Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 8 November 2026

D. Connolly
SandboxAQ
R. Barnes
Cisco
P. Grubbs
University of Michigan

7 May 2026

Hybrid PQ/T Key Encapsulation Mechanisms
draft-irtf-cfrg-hybrid-kems-11

Abstract

This document defines generic constructions for hybrid Key Encapsulation Mechanisms (KEMs) based on combining a post-quantum (PQ) KEM with a traditional cryptographic component. Hybrid KEMs built using these constructions provide strong security properties as long as either of the underlying algorithms are secure.

Discussion Venues

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

Discussion of this document takes place on the Crypto Forum Research Group mailing list (cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cfrg.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-hybrid-kems.

Status of This Memo

Table of Contents

1. Introduction

Post-quantum (PQ) cryptographic algorithms are based on problems that are conjectured to be resistant to attacks possible on a quantum computer. Key Encapsulation Mechanisms (KEMs) are a standardized class of cryptographic scheme that can be used to build protocols in lieu of traditional, quantum-vulnerable variants such as finite field or elliptic curve Diffie-Hellman (DH) based protocols.

Given the novelty of PQ algorithms, however, there is some concern that PQ algorithms currently believed to be secure will be broken. Hybrid constructions that combine both PQ and traditional algorithms can help moderate this risk while still providing security against quantum attack. If constructed properly, a hybrid KEM will retain certain security properties even if one of the two constituent KEMs is compromised. If the PQ KEM is broken, then the hybrid KEM should continue to provide security against non-quantum attackers by virtue of its traditional KEM component. If the traditional KEM is broken by a quantum computer, then the hybrid KEM should continue to resist quantum attack by virtue of its PQ KEM component.

In addition to guarding against algorithm weaknesses, this property also guards against flaws in implementations, such as timing attacks. Hybrid KEMs can also facilitate faster deployment of PQ security by allowing applications to incorporate PQ algorithms while still meeting compliance requirements based on traditional algorithms.

In this document, we define generic frameworks for constructing hybrid KEMs from a PQ KEM and a traditional algorithm. The aim of this document is provide a small set of techniques to achieve specific security properties given conforming component algorithms, which should make these techniques suitable for a broad variety of use cases.

We define four generic frameworks as variants of a common overall scheme. The variations are based on (1) what type of cryptographic object is being used for the traditional component, and (2) whether

the PQ KEM is assumed to have an additional property known as Ciphertext Second Preimage Resistance (C2PRI). Hybrid KEMs built using PQ KEMs that satisfy C2PRI can achieve the same security level with more efficient computations, trading off performance for an additional security assumption.

The remainder of this document is structured as follows: first, in Section 4 and Section 5, we define the abstractions on which the frameworks are built, and then the frameworks themselves. Then, in Section 6, we lay out the security analyses that support these frameworks, including the security requirements for constituent components and the security notions satisfied by hybrid KEMS constructed according to the frameworks in the document Section 6.2.1. Finally, we discuss some "path not taken", related topics that might be of interest to readers, but which are not treated in depth.

2. Requirements Notation

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.

3. Notation

This document is consistent with all terminology defined in [I-D.ietf-pquip-pqt-hybrid-terminology].

The following terms are used throughout this document:

* random(n): return a pseudorandom byte string of length n bytes produced by a cryptographically-secure random number generator.

* concat(x0, ..., xN): Concatenation of byte strings. concat(0x01, 0x0203, 0x040506) = 0x010203040506.

* split(N1, N2, x): Split a byte string x of length N1 + N2 into its first N1 bytes and its last N2 bytes. This function is the inverse of concat(x1, x2) when x1 is N1 bytes long and x2 is N2 bytes long. It is an error to call this function with a byte string that does not have length N1 + N2. Since this function operates over secret data x, it MUST be constant-time for a given N1 and N2.

When x is a byte string, we use the notation x[..i] and x[i..] to denote the slice of bytes in x starting from the beginning of x and leading up to index i, including the i-th byte, and the slice the bytes in x starting from index i to the end of x, respectively. For example, if x = [0, 1, 2, 3, 4], then x[..2] = [0, 1] and x[2..] = [2, 3, 4].

A set is denoted by listing values in braces: {a,b,c}.

A vector of set elements of length n is denoted with exponentiation, such as for the n-bit value: {0,1}^n.

Drawing uniformly at random from an n-bit vector into a value x is denoted: x $← {0,1}^n.

A function f that maps from one domain to another is denoted using a right arrow to separate inputs from outputs: f : inputs → outputs.

4. Cryptographic Dependencies

The generic hybrid PQ/T KEM frameworks we define depend on the following cryptographic primitives:

* Key Encapsulation Mechanisms (Section 4.1)

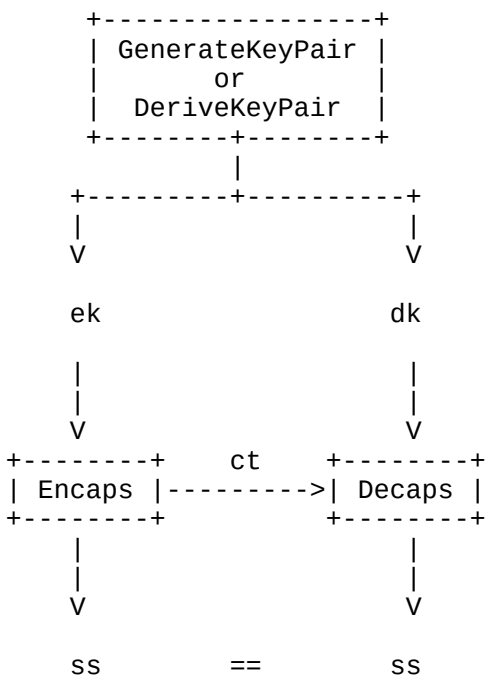
* Nominal Groups (Section 4.2)

* Pseudorandom Generators (Section 4.3)

* Key Derivation Functions (Section 4.4)

In the remainder of this section, we describe functional aspects of these mechanisms. The security properties we require in order for the resulting hybrid KEM to be secure are discussed in Section 6.

4.1. Key Encapsulation Mechanisms



A Key Encapsulation Mechanism (KEM) comprises the following algorithms:

* GenerateKeyPair() -> (dk, ek): A randomized algorithm that generates a secret decapsulation key dk and a public encapsulation key ek, each of which are byte strings.

* DeriveKeyPair(seed) -> (dk, ek): A deterministic algorithm that takes as input a seed seed and generates a secret decapsulation key dk and a public encapsulation key ek, each of which are byte strings.

* Encaps(ek) -> (ss, ct): A probabilistic encapsulation algorithm, which takes as input a public encapsulation key ek and outputs a shared secret ss and ciphertext ct.

* Decaps(dk, ct) -> ss: A deterministic decapsulation algorithm, which takes as input a secret decapsulation key dk and ciphertext ct and outputs a shared secret ss.

In this document, Decaps is modeled as always returning a shared secret and never returning an error. Component KEMs that use implicit rejection (such as ML-KEM) produce a deterministic pseudorandom output on invalid ciphertexts, which propagates through the combiner's KDF.

We also make use of internal algorithms such as:

* expandDecapsulationKey(dk) -> (dk, ek): A deterministic algorithm that takes as input a decapsulation key dk and generates keypair intermediate values for computation.

