Hybrid Key Establishment in Production

Deirdre Connolly | July 15, 2025

Key Establishment
Signatures
‘Fancy Crypto’

Key Establishment

Signatures

‘Fancy Crypto’

Key Establishment

Signatures

‘Fancy Crypto’

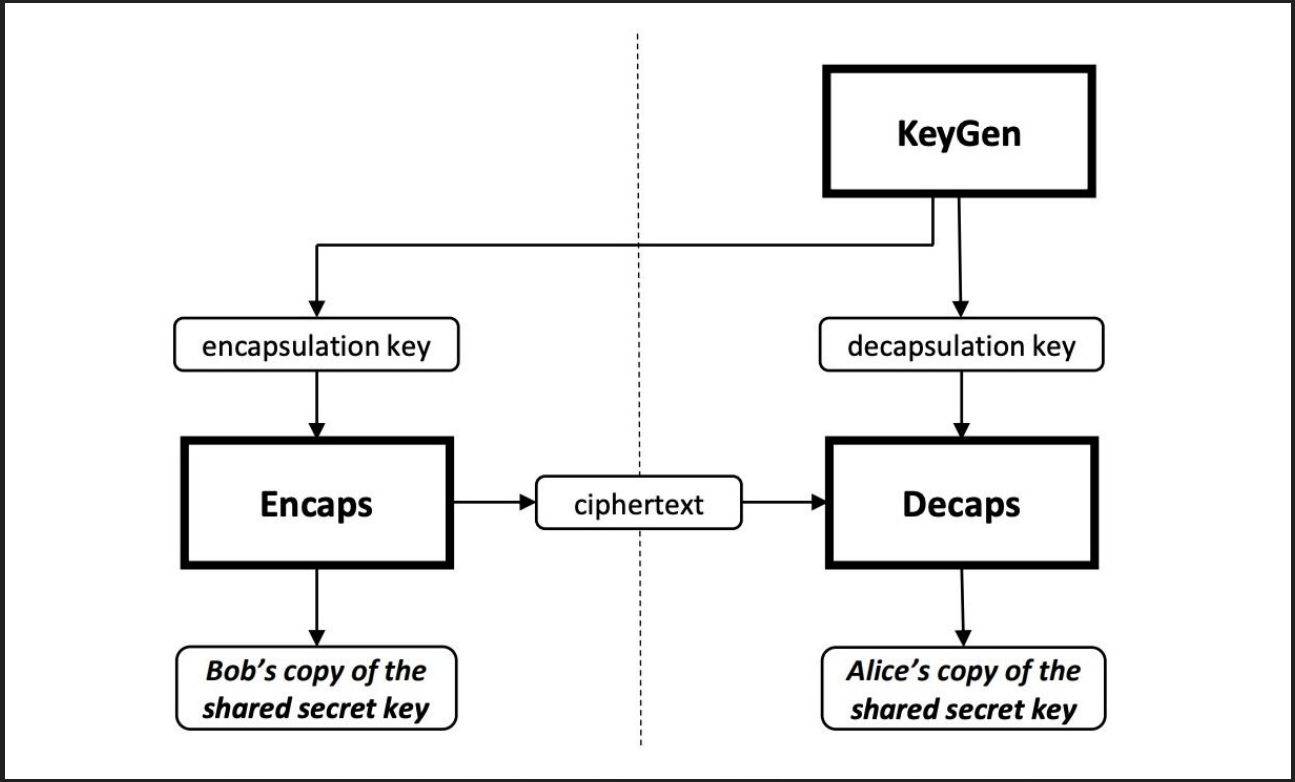
☠️ RIP Diffie-Hellman ☠️

Alice		Bob
①	Choose sk_A	Choose sk_B
②	$k_A=sk_AG(p,a)$	$k_B=sk_BG(p,a)$
③	k_A →	← k_B
④	$k_{AB}=sk_Ak_B$	$k_{BA}=sk_Bk_A$

KEMs

KEMs

Key Encapsulation Mechanisms


KeyGen
encapsulation key
decapsulation key
Encaps
ciphertext
Decaps
Bob's copy of the shared secret key
Alice's copy of the shared secret key

$\text{IND-CCA}^{\mathsf{KEM}}_{\mathcal{A}}$:

