ML-KEM for HPKE

draft-connolly-cfrg-hpke-mlkem

https://github.com/dconnolly/draft-connolly-cfrg-hpke-mlkem
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
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 $N_{secret}$. 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
The ML-KEM construction. At a high level, the ML-KEM construction proceeds in two steps. First, the idea mentioned above is used to construct a public-key encryption scheme from the MLWE problem. Second, this public-key encryption scheme is converted into a key-encapsulation mechanism using the so-called Fujisaki-Okamoto (FO) transform [11, 12]. In addition to producing a KEM, the FO transform is also intended to provide security in a significantly more general adversarial attack model. As a result, ML-KEM is believed to satisfy so-called IND-CCA security [1, 4, 13].
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

Re-encapsulation attack in Signal PQXDH v1

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 v1¹

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\textsuperscript{1} and Key Schedule
HPKE’s DHKEM¹ and Key Schedule

```python
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 = ExtractAnd Expand(dh, kem_context)
    return shared_secret
```
HPKE’s DHKEM and Key Schedule

```python
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(mode, psk_id_hash, info_id)
    shared_secret = ExtractAndExpand(dh, kem_context)
    return shared_secret, enc

def Decap(enc, skR):
    pkE = DeserializePublicKey(enc)
    dh = DH(skR, pkE)
    pkRm = SerializePublicKey(pk(pkR))
    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 DHKEM¹ and Key Schedule

```python
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(mode, psk_id_hash, info_hash)

    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-dhkm
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 DHKEM¹ and Key Schedule

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

```python
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 DHKEM¹ and Key Schedule

```python
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(pkR))
    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 bound² by shared_secret

¹ https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem
HPKE’s DHKEM¹ 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](https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem)

HPKE’s DHKEM¹ Binding Properties

- DHKEM is MAL-BIND-K-CT and MAL-BIND-K-PK secure as analyzed in the symbolic model²
- These give the strongest protections against re-encapsulation attacks from a malicious adversary manipulating key material however they like (MAL)

**HPKE’s DHKEM**¹ Binding Properties

- DHKEM is MAL-BIND-K-CT and MAL-BIND-K-PK secure as analyzed in the symbolic model²
- 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

HPKE’s **DHKEM**¹ Binding Properties

- DHKEM is MAL-BIND-K-CT and MAL-BIND-K-PK secure as analyzed in the symbolic model²
- 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?

ML-KEM¹ Binding Properties²

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

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

**Validated input:** ciphertext $c \in \mathbb{B}^{32(d_k+\ell_c)}$.

**Validated input:** decapsulation key $dk \in \mathbb{B}^{768k+96}$.

**Output:** shared key $K \in \mathbb{B}^{32}$.

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

ML-KEM$^1$ Binding Properties$^2$

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

---

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

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

**Validated input:** ciphertext $c \in \mathbb{B}^{32(d_k + d_\ell)}$.

**Validated input:** decapsulation key $dk \in \mathbb{B}^{768k + 96}$.

**Output:** shared key $K \in \mathbb{B}^{32}$.

1. $dk_{PKE} \leftarrow dk[0 : 384k]$ ▷ extract (from KEM decaps key) the PKE decryption key
2. $ek_{PKE} \leftarrow dk[384k : 768k + 32]$ ▷ extract PKE encryption key
3. $h \leftarrow dk[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
4. $z \leftarrow dk[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
5. $m' \leftarrow K$-$PKE$.Decrypt($dk_{PKE}, c$) ▷ decrypt ciphertext
6. $(K', r') \leftarrow G(m' || h)$
7. $\bar{K} \leftarrow J(z||c, 32)$
8. $c' \leftarrow K$-$PKE$.Encrypt($ek_{PKE}, m', r'$) ▷ 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 $K'$

---


$^3$ [https://eprint.iacr.org/2024/039.pdf](https://eprint.iacr.org/2024/039.pdf)
ML-KEM¹ Binding Properties²

- 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

---

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

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

**Validated input:** ciphertext $c \in \mathbb{B}^{32(h_0 + d_0)}$.

**Validated input:** decapsulation key $dk \in \mathbb{B}^{768k + 96}$.

**Output:** shared key $K \in \mathbb{B}^{32}$.

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

---

³ [https://eprint.iacr.org/2024/039.pdf](https://eprint.iacr.org/2024/039.pdf)
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 resistant³: implies LEAK-BIND-K-CT

---

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

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

<table>
<thead>
<tr>
<th>Algorithm 17 ML-KEM.Decaps(c, dk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses the decapsulation key to produce a shared key from a ciphertext.</td>
</tr>
</tbody>
</table>

**Validated input**: ciphertext $c \in \mathbb{B}^{32(d_k+d_c)}$.

**Validated input**: decapsulation key $dk \in \mathbb{B}^{768k+96}$.

**Output**: shared key $K \in \mathbb{B}^{32}$.

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

---

³ https://eprint.iacr.org/2024/039.pdf
ML-KEM¹ Binding Properties²

- 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 resistant³: 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

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

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

Validated input: ciphertext c ∈ B^{32(d_k+d_e)}.
Validated input: decapsulation key dk ∈ B^{768k+96}.
Output: shared key K ∈ B^{32}.

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

3 https://eprint.iacr.org/2024/039.pdf
But does this matter for HPKE?
A counterexample: Classic McEliece¹

¹ https://classic.mceliece.org/mceliece-spec-20221023.pdf
A counterexample: Classic McEliece

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 $s \in \mathbb{F}_q$ 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 $K = H(b, e, C)$; 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 McEliece¹

5.6 Decapsulation

The following algorithm \textsc{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 \( K = H(b, e, C) \); 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 McEliece¹

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 $e = \bot$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = H(b, e, C)$; 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 McEliece¹

5.6 Decapsulation

The following algorithm \( \text{DECAP} \) 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 \( s \in \mathbb{F}_q^* \) 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 \( K = H(b, e, C) \); 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 robustness³

¹ https://classic.mceliece.org/mceliece-spec-20221023.pdf
A counterexample: Classic McEliece¹

5.6 Decapsulation

The following algorithm \textsc{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 $e = \bot$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = H(b, e, C)$; 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 robustness³
- [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
A counterexample: Classic McEliece¹

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 $s \in \mathbb{F}_q$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \bot$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = H(b, e, C)$; 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 robustness³
- [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
A counterexample: Classic McEliece¹

5.6 Decapsulation

The following algorithm $\text{DECAP}$ 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 $s \in \mathbb{F}_q^\beta$ and $\Gamma' = (g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow \text{DECODE}(C, \Gamma')$. If $e = \bot$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = H(b, e, C)$; 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 robustness³
- [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
Since HPKE does not bind KEM PK or CT outside of DHKEM, the binding properties of the KEM itself matter.
Let’s match\textsuperscript{1} DHKEM
Let's match¹ DHKEM

```python
def Encap(pkR):
    ss, ct = MLKEM.Encaps(pkR)
    shared_secret = ExtractAndExpand(ss, ct)
    return shared_secret, ct
```

```python
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 match¹ DHKEM

```python
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

- Another ExtractAndExpand() to bind to the CT: MAL-BIND-K-CT
Let's match¹ DHKEM

```python
def Encap(pkR):
    ss, ct = MLKEM.Encaps(pkR)
    shared_secret = ExtractAndExpand(ss, ct)
    return shared_secret, ct
```

```python
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 match¹ DHKEM

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

    shared_secret = ExtractAndExpand(ss, ct)

    return shared_secret, ct
```

```python
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
Questions?
ML-KEM for HPKE

draft-connolly-cfrg-hpke-mlkem

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