We assume that the values produced and consumed by the above functions are all byte strings, with fixed lengths:

* Nseed: The length in bytes of a key seed

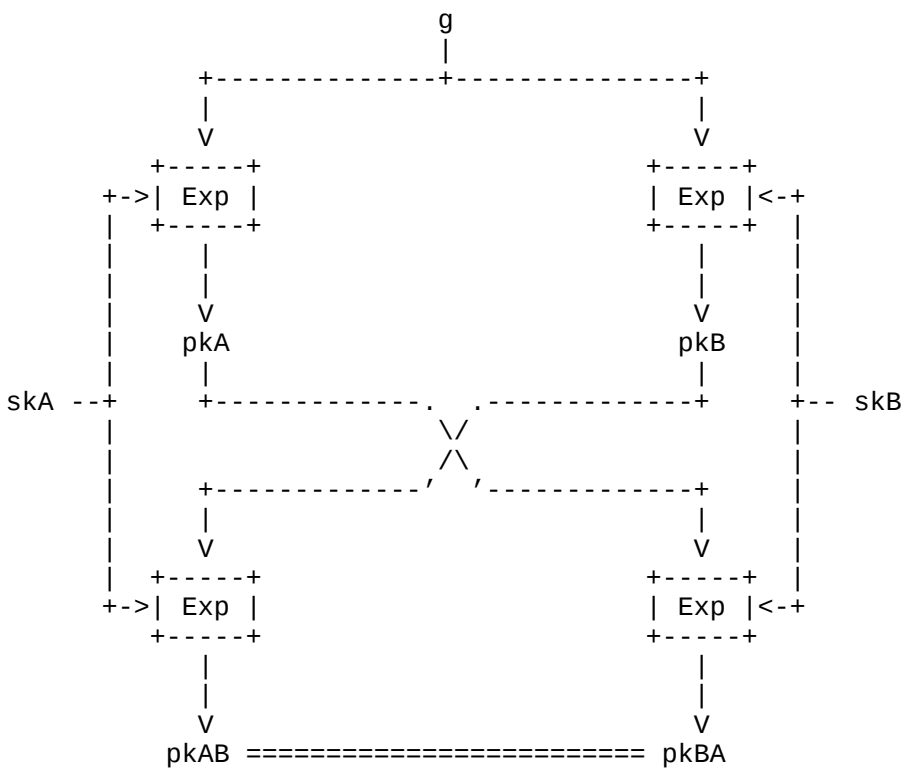
* Nek: The length in bytes of a public encapsulation key

* Ndk: The length in bytes of a secret decapsulation key

* Nct: The length in bytes of a ciphertext produced by Encaps

* Nss: The length in bytes of a shared secret produced by Encaps or Decaps

4.2. Nominal Groups



Nominal groups are an abstract model of elliptic curve groups, over which we instantiate Diffie-Hellman key agreement [ABH_21]. A nominal group comprises a set G together with a distinguished basis element g, an "exponentiation" map, and some auxiliary functions:

* Exp(p, x) -> q: An algorithm that produces an element q of G from an element p and an integer x.

  - The integers x are called "scalars" to distinguish them from group elements.

  - Exp must respect multiplication in its scalar argument x, so that Exp(Exp(p, x), y) = Exp(p, x * y).

* RandomScalar(seed) -> k: Produce a uniform pseudo-random scalar from the uniformly pseudo-random byte string seed.

* ElementToSharedSecret(P) -> ss: Extract a shared secret from an element of the group (e.g., by taking the X coordinate of an elliptic curve point).

We assume that scalars and group elements are represented by byte strings with fixed lengths:

* Nseed: The length in bytes of a seed (input to RandomScalar)

* Nscalar: The length in bytes of a scalar

* Nelem: The length in bytes of a serialized group element

* Nss: The length in bytes of a shared secret produced by ElementToSharedSecret

Groups used with the hybrid KEM framework in this document should be secure with respect to the strong Diffie-Hellman problem (see Section 6.1.3).

4.3. Pseudorandom Generators

A pseudorandom generator (PRG) is a deterministic function whose outputs are longer than its inputs. When the input is chosen uniformly at random, this induces a certain distribution over the possible output. The output distribution is pseudorandom if it is indistinguishable from the uniform distribution.

The PRGs used in this document have a simpler form, with fixed output lengths:

* Nout: The length in bytes of an output from this PRG.

* PRG(seed) -> output: Produce a byte string of length Nout from an input byte string seed.

The fixed sizes are for both security and simplicity.

PRGs used with the frameworks in this document MUST provide the bit-security required to source input randomness for PQ/T components from a seed that is expanded to a output length, of which a subset is passed to the component key generation algorithms.

The security requirements for PRGs used with the frameworks in this document are laid out in Section 6.1.6.

4.4. Key Derivation Functions

A Key Derivation Function (KDF) is a function that produces keying material based on an input secret and other information.

While KDFs in the literature can typically consume and produce byte strings of arbitrary length, the KDFs used in this document have a simpler form, with fixed output lengths:

* Nout: The length in bytes of an output from this KDF.

* KDF(input) -> output: Produce a byte string of length Nout from an input byte string.

The fixed sizes are for both security and simplicity.

Any KDF that utilizes HKDF [HKDF] MUST fully specify HKDF's salt, IKM, info, and L arguments.

The security requirements for KDFs used with the frameworks in this document are laid out in Section 6.1.5.

5. Hybrid KEM Frameworks

In this section, we define four frameworks for building hybrid KEMs. These frameworks are based on a common set of subroutines for things like key generation and computing a final shared secret.

The four frameworks vary along two axes:

1. Whether traditional component is a nominal group or a KEM

2. Whether to rely on the C2PRI property for the post-quantum component

The choice of which framework to use when building a hybrid KEM will depend on the application's needs along these two axes.

Name	PQ C2PRI?	T component
UG	No	Nominal group
UK	No	KEM
CG	Yes	Nominal group
CK	Yes	KEM

Table 1: Hybrid KEM frameworks

Instantiating one of these frameworks creates a hybrid KEM KEM_H based on the following constituent components:

* A traditional component that is either a nominal group or a KEM:
  - Group_T: A nominal group
  - KEM_T: A traditional KEM
* KEM_PQ: A post-quantum KEM
* PRG: A PRG producing byte strings of length KEM_PQ.Nseed + Comp_T.Nseed (PRG.Nout == KEM_PQ.Nseed + Comp_T.Nseed)
* KDF: A KDF producing byte strings of length KEM_H.Nss (KDF.Nout == KEM_H.Nss)
* Label - A byte string used to label the specific combination of the above components being used, as well as which framework is being instantiated. This value should be registered in the Hybrid KEM Labels IANA registry to avoid conflict with other instantiations (see Section 7).

KEM_PQ, Group_T, PRG, and KDF MUST meet the interfaces described in Section 4 and MUST meet the security requirements described in Section 6.2.1.

The constants for public values are derived from the concatenation of encapsulation keys and ciphertexts:

KEM_H.Nek = KEM_PQ.Nek + (KEM_T.Nek or Group_T.Nelem)
KEM_H.Nct = KEM_PQ.Nct + (KEM_T.Nct or Group_T.Nelem)

The Nseed and Nss constants should reflect the overall security level of the combined KEM, with the following recommended values:

KEM_H.Nseed = max(KEM_PQ.Nseed, (KEM_T.Nseed or Group_T.Nseed))
KEM_H.Nss = min(KEM_PQ.Nss, (KEM_T.Nss or Group_T.Nss))

Since we use the seed as the decapsulation key, Ndk = Nseed. For legacy cases where it is not possible to derive per-component decapsulation keys from a common seed, see Section 5.2.

5.1. Subroutines

The four hybrid KEM frameworks share a substantial amount of structure, which we capture in a set of subroutines.