$(sk, pk) \leftarrow\$ \mathsf{KeyGen}()$

$(c_0, k_0) \leftarrow\$ \mathsf{Encaps}(pk)$

$k_1 \leftarrow\$ \mathcal{K}$

$b \leftarrow\$ \{0, 1\}$

$b' \leftarrow\$ \mathcal{A}^{D(sk,pk,\cdot)}(c_0, k_b, pk)$

return $b = b'$

$D(sk, pk, c)$:

if $c \neq c_0$ **then**

$k \leftarrow \mathsf{Decaps}(ct, sk)$

return k

Fujisaki-Okamoto (FO) transform, implicit-rejection

$\underline{\mathsf{Encaps}(pk)}$	$\underline{\mathsf{Decaps}^{\not\perp}(sk, c)}$
01 $m \xleftarrow{\$} \mathcal{M}$	05 $m' := \mathsf{Dec}'(sk, c)$
02 $c \leftarrow \mathsf{Enc}'(pk, m)$	06 **if** $m' = \perp$
03 $K := \mathsf{H}(m, c)$	07 **return** $K := \mathsf{H}(\mathrm{DS}, c)$
04 **return** (K, c)	08 **else return** $K := \mathsf{H}(m', c)$

Fujisaki-Okamoto (FO) transform

Fujisaki-Okamoto (FO) transform

- Turns an IND-CPA-secure public key encryption scheme into a IND-CCA key encapsulation scheme

Fujisaki-Okamoto (FO) transform

- Turns an IND-CPA-secure public key encryption scheme into a IND-CCA key encapsulation scheme
- Popular amongst all the NIST PQC KEM candidates

Fujisaki-Okamoto (FO) transform

- Turns an IND-CPA-secure public key encryption scheme into a IND-CCA key encapsulation scheme
- Popular amongst all the NIST PQC KEM candidates
- Explicit and implicit rejection forms, the KEMs we care about are all implicit rejection (no error codes or panics, returns a non-zero pseudorandom value always)

https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4

NIST PQC Standardization Process | HQC Announced as a 4th Round Selection

March 11, 2025

https://csrc.nist.gov/News/2025/hqc-announced-as-a-4th-round-selection

Algorithms to be Standardized

Public-Key Encryption/KEMs	Digital Signatures
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS$^+$

Algorithms to be Standardized	
Public-Key Encryption/KEMs	**Digital Signatures**
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS$^+$

CRYSTALS-KYBER

Algorithm Specifications And Supporting Documentation

(version 3.02)

Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint,
Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé

August 4, 2021

Algorithm 7 KYBER.CCAKEM.KeyGen()

Output: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$

1: $z \leftarrow \mathcal{B}^{32}$
2: $(pk, sk') := \text{KYBER.CPAPKE.KeyGen}()$
3: $sk := (sk' \| pk \| \text{H}(pk) \| z)$
4: **return** (pk, sk)

Algorithm 8 KYBER.CCAKEM.Enc(pk)

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Output: Shared key $K \in \mathcal{B}^*$

1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow \mathrm{H}(m)$
3: $(\bar{K}, r) := \mathrm{G}(m \| \mathrm{H}(pk))$
4: $c := \text{KYBER.CPAPKE.Enc}(pk, m, r)$
5: $K := \mathsf{KDF}(\bar{K} \| \mathrm{H}(c))$
6: **return** (c, K)

Algorithm 9 KYBER.CCAKEM.Dec(c, sk)

Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Input: Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$
Output: Shared key $K \in \mathcal{B}^*$

$pk := sk + 12 \cdot k \cdot n/8$
$h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
$z := sk + 24 \cdot k \cdot n/8 + 64$
$m' := \text{KYBER.CPAPKE.Dec}(sk, c)$
$(\bar{K}', r') := \text{G}(m' \| h)$
$c' := \text{KYBER.CPAPKE.Enc}(pk, m', r')$
if $c = c'$ **then**
    return $K := \text{KDF}(\bar{K}' \| \text{H}(c))$
else
    return $K := \text{KDF}(z \| \text{H}(c))$
end if
return K

FIPS 203

Federal Information Processing Standards Publication

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Category: Computer Security **Subcategory: Cryptography**

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

This publication is available free of charge from:
https://doi.org/10.6028/NIST.FIPS.203

Published August 13, 2024

https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf

Algorithm 15 ML-KEM.KeyGen()

Generates an encapsulation key and a corresponding decapsulation key.

Output: Encapsulation key $\mathsf{ek} \in \mathbb{B}^{384k+32}$.
Output: Decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.

1: $z \xleftarrow{\$} \mathbb{B}^{32}$ ▷ z is 32 random bytes (see Section 3.3)
2: $(\mathsf{ek}_{\text{PKE}}, \mathsf{dk}_{\text{PKE}}) \leftarrow$ K-PKE.KeyGen() ▷ run key generation for K-PKE
3: $\mathsf{ek} \leftarrow \mathsf{ek}_{\text{PKE}}$ ▷ KEM encaps key is just the PKE encryption key
4: $\mathsf{dk} \leftarrow (\mathsf{dk}_{\text{PKE}} \| \mathsf{ek} \| H(\mathsf{ek}) \| z)$ ▷ KEM decaps key includes PKE decryption key
5: **return** $(\mathsf{ek}, \mathsf{dk})$

Algorithm 16 ML-KEM.Encaps(ek)

Uses the encapsulation key to generate a shared key and an associated ciphertext.

