Going Post-Quantum

Deirdre Connolly - SandboxAQ

Key Agreement

Signatures

‘Fancy Crypto’

Key Agreement

Signatures

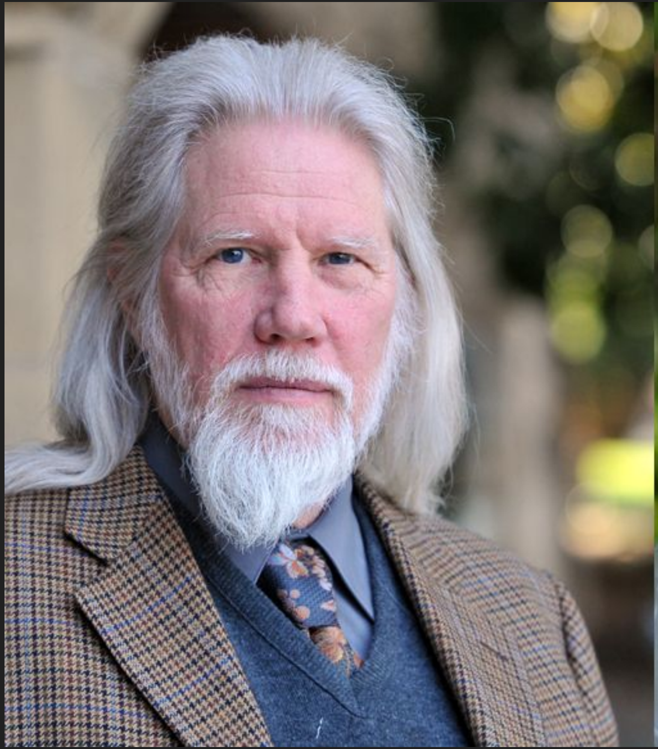
‘Fancy Crypto’

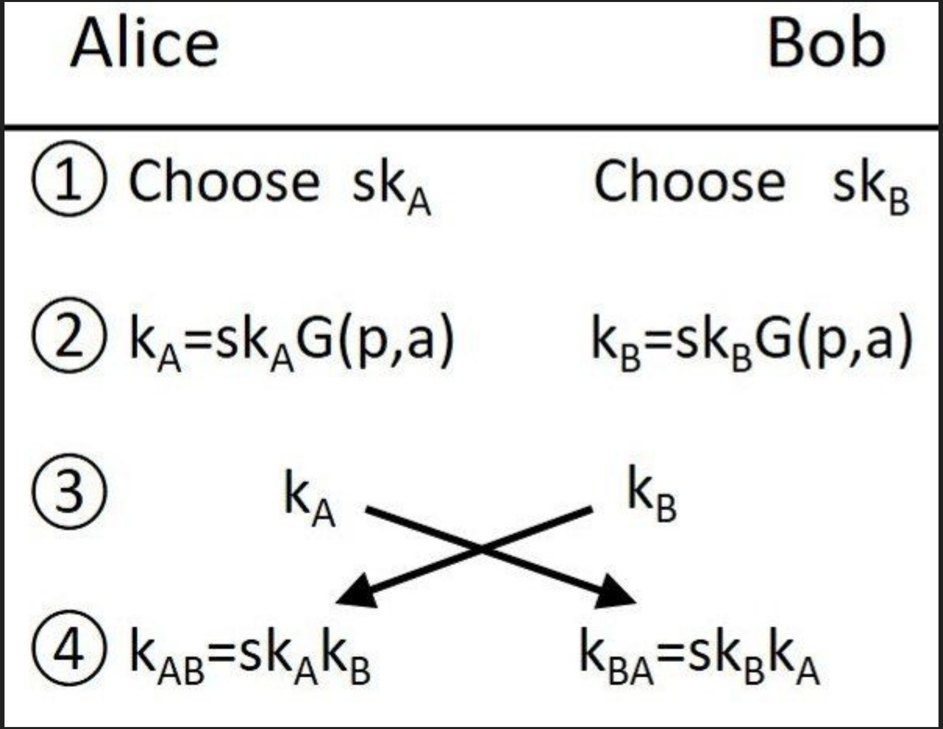
Key Agreement

Signatures

‘Fancy Crypto’

RIP Diffie-Hellman

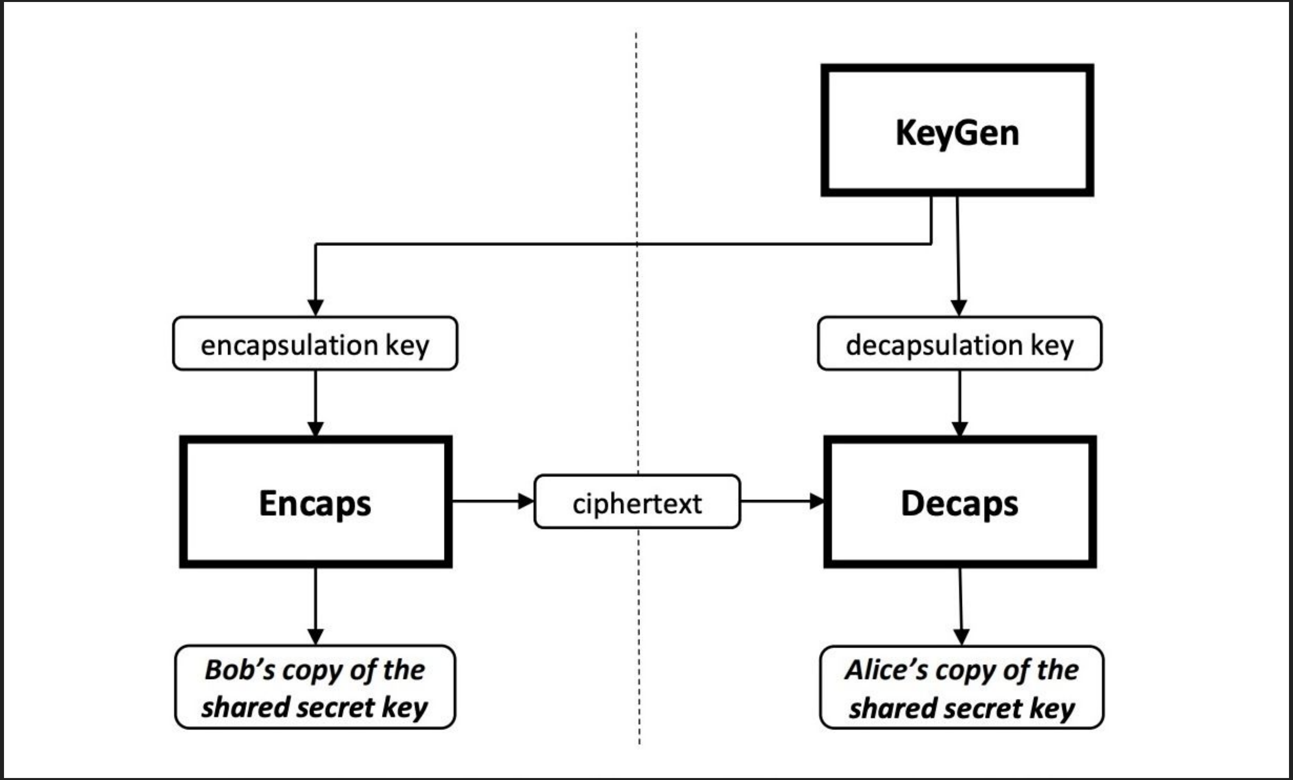


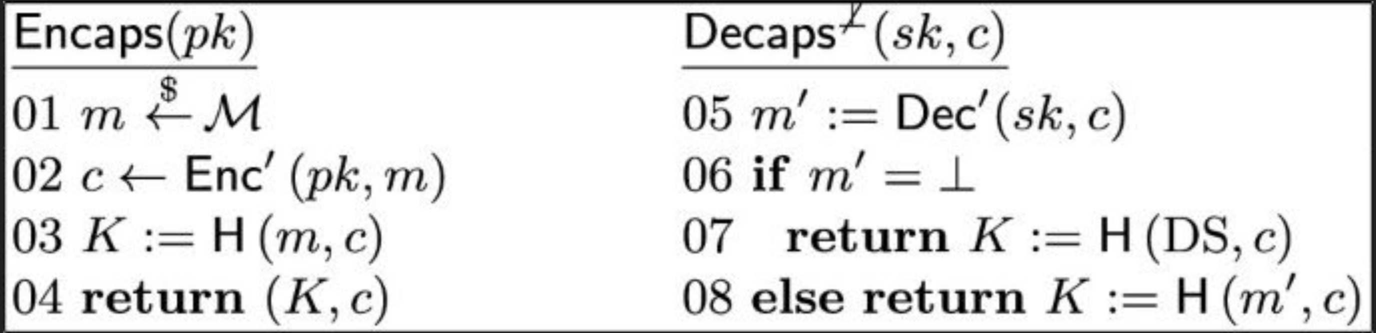

Alice
Bob
① Choose sk_A
Choose sk_B
② $k_A=sk_A G(p,a)$
$k_B=sk_B G(p,a)$
③
k_A
k_B
④ $k_{AB}=sk_A k_B$
$k_{BA}=sk_B k_A$

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



```
Encaps($pk$)
01 $m \xleftarrow{\$} \mathcal{M}$
02 $c \leftarrow \mathsf{Enc}'(pk, m)$
03 $K := \mathsf{H}(m, c)$
04 **return** $(K, c)$
```

```
Decaps$^{\not\perp}(sk, c)$
05 $m' := \mathsf{Dec}'(sk, c)$
06 **if** $m' = \perp$
07     **return** $K := \mathsf{H}(\mathrm{DS}, c)$
08 **else return** $K := \mathsf{H}(m', c)$
```