5.1.1. Using a Nominal Group

Hybrid KEM frameworks that use a nominal group for the traditional component invoke the DeriveKeyPair, Encaps, and Decaps functions of PQ KEMs, alongside analogous functions of the nominal group. The "encapsulation key" is the receiver’s public key group element; the "ciphertext" is an ephemeral group element; and the shared secret is the secret value resulting from an ephemeral-static Diffie-Hellman exchange.

```
def expandDecapsKeyG(seed):
    seed_full = PRG(seed)
    (seed_PQ, seed_T) = split(KEM_PQ.Nseed, Group_T.Nseed, seed_full)

    (dk_PQ, ek_PQ) = KEM_PQ.DeriveKeyPair(seed_PQ)
    dk_T = Group_T.RandomScalar(seed_T)
    ek_T = Group_T.Exp(Group_T.g, dk_T)

    return (ek_PQ, ek_T, dk_PQ, dk_T)

def prepareEncapsG(ek_PQ, ek_T):
    (ss_PQ, ct_PQ) = KEM_PQ.Encaps(ek_PQ)
    sk_E = Group_T.RandomScalar(random(Group_T.Nseed))
    ct_T = Group_T.Exp(Group_T.g, sk_E)
    ss_T = Group_T.ElementToSharedSecret(Group_T.Exp(ek_T, sk_E))
    return (ss_PQ, ss_T, ct_PQ, ct_T)

def prepareDecapsG(ct_PQ, ct_T, dk_PQ, dk_T):
    ss_PQ = KEM_PQ.Decaps(dk_PQ, ct_PQ)
    ss_T = Group_T.ElementToSharedSecret(Group_T.Exp(ct_T, dk_T))
    return (ss_PQ, ss_T)
```

5.1.2. Using a Traditional KEM

Hybrid KEM frameworks that use a KEM for the traditional component invoke the DeriveKeyPair, Encaps, and Decaps functions of the traditional and PQ KEMs in parallel.

```
   def expandDecapsKeyK(seed):
       seed_full = PRG(seed)
       (seed_PQ, seed_T) = split(KEM_PQ.Nseed, KEM_T.Nseed, seed_full)
       (dk_PQ, ek_PQ) = KEM_PQ.DeriveKeyPair(seed_PQ)
       (dk_T, ek_T) = KEM_T.DeriveKeyPair(seed_T)
       return (ek_PQ, ek_T, dk_PQ, dk_T)

   def prepareEncapsK(ek_PQ, ek_T):
       (ss_PQ, ct_PQ) = KEM_PQ.Encaps(ek_PQ)
       (ss_T, ct_T) = KEM_T.Encaps(ek_T)
       return (ss_PQ, ss_T, ct_PQ, ct_T)

   def prepareDecapsK(ct_PQ, ct_T, dk_PQ, dk_T):
       ss_PQ = KEM_PQ.Decaps(dk_PQ, ct_PQ)
       ss_T = KEM_T.Decaps(dk_T, ct_T)
       return (ss_PQ, ss_T)
```

5.1.3. Combiners

A combiner function uses the KDF used in the hybrid KEM to combine the shared secrets output by the component algorithms with contextual information.

The two combiner functions defined in this document are as follows:

```
   def UniversalCombiner(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, label):
       return KDF(concat(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, label))

   def C2PRICombiner(ss_PQ, ss_T, ct_T, ek_T, label):
       return KDF(concat(ss_PQ, ss_T, ct_T, ek_T, label))
```

Note that while the names of the inputs are suggestive of the shared secret, ciphertext, and encapsulation key outputs of a KEM, the inputs to this function in the hybrid KEM framework are not necessarily the output of a secure KEM. In particular, when the framework is instantiated with a nominal group, the "ciphertext" component is an ephemeral group element, and the "encapsulation key" is the group element that functions as the recipient’s public key.

The choice of combiner brings with it certain assumptions under which the resulting hybrid KEM is secure.

The UniversalCombiner combiner explicitly computes over shared secrets, ciphertexts, and encapsulation keys from both components. This allows the resulting hybrid KEM to be secure as long as either component is secure, with no further assumptions on the components.

The C2PRICombiner combiner does not compute over the ciphertext or encapsulation key from the PQ component. The resulting hybrid KEM will be secure if the PQ component is IND-CCA secure, or, the traditional component is secure and the PQ component also satisfies the C2PRI property.

5.2. Key Generation

All four frameworks share a common key generation function, and a function to compute the encapsulation key that corresponds to a decapsulation key:

```
def GenerateKeyPair():
    seed = random(Nseed)
    return DeriveKeyPair(seed)

def DecapsToEncaps(dk):
    # The dk is always in seed format
    (_, ek) = DeriveKeyPair(dk)
    return ek
```

In some deployment environments, it is not possible to instantiate this process. Some implementations of component schemes do not support the DeriveKeyPair function, only GenerateKeyPair. Likewise in the nominal group case, a (scalar, group element) pair will only be generated when the scalar is generated internal to the implementation.

An implementation of a hybrid KEM in such environments MAY deviate from the above description in the following ways:

* DeriveKeyPair is not implemented.

* The decapsulation key returned by GenerateKeyPair and consumed by Decaps is a tuple (dk_PQ, dk_T) of per-constituent decapsulation keys (or pointers/handles to keys).

* The expandDecapsKeyG and expandDecapsKeyK functions are replaced by the following, where decapsToEncaps() is a function that returns the encapsulation key associated with a decapsulation key:

```
def expandDecapsKey(dk):
    (dk_PQ, dk_T) = dk # depending on the private key storage format
    ek_PQ = decapsToEncaps(dk_PQ)
    ek_T = decapsToEncaps(dk_T)
    return (ek_PQ, ek_T, dk_PQ, dk_T)
```

These deviations have both interoperability and security impacts.

From an interoperability point of view, the use of a second format for the hybrid KEM decapsulation key (other than the shared seed) introduces the risk of incompatibilities in cases where a private key needs to be moved from one system to another.

Separate key generation / handling also reduces binding properties from MAL-BIND-P-Q to LEAK-BIND-P-Q. As discussed below, binding properties can address a variety of attack scenarios, including LEAK scenarios in which an attacker has passive access to the decapsulation key and MAL scenarios in which an attacker can cause the victim to use a crafted decapsulation key. The above hybrid KEM framework assures binding properties in the face of a LEAK attacker, irrespective of how key generation is done. The additional protection provided by the default "shared seed" key generation upgrades this to protection against a MAL attacker.

Allowing for separate private key generation and handling also introduces a risk of inappropriate key reuse and cross-protocol attacks. A given key pair MUST never be used in both a hybrid KEM and with a non-hybrid algorithm. A pair of key pairs generated for a hybrid algorithm MUST only be used with that hybrid algorithm, not separately with their component algorithms. Likewise, key pairs generated outside of the context of a hybrid KEM MUST NOT be used with a hybrid KEM. The "shared seed" style of key generation prevents such reuse, because the per-component private keys are derived internally to the hybrid KEM.

As a result, this alternative style of key generation should only be used in environments where implementations of component algorithms do not allow decapsulation keys to be imported or exported. In scenarios where separate key generation is used and decapsulation keys can be imported/exported, additional measures should be put in place to mitigate the key reuse risks noted above.