Validated input: encapsulation key $\mathsf{ek} \in \mathbb{B}^{384k+32}$.
Output: shared key $K \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

1: $m \xleftarrow{\$} \mathbb{B}^{32}$ ▷ m is 32 random bytes (see Section 3.3)
2: $(K, r) \leftarrow G(m \| H(\mathsf{ek}))$ ▷ derive shared secret key K and randomness r
3: $c \leftarrow$ K-PKE.Encrypt(ek, m, r) ▷ encrypt m using K-PKE with randomness r
4: **return** (K, c)

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 $K \in \mathbb{B}^{32}$.

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

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 $K \in \mathbb{B}^{32}$.

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

Why hybrid?

Why hybrid?

- Want to move to a quantum-secure posture now, to mitigate store-now, decrypt-later threat
- Post-quantum primitives are new and the adoption risk[1] is not fully understood
- Moving to a hybrid post-quantum/traditional construction theoretically provides security if either component breaks (for cryptographic reasons or just implementation or side-channel[2] reasons)

[1] Cryptographically, implementation-wise, etc
[2] KyberSlash: https://eprint.iacr.org/2024/1049

No really, why hybrid?

Elliptic curves are small and computationally ~cheap.

Our current PQ primitives are large and computationally cheap.

Otherwise we wouldn’t be considering PQ/T hybrid constructions…which are ~all based on elliptic curves.

Hybrid Protocols
Hybrid Primitives

Hybrid Protocols
Hybrid Primitives

TLS 1.3

TLS 1.3 hybrid key agreement[1]

```
In other words, the shared secret is calculated as

   concatenated_shared_secret = shared_secret_1 || shared_secret_2

and inserted into the TLS 1.3 key schedule in place of the (EC)DHE
shared secret, as shown in Figure 1.
```

```
                                    |
                                    v
                          Derive-Secret(., "derived", "")
                                    |
                                    v
concatenated_shared_secret -> HKDF-Extract = Handshake Secret
^^^^^^^^^^^^^^^^^^^^^^^^^^          |
                                    +-----> Derive-Secret(...)
                                    +-----> Derive-Secret(...)
                                    |
                                    v
                          Derive-Secret(., "derived", "")
                                    |
                                    v
```

```rust
struct CombinedSecret([u8; COMBINED_SHARED_SECRET_LEN]);

impl CombinedSecret {
    fn combine(x25519: SharedSecret, kyber: kem::SharedSecret) -> Self {
        let mut out = CombinedSecret([0u8; COMBINED_SHARED_SECRET_LEN]);
        out.0[..X25519_LEN].copy_from_slice(x25519.secret_bytes());
        out.0[X25519_LEN..].copy_from_slice(kyber.as_ref());
        out
    }
}
```

[1] https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design
[2] https://github.com/rustls/rustls/blob/main/rustls-post-quantum/src/lib.rs

TLS 1.3 hashes in everything[1]

[1] https://datatracker.ietf.org/doc/html/rfc8446#section-4.4.1

iMessage

iMessage PQ3[1]

10 : // Derive session identifier and root/chain keys

11 : $\pi_{S,s}.\mathsf{sid}[\mathtt{asym}_0] \leftarrow \mathsf{id}_S \| \mathsf{idecvk}_S \| \mathsf{id}_R \| \mathsf{idecvk}_R \| \mathtt{label}_{\mathrm{start}} \| \mathsf{preecpk}_{R,i} \| \mathsf{rchecpk}_{S,s,1} \| \mathsf{prepqct}_{S,s} \| \mathsf{prepqpk}_{R,i}$

12 : $(rk, ck) \leftarrow \mathsf{KDFRKCK}(0, \mathsf{ecss}, \mathsf{pqss}, \pi_{S,s}.\mathsf{sid}[\mathtt{asym}_0])$

$\mathsf{KDFRKCK}(rk, ss_1, ss_2, sid)$

1 : $ext_1 \leftarrow \mathsf{HKDF.Extract}(ikm = ss_1, salt = rk)$

2 : **if** $ss_2 = \perp$ **then** $ext_2 \leftarrow \mathsf{HKDF.Extract}(ikm = ext_1, salt = 0)$

3 : **else** $ext_2 \leftarrow \mathsf{HKDF.Extract}(ikm = ext_1, salt = ss_2)$

4 : $z \leftarrow \mathsf{HKDF.Expand}(ext_2, \mathtt{label}_{\mathrm{rootkeyderivation}} \| sid, 512)$

