HPKE v2?

Deirdre Connolly, Karthik Bhargavan, Franziskus Kiefer

Problems in HPKEv1

1: Lack of binding in any KEM that's not DH-KEM

2: Does not define a signature-based AuthKEM

3: Does not provide a way of mixing multiple Encap/AuthEncap's or multiple PSKs.

Fix 1: Fix binding, backwards-compatible

```
KEM.Encap(pkR)
KEM.Decap(enc,skR)

def KEM.EncapWithBinding(pkR):
      ss,enc = KEM.Encap(pkR)
      pkRm = SerializePublicKey(pkR)
      kem_context = concat(enc, pkRm)
      shared_secret = ExtractAndExpand(ss, kem_context)
      return shared_secret, enc

def KEM.DecapWithBinding(enc, skR):
      ss = KEM.Decap(enc,skR)
      pkRm = SerializePublicKey(pk(skR))
      kem_context = concat(enc, pkRm)
      shared_secret = ExtractAndExpand(ss, kem_context)
      return shared_secret
```

- Any KEM can either meet the first API Encap/Decap, or it can define EncapWithBinding/DecapWithBinding
- Doing the binding in HPKE always is preferred: the guaranteed security is more important than any perf penalty due to additional hashing/serialization.
- This reformulation is fully backwards compatible with DH-KEM in HPKEv1 and the same thing can be done with AuthEncap/AuthDecap (this time also binding the sender's public key)

Fix 2: A generic signature-based AuthKEM mode

```
def SigKEM.AuthEncap(sig_kS, pkR):
      ss,enc = KEM.Encap(pkR)
      pkRm = SerializePublicKey(pkR)
      pkSm = SerializePublicKey(Sig.pk(sig_kS))
      sig_kem_context = concat(enc, pkRm, pkSm)
      sig = Sig.Sign(sig_kS, sig_kem_context)
      enc_sig = concat(enc, sig)
      shared_secret = ExtractAndExpand(ss, kem_context)
      return shared_secret, enc_sig

def SigKEM.AuthDecap(enc, skR, verif_kS):
      enc, sig = split(enc, enc.len() - Sig.sig_len)
      ss = KEM.Decap(enc, skR)
      pkRm = SerializePublicKey(pk(skR))
      pkSm = SerializePublicKey(pkS)
      sig_kem_context = concat(enc, pkRm, pkSm)
      if Sig.verify(verif_kS, sig_kem_context, sig) then
         shared_secret = ExtractAndExpand(ss, kem_context)
         return shared_secret
      else Error
```

- The authentication also authenticates the entire `kem_context`; this is the general pattern we need to compose multiple KEMs and multiple Signatures: we construct a `context` that contains the public keys and any other context used in both schemes and make sure that this context is bound into both cryptographic operations.
- To improve privacy, we could also modify this scheme to encrypt and send the sender's public key within the `enc_sig`. This would ameliorate the risk that the application would send the sender's public key in the clear outside HPKE and makes HPKE self-contained.
- To improve key compromise impersonation protection, we could define a special OneShot mode, where the hash of the payload is mixed into the `sig_kem_context` and hence included in the signature. This would prevent the "insider" weakness identified by Alwen et al.

Fix 3: Generalize key schedule to allow N *Encap()'s?

- Generalize the HPKE key schedule to allow the Encap/AuthEncap computations to be repeated any number of times?
- The key idea would be to ensure that we incrementally construct and bind a `context` that includes all the elements of the encaps we are using.
- The key advantage of defining this combination within HPKE would be that we could allow people to combine any number of PQ/non-PK KEMS, Signatures, and PSKs in a one-shot HPKE construction.
    - In practice, people will likely want at most 1 DH, 1 PQ-KEM, 1 Signature, 1 PQ-Signature, 1 PSK, but even this is quite a lot, so a generic composition construction may have value.

Next Steps?

- Should we propose all of these improvements in a single RFC, which would be a motivation for HPKEv2, or should they be piecemeal extensions?
- Each of these changes requires new security analyses, although because we are backwards compatible, the proofs for HPKEv1 continue to hold for the usual DH-KEM case.

Questions?