5.3. UG Framework: Universal Combiner with a Nominal Group

This framework combines a PQ KEM with a nominal group, using the universal combiner function. It should be used in cases where the application wants to use a nominal group for the traditional component, and does not want to rely on the C2PRI assumption for the PQ KEM.

```
def DeriveKeyPair(seed):
    (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyG(seed)
    return (seed, concat(ek_PQ, ek_T))

def Encaps(ek):
    (ek_PQ, ek_T) = split(KEM_PQ.Nek, Group_T.Nelem, ek)
    (ss_PQ, ss_T, ct_PQ, ct_T) = prepareEncapsG(ek_PQ, ek_T)
    ss_H = UniversalCombiner(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, Label)
    ct_H = concat(ct_PQ, ct_T)
    return (ss_H, ct_H)

def Decaps(dk, ct):
    (ct_PQ, ct_T) = split(KEM_PQ.Nct, Group_T.Nelem, ct)
    (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyG(dk)
    (ss_PQ, ss_T) = prepareDecapsG(ct_PQ, ct_T, dk_PQ, dk_T)
    ss_H = UniversalCombiner(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, Label)
    return ss_H
```

5.4. UK Framework: Universal Combiner with a KEM

This framework combines a PQ KEM with a traditional KEM, using the universal combiner function. It should be used in cases where the application wants to use a KEM for the traditional component, and does not want to rely on the C2PRI assumption for the PQ KEM.

```
def DeriveKeyPair(seed):
    (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyK(seed)
    return (seed, concat(ek_PQ, ek_T))

def Encaps(ek):
    (ek_PQ, ek_T) = split(KEM_PQ.Nek, KEM_T.Nek, ek)
    (ss_PQ, ss_T, ct_PQ, ct_T) = prepareEncapsK(ek_PQ, ek_T)
    ss_H = UniversalCombiner(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, Label)
    ct_H = concat(ct_PQ, ct_T)
    return (ss_H, ct_H)

def Decaps(dk, ct):
    (ct_PQ, ct_T) = split(KEM_PQ.Nct, KEM_T.Nct, ct)
    (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyK(dk)
    (ss_PQ, ss_T) = prepareDecapsK(ct_PQ, ct_T, dk_PQ, dk_T)
    ss_H = UniversalCombiner(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, Label)
    return ss_H
```

5.5. CG Framework: C2PRI Combiner with a Nominal Group

This framework combines a PQ KEM with a nominal group, using the C2PRI combiner function. It should be used in cases where the application wants to use a nominal group for the traditional component, and is comfortable relying on the C2PRI assumption for the PQ KEM.

```
def DeriveKeyPair(seed):
    (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyG(seed)
    return (seed, concat(ek_PQ, ek_T))

def Encaps(ek):
    (ek_PQ, ek_T) = split(KEM_PQ.Nek, Group_T.Nelem, ek)
    (ss_PQ, ss_T, ct_PQ, ct_T) = prepareEncapsG(ek_PQ, ek_T)
    ss_H = C2PRICombiner(ss_PQ, ss_T, ct_T, ek_T, Label)
    ct_H = concat(ct_PQ, ct_T)
    return (ss_H, ct_H)

def Decaps(dk, ct):
    (ct_PQ, ct_T) = split(KEM_PQ.Nct, Group_T.Nelem, ct)
    (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyG(dk)
    (ss_PQ, ss_T) = prepareDecapsG(ct_PQ, ct_T, dk_PQ, dk_T)
    ss_H = C2PRICombiner(ss_PQ, ss_T, ct_T, ek_T, Label)
    return ss_H
```

5.6. CK Framework: C2PRI Combiner with a KEM

This framework combines a PQ KEM with a traditional KEM, using the C2PRI combiner function. It should be used in cases where the application wants to use a KEM for the traditional component, and is comfortable relying on the C2PRI assumption for the PQ KEM.

```
   def DeriveKeyPair(seed):
       (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyK(seed)
       return (seed, concat(ek_PQ, ek_T))

   def Encaps(ek):
       (ek_PQ, ek_T) = split(KEM_PQ.Nek, KEM_T.Nek, ek)
       (ss_PQ, ss_T, ct_PQ, ct_T) = prepareEncapsK(ek_PQ, ek_T)
       ss_H = C2PRICombiner(ss_PQ, ss_T, ct_T, ek_T, Label)
       ct_H = concat(ct_PQ, ct_T)
       return (ss_H, ct_H)

   def Decaps(dk, ct):
       (ct_PQ, ct_T) = split(KEM_PQ.Nct, KEM_T.Nct, ct)
       (ek_PQ, ek_T, dk_PQ, dk_T) = expandDecapsKeyK(dk)
       (ss_PQ, ss_T) = prepareDecapsK(ct_PQ, ct_T, dk_PQ, dk_T)
       ss_H = C2PRICombiner(ss_PQ, ss_T, ct_T, ek_T, Label)
       return ss_H
```

6. Security Considerations

Hybrid KEMs provide security by combining two or more schemes so that security is preserved if all but one scheme is broken. Informally, these hybrid KEMs are secure if the KDF is secure, and either the traditional component is secure, or the post-quantum KEM is secure: this is the ’hybrid’ property.

In this section, we review the important security properties for hybrid KEMs, and discuss how these security properties are provided by hybrid KEMs constructed according to the framework in this document.

6.1. Security Properties for Component Algorithms

In order to precisely define our security objectives for a hybrid KEM, we need to describe some properties that we will require from the component algorithms.

6.1.1. Indistinguishability under Chosen Ciphertext Attack (IND-CCA)

The first goal we have for our hybrid KEM constructions is indistinguishability under adaptive chosen ciphertext attack, or IND-CCA [BHK09]. This is most common security goal for KEMs and public-key encryption.

For KEMs, IND-CCA requires that no efficient adversary, given a ciphertext obtained by running Encaps() with an honestly-generated public key, can distinguish whether it is given the "real" secret output from Encaps(), or a random string unrelated to the Encaps()

call that created that ciphertext. (Readers should note that this definition is slightly different than the corresponding definitions for public-key encryption [BHK09].)

Whether a given KEM provides IND-CCA depends on whether the attacker is assumed to have access to quantum computing capabilities or not (assuming the scheme is without bugs and the implementation is correct). Post-quantum KEMs are intended to provide IND-CCA security against such an attacker. Traditional KEMs are not.

IND-CCA is the standard security notion for KEMs; most PQ KEMs were explicitly designed to achieve this type of security against both a quantum attacker and a traditional one.

For traditional algorithms, things are less clear. The DHKEM construction in [RFC9180] is an IND-CCA KEM based on Diffie-Hellman [ABH_21], but "raw" ephemeral-static Diffie-Hellman, interpreting the ephemeral public key as the ciphertext, is not IND-CCA secure. RSA-KEM is IND-CCA secure [ISO18033-2], and RSA-OAEP public-key encryption can be used to construct an IND-CCA KEM, but "classical" RSA encryption (RSAES-PKCS1-v1_5 as defined in [RFC8017]) is not even IND-CCA secure as a public-key encryption algorithm.

6.1.2. Ciphertext Second-Preimage Resistance (C2PRI)

Ciphertext Second-Preimage Resistance (C2PRI) is the property that given an honestly generated ciphertext, it is difficult for an attacker to generate a different ciphertext that decapsulates to the same shared secret. In other words, if an honest party computes (ss, ct) = Encaps(ek), then it is infeasible for an attacker to find another ciphertext ct’ such that Decaps(dk, ct’) == ss (where dk is the decapsulation key corresponding to the encapsulation key ek).

A related notion in the literature is chosen-ciphertext resistance (CCR) [CDM23]. C2PRI targets preimage-resistance, whereas CCR targets collision-resistance, much like the analogous properties for hash functions. In the language of the binding properties discussed in Section 6.1.4, CCR is equivalent to the property LEAK-BIND-K,PK-CT.

C2PRI is a weaker property than CCR / LEAK-BIND-K,PK-CT because it requires the attacker to match a specific, honestly generated ciphertext, as opposed to finding an arbitrary pair.

Several PQ KEMs have been shown to have C2PRI. ML-KEM [FIPS203] was shown to have this property in [XWING], and [CHH_25] proves C2PRI for several other algorithms, including FrodoKEM, HQC, Classic McEliece, and sntrup.

6.1.3. Strong Diffie-Hellman Problem (SDH)

The standard Diffie-Hellman problem is whether an attacker can compute g^xy given access to g^x and g^y and an oracle DH(Y, Z) that answers whether Y^x = Z. (This is the notion specified in [XWING], not the notion of the same name used in the context of bilinear pairings [Cheon06].)