5 : $rk \leftarrow z[0 \ldots 255], ck \leftarrow z[256 \ldots 511]$

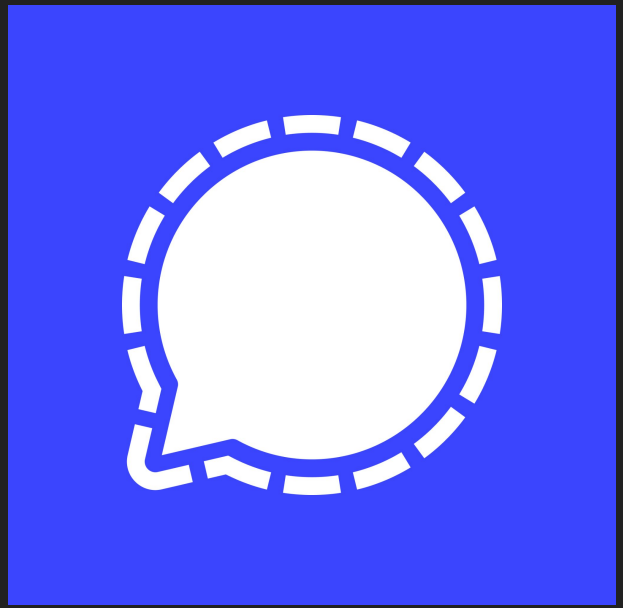
6 : **return** (rk, ck)

[1] https://eprint.iacr.org/2024/357
[2] https://github.com/rustls/rustls/blob/main/rustls-post-quantum/src/lib.rs

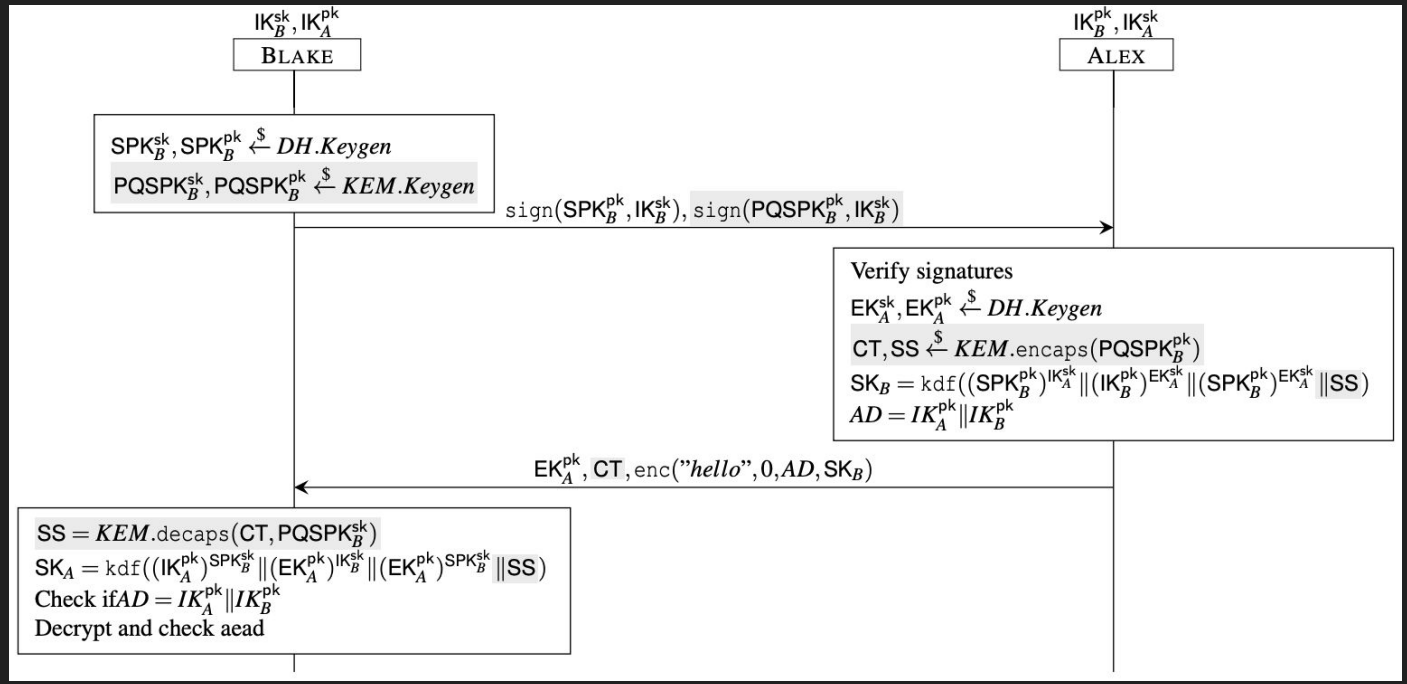
iMessage PQ3 hashes in everything

1

Signal



Signal: PQXDH v1 (simplified) [1] [2]