Fujisaki-Okamoto (FO) transform

Fujisaki-Okamoto (FO) transform

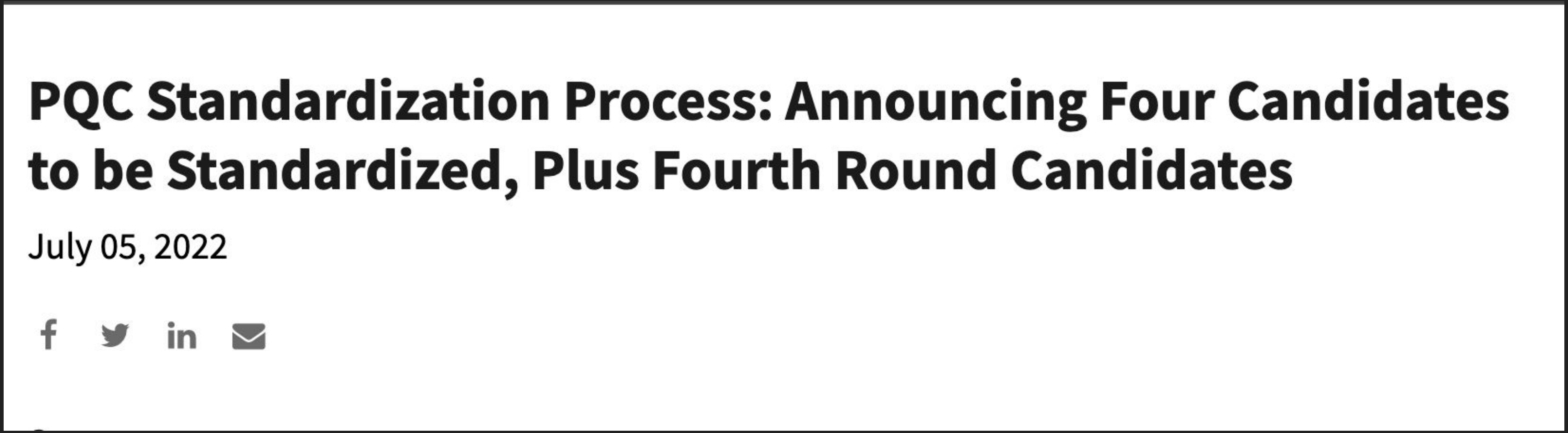
- Turns an IND-CPA-secure public key encryption scheme into a IND-CCA key encapsulation scheme

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

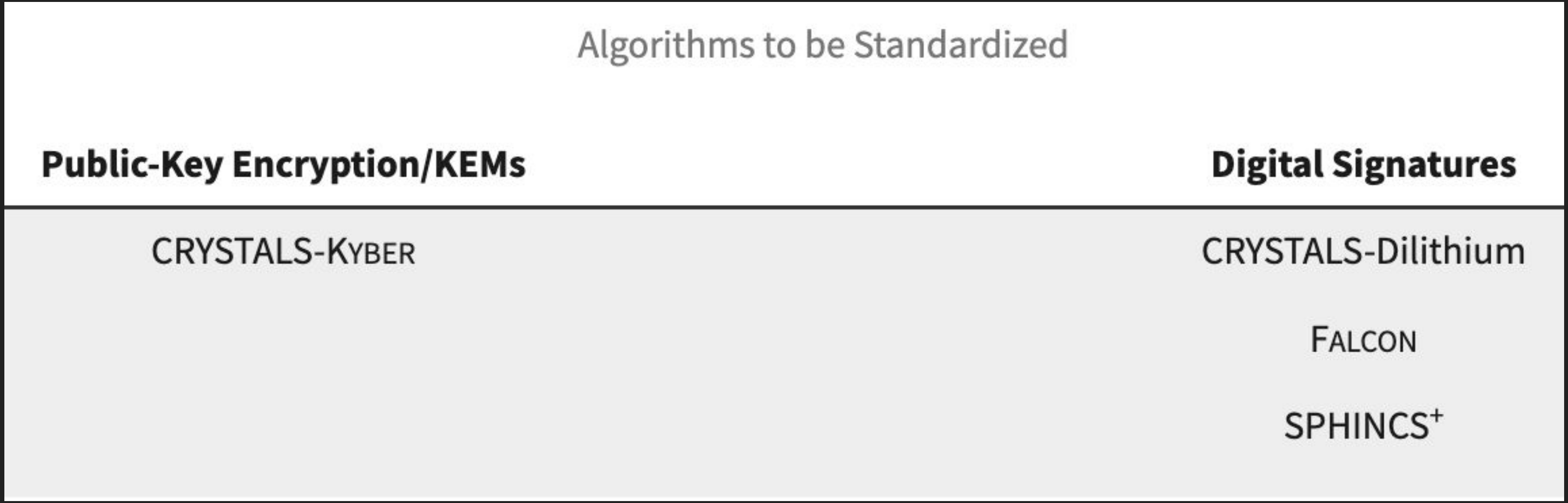
- Turns an IND-CPA-secure public key encryption scheme into a IND-CCA key encapsulation scheme
- Popular amongst all the NIST PQC KEM candidates
- Explicit and implicit rejection forms, the KEMs we care about are all implicit rejection (no error codes or panics, returns a non-zero pseudorandom value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



PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates

July 05, 2022

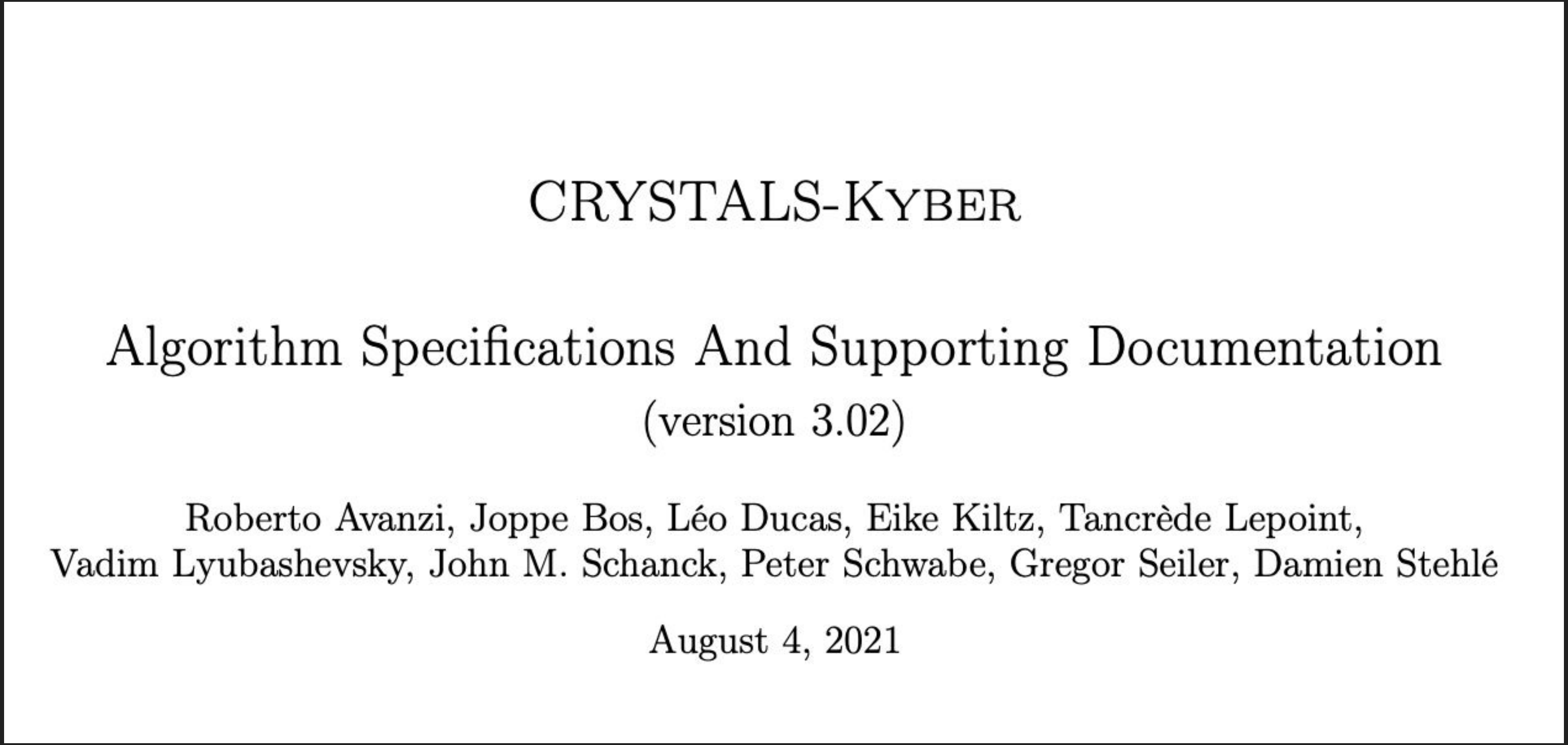
https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4