HPKE v2?

Deirdre Connolly, Karthik Bhargavan, Franziskus Kiefer

✨ Backup Slides ✨

Goal: a pure-PQ ciphersuite for HPKE

- HPKE is the modern standard for public-key encryption
- No purely post-quantum ciphersuites
  - X-Wing hybrid KEM has recently been mentioned
- The upcoming FIPS standard ML-KEM would seem to fit nicely
- Would provide FIPS compatibility with HKDF-SHA2, -AES-GCM
- An Auth- mode with PQ KEMs doesn’t seem well-understood, we’ll stick with the Base- mode

HPKE KEM Requirements[1]

9.2. Security Requirements on a KEM Used within HPKE

A KEM used within HPKE MUST allow HPKE to satisfy its desired security properties described in Section 9.1. Section 9.6 lists requirements concerning domain separation.

In particular, the KEM shared secret MUST be a uniformly random byte string of length `Nsecret`. This means, for instance, that it would not be sufficient if the KEM shared secret is only uniformly random as an element of some set prior to its encoding as a byte string.

9.2.1. Encap/Decap Interface

As mentioned in Section 9, [CS01] provides some indications that if the KEM's `Encap()`/`Decap()` interface (which is used in the Base and PSK modes) is IND-CCA2-secure, HPKE is able to satisfy its desired security properties. An appropriate definition of IND-CCA2 security for KEMs can be found in [CS01] and [BHK09].

[1] IND-CCA2 == KEM IND-CCA

FIPS 203 (draft[1]): ML-KEM Properties[2]

589 **The ML-KEM construction.** At a high level, the ML-KEM construction proceeds in two
590 steps. First, the idea mentioned above is used to construct a public-key encryption scheme
591 from the MLWE problem. Second, this public-key encryption scheme is converted into a
592 key-encapsulation mechanism using the so-called Fujisaki-Okamoto (FO) transform [11, 12].
593 In addition to producing a KEM, the FO transform is also intended to provide security in a
594 significantly more general adversarial attack model. As a result, ML-KEM is believed to satisfy
595 so-called IND-CCA security [1, 4, 13].

[1] IND-CCA security not expected to change
[2] IND-CCA: ciphertext indistinguishability under adaptive chosen ciphertext attacks for KEMs

Done, right?

Re-encapsulation Attacks

Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols

Version 1.0.5, March 5, 2024*

Cas Cremers, Alexander Dax, and Niklas Medinger

CISPA Helmholtz Center for Information Security
{cremers,alexander.dax,niklas.medinger}@cispa.de

https://eprint.iacr.org/2023/1933.pdf

Re-encapsulation attack in Signal PQXDH v1[1]

KEM Re-Encapsulation Attack

We show that when using an IND-CCA secure public key encryption scheme to build an IND-CCA secure KEM, an attacker can make two parties compute the same key, even though both used a distinct PQPK, as soon as only one of the PQPK was compromised. This attack is a new attack in the class of re-encapsulation attacks as introduced by Cremers, Dax, and Medinger.

Consider the following execution:

1. An attacker is able to compromise some PQPK of responder B, while another PQPK2 of the same responder is uncompromised.
2. The attacker makes initiator A use PQPK, and obtain a ciphertext CT, from which it can learn the shared secret SS, as PQPK was compromised.
3. Now, the attacker, not violating IND-CCA, comes up with a new ciphertext CT', valid for PQPK2, such that the decapsulation of CT' is also SS.
4. The attacker then forwards to B the message from A, but swaps CT by CT', and the key identifier of PQPK by PQPK2.
5. The responder B succeeds in computing the key using PQPK2.

The main issue here is that the compromise of a single PQPK in fact enables an attacker to compromise all future KEM shared secrets of the responder, and this even after the responder deleted the compromised PQPK.

[1] https://cryspen.com/post/pqxdh/

Re-encapsulation attack in Signal PQXDH v1[1]

As this attack can be carried out without violating the IND-CCA assumption, it turns out that the IND-CCA security of the KEM scheme is not enough to show the full security of PQXDH. We in fact require an additional assumption, which is not a classical cryptographic one, but which informally captures that the shared secret is strongly linked to the public key. While many schemes such as Kyber/ML-KEM do include the public key in the shared secret derivation, it may be prudent to add PQPK somewhere else in the protocol, for instance in the associated data of the AEAD encrypted message or directly in the KDF. Such changes are considered for a next version of the PQXDH protocol.