[1] https://signal.org/docs/specifications/pqxdh/
[2] https://www.usenix.org/system/files/usenixsecurity24-bhargavan.pdf

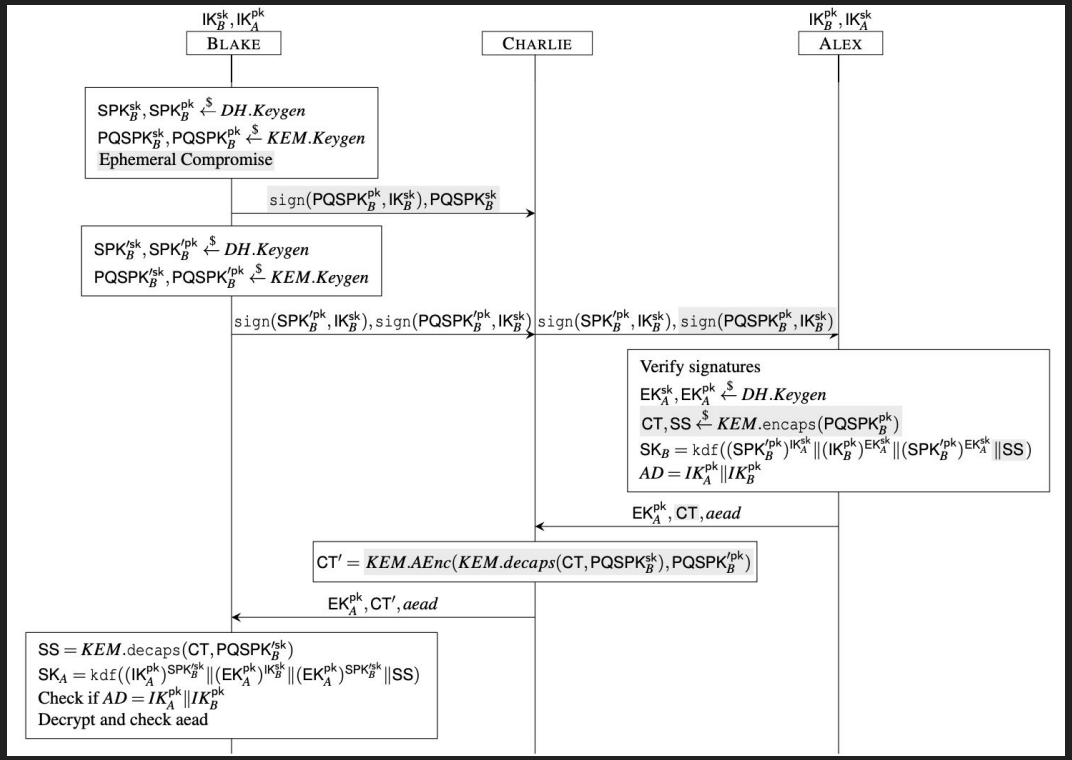
Signal PQXDH (v1) hashes in…just the shared secrets

Signal PQXDH (v1) hashes in…just the shared secrets

(And includes elliptic curve public ID keys in the AEAD additional data)

Re-encapsulation Attacks

Re-encapsulation attack in Signal PQXDH v1



https://www.usenix.org/system/files/usenixsecurity24-bhargavan.pdf

Re-encapsulation attack[1] 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 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/

Fix: Signal PQXDH v2[1], v3[2]

Alice then calculates an "associated data" byte sequence AD that contains identity information for both parties:

$$\mathrm{AD} = \mathrm{EncodeEC}(\mathrm{IK_A}) \,||\, \mathrm{EncodeEC}(\mathrm{IK_B})$$

~~Alice~~ If *pqkem* does not incorporate $PQPK_B$ into the ciphertext, Alice must also append *EncodeKEM*$(PQPK_B)$ to AD (see the discussion in Section 4.12). Alice may optionally append additional information to AD, such as Alice and Bob's usernames, certificates, or other identifying information.

[1] https://github.com/Inria-Prosecco/pqxdh-analysis/blob/main/revision2/pqxdh-diff-rev-1-to-2.pdf
[2] https://signal.org/docs/specifications/pqxdh/

IND-CCA is not sufficient

IND-CCA is not sufficient

Not every protocol can afford to hash everything in, always

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

{HON, LEAK}-BIND-P-Q Security Game

$X\text{-}BIND\text{-}P\text{-}Q^{\mathsf{KEM}}_{\mathcal{A}}$:

$\mathsf{sk}_0, \mathsf{pk}_0 \leftarrow \mathsf{KeyGen}()$

$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{KeyGen}()$

if $\mathsf{pk} \in Q$: $b \leftarrow 1$

else if $\mathsf{pk} \in P$: $b \leftarrow 0$

else : $b \in \{0,1\}, st \leftarrow \mathcal{A}()$

$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{sk}_b, \mathsf{pk}_b$

if $X = HON$: $\mathsf{ct}_0, \mathsf{ct}_1 \leftarrow \mathcal{A}^{D_{b'}(\mathsf{sk}_{b'}, \cdot)}(\mathsf{pk}_0, \mathsf{pk}_1, st)$

if $X = LEAK$: $\mathsf{ct}_0, \mathsf{ct}_1 \leftarrow \mathcal{A}(\mathsf{pk}_0, \mathsf{sk}_0, \mathsf{pk}_1, \mathsf{sk}_1, st)$

$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct}_0)$

$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct}_1)$

if $k_0 = \perp \vee k_1 = \perp$: **return** 0

// $\mathcal{A}$ wins if $\neg\big((\forall x \in P \,.\, x_0 = x_1) \implies (\forall y \in Q \,.\, y_0 = y_1)\big)$

return $\forall x \in P \,.\, x_0 = x_1 \wedge \exists y \in Q \,.\, y_0 \neq y_1$

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

X-BIND-K-CT, X-BIND-K-PK: collision resistance

- Against an adversary X, the Pr. of X finding two ciphertexts (or PKs) that result in the same shared secret K from Decaps() is negligible
- K ‘binds’ the ciphertext CT, or ‘binds’ the encapsulation key PK
- Can hold even if the IND-CCA security of the KEM falls apart
- The adversary can be HONest (no access to KEM decaps keys), LEAK (have access to decaps keys), or MALicious (have access to honestly-generated decaps keys and can manipulate them)
- The LEAK adversary model seems the most salient to existing protocols (see Signal PQXDH)
- A LEAK-BIND-K-PK KEM in PQXDH v1 protects against that attack

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

Kyber in PQXDH v1 protects against that attack

- Signal PQXDH v1 only specified an IND-CCA KEM, but used Kyber v3 in its prototype
- Bhargavan et al. showed that Kyber's 'semi-honest collision resistance' security (SH-CR) protected against the re-encapsulation attack

SH-CR Advantage

Definition 1 (SH-CR). *We define the advantage of an adversary $\mathcal{A}$ against a KEM* ($\mathtt{keygen}, \mathtt{encaps}, \mathtt{decaps}$) *semi-honest collision resistance as:*

$$\mathtt{Adv}^{SH\text{-}CR}_{\mathcal{A},KEM} = \mathtt{Pr}\left[\begin{array}{l|l} decaps(ct', \mathsf{sk}) = ss \wedge & \mathsf{sk}, \mathsf{pk} \xleftarrow{\$} keygen() \\ (ct \neq ct' \vee \mathsf{pk} \neq \mathsf{pk}') & \mathsf{pk}' \xleftarrow{\$} \mathcal{A}(\mathsf{sk}) \\ & ss, ct \xleftarrow{\$} encaps(\mathsf{pk}') \\ & ct' \xleftarrow{\$} \mathcal{A}(\mathsf{sk}, ss, ct) \end{array} \right]$$

https://www.usenix.org/system/files/usenixsecurity24-bhargavan.pdf

Kyber in PQXDH v1 protects against that attack

- A SH-CR-secure KEM’s shared secret K in combination with the ciphertext ‘binds’ the public encapsulation key against an adversary with access to the honestly-generated decapsulation key and a possibly modified public encapsulation key
- MAL-BIND-K,CT-PK implies SH-CR
- A SH-CR KEM protects against key impersonation and preserves session independence
- But Kyber != ML-KEM - the CT is no longer directly hashed in! 🤔

Not just a PQ concern!

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[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[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[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[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

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

[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/archive/2024/039/1704824081.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
- If you use the standardized seed variant for the keys, you get MAL-BIND-K-CT, but the fix[4] to get MAL-BIND-K-PK was not adopted
- It seems to protect against an SH-CR adversary

[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/archive/2024/039/1704824081.pdf
[4] https://eprint.iacr.org/2024/523.pdf

Fix: Signal PQXDH v2[1], v3[2]

Alice then calculates an "associated data" byte sequence AD that contains identity information for both parties:

$$\mathrm{AD} = \mathrm{EncodeEC}(\mathrm{IK_A}) \,||\, \mathrm{EncodeEC}(\mathrm{IK_B})$$

~~Alice~~ If *pqkem* does not incorporate $PQPK_B$ into the ciphertext, Alice must also append *EncodeKEM*$(PQPK_B)$ to AD (see the discussion in Section 4.12). Alice may optionally append additional information to AD, such as Alice and Bob's usernames, certificates, or other identifying information.

