Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 22 January 2026

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

21 July 2025

Hybrid PQ/T Key Encapsulation Mechanisms
draft-irtf-cfrg-hybrid-kems-05

Abstract

This document defines generic constructions for hybrid Key Encapsulation Mechanisms (KEMs) based on combining a traditional cryptographic component and a post-quantum (PQ) KEM. 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 traditional algorithm and a PQ KEM. 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.

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 6.4. 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 the following cryptographic primitives:

* Key Encapsulation Mechanisms (Section 4.1)

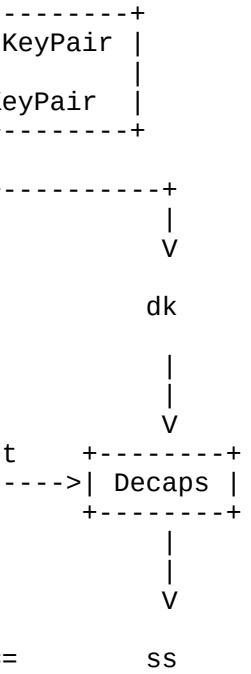
* Nominal Groups (Section 4.2)

* 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 (KEMs) comprises the following algorithms:

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

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

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

* 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.

We also make use of internal algorithms such as:

* expandDecapsulationKey(dk) -> (ek, dk): 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

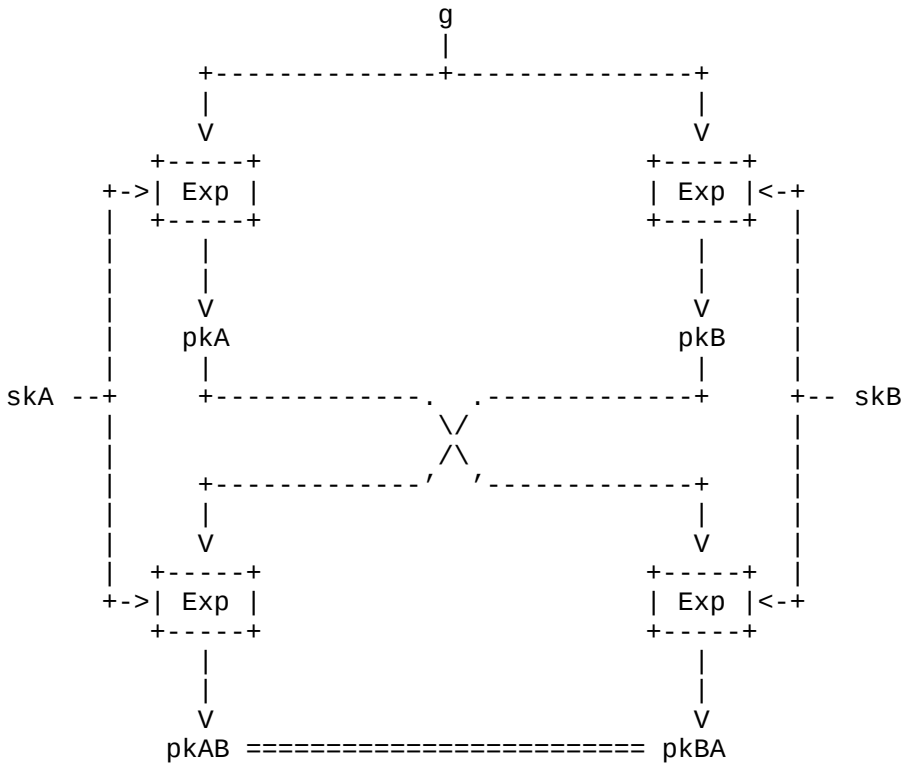
* Nek: The length in bytes of a public encapsulation key

* 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 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

The security requirements for groups used with the frameworks in this document are laid out in Section 6.4.2.

4.3. Pseudorandom Generators

A pseudorandom generator (PRG) is a deterministic function G whose outputs are longer than its inputs. When the input to G is chosen uniformly at random, it 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 a 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.

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 6.4.4.

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 a 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.

For instances of the Extract()/Expand() KDF paradigm such as HKDF, we fix the salt and sizes to fit this form.

The security requirements for KDFs used with the frameworks in this document are laid out in Section 6.4.3.