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$^+$



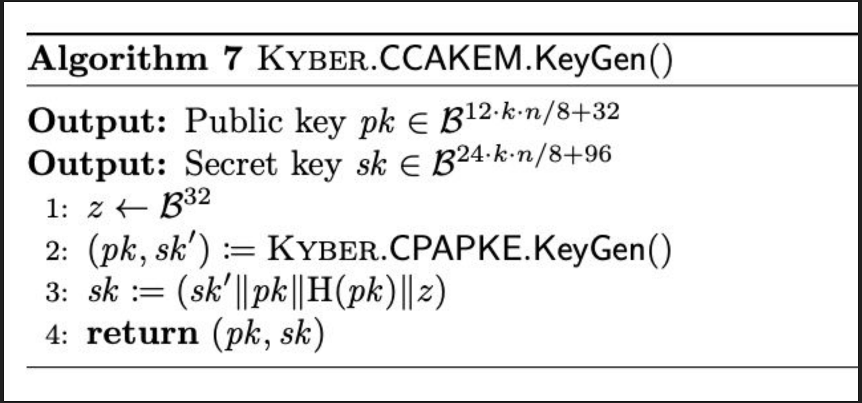
CRYSTALS-KYBER

Algorithm Specifications And Supporting Documentation

(version 3.02)

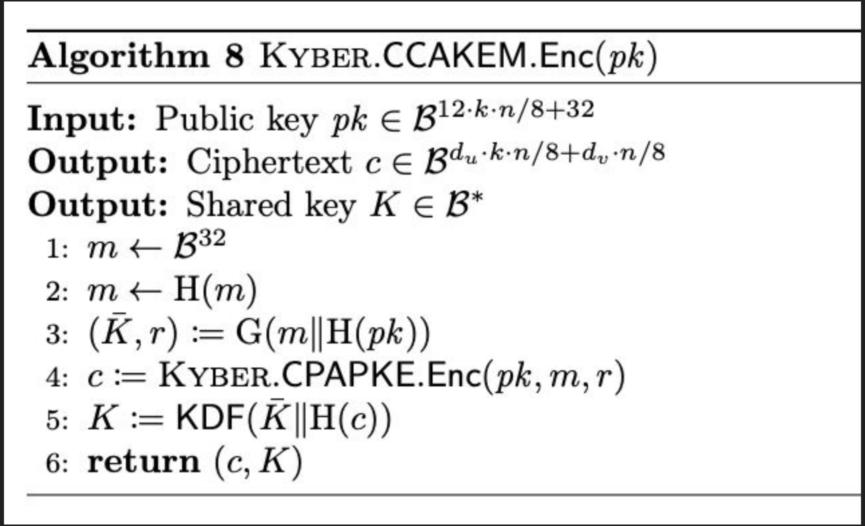
Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint,
Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé

August 4, 2021



Algorithm 7 KYBER.CCAKEM.KeyGen()

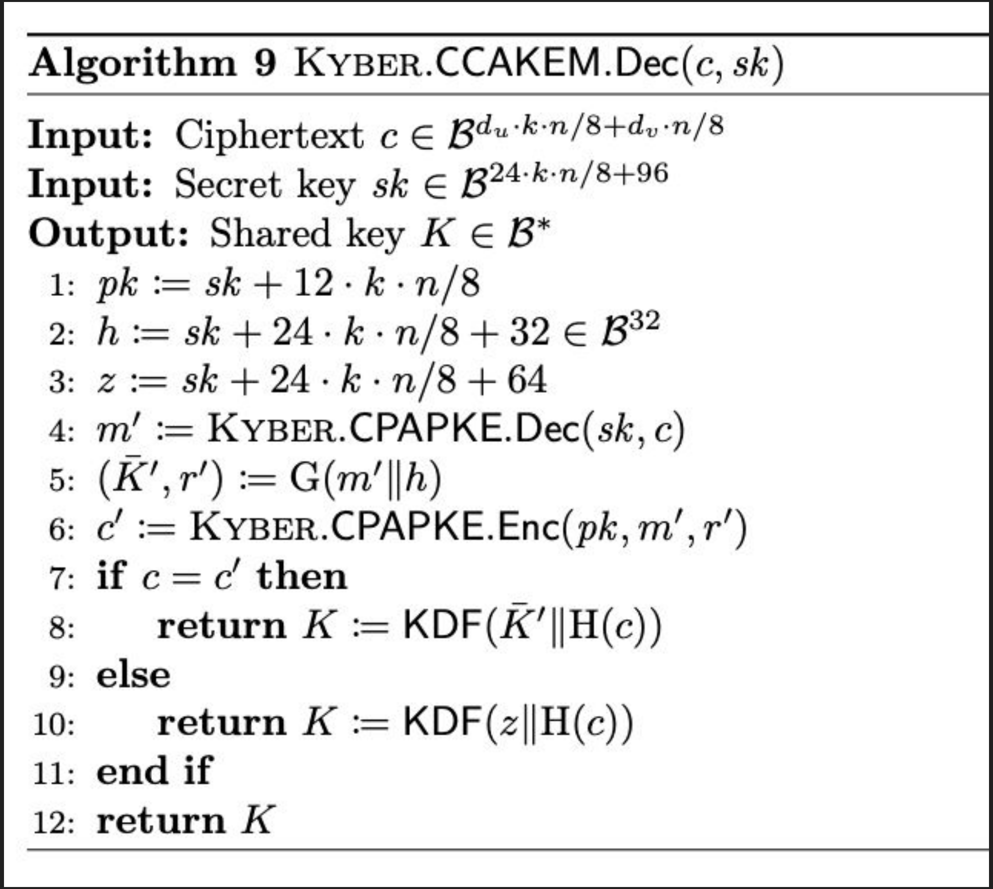
```
Output: Public key $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$
Output: Secret key $sk \in \mathcal{B}^{24\cdot k\cdot n/8+96}$
1: $z \leftarrow \mathcal{B}^{32}$
2: $(pk, sk') := \text{KYBER.CPAPKE.KeyGen}()$
3: $sk := (sk' \| pk \| \mathrm{H}(pk) \| z)$
4: return $(pk, sk)$
```



Algorithm 8 $\textsc{Kyber}.\mathsf{CCAKEM.Enc}(pk)$

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 $K \in \mathcal{B}^*$

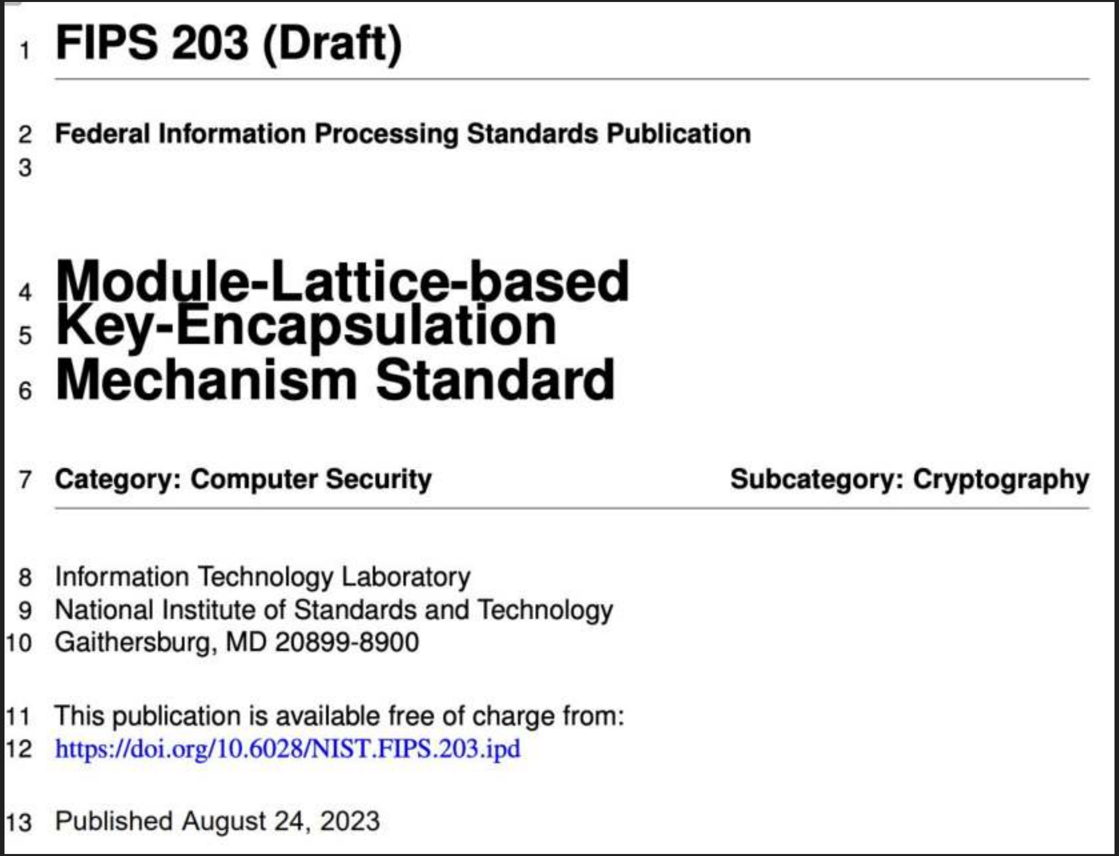
```
1: $m \leftarrow \mathcal{B}^{32}$
2: $m \leftarrow \mathrm{H}(m)$
3: $(\bar{K}, r) := \mathrm{G}(m \| \mathrm{H}(pk))$
4: $c := \textsc{Kyber}.\mathsf{CPAPKE.Enc}(pk, m, r)$
5: $K := \mathsf{KDF}(\bar{K} \| \mathrm{H}(c))$
6: **return** $(c, K)$
```