[1] https://github.com/Inria-Prosecco/pqxdh-analysis/blob/main/revision2/pqxdh-diff-rev-1-to-2.pdf
[2] https://signal.org/docs/specifications/pqxdh/

Takeaways from Signal PQXDH v1 attempt

6.1 Designing Hybrid schemes

The core issues we identify that one should consider when updating a protocol to a post-quantum variant are:

- The added KEM public key and ciphertext should be mutually authenticated, otherwise IND-CCA is probably not enough to prove the protocol secure, and the protocol may in fact be insecure for some specific KEM instantiations.
- Domain separation within the protocol between DH public keys and KEM public keys is crucial, as confusion may lead to a downgrade of security.
- Domain separation between distinct protocols is important. This is a typical case where both the old classical version of the protocol and it's post-quantum version will exist in parallel, so derived keys and other materials should not be confused between the two.

https://www.usenix.org/system/files/usenixsecurity24-bhargavan.pdf

Takeaways from Signal PQXDH v1 hybrid attempt

6.1 Designing Hybrid schemes

The core issues we identify that one should consider when updating a protocol to a post-quantum variant are:

- The added KEM public key and ciphertext should be mutually authenticated, otherwise IND-CCA is probably not enough to prove the protocol secure, and the protocol may in fact be insecure for some specific KEM instantiations.
- Domain separation within the protocol between DH public keys and KEM public keys is crucial, as confusion may lead to a downgrade of security.
- Domain separation between distinct protocols is important. This is a typical case where both the old classical version of the protocol and it's post-quantum version will exist in parallel, so derived keys and other materials should not be confused between the two.

https://www.usenix.org/system/files/usenixsecurity24-bhargavan.pdf

Takeaways from Signal PQXDH v1 hybrid attempt

Theorem 5 (Kyber is SH-CR). *If the hash functions used in the Kyber design are modeled as Random Oracles, Kyber is SH-CR.*

For the reasons mentioned previously in the practical implications of the re-encapsulation attack, it is clear that McEliece does not meet SH-CR, and the situation is unclear for BIKE and HQC.

https://www.usenix.org/system/files/usenixsecurity24-bhargavan.pdf

This is a lot of work to go hybrid!

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]

5.6 Decapsulation

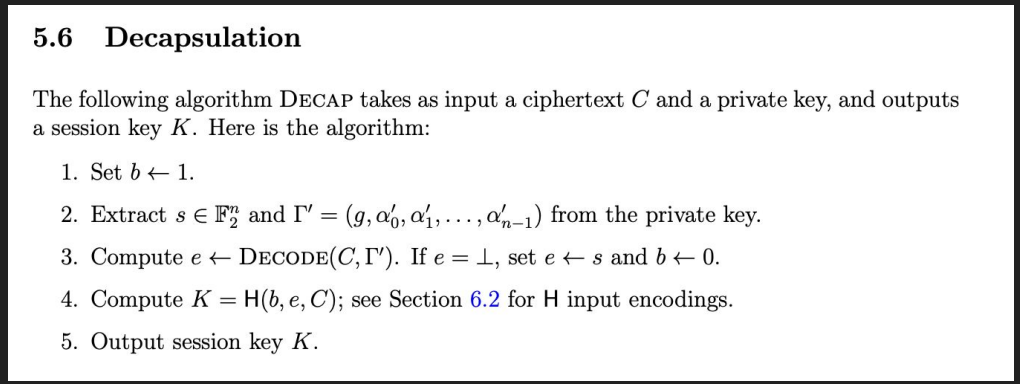
The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

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[2]

[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

Hybrid Protocols
Hybrid Primitives

Hybrid Protocols
Hybrid Primitives

Design a hybrid KEM

Design a hybrid KEM

- Slots in nicely where an IND-CCA KEM is already expected
- Hybrid Public-Key Encryption (HPKE)
- HPKE is a dependency of Encrypted Client Hello (ECH) in TLS 1.3
- Also a key dependency of Messaging Layer Security (MLS)[1]

- Pros: nicely abstracted away
- Cons: might be losing some efficiency/doing extra work vs a custom version of the protocol (eg TLS 1.3)

X-Wing[1][2]

Algorithm KeyGen()
$sk_1, pk_1 \leftarrow\$ \text{ML-KEM-768.KeyGen}()$
$sk_2 \leftarrow\$ \text{random}(32)$
$pk_2 \leftarrow \text{X25519.DH}(sk_2, g_{\text{X25519}})$
$sk \leftarrow (sk_1, sk_2, pk_2)$
$pk \leftarrow (pk_1, pk_2)$
return (sk, pk)