This is an important observation, as some KEM designers explicitly state that "Application designers are encouraged to assume solely the standard IND-CCA2 property" [MCR], and notably, both HQC and BIKE do not directly tie the shared secret to the public key, but only to the ciphertext.

[1] https://cryspen.com/post/pqxdh/

Let’s look under the hood, just in case

HPKE's DHKEM[1] and Key Schedule

HPKE's DHKEM[1] and Key Schedule

```
def ExtractAndExpand(dh, kem_context):
  eae_prk = LabeledExtract("", "eae_prk", dh)
  shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                kem_context, Nsecret)
  return shared_secret

def Encap(pkR):
  skE, pkE = GenerateKeyPair()
  dh = DH(skE, pkR)
  enc = SerializePublicKey(pkE)

  pkRm = SerializePublicKey(pkR)
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret, enc

def Decap(enc, skR):
  pkE = DeserializePublicKey(enc)
  dh = DH(skR, pkE)

  pkRm = SerializePublicKey(pk(skR))
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret
```

HPKE's DHKEM[1] and Key Schedule

```
def ExtractAndExpand(dh, kem_context):
  eae_prk = LabeledExtract("", "eae_prk", dh)
  shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                kem_context, Nsecret)
  return shared_secret

def Encap(pkR):
  skE, pkE = GenerateKeyPair()
  dh = DH(skE, pkR)
  enc = SerializePublicKey(pkE)

  pkRm = SerializePublicKey(pkR)
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret, enc

def Decap(enc, skR):
  pkE = DeserializePublicKey(enc)
  dh = DH(skR, pkE)

  pkRm = SerializePublicKey(pk(skR))
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret
```

```
def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
  VerifyPSKInputs(mode, psk, psk_id)

  psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
  info_hash = LabeledExtract("", "info_hash", info)
  key_schedule_context = concat(mode, psk_id_hash, info_hash)

  secret = LabeledExtract(shared_secret, "secret", psk)

  key = LabeledExpand(secret, "key", key_schedule_context, Nk)
  base_nonce = LabeledExpand(secret, "base_nonce",
                             key_schedule_context, Nn)
  exporter_secret = LabeledExpand(secret, "exp",
                                  key_schedule_context, Nh)

  return Context<ROLE>(key, base_nonce, 0, exporter_secret)
```

HPKE's DHKEM[1] and Key Schedule

```
def ExtractAndExpand(dh, kem_context):
  eae_prk = LabeledExtract("", "eae_prk", dh)
  shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                kem_context, Nsecret)
  return shared_secret

def Encap(pkR):
  skE, pkE = GenerateKeyPair()
  dh = DH(skE, pkR)
  enc = SerializePublicKey(pkE)

  pkRm = SerializePublicKey(pkR)
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret, enc

def Decap(enc, skR):
  pkE = DeserializePublicKey(enc)
  dh = DH(skR, pkE)

  pkRm = SerializePublicKey(pk(skR))
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret
```

```
def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
  VerifyPSKInputs(mode, psk, psk_id)

  psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
  info_hash = LabeledExtract("", "info_hash", info)
  key_schedule_context = concat(mode, psk_id_hash, info_hash)

  secret = LabeledExtract(shared_secret, "secret", psk)

  key = LabeledExpand(secret, "key", key_schedule_context, Nk)
  base_nonce = LabeledExpand(secret, "base_nonce",
                             key_schedule_context, Nn)
  exporter_secret = LabeledExpand(secret, "exp",
                                  key_schedule_context, Nh)

  return Context<ROLE>(key, base_nonce, 0, exporter_secret)
```

