>.

[LL24] Lyu, Y. and S. Liu, "Hybrid Password Authentication Key Exchange in the UC Framework", n.d., <https://eprint.iacr.org/2024/1630>.

[LLH24] Lyu, Y., Liu, S., and S. Han, "Efficient Asymmetric PAKE Compiler from KEM and AE", n.d., <https://eprint.iacr.org/2024/1400>.

[OPAQUE] Bourdrez, D., Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Augmented PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-18, 21 November 2024, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-18>.

[SPAKE2PLUS]
Taubert, T. and C. A. Wood, "SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol", RFC 9383, DOI 10.17487/RFC9383, September 2023, <https://www.rfc-editor.org/rfc/rfc9383>.

Appendix A. Deriving parameters

This section discusses how to generate parameters, given an upper bound on an adversary's advantage in breaking the hybrid (a)PAKE. The parameters in this standard correspond to a classical hardness of 117 bits (considering the attacker can break CPace) and a quantum hardness of 100 bits. We assume that an adversary can perform at most 2^qq queries to random oracles or (a)PAKE sessions. We use qq = 64. The derivation below uses some approximations, ignoring small constants in the exponent such as 1 and 1.6. We also only study dominant terms in the advantage equations.

A.1. Parameters for CPaceOQUAKE+

We have the following requirements:

* Nseed * 8 + Nverifier * 8 >= 2 * qq + classical hardness
* Nverifier * 8 >= qq + classical hardness
* Nkc * 8 >= qq + classical hardness
* KEM failure <= -qq - classical hardness
* KEM ind vs classical <= -qq - classical hardness
* KEM ind vs quantum <= -qq - quantum hardness
* CPaceOQUAKE vs classical <= classical hardness
* CPaceOQUAKE vs quantum <= quantum hardness

For ML-KEM we have Nseed = 32. For consistency, the spec uses Nverifier = 32. ML-KEM768's failure probability is 2^-165.2 and ML-KEM1024's failure probability is 2^-175.2. Both are slightly too large, but we deem them acceptable: the chance that an adversary encounters a failure is purely statistical and very small.

The following subsection discusses the parameters and hardness of CPaceOQUAKE.

A.2. Parameters for CPaceOQUAKE

For the security of CPaceOQUAKE+, we require that CPaceOQUAKE provides:

* CPaceOQUAKE vs classical <= classical hardness

* CPaceOQUAKE vs quantum <= quantum hardness

We have the following requirements when CPaceOQUAKE relies on CPace's security:

* CPace vs classical <= classical hardness

* Nkey * 8 >= qq + classical hardness

* KEM failure <= -qq - classical hardness

We have the following requirements when CPaceOQUAKE relies on OQUAKE's security:

* OQUAKE vs classical <= classical hardness

* OQUAKE vs quantum <= quantum hardness

* Nkey * 8 >= 2*qq + classical hardness

So, the smallest Nkey = 32. We ignore the KEM failure following the same reasoning as above.

The following subsections discuss the parameters and hardness of CPace and OQUAKE.

A.3. Parameters for CPace

We refer to the CPace [CPACE]. This standard requires Nkey, the number of bytes in CPace's session key, to be 32, so one must set H.bmax_in_bytes = 32.

A.4. Parameters for OQUAKE

We have the following requirements:

* BUKEM ind vs classical <= -qq - classical hardness

* BUKEM ind vs quantum <= -qq - quantum hardness

* BUKEM public key uniformity vs classical <= -qq - classical hardness

* BUKEM public key uniformity vs quantum <= -qq - quantum hardness

* BUKEM ciphertext uniformity vs classical <= -qq - classical hardness

* BUKEM ciphertext uniformity vs quantum <= -qq - quantum hardness

* BUKEM key * 8 >= qq + classical hardness

* KEM failure <= -qq - classical hardness

For, ML-BUKEM it is as hard or harder to break public key and ciphertext uniformity as it is to break indistinguishability, so we discuss all three properties at once.

For ML-BUKEM768, the resistance to classical attacks is approximately 181 - qq bits of security. So for qq = 64, classical hardness is approximately 117 bits of security. The resistance to quantum attacks is approximately 164 - qq bits of security. So for qq = 64, quantum hardness is approximately 100 bits of security.

For ML-BUKEM1024, this would come out to 253 - 64 = 189 bits of security for classical attacks and 230 - 64 = 166 bits of security for quantum attacks.

The ML-BUKEM key is 32 bytes, so this satisfies the requirements. We ignore the KEM failure following the same reasoning as above.

Authors' Addresses

Jelle Vos
Email: jvos@apple.com

Stanislaw Jarecki
University of California, Irvine
Email: sjarecki@ics.uci.edu

Christopher A. Wood
Apple, Inc.
Email: caw@heapingbits.net