When we say that the strong Diffie-Hellman problem is hard in a group, we always mean this in the context of classical attackers, without access to quantum computers. An attacker with access to a quantum computer that can execute Shor's algorithm for a group can efficiently solve the discrete log problem in that group, which implies the ability to solve the strong Diffie-Hellman problem.

As shown in [ABH_21], this problem is hard in prime-order groups such as the NIST elliptic curve groups P-256, P-384, and P-521, as well as in the Montgomery curves Curve25519 and Curve448.

6.1.4. Binding Properties

It is often useful for a KEM to have certain "binding" properties, by which certain parameters determine certain others. Recent work [CDM23] gave a useful framework of definitions for these binding properties. Binding for KEMs is related to other properties for KEMs and public-key encryption, such as robustness [GMP22] [ABN10], and collision-freeness [MOHASSEL10].

The framework given by [CDM23] refers to these properties with labels of the form X-BIND-P-Q. The first element X is the model for how the attacker can access the decapsulation key: HON for the case where the attacker never accesses the decapsulation key, LEAK for the case where the attacker has access to the honestly-generated decapsulation key, or MAL for the case where the attacker can choose or manipulate the keys used by the victim. P,Q means that given the value P, it is hard to produce another Q that causes Decaps to succeed. For example, LEAK-BIND-K-PK means that for a given shared secret (K), there is a unique encapsulation key (PK) that could have produced it, even if all of the secrets involved are given to the adversary after the encapsulation operation is completed (LEAK).

There is quite a bit of diversity in the binding properties provided by KEMs. Table 5 of [CDM23] shows the binding properties of a few KEMs. For example: DHKEM provides MAL-level binding for several properties. ML-KEM provides only LEAK-level binding [SCHMIEG2024]. Classic McEliece provides MAL-BIND-K-CT, but no assurance at all of X-BIND-K-PK.

6.1.5. Indifferentiability from a Random Oracle

The KDF used with a hybrid KEM MUST be indifferentiable from a random oracle (RO) [MRH03], even to a quantum attacker [BDFL_10] [ZHANDRY19]. This is a conservative choice given a review of the existing security analyses for our hybrid KEM constructions: most IND-CCA analyses for the four frameworks require only that the KDF is some kind of pseudorandom function, but the SDH-based IND-CCA analysis of CG in [XWING], and the corresponding analysis for UG [CG26] relies on the KDF being a RO. Proofs of our target binding properties for our hybrid KEMs require the KDF is a collision-resistant function.

If the KDF is a RO, the key derivation step in the hybrid KEMs can be viewed as applying a (RO-based) pseudorandom function - keyed with the shared secrets output by the constituent KEMs - to the other inputs. Thus, analyses which require the KDF to be a PRF, such as the one given in [GHP18] for UK, or the standard-model analysis of CG in [XWING], apply.

Sponge-based constructions such as SHA-3 [FIPS202] have been shown to be indifferentiable against classical [BDP_08] as well as quantum adversaries [ACM_25].

HKDF has been shown to be indifferentiable from a random oracle under specific constraints [LBB20]:

* that HMAC is indifferentiable from a random oracle, which for HMAC-SHA-256 has been shown in [DRS_13] when the compression function underlying SHA-256 is a random oracle, which is a regular assumption in the literature.

* the values of HKDF’s IKM input do not collide with values of info || 0x01. This MUST be enforced by the concrete instantiations that use HKDF as its KDF.

Using HKDF as a KDF in the sense defined in this document requires mapping the single input defined here to the IKM, salt, and info inputs required by HKDF. Concrete instantiations MUST define this mapping in such a way that no input value will ever map to colliding IKM and info values.

The choice of the KDF security level SHOULD be made based on the security level provided by the constituent KEMs. The KDF SHOULD at least have the security level of the strongest constituent KEM.

6.1.6. Security Requirements for PRGs

The functions used to expand a key seed to multiple key seeds is closer to a pseudorandom generator (PRG) in its security requirements [AOB_24]. A secure PRG is an algorithm PRG : {0, 1}^n → {0, 1}^m, such that no polynomial-time adversary can distinguish between PRG(r) (for r $← {0, 1}^n) and a random z $← {0, 1}^m [Rosulek]. The uniform string r ∈ {0, 1}^n is called the seed of the PRG.

A PRG is not to be confused with a random (or pseudorandom) _number_ generator (RNG): a PRG requires the seed randomness to be chosen uniformly and extend it; an RNG takes sources of noisy data and transforms them into uniform outputs.

PRGs are related to extendable output functions (XOFs) which can be built from random oracles. Examples include SHAKE256.

6.2. Security Goals for Hybrid KEMs

The security notions for hybrid KEMs are largely the same as for other algorithms, but they are contingent on the security properties of the component algorithms. In this section we discuss the intended security properties for hybrid KEMs and the requirements that the component algorithms must meet in order for those properties to hold.

6.2.1. IND-CCA Security

The idea of a hybrid KEM is that it should maintain its security if only one of the two component KEMs is secure. For a PQ/T hybrid KEM, this means that the hybrid KEM should be secure against a quantum attacker if the T component is broken, and secure against at least a classical attacker if the PQ component is broken.

More precisely, the hybrid KEM should meet two different notions of IND-CCA security, under different assumptions about the component algorithms:

* IND-CCA against a classical attacker all of the following are true:
  - KDF is indifferentiable from a random oracle
  - If using Group_T: The strong Diffie-Hellman problem is hard in Group_T
  - If using KEM_T: KEM_T is IND-CCA against a classical attacker
  - If using C2PRICombiner: KEM_PQ is C2PRI

* IND-CCA against a quantum attacker if all of the following are true:

  - KDF is indifferentiable from a random oracle

  - KEM_PQ is IND-CCA against a quantum attacker

Some IND-CCA analyses do not strictly require the KDF to be indifferentiable from a random oracle; they instead only require a kind of PRF assumption on the KDF. For simplicity we ignore this here; the security analyses described below for our constructions will elaborate on this point when appropriate.

6.2.2. Binding Properties

The most salient binding properties for a hybrid KEM to be used in Internet protocols are LEAK-BIND-K-PK and LEAK-BIND-K-CT.

The LEAK attack model is most appropriate for Internet protocols. There have been attacks in the LEAK model [BJKS24] [FG24], so a hybrid KEM needs to be resilient at least to LEAK attacks (i.e., HON is too weak). Internet applications generally assume that private keys are honestly generated, so MAL is too strong an attack model to address.

The LEAK-BIND-K-PK and LEAK-BIND-K-CT properties are naturally aligned with the needs of protocol design. Protocols based on traditional algorithms frequently need to incorporate transcript hashing in order to protect against key confusion attacks [FG24] or KEM re-encapsulation attacks [BJKS24]. The LEAK-BIND-K-PK and LEAK-BIND-K-CT properties prevent these attacks at the level of the hybrid KEM. Protocol designers may still need or want to include the ciphertext or encapsulation key into their protocol or key schedule for other reasons, but that can be independent of the specific properties of the KEM and its resulting shared secret.

Implementors should not interpret the paragraph above as absolving them of their responsibility to carefully think through whether MAL-BIND attacks apply in their settings.

6.3. Security Non-goals for Hybrid KEMs

Security properties not targeted by these designs are listed in Section 8.

6.4. Security Analysis

In this section, we describe how the hybrid KEM framework in this document provides the security properties described above.

6.4.1. IND-CCA analyses

The UG construction has two complementary IND-CCA analyses: one for when the SDH problem holds but the PQ KEM is broken, and one for the reverse. Both are technically novel but are substantially similar to the existing peer-reviewed analyses of the CG [XWING] and UK [GHP18] constructions. [CG26] by the editorial team describes the analysis of UG in detail.