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem

HPKE's DHKEM[1] and Key Schedule

```
def ExtractAndExpand(dh, kem_context):
  eae_prk = LabeledExtract("", "eae_prk", dh)
  shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                kem_context, Nsecret)
  return shared_secret

def Encap(pkR):
  skE, pkE = GenerateKeyPair()
  dh = DH(skE, pkR)
  enc = SerializePublicKey(pkE)

  pkRm = SerializePublicKey(pkR)
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret, enc

def Decap(enc, skR):
  pkE = DeserializePublicKey(enc)
  dh = DH(skR, pkE)

  pkRm = SerializePublicKey(pk(skR))
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret
```

```
def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
  VerifyPSKInputs(mode, psk, psk_id)

  psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
  info_hash = LabeledExtract("", "info_hash", info)
  key_schedule_context = concat(mode, psk_id_hash, info_hash)

  secret = LabeledExtract(shared_secret, "secret", psk)

  key = LabeledExpand(secret, "key", key_schedule_context, Nk)
  base_nonce = LabeledExpand(secret, "base_nonce",
                             key_schedule_context, Nn)
  exporter_secret = LabeledExpand(secret, "exp",
                                  key_schedule_context, Nh)

  return Context<ROLE>(key, base_nonce, 0, exporter_secret)
```

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem

HPKE's DHKEM[1] and Key Schedule

```
def ExtractAndExpand(dh, kem_context):
  eae_prk = LabeledExtract("", "eae_prk", dh)
  shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                kem_context, Nsecret)
  return shared_secret

def Encap(pkR):
  skE, pkE = GenerateKeyPair()
  dh = DH(skE, pkR)
  enc = SerializePublicKey(pkE)

  pkRm = SerializePublicKey(pkR)
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret, enc

def Decap(enc, skR):
  pkE = DeserializePublicKey(enc)
  dh = DH(skR, pkE)

  pkRm = SerializePublicKey(pk(skR))
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret
```

```
def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
  VerifyPSKInputs(mode, psk, psk_id)

  psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
  info_hash = LabeledExtract("", "info_hash", info)
  key_schedule_context = concat(mode, psk_id_hash, info_hash)

  secret = LabeledExtract(shared_secret, "secret", psk)

  key = LabeledExpand(secret, "key", key_schedule_context, Nk)
  base_nonce = LabeledExpand(secret, "base_nonce",
                             key_schedule_context, Nn)
  exporter_secret = LabeledExpand(secret, "exp",
                                  key_schedule_context, Nh)

  return Context<ROLE>(key, base_nonce, 0, exporter_secret)
```

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem

HPKE's DHKEM[1] and Key Schedule

```
def ExtractAndExpand(dh, kem_context):
  eae_prk = LabeledExtract("", "eae_prk", dh)
  shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                kem_context, Nsecret)
  return shared_secret

def Encap(pkR):
  skE, pkE = GenerateKeyPair()
  dh = DH(skE, pkR)
  enc = SerializePublicKey(pkE)

  pkRm = SerializePublicKey(pkR)
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret, enc

def Decap(enc, skR):
  pkE = DeserializePublicKey(enc)
  dh = DH(skR, pkE)

  pkRm = SerializePublicKey(pk(skR))
  kem_context = concat(enc, pkRm)

  shared_secret = ExtractAndExpand(dh, kem_context)
  return shared_secret
```

```
def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
  VerifyPSKInputs(mode, psk, psk_id)

  psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
  info_hash = LabeledExtract("", "info_hash", info)
  key_schedule_context = concat(mode, psk_id_hash, info_hash)

  secret = LabeledExtract(shared_secret, "secret", psk)

  key = LabeledExpand(secret, "key", key_schedule_context, Nk)
  base_nonce = LabeledExpand(secret, "base_nonce",
                             key_schedule_context, Nn)
  exporter_secret = LabeledExpand(secret, "exp",
                                  key_schedule_context, Nh)

  return Context<ROLE>(key, base_nonce, 0, exporter_secret)
```

pkR and pkE are bound[2] by shared_secret

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem

[2] https://eprint.iacr.org/2023/1933.pdf

HPKE's DHKEM[1] Binding Properties

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem

HPKE's DHKEM[1] Binding Properties

- DHKEM is MAL-BIND-K-CT and MAL-BIND-K-PK secure as analyzed in the symbolic model[2]