Algorithm 9 $\text{Kyber.CCAKEM.Dec}(c, sk)$

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 $K \in \mathcal{B}^*$

```
 1: $pk := sk + 12 \cdot k \cdot n/8$
 2: $h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
 3: $z := sk + 24 \cdot k \cdot n/8 + 64$
 4: $m' := \text{Kyber.CPAPKE.Dec}(sk, c)$
 5: $(\bar{K}', r') := \mathrm{G}(m' \| h)$
 6: $c' := \text{Kyber.CPAPKE.Enc}(pk, m', r')$
 7: if $c = c'$ then
 8:     return $K := \text{KDF}(\bar{K}' \| \mathrm{H}(c))$
 9: else
10:     return $K := \text{KDF}(z \| \mathrm{H}(c))$
11: end if
12: return $K$
```



FIPS 203 (Draft)

Federal Information Processing Standards Publication

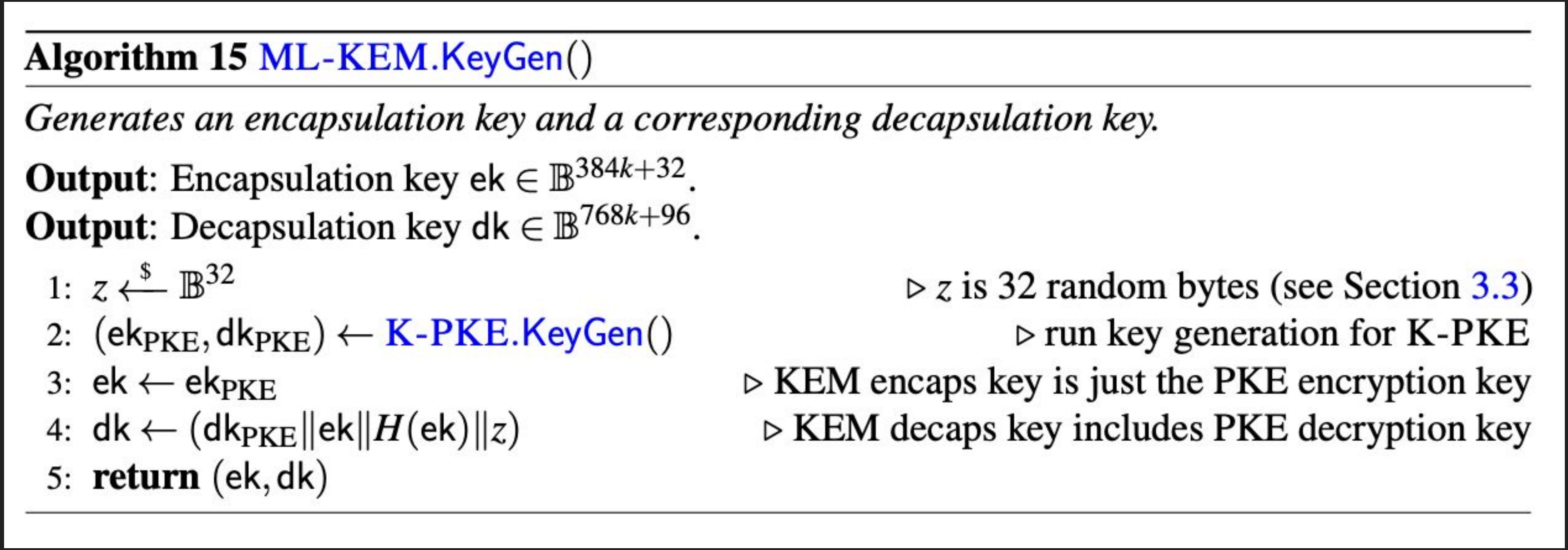
Module-Lattice-based Key-Encapsulation Mechanism Standard

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.ipd

Published August 24, 2023

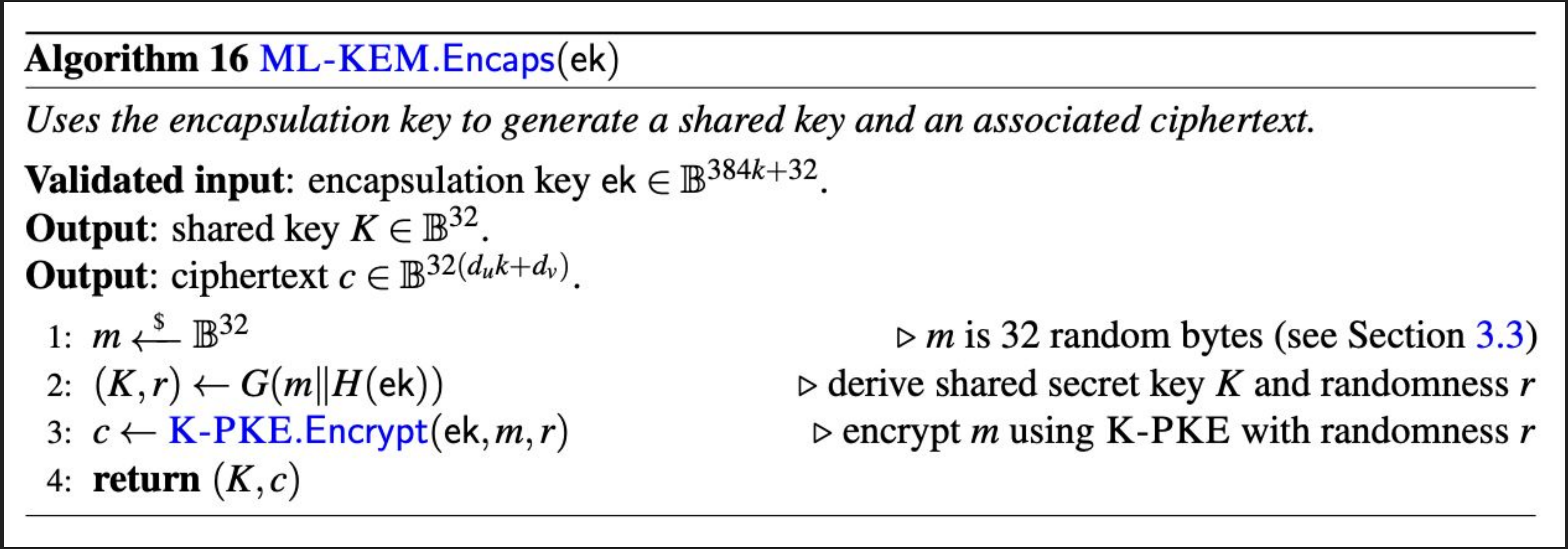


Algorithm 15 ML-KEM.KeyGen()

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** $(\mathsf{ek}, \mathsf{dk})$
```

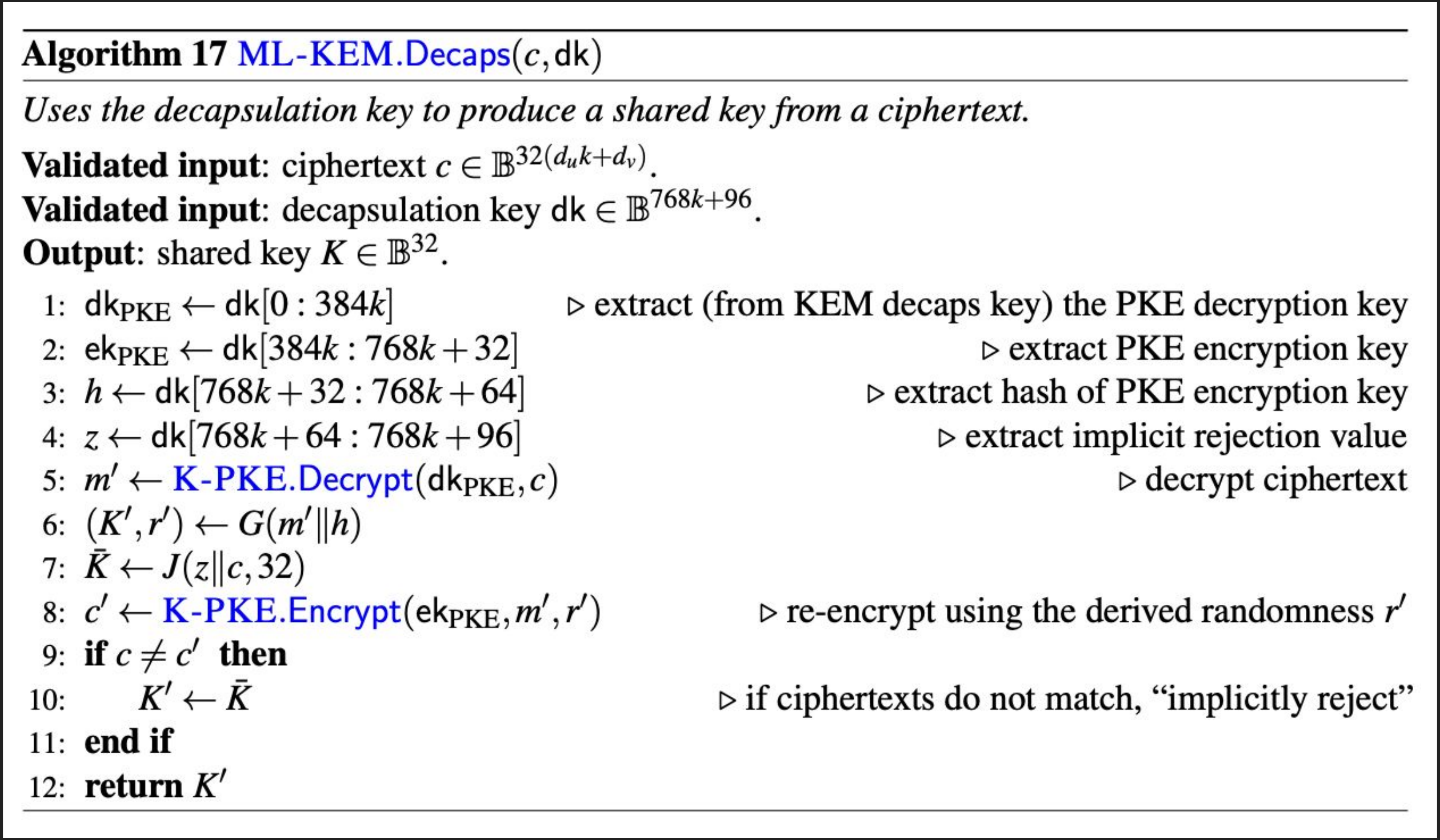


Algorithm 16 ML-KEM.Encaps(ek)

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$(\mathsf{ek}, m, r)$       ▷ encrypt $m$ using K-PKE with randomness $r$
4: return $(K, c)$
```