The first IND-CCA analysis, based on SDH, is similar to the corresponding analysis of CG given in [XWING]: it gives a straightforward reduction to the SDH hardness in the underlying group. Notably, since the PQ KEM key and ciphertext are hashed, the C2PRI security of the PQ KEM does not appear in the bound.

The second IND-CCA analysis is a straightforward reduction to the IND-CCA security of the PQ KEM, and the PRF security of the RO when keyed with the PQ KEM’s shared secret.

This document’s UK construction does not have a dedicated IND-CCA analysis; the [GHP18] paper on which the construction is based gives a slightly different version, namely they do not include the public encapsulation keys in the KDF. However, we argue that the proof goes through with trivial modifications if the public encapsulation keys are included in the KDF. The relevant step is claim 3 of Theorem 1, which reduces to the split-key pseudorandomness of the KDF. ([GHP18] call the KDF a "core" function, and denote it as W.) We observe that adding the public encapsulation keys to the inputs only changes the concrete contents of the reduction’s queries to its oracle. Since the reduction chooses the public encapsulation keys itself, they can be added to the oracle inputs, and the remainder of the proof goes through unmodified.

We reiterate that modulo some low-level technical details, our requirement that the KDF is indifferentiable from an RO implies that, in the ROM, the KDF used in [GHP18] meets the split-key pseudorandomness property used in [GHP18]’s analysis: this is shown in [GHP18], Lemma 6, where a pseudorandom skPRF is constructed from any almost-uniform keymixing function in the random oracle model by H(g(k1,...,kn), x) , where H is modeled as a random oracle and g is -almost uniform. Example 3 from [GHP18] qualifies g(k_1,...,k_n) = k_1 || ... || k_n as -almost uniform with = 1/len(k_1 || ... || k_n).

Like UG, the CG construction has two complementary IND-CCA analyses. Both were given in [XWING]. We summarize them but elide some details.

One analysis (Theorem 1) [XWING] shows that if the KDF is modelled as a RO, IND-CCA holds if the PQ KEM is broken, as long as the SDH problem holds in the nominal group and the PQ KEM satisfies C2PRI. The other (Theorem 2) [XWING] shows that if the PQ-KEM is IND-CCA and the KDF is a PRF keyed on the PQ-KEM's shared secret, IND-CCA holds.

As long as the aforementioned security requirements of the component parts are met, these analyses imply that this document's CG construction satisfies IND-CCA security.

The CK construction has two complementary IND-CCA analyses: one for when the IND-CCA security of the traditional PKE-based KEM holds but the PQ KEM is broken, except for the PQ KEM's C2PRI security, and one for where the IND-CCA security of the PQ KEM holds. Both are technically novel but are substantially similar to the existing peer-reviewed analyses of the CG [XWING] and UK [GHP18] constructions. [COS_26] by the editorial team and collaborators describes the analysis of UG in detail.

Therefore all four hybrid KEMs in this document are IND-CCA when instantiated with cryptographic components that meet the security requirements described above. Any changes to the algorithms, including key generation/derivation, are not guaranteed to produce secure results.

The IND-CCA analyses of UG in [CG26], CG in [XWING], and CK in [COS_26] all model component key generation as sampling the two component key pairs independently, whereas the default key generation in this document derives both component key pairs from a single seed via the PRG (Section 5.2). This apparent mismatch is resolved by a standard hybrid argument: by the PRG security required in Section 6.1.6, the joint distribution of component key pairs derived from a single seed is computationally indistinguishable from the joint distribution of independently generated component key pairs. Thus, any IND-CCA adversary against the shared-seed scheme implies an IND-CCA adversary against the independent-keys scheme with at most an additive PRG-distinguishing term in the bound. The same observation underlies the binding sketches in Section Section 6.4.2.

6.4.2. Binding analyses

There are four hybrid KEM frameworks, and two target binding properties, so we need eight total analyses. The CG and CK binding analyses additionally require the corresponding LEAK-BIND property of KEM_PQ. None of these exact results were known; thus the following are results by the editorial team. We include informal justifications here and defer rigorous proofs to a forthcoming paper.

We note that these sketches implicitly ignore the fact that in our hybrid KEMs, both key pairs are derived from a common random seed; we instead implicitly think of them as two runs of DeriveKeyPair with independent random seeds. We justify this simplification by noting that in the LEAK model - in which the adversary is given the key pairs resulting from an honest run of KeyGen - the pseudorandomness of the seed expansion implies the adversary's input distributions in the two cases are computationally indistinguishable. The derivation of component scheme key pairs from the common random seed provides further protection against manipulation or corruption of keys such that it can contribute to stronger binding properties against a MAL adversary, as well as operational benefits in practice, but we do not prove that here.

The paper that establishes the IND-CCA security of the UG construction ([CG26]) does not also include a proof of its binding properties. Instead, [CG26] Section 4.3 observes that the binding arguments for UK transfer to UG essentially unmodified, since both constructions include the relevant ciphertexts and encapsulation keys in the KDF input; we make that argument concrete in the sketches below. The sketches rely on collision resistance of the KDF (Section 6.1.5), and for CG and CK, additionally on the corresponding LEAK-BIND property of the PQ KEM.

6.4.2.1. UG Binding

6.4.2.1.1. LEAK-BIND-K-CT of UG

Claim: If KDF is collision-resistant, then UG is LEAK-BIND-K-CT.

Justification: To win LEAK-BIND-K-CT, given knowledge of two honestly-generated UG secret keys, the adversary must construct two distinct UG ciphertexts that decapsulate to the same (non-bot) key. Since UG includes the ciphertexts in the key derivation, the condition that the ciphertexts are distinct directly implies that a LEAK-BIND-K-CT win gives a collision in the KDF.

6.4.2.1.2. LEAK-BIND-K-PK of UG

Claim: If KDF is collision-resistant, then UG is LEAK-BIND-K-PK.

Justification: As described above, in the LEAK-BIND-K-PK game, to win the adversary must construct two ciphertexts that decapsulate to the same non-bot key, for distinct UG public keys. Again, since UG includes the public keys in the KDF, the distinctness condition implies a LEAK-BIND-K-PK win must collide the KDF.

6.4.2.2. UK Binding

6.4.2.2.1. LEAK-BIND-K-CT of UK

Claim: If KDF is collision-resistant, then UK is LEAK-BIND-K-CT.

Justification: To win LEAK-BIND-K-CT, given knowledge of two honestly-generated UK secret keys, the adversary must construct two distinct UK ciphertexts that decapsulate to the same (non-bot) key. Since UK includes the ciphertexts in the key derivation, the condition that the ciphertexts are distinct directly implies that a LEAK-BIND-K-CT win gives a collision in the KDF.

6.4.2.2.2. LEAK-BIND-K-PK of UK

Claim: If KDF is collision-resistant, then UK is LEAK-BIND-K-PK.

Justification: As described above, in the LEAK-BIND-K-PK game, to win the adversary must construct two ciphertexts that decapsulate to the same non-bot key, for distinct UK public keys. Again, since UK includes the public keys in the KDF, the distinctness condition implies a LEAK-BIND-K-PK win must collide the KDF.

6.4.2.3. CG Binding

The LEAK-BIND proofs for CG are a bit more subtle than for UK; the main reason for this is CG’s omission of the PQ KEM key and ciphertext from the KDF. We will show that CG still has our target LEAK-BIND properties as long as the underlying PQ-KEM also has the corresponding LEAK-BIND property. We note that our preliminary results suggest that a different proof strategy, which instead directly uses properties of the nominal group, may work here; we present the PQ-KEM route for concreteness.

6.4.2.3.1. LEAK-BIND-K-CT of CG

Claim: If KDF is collision-resistant and the PQ KEM is LEAK-BIND-K-CT, then CG is LEAK-BIND-K-CT.