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem
[2] https://eprint.iacr.org/2023/1933.pdf

HPKE's DHKEM[1] Binding Properties

- DHKEM is MAL-BIND-K-CT and MAL-BIND-K-PK secure as analyzed in the symbolic model[2]
- These give the strongest protections against re-encapsulation attacks from a malicious adversary manipulating key material however they like (MAL)

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem
[2] https://eprint.iacr.org/2023/1933.pdf

HPKE's DHKEM[1] Binding Properties

- DHKEM is MAL-BIND-K-CT and MAL-BIND-K-PK secure as analyzed in the symbolic model[2]
- These give the strongest protections against re-encapsulation attacks from a malicious adversary manipulating key material however they like (MAL)
- It is SAFE to just take the raw shared_secret from DHKEM and use it in HPKE's KeySchedule() without including any other KEM 'transcript' context

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem
[2] https://eprint.iacr.org/2023/1933.pdf

HPKE's DHKEM[1] Binding Properties

- DHKEM is MAL-BIND-K-CT and MAL-BIND-K-PK secure as analyzed in the symbolic model[2]
- These give the strongest protections against re-encapsulation attacks from a malicious adversary manipulating key material however they like (MAL)
- It is SAFE to just take the raw shared_secret from DHKEM and use it in HPKE's KeySchedule() without including any other KEM 'transcript' context
- What about ML-KEM?

[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem
[2] https://eprint.iacr.org/2023/1933.pdf

ML-KEM[1] Binding Properties[2]

Algorithm 17 ML-KEM.Decaps(c, dk)

Uses the decapsulation key to produce a shared key from a ciphertext.

Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

1: $\mathsf{dk}_{\text{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$ ▷ extract (from KEM decaps key) the PKE decryption key
2: $\mathsf{ek}_{\text{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$ ▷ extract PKE encryption key
3: $h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
4: $z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
5: $m' \leftarrow$ K-PKE.Decrypt$(\mathsf{dk}_{\text{PKE}}, c)$ ▷ decrypt ciphertext
6: $(K', r') \leftarrow G(m' \| h)$
7: $\bar{K} \leftarrow J(z \| c, 32)$
8: $c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\text{PKE}}, m', r')$ ▷ re-encrypt using the derived randomness r'
9: **if** $c \neq c'$ **then**
10: $K' \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
11: **end if**
12: **return** K'

[1] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf
[2] https://eprint.iacr.org/2023/1933.pdf

ML-KEM[1] Binding Properties[2]

Algorithm 17 ML-KEM.Decaps(c, dk)

Uses the decapsulation key to produce a shared key from a ciphertext.

Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

1: $\mathsf{dk}_{\text{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$ ▷ extract (from KEM decaps key) the PKE decryption key
2: $\mathsf{ek}_{\text{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$ ▷ extract PKE encryption key
3: $h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
4: $z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
5: $m' \leftarrow$ K-PKE.Decrypt$(\mathsf{dk}_{\text{PKE}}, c)$ ▷ decrypt ciphertext
6: $(K', r') \leftarrow G(m' \| h)$
7: $\bar{K} \leftarrow J(z \| c, 32)$
8: $c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\text{PKE}}, m', r')$ ▷ re-encrypt using the derived randomness r'
9: **if** $c \neq c'$ **then**
10: $K' \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
11: **end if**
12: **return** K'

- ML-KEM's shared secret K binds ekPKE (PK) via hashing in the hash of ekPKE: MAL-BIND-K-PK

[1] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2024/039.pdf

ML-KEM[1] Binding Properties[2]

Algorithm 17 ML-KEM.Decaps(c, dk)

Uses the decapsulation key to produce a shared key from a ciphertext.

Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

1: $\mathsf{dk}_{\text{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$ ▷ extract (from KEM decaps key) the PKE decryption key
2: $\mathsf{ek}_{\text{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$ ▷ extract PKE encryption key
3: $h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
4: $z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
5: $m' \leftarrow$ K-PKE.Decrypt$(\mathsf{dk}_{\text{PKE}}, c)$ ▷ decrypt ciphertext
6: $(K', r') \leftarrow G(m' \| h)$
7: $\bar{K} \leftarrow J(z \| c, 32)$
8: $c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\text{PKE}}, m', r')$ ▷ re-encrypt using the derived randomness r'
9: **if** $c \neq c'$ **then**
10: $K' \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
11: **end if**
12: **return** K'

- ML-KEM's shared secret K binds ekPKE (PK) via hashing in the hash of ekPKE: MAL-BIND-K-PK
- Binding the ciphertext *c* relies on the robustness properties of K-PKE

[1] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2024/039.pdf

ML-KEM[1] Binding Properties[2]

Algorithm 17 ML-KEM.Decaps(c, dk)

Uses the decapsulation key to produce a shared key from a ciphertext.

Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

1: $\mathsf{dk}_{\text{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$ ▷ extract (from KEM decaps key) the PKE decryption key
2: $\mathsf{ek}_{\text{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$ ▷ extract PKE encryption key
3: $h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
4: $z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
5: $m' \leftarrow \text{K-PKE.Decrypt}(\mathsf{dk}_{\text{PKE}}, c)$ ▷ decrypt ciphertext
6: $(K', r') \leftarrow G(m' \| h)$
7: $\bar{K} \leftarrow J(z \| c, 32)$
8: $c' \leftarrow \text{K-PKE.Encrypt}(\mathsf{ek}_{\text{PKE}}, m', r')$ ▷ re-encrypt using the derived randomness r'
9: **if** $c \neq c'$ **then**
10: $K' \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
11: **end if**
12: **return** K'

- ML-KEM's shared secret K binds ekPKE (PK) via hashing in the hash of ekPKE: MAL-BIND-K-PK
- Binding the ciphertext *c* relies on the robustness properties of K-PKE
- Shown to be *chosen ciphertext resistant*[3]: implies *LEAK*-BIND-K-CT

[1] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2024/039.pdf

ML-KEM[1] Binding Properties[2]

Algorithm 17 ML-KEM.Decaps(c, dk)

Uses the decapsulation key to produce a shared key from a ciphertext.

Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

1: $\mathsf{dk}_{\text{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$ ▷ extract (from KEM decaps key) the PKE decryption key
2: $\mathsf{ek}_{\text{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$ ▷ extract PKE encryption key
3: $h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
4: $z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
5: $m' \leftarrow$ K-PKE.Decrypt$(\mathsf{dk}_{\text{PKE}}, c)$ ▷ decrypt ciphertext
6: $(K', r') \leftarrow G(m' \| h)$
7: $\bar{K} \leftarrow J(z \| c, 32)$
8: $c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\text{PKE}}, m', r')$ ▷ re-encrypt using the derived randomness r'
9: **if** $c \neq c'$ **then**
10: $K' \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
11: **end if**
12: **return** K'

- ML-KEM's shared secret K binds ekPKE (PK) via hashing in the hash of ekPKE: MAL-BIND-K-PK
- Binding the ciphertext *c* relies on the robustness properties of K-PKE
- Shown to be *chosen ciphertext resistant*[3]: implies *LEAK*-BIND-K-CT
- LEAK is resistant to adversaries with access to leaked, honestly-generated key pairs, strictly weaker security than MAL
- Strictly weaker binding properties as a KEM than DHKEM

[1] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2024/039.pdf

But does this matter for HPKE?

A counterexample: Classic McEliece[1]

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf

A counterexample: Classic McEliece[1]

5.6 Decapsulation

The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

1. Set $b \leftarrow 1$.
2. Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = \mathsf{H}(b, e, C)$; see Section 6.2 for H input encodings.
5. Output session key K.

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf

A counterexample: Classic McEliece[1]

- IND-CCA ✅

5.6 Decapsulation

The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

1. Set $b \leftarrow 1$.
2. Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = \mathsf{H}(b, e, C)$; see Section 6.2 for H input encodings.
5. Output session key K.