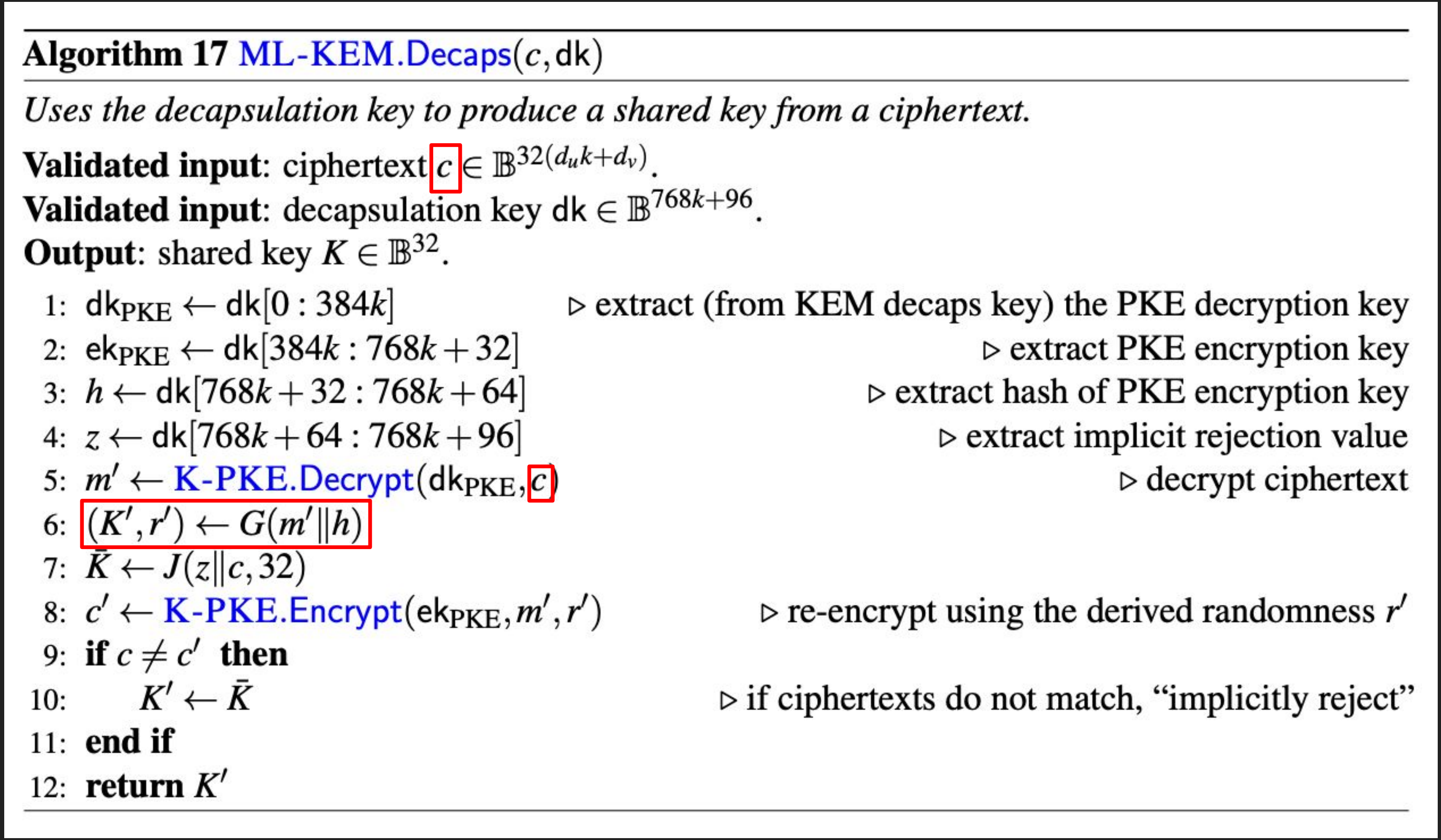


Algorithm 17 ML-KEM.Decaps(c, dk)

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

```
 1: $\mathsf{dk}_{\mathrm{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$        ▷ extract (from KEM decaps key) the PKE decryption key
 2: $\mathsf{ek}_{\mathrm{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$        ▷ 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$(\mathsf{dk}_{\mathrm{PKE}}, c)$        ▷ decrypt ciphertext
 6: $(K', r') \leftarrow G(m' \| h)$
 7: $\bar{K} \leftarrow J(z \| c, 32)$
 8: $c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\mathrm{PKE}}, m', r')$        ▷ 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 $K'$
```



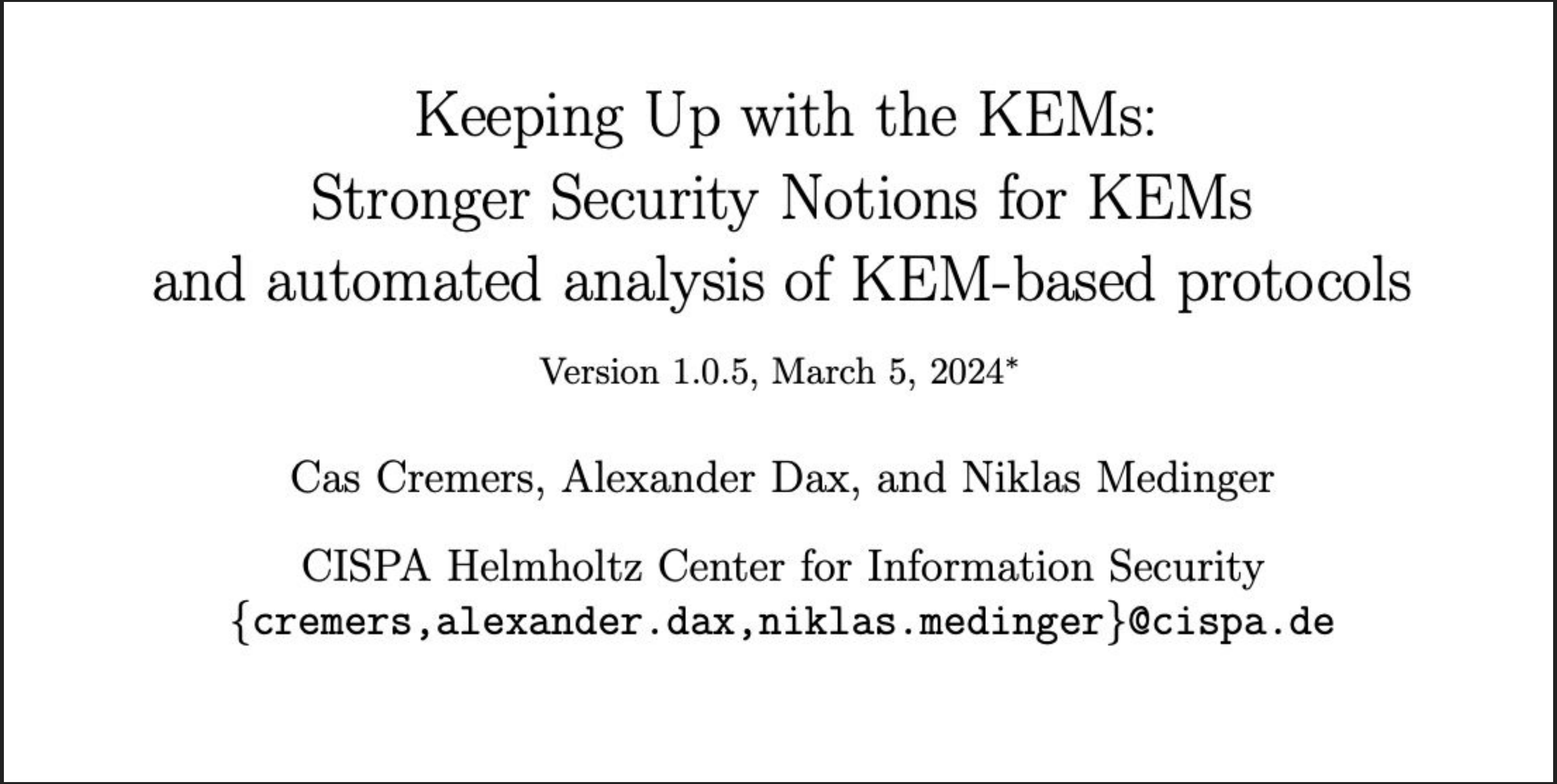
Algorithm 17 ML-KEM.Decaps(c, dk)