Justification: To win the adversary must construct two distinct CG ciphertexts that decapsulate to the same non-bot key. Call the CG ciphertexts output by the adversary (ct_PQ^0, ct_T^0) and (ct_PQ^1, ct_T^1). Distinctness implies (ct_PQ^0, ct_T^0) != (ct_PQ^1, ct_T^1). Since ct_T is included in the KDF, if ct_T^0 != ct_T^1, a win must collide the KDF.

Thus we can restrict attention to the case where ct_PQ^0 != ct_PQ^1 but ct_T^0 = ct_T^1. In this case, there are two relevant sub-cases: either ss_PQ^0 (:= KEM_PQ.Decaps(dk_PQ^0, ct_PQ^0)) is not equal to ss_PQ^1 (:= KEM_PQ.Decaps(dk_PQ^1, ct_PQ^1)), or they are equal. If they are not equal, the KDF inputs are again distinct, so a LEAK-BIND-K-CT win must collide the KDF.

If ss_PQ^0 = ss_PQ^1, we can show a reduction to the LEAK-BIND-K-CT security of the PQ KEM. The reduction is given two PQ KEM key pairs as input and must output two distinct PQ KEM ciphertexts that decapsulate to the same key. The reduction does this by generating two nominal-group key pairs and running the CG LEAK-BIND-K-CT adversary on all keys. Then the reduction outputs the PQ KEM ciphertexts output by the adversary. The probability that the adversary wins and ss_PQ^0 = ss_PQ^1 and ct_PQ^0 != ct_PQ^1 and ct_T^0 = ct_T^1 is a lower bound on the probability of the reduction winning the LEAK-BIND-K-CT game against the PQ KEM.

We conclude by noting these cases are exhaustive.

6.4.2.3.2. LEAK-BIND-K-PK of CG

Claim: If KDF is collision-resistant and the PQ KEM is LEAK-BIND-K-PK, then CG is LEAK-BIND-K-PK.

Justification: Similar to the above, we proceed by a case analysis on the win condition of the LEAK-BIND-K-PK game. The condition is (ek_PQ^0, ek_T^0) != (ek_PQ^1, ek_T^1) and ss_H^0 = ss_H^1. Again, as above we argue that the only nontrivial case is the one where ek_PQ^0 != ek_PQ^1 but ek_T^0 = ek_T^1: in the other case we can directly get a KDF collision from a winning output. In this case the result of KEM_PQ.Decaps for the two PQ KEM keys can either be the same or different. IF they are different, we again get a KDF collision from a win. If they are the same, in a similar way as above, we can build a reduction to the LEAK-BIND-K-PK of PQ KEM.

Again, we conclude by noting that these cases are exhaustive.

6.4.2.4. CK Binding

6.4.2.4.1. LEAK-BIND-K-CT of CK

Claim: If KDF is collision-resistant and the PQ KEM is LEAK-BIND-K-CT, then CK is LEAK-BIND-K-CT.

Justification: To win the adversary must construct two distinct CK ciphertexts that decapsulate to the same non-bot key. Call the CK ciphertexts output by the adversary (ct_PQ^0, ct_T^0) and (ct_PQ^1, ct_T^1). Distinctness implies (ct_PQ^0, ct_T^0) != (ct_PQ^1, ct_T^1). Since ct_T is included in the KDF, if ct_T^0 != ct_T^1, a win must collide the KDF.

Thus we can restrict attention to the case where ct_PQ^0 != ct_PQ^1 but ct_T^0 = ct_T^1. In this case, there are two relevant sub-cases: either ss_PQ^0 (:= KEM_PQ.Decaps(dk_PQ^0, ct_PQ^0)) is not equal to ss_PQ^1 (:= KEM_PQ.Decaps(dk_PQ^1, ct_PQ^1)), or they are equal. If they are not equal, the KDF inputs are again distinct, so a LEAK-BIND-K-CT win must collide the KDF.

If ss_PQ^0 = ss_PQ^1, we can show a reduction to the LEAK-BIND-K-CT security of the PQ KEM. The reduction is given two PQ KEM key pairs as input and must output two distinct PQ KEM ciphertexts that decapsulate to the same key. The reduction does this by generating two traditional KEM key pairs and running the CK LEAK-BIND-K-CT adversary on all keys. Then the reduction outputs the PQ KEM ciphertexts output by the adversary. The probability that the adversary wins and ss_PQ^0 = ss_PQ^1 and ct_PQ^0 != ct_PQ^1 and ct_T^0 = ct_T^1 is a lower bound on the probability of the reduction winning the LEAK-BIND-K-CT game against the PQ KEM.

We conclude by noting these cases are exhaustive.

6.4.2.4.2. LEAK-BIND-K-PK of CK

Claim: If KDF is collision-resistant and the PQ KEM is LEAK-BIND-K-PK, then CK is LEAK-BIND-K-PK.

Justification: Similar to the above, we proceed by a case analysis on the win condition of the LEAK-BIND-K-PK game. The condition is (ek_PQ^0, ek_T^0) != (ek_PQ^1, ek_T^1) and ss_H^0 = ss_H^1. Again, as above we argue that the only nontrivial case is the one where ek_PQ^0 != ek_PQ^1 but ek_T^0 = ek_T^1: in the other case we can directly get a KDF collision from a winning output. In this case the result of KEM_PQ.Decaps for the two PQ KEM keys can either be the same or different. IF they are different, we again get a KDF collision from a win. If they are the same, in a similar way as above, we can build a reduction to the LEAK-BIND-K-PK of PQ KEM.

Again, we conclude by noting that these cases are exhaustive.

6.5. Other Considerations

6.5.1. Domain Separation

ASCII-encoded bytes provide oracle cloning [BDG20] in the security game via domain separation. The IND-CCA security of hybrid KEMs often relies on the KDF function KDF to behave as an independent random oracle, which the inclusion of the label achieves via domain separation [GHP18].

By design, the calls to KDF in these frameworks and usage anywhere else in higher level protocol use separate input domains unless intentionally duplicating the ’label’ per concrete instance with fixed parameters. This justifies modeling them as independent functions even if instantiated by the same KDF. This domain separation is achieved by using prefix-free sets of label values. Recall that a set is prefix-free if no element is a prefix of another within the set.

Length differentiation is sometimes used to achieve domain separation but as a technique it is brittle and prone to misuse [BDG20] in practice so we favor the use of an explicit post-fix label.

6.5.2. Fixed-length

Variable-length secrets are generally dangerous. In particular, using key material of variable length and processing it using hash functions may result in a timing side channel. In broad terms, when the secret is longer, the hash function may need to process more blocks internally. In some unfortunate circumstances, this has led to timing attacks, e.g. the Lucky Thirteen [LUCKY13] and Raccoon [RACCOON] attacks.

Furthermore, [AVIRAM] identified a risk of using variable-length secrets when the hash function used in the key derivation function is no longer collision-resistant.

If concatenation were to be used with values that are not fixed-length, a length prefix or other unambiguous encoding would need to be used to ensure that the composition of the two values is injective and requires a mechanism different from that specified in this document.

Therefore, this specification MUST only be used with algorithms which have fixed-length shared secrets.

7. IANA Considerations

This document requests that IANA create a registry "Hybrid KEM Labels", which lists labels that uniquely identify instantiations of the frameworks in this document. The registry should be administered under the First Come First Served policy [RFC8126].