5. Hybrid KEM Frameworks

In this section, we define three generic frameworks for building for hybrid KEMs:

GHP: A generic framework that is suitable for use with any choice of traditional and PQ KEMs, with minimal security assumptions on the constituent KEMs

PRE: A performance optimization of GHP for the case where encapsulation keys are large and frequently reused

QSF: An optimized generic framework for the case where the traditional component is a nominal group and the PQ component has strong binding properties

These frameworks share a common overall structure, differing mainly in how they compute the final shared secret and the security requirements of their components.

5.1. GHP

The GHP hybrid KEM depends on the following constituent components:

* KEM_T: A traditional KEM

* KEM_PQ: A post-quantum KEM

* PRG: A PRG producing byte strings of length KEM_T.Nseed + KEM_PQ.Nseed (PRG.Nout == KEM_T.Nseed + KEM_PQ.Nseed)

* KDF: A KDF producing byte strings of length GHP.Nss (KDF.Nout == GHP.Nss)

* Label - A byte string used to label the specific combination of the above constituents being used.

The KEMs, groups, KDFs, and PRGs MUST meet the security requirements in Section 6.4.

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

```
Nek = KEM_T.Nek + KEM_PQ.Nek
Nct = KEM_T.Nct + KEM_PQ.Nct
```

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

```
Nseed = max(KEM_T.Nseed, KEM_PQ.Nseed)
Nss = min(KEM_T.Nss, KEM_PQ.Nss)
```

Since we use the seed as the decapsulation key, Ndk = Nseed.

Given these constituent parts, the GHP hybrid KEM is defined as follows:

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

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

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

   def Encaps(ek):
       (ek_T, ek_PQ) = split(KEM_T.Nek, KEM_PQ.Nek, ek)
       (ss_T, ct_T) = KEM_T.Encap(pk_T)
       (ss_PQ, ct_PQ) = KEM_PQ.Encap(pk_PQ)
       ss_H = KDF(concat(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, label))
       ct_H = concat(ct_T, ct_PQ)
       return (ss_H, ct_H)

   def Decaps(dk, ct):
       (ek_T, ek_PQ, dk_T, dk_PQ) = expandDecapsulationKey(dk)

       (ct_T, ct_PQ) = split(KEM_T.Nct, KEM_PQ.Nct, ct)
       ss_T = KEM_T.Decap(dk_T, ct_T)
       ss_PQ = KEM_PQ.Decap(dk_PQ, ct_PQ)

       ss_H = KDF(concat(ss_PQ, ss_T, ct_PQ, ct_T, ek_PQ, ek_T, label))
       return ss_H
```

5.2. PRE

The PRE hybrid KEM is a performance optimization of the GHP KEM, optimized for the case where encapsulation keys are large and frequently reused. In such cases, hashing the entire encapsulation key is expensive, and the same value is hashed repeatedly. The PRE KEM thus computes an intermediate hash of the hybrid encapsulation key, so that the hash value can be computed once and used across many encapsulation or decapsulation operations.

The PRE KEM is identical to the GHP KEM except for the shared secret computation. One additional KDF is required:

* KeyHash: A KDF producing byte strings of length GHP.Nss (KeyHash.Nout == GHP.Nss)

The GenerateKeyPair and DeriveKeyPair algorithms for PRE are identical to those of the GHP KEM. The Encaps and Decaps method use a modified shared secret computation:

```
def Encaps(ek):
    (ek_T, ek_PQ) = split(KEM_T.Nek, KEM_PQ.Nek, ek)
    (ss_T, ct_T) = KEM_T.Encap(pk_T)
    (ss_PQ, ct_PQ) = KEM_PQ.Encap(pk_PQ)

    ekh = KeyHash(concat(ek_T, ek_PQ))
    ss_H = KDF(concat(ss_PQ, ss_T, ct_PQ, ct_T, ekh, label))

    ct_H = concat(ct_T, ct_PQ)
    return (ss_H, ct_H)

def Decaps(dk, ct):
    (ek_T, ek_PQ, dk_T, dk_PQ) = expandDecapsulationKey(dk)