Algorithm Enc(pk)
$(pk_1, pk_2) \leftarrow pk$
$sk_e \leftarrow\$ \text{random}(32)$
$c_2 \leftarrow \text{X25519.DH}(sk_e, g_{\text{X25519}})$
$k_1, c_1 \leftarrow\$ \text{ML-KEM-768.Enc}(pk)$
$k_2 \leftarrow \text{X25519.DH}(sk_e, pk_2)$
$s \leftarrow \texttt{"\./\/^\"} \| k_1 \| k_2 \| c_2 \| pk_2$
$k \leftarrow \text{SHA3-256}(s)$
$c \leftarrow (c_1, c_2)$
return (k, c)

Algorithm Dec(c, sk)
$(sk_1, sk_2, pk_2) \leftarrow sk$
$(c_1, c_2) \leftarrow c$
$k_1 \leftarrow \text{ML-KEM-768.Dec}(c_1, sk_1)$
$k_2 \leftarrow \text{X25519.DH}(sk_2, c_2)$
$s \leftarrow \texttt{"\./\/^\"} \| k_1 \| k_2 \| c_2 \| pk_2$
$k \leftarrow \text{SHA3-256}(s)$
return k

[1] https://eprint.iacr.org/2024/039.pdf
[2] https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/

X-Wing[1][2]

- Hybrid KEM based on ML-KEM-768, X25519, SHA-3, SHAKE256
- Defines notion of Ciphertext Second Preimage Resistance (C2PRI)
  - A weaker notion than LEAK-BIND-K-CT and Ciphertext Collision Resistance (CCR)
- Proves IND-CCA in the standard of generic construction when the KEM is C2PRI and the KDF is a PRF, and proves IND-CCA in the ROM when strong DH assumption holds in the nominal group and the KDF is an RO
- Shows that ML-KEM is C2PRI, and X25519 is a nominal group
- C2PRI security of ML-KEM allows the large public encaps key and ciphertext to be left out of the KDF input

[1] https://eprint.iacr.org/2024/039.pdf
[2] https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/

IRTF CFRG Hybrid KEMs[1] [2]

- Trying to collect generic frameworks for constructing hybrid KEMs
- Current draft includes
  - GHP: hash everything in, easiest requirements
  - QSF: X-Wing style, requires C2PRI KEM and elliptic curve group
  - PRE: GHP but with pre-hashed encaps keys, cacheable
- Three concrete instantiations so far, including X-Wing, all QSF
- Ongoing research[3] indicates that C2PRI may be a common property of popular FO-transform KEMs, making QSF quite applicable

[1] https://datatracker.ietf.org/doc/draft-irtf-cfrg-hybrid-kems/
[2] https://datatracker.ietf.org/doc/draft-irtf-cfrg-concrete-hybrid-kems/
[3] Under review

NIST SP 800-227 IPD[1] Composite KEM

- ~Matches the GHP construction in CFRG draft
- Conservative
- Does not apply to US National Security Systems (they are going pure ML-KEM-1024 except for IKEv2)

[1] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-227.ipd.pdf

And more

- NIST SP 800-227 IPD Composite KEM
  - Matches the GHP construction in CFRG draft
  - https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-227.ipd.pdf

- IETF LAMPS Composite KEMs
  - Many combinations of KEMs/components
  - Has evolved many times
  - Seems to also be consolidating around ML-KEM only and the QSF framework

Hybrid KEM Adoption

- X-Wing
  - has been adopted by Apple CryptoKit for HPKE
  - implemented in Google’s BoringSSL for Encrypted Client Hello in TLS 1.3
  - MLS
- CFRG Hybrid KEMs
  - MLS
  - General HPKE use
  - Open Compute HSMs
  -

Hash in EVERYTHING.

If you can go PQ-only, do. It’s cleaner.

If you must go hybrid, hash in everything.

Be like TLS 1.3: hash in everything.

The future is going to be hybrid

The future is going to be hybrid, in more ways than one

The future is going to be hybrid, in more ways than one, at least for a while.

The future is going to be hybrid, in more ways than one, at least for a while.

(Talk to your local cryptographer about going full-PQ!)

Questions?

Hybrid Key Establishment in Production

Deirdre Connolly | July 15, 2025

⭐ Bonus slides ⭐

And more protocols

- SSH hybrid key establishment: https://datatracker.ietf.org/doc/draft-ietf-sshm-mlkem-hybrid-kex/
- IPSEC/IKEv2: https://datatracker.ietf.org/doc/draft-ietf-ipsecme-ikev2-mlkem/
- Alternative option for MLS, maintain dual PQ and T sessions, support hybrid-PQ key updates at configurable intervals: https://datatracker.ietf.org/doc/draft-ietf-mls-combiner/
- Signal's approach to better PQ security in ratcheting, amortizing large ephemeral KEMs over multiple messages: https://eprint.iacr.org/2025/078

And more primitives

- Hybrid KEM certificates (LAMPS)
- KEM TLS?
- Multi-recipient KEMs
- PAKEs!
- Obfuscated KEMs, hybrid and not