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

Validated input: ciphertext $\boxed{c} \in \mathbb{B}^{32(d_u k + d_v)}$.
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$(\mathsf{dk}_{\text{PKE}}, \boxed{c})$    ▷ decrypt ciphertext
 6: $\boxed{(K', r') \leftarrow G(m' \| h)}$
 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 $K'$
```

Re-encapsulation Attacks



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

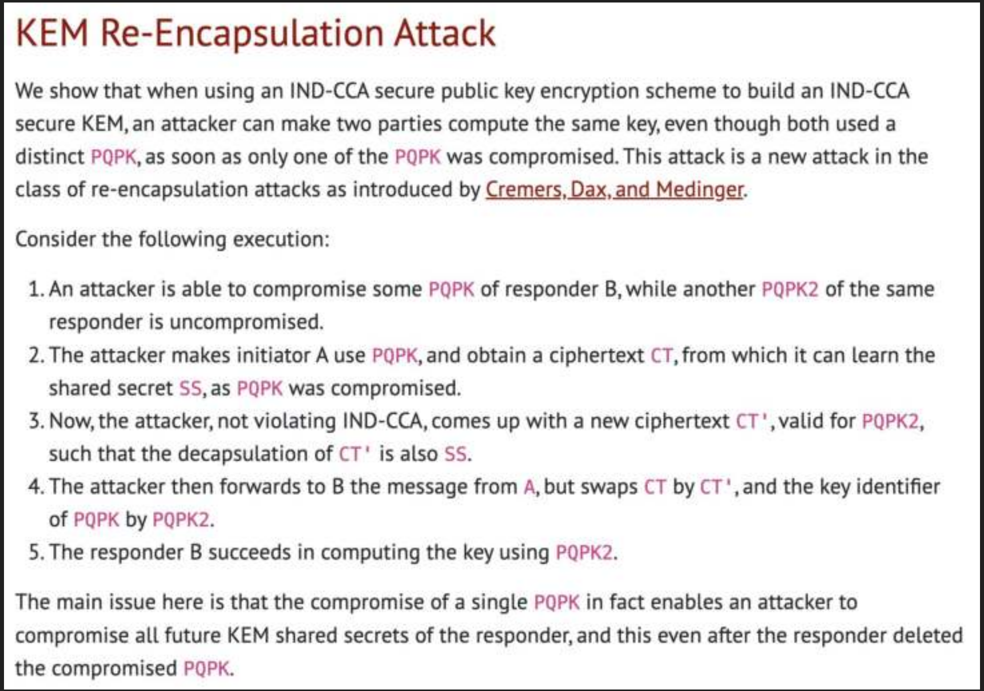
Version 1.0.5, March 5, 2024*

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.

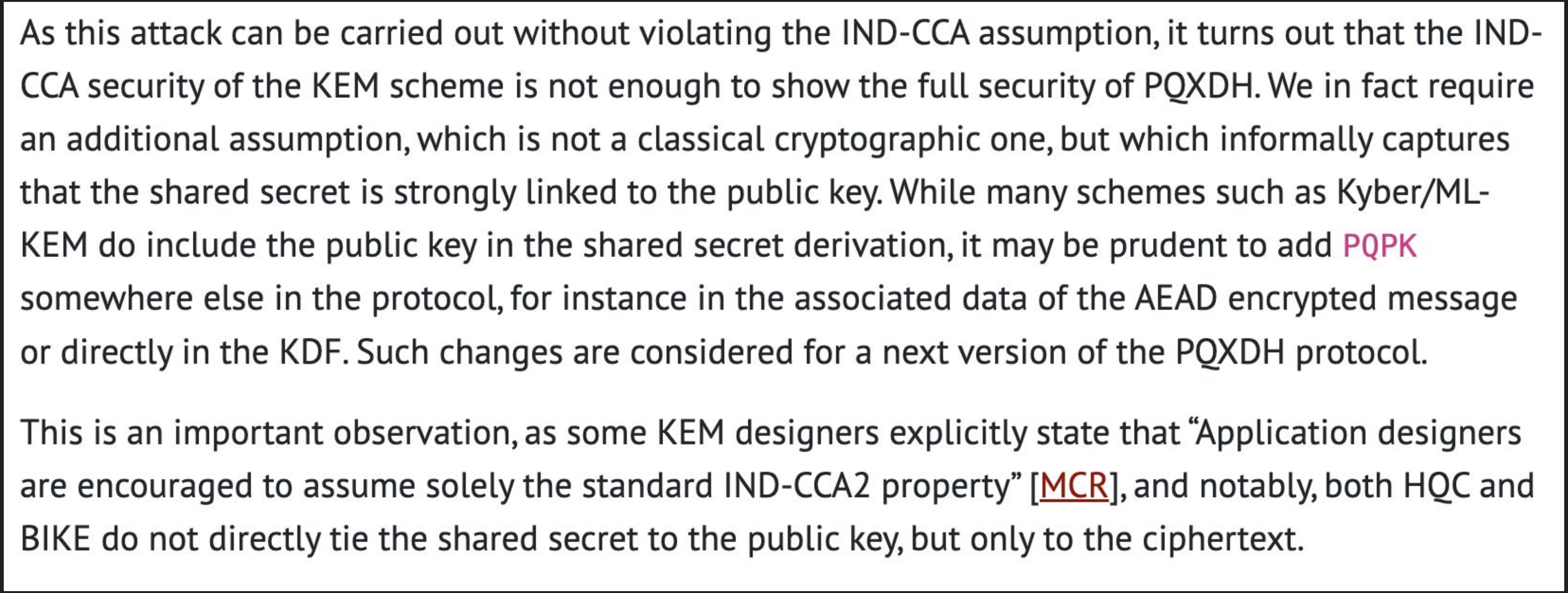
Consider the following execution:

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/

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

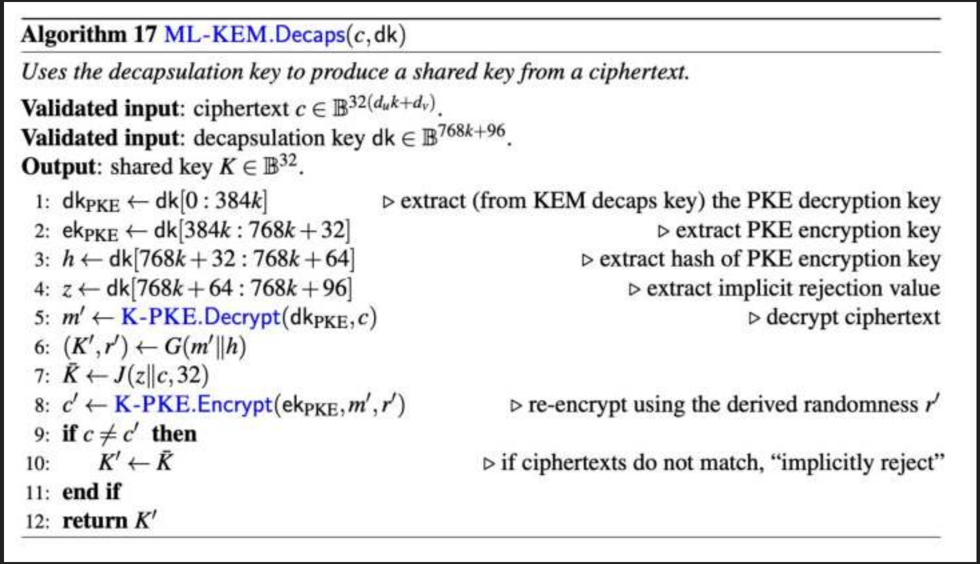
[1] https://www.rfc-editor.org/rfc/rfc9180.html#name-dh-based-kem-dhkem
[2] https://eprint.iacr.org/2023/1933.pdf

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)

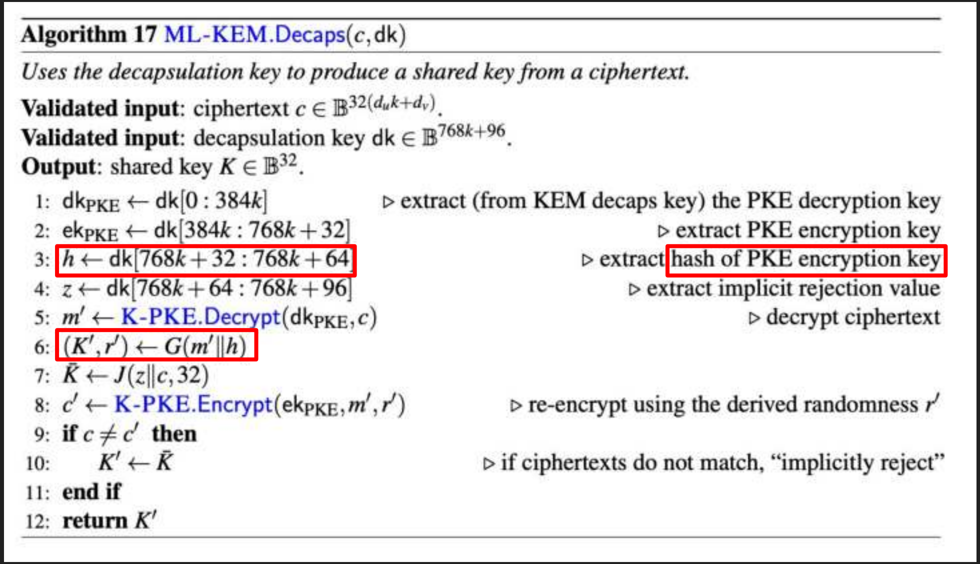
Uses the decapsulation key to produce a shared key from a ciphertext.

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:  $\mathsf{dk}_{\mathrm{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$    ▷ extract (from KEM decaps key) the PKE decryption key
2:  $\mathsf{ek}_{\mathrm{PKE}} \leftarrow \mathsf{dk}[384k : 768k+32]$    ▷ 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$(\mathsf{dk}_{\mathrm{PKE}}, c)$    ▷ decrypt ciphertext
6:  $(K', r') \leftarrow G(m' \| h)$
7:  $\bar{K} \leftarrow J(z \| c, 32)$
8:  $c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\mathrm{PKE}}, m', r')$    ▷ 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.

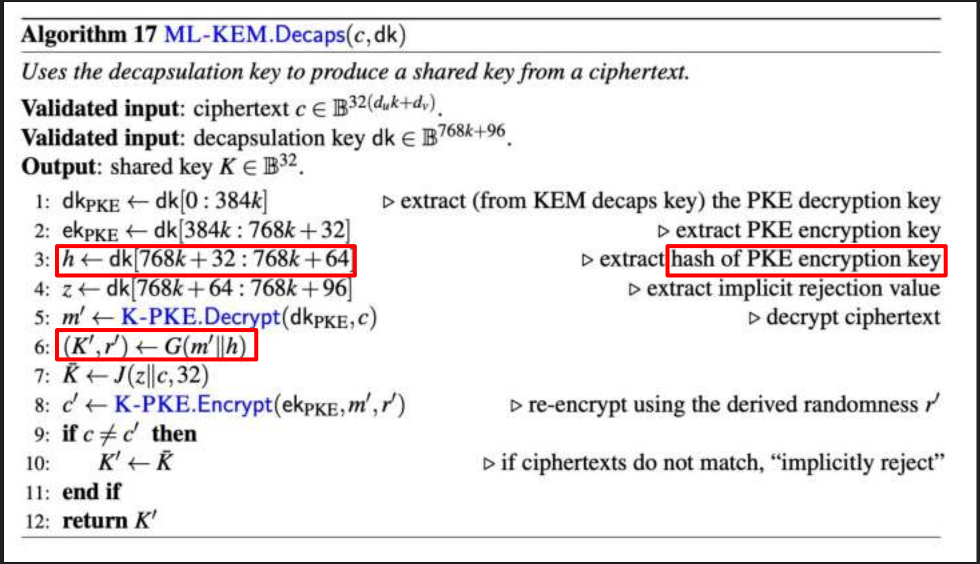
Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

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

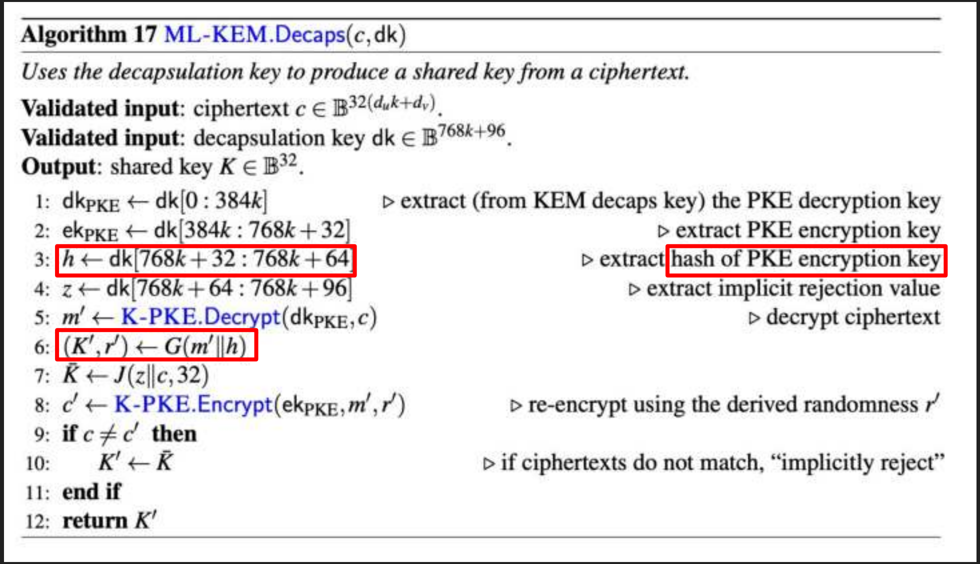
Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

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

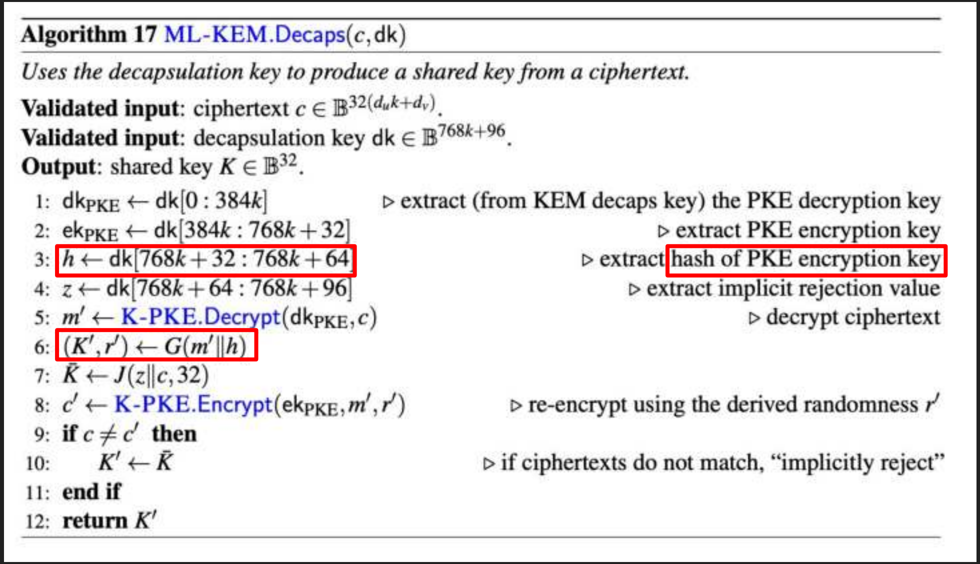
Validated input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Validated input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Output: shared key $K \in \mathbb{B}^{32}$.