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf

A counterexample: Classic McEliece[1]

- IND-CCA ✅
- Binds the ciphertext C: MAL-BIND-K-CT[2]

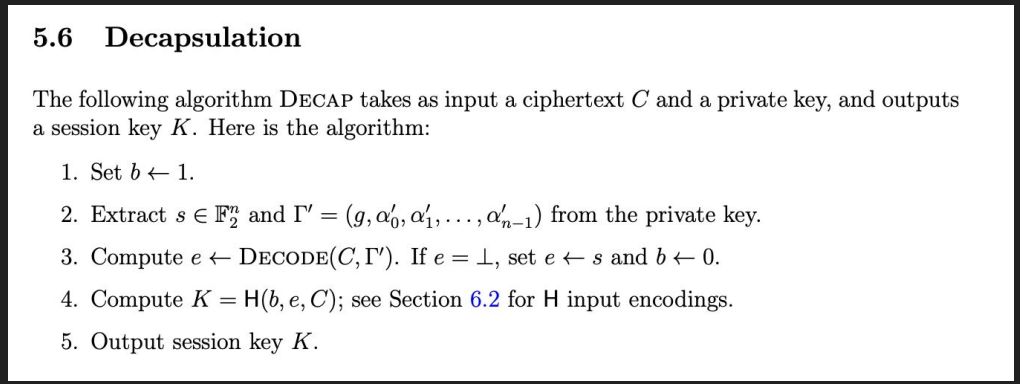
5.6 Decapsulation

The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

1. Set $b \leftarrow 1$.
2. Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = \mathsf{H}(b, e, C)$; see Section 6.2 for H input encodings.
5. Output session key K.

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf
[2] https://eprint.iacr.org/2023/1933.pdf

A counterexample: Classic McEliece[1]



5.6 Decapsulation

The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

1. Set $b \leftarrow 1$.
2. Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = \mathsf{H}(b, e, C)$; see Section 6.2 for H input encodings.
5. Output session key K.

- IND-CCA ✅
- Binds the ciphertext C: MAL-BIND-K-CT
- Encapsulation key binding depends on PKE robustness[3]

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2021/708.pdf

A counterexample: Classic McEliece[1]

5.6 Decapsulation

The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

1. Set $b \leftarrow 1$.
2. Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = \mathsf{H}(b, e, C)$; see Section 6.2 for H input encodings.
5. Output session key K.

- IND-CCA ✅
- Binds the ciphertext C: MAL-BIND-K-CT
- Encapsulation key binding depends on PKE robustness[3]
- [3]: ‘for any plaintext m, they find that it is possible to construct a single ciphertext c that always decrypts to m under any Classic McEliece private key’

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2021/708.pdf

A counterexample: Classic McEliece[1]

5.6 Decapsulation

The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

1. Set $b \leftarrow 1$.
2. Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = \mathsf{H}(b, e, C)$; see Section 6.2 for H input encodings.
5. Output session key K.

- IND-CCA ✅
- Binds the ciphertext C: MAL-BIND-K-CT
- Encapsulation key binding depends on PKE robustness[3]
- [3]: ‘for any plaintext m, they find that it is possible to construct a single ciphertext c that always decrypts to m under any Classic McEliece private key’
- Therefore offers *no PK binding at all*

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2021/708.pdf

A counterexample: Classic McEliece[1]

5.6 Decapsulation

The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

1. Set $b \leftarrow 1$.
2. Extract $s \in \mathbb{F}_2^n$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = \mathsf{H}(b, e, C)$; see Section 6.2 for H input encodings.
5. Output session key K.

- IND-CCA ✅
- Binds the ciphertext C: MAL-BIND-K-CT
- Encapsulation key binding depends on PKE robustness[3]
- [3]: ‘for any plaintext m, they find that it is possible to construct a single ciphertext c that always decrypts to m under *any* Classic McEliece private key’
- Therefore offers *no PK binding at all*
- If used in place of DHKEM, allows an HPKE payload to be decrypted under *any* key pair, not just the one used to encrypt it

[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2021/708.pdf

Since HPKE does not bind KEM PK or CT outside of DHKEM, the binding properties of the KEM itself matter

Let’s match[1] DHKEM

Let’s match[1] DHKEM

```
def Encap(pkR):
  ss, ct = MLKEM.Encaps(pkR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
~~~

~~~
def Decap(enc, skR):
  ss, ct = MLKEM.Decaps(enc, skR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
```