    (ct_T, ct_PQ) = split(KEM_T.Nct, KEM_PQ.Nct, ct)
    ss_T = KEM_T.Decap(dk_T, ct_T)
    ss_PQ = KEM_PQ.Decap(dk_PQ, ct_PQ)

    ekh = KeyHash(concat(ek_T, ek_PQ))
    ss_H = KDF(concat(ss_PQ, ss_T, ct_PQ, ct_T, ekh, label))
    return ss_H
```

5.3. QSF

The QSF hybrid KEM (QSF below) depends on the following constituent components:

* Group_T: A nominal group
* KEM_PQ: A post-quantum KEM
* PRG: A PRG producing byte strings of length Group_T.Nseed + KEM_PQ.Nseed (Expand.Nout == Group_T.Nseed + KEM_PQ.Nseed)
* KDF: A KDF producing byte strings of length QSF.Nss (KDF.Nout == KDF.Nss)
* Label - A byte string used to label the specific combination of the above constituents being used.

We presume that Group_T, KEM_PQ, and the KDFs meet the interfaces described in Section 4 and MUST meet the security requirements described in Section 6.4.

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

```
Nek = Group_T.Nelem + KEM_PQ.Nek
Nct = Group_T.Nelem + KEM_PQ.Nct
```

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

```
Nseed = max(Group_T.Nseed, KEM_PQ.Nseed)
Nss = min(Group_T.Nss, KEM_PQ.Nss)
```

Since we use the seed as the decapsulation key, Ndk = Nseed.

Given these constituent parts, we define the QSF hybrid KEM as follows:

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

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

       return (ek_T, ek_PQ, dk_T, dk_PQ)

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

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

   def Encaps(ek):
       (ek_T, ek_PQ) = split(Group_T.Nek, KEM_PQ.Nek, ek)

       sk_E = Group_T.RandomScalar(random(GroupT.Nseed))
       ct_T = Group_T.Exp(GroupT.g, sk_E)
       ss_T = Group_T.ElementToSharedSecret(Group_T.Exp(ek_T, sk_E))
       (ss_PQ, ct_PQ) = KEM_PQ.Encap(ek_PQ)

       ss_H = KDF(concat(ss_PQ, ss_T, ct_T, ek_T, Label))
       ct_H = concat(ct_T, ct_PQ)
       return (ss_H, ct_H)

   def Decaps(dk, ct):
       (ek_T, ek_PQ, dk_T, dk_PQ) = expandDecapsulationKey(dk)

       ss_T = Group_T.ElementToSharedSecret(Group_T.Exp(ct_T, dk_T))
       ss_PQ = KEM_PQ.Decap(dk_PQ, ct_PQ)

       ss_H = KDF(concat(ss_PQ, ss_T, ct_T, ek_T, Label))
       return ss_H
```

6. Security Considerations

Hybrid KEM constructions aim to provide security by combining two or more schemes so that security is preserved if all but one schemes are replaced by an arbitrarily bad scheme. 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. Next we describe this document’s specific security goals for hybrid KEMs.

6.1. Cryptographic Security Goals for Hybrid KEMs

6.1.1. IND-CCA Security

The first goal we have for our hybrid KEM constructions is indistinguishability under adaptive chosen ciphertext attack, or IND-CCA. 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 one for public-key encryption [RS92].)

6.2. 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 attack model---HON, LEAK, or MAL. 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).

We treat LEAK-BIND-K-PK and LEAK-BIND-K-CT to be target binding properties. The HON-BIND security model seems too weak for real applications---real attacks in the LEAK model are known [BJKS24] [FG24]. We are not aware of any common settings where the MAL-BIND security model is needed; thus, LEAK-BIND seems a sensible middle ground.

The LEAK-BIND-K-PK and LEAK-BIND-K-CT properties independently allow using a KEM shared secret such that the likelihood of finding a colliding value with the encapsulation key used in its computation or the ciphertext used in its computation is negligible. Such properties are attractive when integrating KEMs into protocols where once protocol designers would have used Diffie-Hellman, as they can

use the smaller shared secret value alone as an input to a protocol key schedule for example, without necessarily also needing to include the much larger ciphertext or the encapsulation key to be protected against key confusion attacks [FG24] or KEM re-encapsulation attacks [BJKS24]. 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 that were considered and not included in these designs:

Anonymity [GMP22], Deniability, Obfuscation, other forms of key-robustness or binding [GMP22], [CDM23]

6.4. Security Requirements for Constituent Components

6.4.1. Security Requirements for KEMs

Component KEMs MUST be IND-CCA-secure [GHP2018] [XWING].

For instances of QSF, the component KEM MUST also be ciphertext second preimage resistant (C2PRI) [XWING]: this allows the component KEM encapsulation key and ciphertext to be left out from the KDF input.