Template:

* Label: The name of the wire format
* Framework: The framework used in the hybrid KEM. This value MUST be one of the following values: "UG", "UK", "CG", or "CK".
* PQ component: The name of the post-quantum KEM used in the hybrid KEM.
* Traditional component: The name of the traditional KEM or nominal group used in the hybrid KEM.
* KDF: The name of the Key Derivation Function used in the hybrid KEM.
* PRG: The name of the Pseudo-Random Generator used in the hybrid KEM.
* Nseed: An integer representing the size of a seed for this hybrid KEM.
* Nss: An integer representing the size of a shared secret for this hybrid KEM.
* Reference (optional): The document where this hybrid KEM is defined

The registry should initially be empty.

8. Out of Scope

Security properties and design considerations that were considered and not included in these designs:

* Anonymity [GMP22], deniability, obfuscation, other forms of key-robustness or binding [GMP22], [CDM23]

* More than two components: this document restricts the scope to two components: one post-quantum component and one traditional component

* Parameterized output length: not analyzed as part of any security proofs in the literature, and a complication deemed unnecessary

9. References

9.1. Normative References

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[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>.

9.2. Informative References

[ABH_21] Jol Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel, "Analysing the HPKE standard.", April 2021.

[ABN10] "Robust Encryption", 2010, <https://eprint.iacr.org/2008/440.pdf>.

[ACM_25] "The Sponge is Quantum Indifferentiable", 2025, <https://eprint.iacr.org/2025/731.pdf>.

[AOB_24] "Formally verifying Kyber Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt", 2024, <https://eprint.iacr.org/2024/843.pdf>.

[AVIRAM] Nimrod Aviram, Benjamin Dowling, Ilan Komargodski, Kenny Paterson, Eyal Ronen, and Eylon Yogev, "[TLS] Combining Secrets in Hybrid Key Exchange in TLS 1.3", 1 September 2021, <https://mailarchive.ietf.org/arch/msg/tls/F4SVeL2xbGPaPB2GW_GkBbD_a5M/>.

[BDFL_10] "Random Oracles in a Quantum World", 2010, <https://eprint.iacr.org/2010/428.pdf>.

[BDG20] "Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability", 2020, <https://eprint.iacr.org/2020/241.pdf>.

[BDP_08] "On the Indifferentiability of the Sponge Construction", 2008, <https://www.iacr.org/archive/eurocrypt2008/49650180/49650180.pdf>.

[BHK09] Bellare, M., Hofheinz, D., and E. Kiltz, "Subtleties in the Definition of IND-CCA: When and How Should Challenge-Decryption be Disallowed?", 2009, <https://eprint.iacr.org/2009/418>.

[BJKS24] "Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging", 2024, <https://www.usenix.org/system/files/usenixsecurity24-bhargavan.pdf>.

[CDM23] Cremers, C., Dax, A., and N. Medinger, "Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols", 2023, <https://eprint.iacr.org/2023/1933.pdf>.

[CG26] "*** BROKEN REFERENCE ***".

[Cheon06] Cheon, J. H., "Security Analysis of the Strong Diffie-Hellman Problem", 2006.

[CHH_25] "Starfighters — on the general applicability of X-Wing", 2025, <https://eprint.iacr.org/2025/1397>.

[COS_26] "StarHunters— Secure Hybrid Post-Quantum KEMs From IND-CCA2 PKEs", 2026, <https://eprint.iacr.org/2026/427>.

[DRS_13] "To Hash or Not to Hash Again? (In)differentiability Results for H^2 and HMAC", 2013, <https://eprint.iacr.org/2013/382.pdf>.

[FG24] "Security Analysis of Signal's PQXDH Handshake", 2024, <https://link.springer.com/chapter/10.1007/978-3-031-91823-0_5>.

[FIPS202] "SHA-3 standard :: permutation-based hash and extendable-output functions", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.202, 2015, <https://doi.org/10.6028/nist.fips.202>.

[FIPS203] "Module-lattice-based key-encapsulation mechanism standard", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.203, August 2024, <https://doi.org/10.6028/nist.fips.203>.

[GHP18] Giacon, F., Heuer, F., and B. Poettering, "KEM Combiners", 2018, <https://eprint.iacr.org/2018/024.pdf>.

[GMP22] Grubbs, P., Maram, V., and K.G. Paterson, "Anonymous, Robust Post-Quantum Public-Key Encryption", 2022, <https://eprint.iacr.org/2021/708.pdf>.

[HKDF] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[I-D.ietf-pquip-pqt-hybrid-terminology] D, F., P, M., and B. Hale, "Terminology for Post-Quantum Traditional Hybrid Schemes", Work in Progress, Internet-Draft, draft-ietf-pquip-pqt-hybrid-terminology-06, 10 January 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-pquip-pqt-hybrid-terminology-06>.

[ISO18033-2] "Information technology -- Security techniques -- Encryption algorithms -- Part 2: Asymmetric ciphers", 2006, <https://www.iso.org/standard/37971.html>.

[LBB20] "A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol", 2019, <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8806752>.

[LUCKY13] Al Fardan, N. J. and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS record protocols", n.d., <https://ieeexplore.ieee.org/iel7/6547086/6547088/06547131.pdf>.

[MOHASSEL10] "A closer look at anonymity and robustness in encryption schemes.", 2010, <https://www.iacr.org/archive/asiacrypt2010/6477505/6477505.pdf>.

[MRH03] "Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology", 2003, <https://eprint.iacr.org/2003/161.pdf>.

[RACCOON] Merget, R., Brinkmann, M., Aviram, N., Somorovsky, J., Mittmann, J., and J. Schwenk, "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)", September 2020, <https://raccoon-attack.com/>.

[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>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

[Rosulek] "The Joy of Cryptography", 2021, <https://joyofcryptography.com/pdf/book.pdf>.

[SCHMIEG2024] Schmieg, S., "Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK", 2024, <https://eprint.iacr.org/2024/523.pdf>.

[XWING] "X-Wing: The Hybrid KEM You’ve Been Looking For", 2024, <https://eprint.iacr.org/2024/039.pdf>.

[ZHANDRY19] "How to Record Quantum Queries, and Applications to Quantum Indifferentiability", 2019, <https://doi.org/10.1007/978-3-030-26951-7_9>.

Appendix A. Deterministic Encapsulation

When verifying the behavior of a KEM implementation (e.g., by generating or verifying test vectors), it is useful for the implementation to expose a "derandomized" version of the Encaps algorithm:

* EncapsDerand(ek, randomness) -> (shared_secret, ct): A deterministic encapsulation algorithm, which takes as input a public encapsulation key ek and randomness randomness, and outputs a shared secret shared_secret and ciphertext ct.

An implementation that exposes EncapsDerand must also define a required amount of randomness:

* Nrandom: The length in bytes of the randomness provided to EncapsDerand

The corresponding change for a nominal group is to replace randomly-generated inputs to RandomScalar with deterministic ones. In other words, for a nominal group, Nrandom = Nseed.

When a hybrid KEM is instantiated with constituents that support derandomized encapsulation (either KEMs or groups), the hybrid KEM can also support EncapsDerand(), with Nrandom = PQ.Nrandom + T.Nrandom. The structure of the hybrid KEM’s EncapsDerand algorithm is the same as its Encaps method, with the following differences:

* The EncapsDerand algorithm also takes a randomness parameter, which is a byte string of length Nrandom.

* Invocations of Encaps or RandomScalar (with a random input) in the constituent algorithms are replaced with calls to EncapsDerand or RandomScalar with a deterministic input.

* The randomness used by the PQ constituent is the first PQ.Nrandom bytes of the input randomness.

* The randomness used by the traditional constituent is the final T.Nrandom bytes of the input randomness.

Acknowledgments

TODO acknowledge.

Authors’ Addresses

Deirdre Connolly
SandboxAQ
Email: durumcrustulum@gmail.com

Richard Barnes
Cisco
Email: rlb@ipv.sx

Paul Grubbs
University of Michigan
Email: paulgrubbs12@gmail.com