[1] https://github.com/dconnolly/draft-connolly-cfrg-hpke-mlkem

Let’s match[1] DHKEM

```
def Encap(pkR):
  ss, ct = MLKEM.Encaps(pkR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
~~~

~~~
def Decap(enc, skR):
  ss, ct = MLKEM.Decaps(enc, skR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
```

- Another ExtractAndExpand() to bind to the CT: MAL-BIND-K-CT

[1] https://github.com/dconnolly/draft-connolly-cfrg-hpke-mlkem

Let’s match[1] DHKEM

```
def Encap(pkR):
  ss, ct = MLKEM.Encaps(pkR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
~~~

~~~
def Decap(enc, skR):
  ss, ct = MLKEM.Decaps(enc, skR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
```

- Another ExtractAndExpand() to bind to the CT: MAL-BIND-K-CT
- ML-KEM is already MAL-BIND-K-PK

[1] https://github.com/dconnolly/draft-connolly-cfrg-hpke-mlkem

Let’s match[1] DHKEM

```
def Encap(pkR):
  ss, ct = MLKEM.Encaps(pkR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
~~~

~~~
def Decap(enc, skR):
  ss, ct = MLKEM.Decaps(enc, skR)

  shared_secret = ExtractAndExpand(ss, ct)

  return shared_secret, ct
```

- Another ExtractAndExpand() to bind to the CT: MAL-BIND-K-CT
- ML-KEM is already MAL-BIND-K-PK
- Done ✅

[1] https://github.com/dconnolly/draft-connolly-cfrg-hpke-mlkem

Alternative:

Alternative: change HPKE to bind PK and CT?

Alternative: change HPKE to bind PK and CT?

- Changes a fixed standard
- Already implemented and deployed
- Would avoid having to do this analysis for all future KEMs!
- Could enforce this generic wrapper (including PK, not just CT) for all KEMs, regardless of their binding properties: HPKE-KEM wrapper
    - Would allow the (wrapped) Classic McEliece ciphersuite!
- Backwards-compatible!

Alternative: change HPKE to bind PK and CT?

“The design principle of HPKE is to bind the KEM public key and ciphertext to the generated secret to prevent any re-encapsulation attacks. HPKE already hard-codes this for DH, without worrying about whether it is necessary for prime-order groups etc, and we should do the same for all PQ-KEM things. This structure can be better clarified by separating out the inner KEM from the HPKE-KEM wrapper, but it may require some fiddling about for AuthKEMs.”

- An HPKE co-author

Problems/Limitations in HPKEv1

HPKE relies on a KEM scheme that must provide a set of functions (Encap/Decap, Serialize/Deserialize, Derive) but it does not say that Encap/Decap must bind the encapsulated key with the public key of the recipient and the encapsulated ciphertext, even though the RFC does do this binding for DH-KEM. Consequently, new KEM ciphersuites for HPKE do not neatly fit the HPKE spec in that they cannot directly replace Encap/Decap interface.

a. As a consequence, defining a PQ-KEM HPKE ciphersuite is not a simple extension and might need to do something different for each PQ-KEM

Problems/Limitations in HPKEv1

HPKE does not define a signature-based AuthKEM. This means that there is no HPKE mode that is secure against key compromise impersonation (KCI) attacks,

a. As a consequence, defining a PQ-KEM+PQ-SIG HPKE ciphersuite is not immediately simple.
b. As a second consequence, if we do allow a signature-based AuthKEM, we hit the weakness pointed out by Alwen et al. where in one-shot HPKE, we could sign a hash of the message content to provide KCI resistance for the payload.
c. It is also worth pointing out that AuthEncap as defined in HPKE is not as privacy-preserving as it could be in all scenarios. It requires the application to communicate the sender's public key to the recipient outside HPKE.

Problems/Limitations in HPKEv1

HPKE allows for the mixing of one Encap/AuthEncap scheme with one PSK, but does not itself provide a way of mixing multiple Encap/AuthEncap or multiple PSKs.

a. As a consequence, we have to define external KEM ciphersuites for every combination of encap/authencap we wish to support, rather than allowing these combinations within HPKE