6.4.1.1. Ciphertext Second Preimage Resistant (C2PRI) Security

Roughly, C2PRI [XWING] says that an adversary given an honestly-generated key pair (sk, pk) and the result of an _honest_ Encaps(pk) - call it k, c - cannot find a _distinct_ c’ such that Decaps(sk, c’) = k. A related notion has also been described as chosen-ciphertext resistance (CCR) [CDM23]. C2PRI targets preimage-resistance, whereas CCR targets collision-resistance [CAMPBELL25].

6.4.2. Security Requirements for Groups

The groups MUST be modelable as nominal groups in which the strong Diffie-Hellman problem holds [ABH_21] [XWING].

Prime-order groups such as P-256, P-384, and P-521 and the Montgomery curves Curve25519 and Curve448 have been shown to be modelable as nominal groups in [ABH_21], as well as showing the X25519() and X448() functions respectively pertain to the nominal group exp(X, y) function, specifically clamping secret keys when they are generated, instead of clamping secret keys together with exponentiation.

6.4.3. Security Requirements for KDFs

The KDF 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. (In short, most IND-CCA analyses require only that the KDF is some kind of pseudorandom function, but the SDH-based IND-CCA analysis of QSF in [XWING] 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 GHP [GHP2018] or the standard-model analysis of QSF in [XWING], apply.

Sponge-based constructions such as SHA-3 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], assuming the compression function underlying SHA-256 is a random oracle, which it is indifferentiably when used prefix-free.

* 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.

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.

6.4.4. 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.

6.4.5. Security Properties of PRE

The PRE hybrid KEM framework uses a function KeyHash to generate a short digest of the encapsulation keys. This function MUST be collision-resistant.

6.5. Security Properties of Hybrid KEMs Frameworks

6.5.1. IND-CCA analyses

The QSF 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 QSF construction satisfies IND-CCA security.

This document’s exact GHP and PRE constructions do not have IND-CCA analyses; the GHP paper 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. (GHP 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 also argue that this extension applies, again with nearly trivial modifications, to prove security of PRE. Observe that the only difference between GHP and PRE is prehashing of the public encapsulation keys. As long as the hash function is collision-resistant, any event that happens in the IND-CCA game of GHP happens only with negligibly different probability in the IND-CCA game of PRE.

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 GHP and PRE meets the split-key pseudorandomness property used in GHP's analysis.

Therefore all three 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.

6.5.2. Binding analyses

There are three hybrid KEM frameworks, and two target binding properties, so we need six total analyses. None of these exact results were known; thus the following are new 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.

6.5.2.1. GHP Binding

6.5.2.1.1. LEAK-BIND-K-CT of GHP

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

Justification: To win LEAK-BIND-K-CT, given knowledge of two honestly-generated GHP secret keys, the adversary must construct two distinct GHP ciphertexts that decapsulate to the same (non-bot) key. Since GHP 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.

6.5.2.2. LEAK-BIND-K-PK of GHP

Claim: If KDF is collision-resistant, then GHP 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 GHP public keys. Again, since GHP includes the public keys in the KDF, the distinctness condition implies a LEAK-BIND-K-PK win must collide the KDF.

6.5.2.3. PRE Binding

6.5.2.3.1. LEAK-BIND-K-CT of PRE

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

Justification: PRE and GHP do not differ on how they incorporate the ciphertexts into key derivation, so the GHP proof above applies.

6.5.2.3.2. LEAK-BIND-K-PK of PRE

Claim: If KDF and KeyHash are collision-resistant, then PRE is LEAK-BIND-K-PK.

Justification: The only relevant difference between PRE and GHP is key prehashing. This does indeed change the proof, since we can no longer argue the distinctness condition on the public keys _directly_ gives a collision in KDF - the keys are hashed, and only their hash is input into the KDF. However, as long as KeyHash is collision-resistant, the distinctness condition implies the public key hashes are distinct. Thus, for the adversary to win it must either collide KeyHash or KDF.

6.5.2.4. QSF Binding

The LEAK-BIND proofs for QSF are a bit more subtle than for GHP and PRE; the main reason for this is QSF's omission of the PQ KEM key and ciphertext from the KDF. We will show that QSF 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 a different proof strategy, which instead directly uses properties of the nominal group, may work here; we present the PQ-KEM route for concreteness.