```
1:  $\mathsf{dk_{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$                ▷ extract (from KEM decaps key) the PKE decryption key
2:  $\mathsf{ek_{PKE}} \leftarrow \mathsf{dk}[384k : 768k+32]$           ▷ 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($\mathsf{dk_{PKE}}, c$)                ▷ decrypt ciphertext
6:  $(K', r') \leftarrow G(m' \| h)$
7:  $\bar{K} \leftarrow J(z \| c, 32)$
8:  $c' \leftarrow$ K-PKE.Encrypt($\mathsf{ek_{PKE}}, m', r'$)           ▷ 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 $K \in \mathbb{B}^{32}$.

```
$\mathsf{dk}_{\mathrm{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$        ▷ extract (from KEM decaps key) the PKE decryption key
$\mathsf{ek}_{\mathrm{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$        ▷ extract PKE encryption key
$\boxed{h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]}$        ▷ extract $\boxed{\text{hash of PKE encryption key}}$
$z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$        ▷ extract implicit rejection value
$m' \leftarrow$ K-PKE.Decrypt$(\mathsf{dk}_{\mathrm{PKE}}, c)$        ▷ decrypt ciphertext
$\boxed{(K', r') \leftarrow G(m' \| h)}$
$\bar{K} \leftarrow J(z \| c, 32)$
$c' \leftarrow$ K-PKE.Encrypt$(\mathsf{ek}_{\mathrm{PKE}}, m', r')$        ▷ 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 $K'$
```

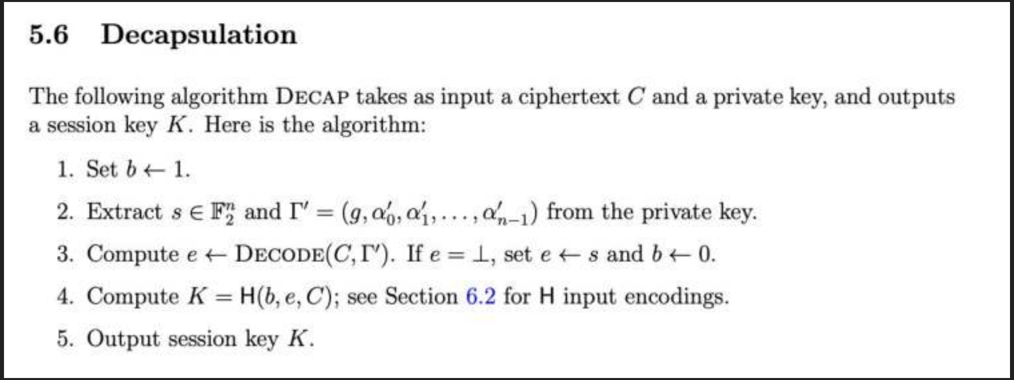
- ML-KEM's shared secret K binds ekPKE (PK) via hashing in the hash of ekPKE: MAL-BIND-K-PK
- Binding the ciphertext *c* relies on the robustness properties of K-PKE
- Shown to be *chosen ciphertext resistant*[3]: implies *LEAK*-BIND-K-CT
- LEAK is resistant to adversaries with access to leaked, honestly-generated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