6.5.2.4.1. LEAK-BIND-K-CT of QSF

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

Justification: To win the adversary must construct two distinct QSF ciphertexts that decapsulate to the same non-bot key. Call the QSF ciphertexts output by the adversary (ct_T^0, ct_PQ^0) and (ct_T^1, ct_PQ^1). Distinctness implies (ct_T^0, ct_PQ^0) != (ct_T^1, ct_PQ^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.Decap(dk_PQ^0, ct_PQ^0)) is not equal to ss_PQ^1 (:= KEM_PQ.Decap(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 QSF 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.

6.5.2.4.2. LEAK-BIND-K-PK of QSF

Claim: If KDF is collision-resistant and the PQ KEM is LEAK-BIND-K-PK, then QSF 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_T^0, ek_PQ^0) != (ek_T^1, ek_PQ^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.Decap 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.

6.6. Other Considerations

6.6.1. Domain Separation

ASCII-encoded bytes provide oracle cloning [BDG2020] 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 [GHP2018].

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 [BDG2020] in practice so we favor the use of an explicit post-fix label.

6.6.2. 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. In Scope

7.1. GHP Framework

The GHP framework works for generic IND-CCA component schemes. GHP also has an IND-CCA proof from [GHP2018]. Including the public encapsulation keys as part of the KDF preimage fits in the 'additional data' parts of the split key PRF proof there, and binds to the encapsulation keys, which is a nice property for protocols integrating concrete instances. GHP also matches NIST SP 800-227 IPD, and gives good binding properties Section 6.2 is generally safe with no caveats on use for constructing concrete instances using a broad array of components.

7.2. QSF Framework

QSF works for most elliptic curve groups and C2PRI-secure quantum-resistant KEMs. It is an optimization that leaves out large ciphertexts and encapsulation keys from the KDF preimage, saving extra hashing, if the PQ KEM meets requirements. More KEMs can be proven to be C2PRI-secure eventually for use with QSF.

7.3. PRE Framework

PRE offers many of the same benefits as GHP, while allowing an optimization to pre-hash static encapsulation keys, which if large, can be a performance improvement.

8. Out of Scope

Considerations that were considered and not included in these designs:

Anonymity [GMP22], Deniability, Obfuscation, other forms of key-robustness or binding [GMP22], [CDM23]

8.1. More than Two Component KEMs

Design team decided to restrict the space to only two components, a traditional and a post-quantum KEM.

8.2. 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

[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>.

[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>.

[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>.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway, "The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES", January 2001.

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

[ANSIX9.62]
ANS, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANS X9.62-2005, November 2005.

[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>.

[BDG2020] "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>.

[BDP_11] "Cryptographic sponge functions", 2011, <https://keccak.team/files/CSF-0.1.pdf>.

[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>.

[CAMPBELL25]
"Re: Binding and QSF-style hybrid KEMs", 2025, <https://mailarchive.ietf.org/arch/msg/cfrg/qp_YxofDEl5fN6W7Xyab-juwaCc/>.

[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>.

[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>.

[FIPS186] "Digital Signature Standard (DSS)", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.186-5, February 2023, <https://doi.org/10.6028/nist.fips.186-5>.

[GHP2018] 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.driscoll-pqt-hybrid-terminology] D, F., "Terminology for Post-Quantum Traditional Hybrid Schemes", Work in Progress, Internet-Draft, draft-driscoll-pqt-hybrid-terminology-02, 7 March 2023, <https://datatracker.ietf.org/doc/html/draft-driscoll-pqt-hybrid-terminology-02>.

[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>.

[KSMW2024] Kraemer, J., Struck, P., and M. Weishaupl, "Binding Security of Implicitly-Rejecting KEMs and Application to BIKE and HQC", n.d., <https://eprint.iacr.org/2024/1233>.

[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/>.

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

[RS92] "Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack.", 1992, <https://link.springer.com/chapter/10.1007/3-540-46766-1_35>.

[RSS11] "Careful with Composition: Limitations of Indifferentiability and Universal Composability", 2011, <https://eprint.iacr.org/2011/339.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>.

[SEC1] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <https://secg.org/sec1-v2.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) -> (ct, shared_secret): A deterministic encapsulation algorithm, which takes as input a public encapsulation key ek and randomness randomness, and outputs a ciphertext ct and shared secret shared_secret.

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 = T.Nrandom + PQ.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 traditional constituent is the first T.Nrandom bytes of the input randomness.

* The randomness used by the PQ constituent is the final PQ.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