A counterexample: Classic McEliece[1]

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

A counterexample: Classic McEliece[1]



5.6 Decapsulation

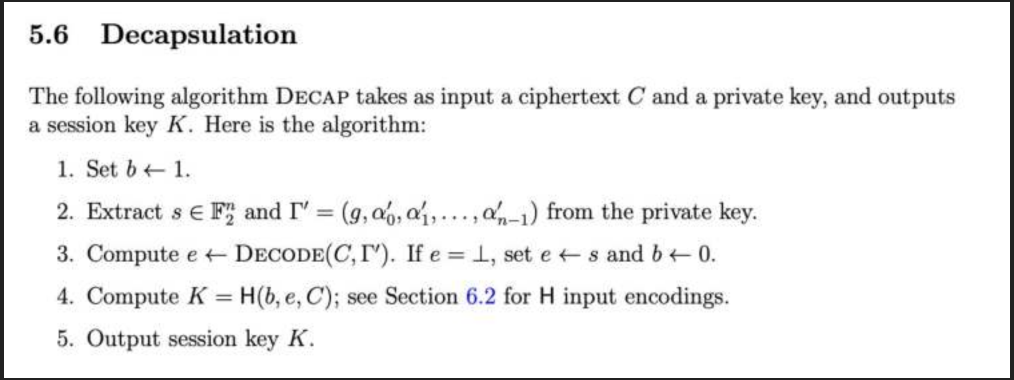
The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

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 ✅



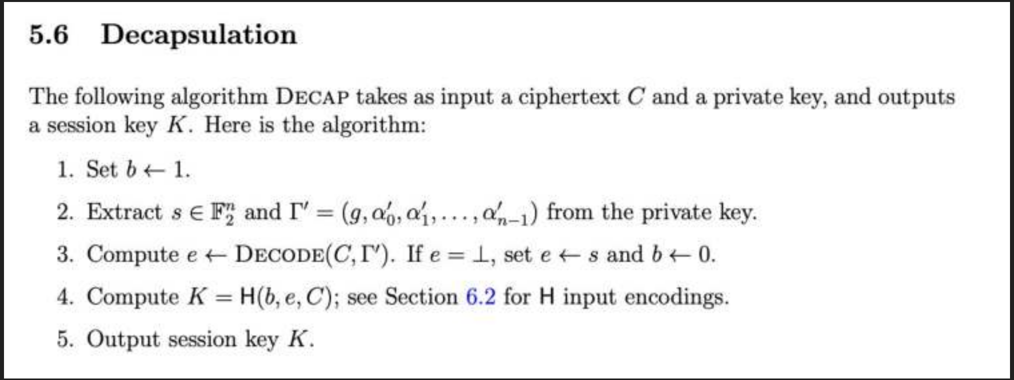
5.6 Decapsulation

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 $e \leftarrow \textsc{Decode}(C, \Gamma')$. 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

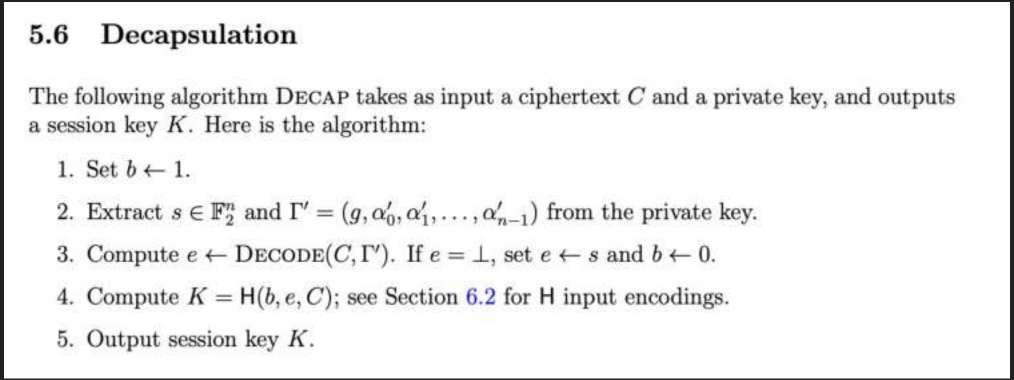
The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

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 $e \leftarrow \textsc{Decode}(C, \Gamma')$. 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

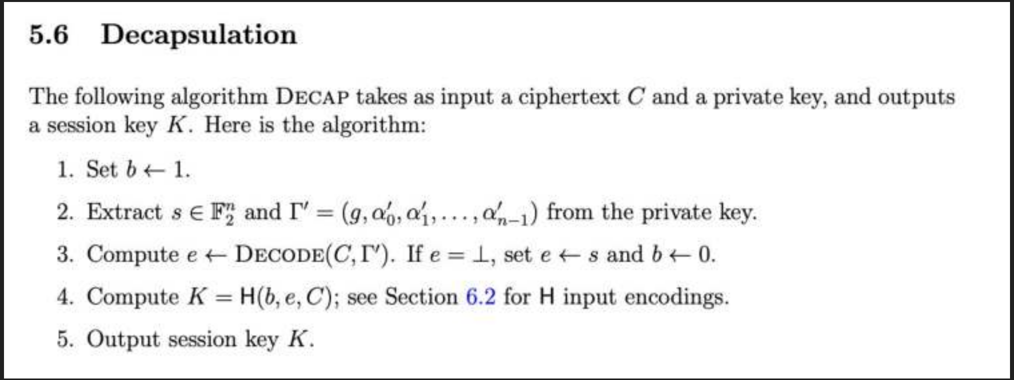
The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

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

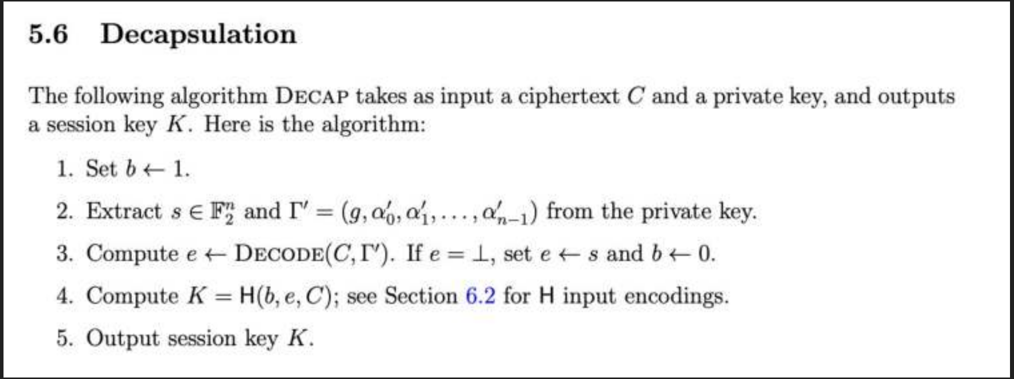
The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

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

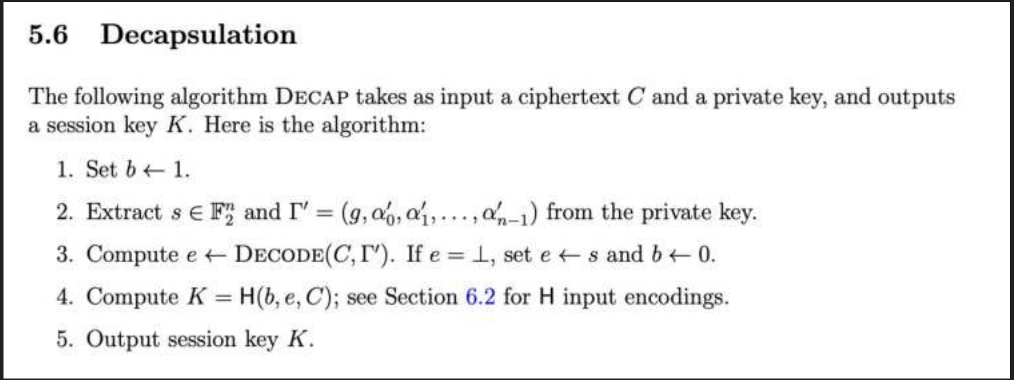
The following algorithm DECAP takes as input a ciphertext C and a private key, and outputs a session key K. Here is the algorithm:

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 $e \leftarrow \textsc{Decode}(C, \Gamma')$. 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

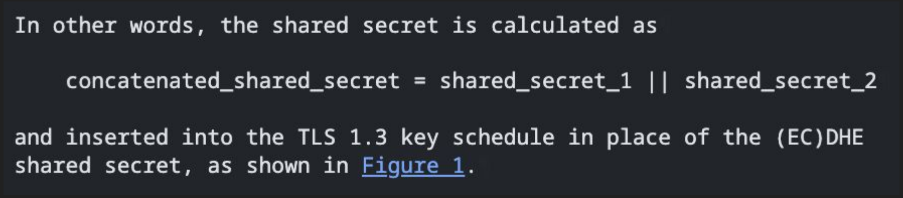
[1] https://classic.mceliece.org/mceliece-spec-20221023.pdf
[2] https://eprint.iacr.org/2023/1933.pdf
[3] https://eprint.iacr.org/2021/708.pdf

Hash in
EVERYTHING.

If you can go PQ-only, do.

If you must go hybrid, hash in everything.

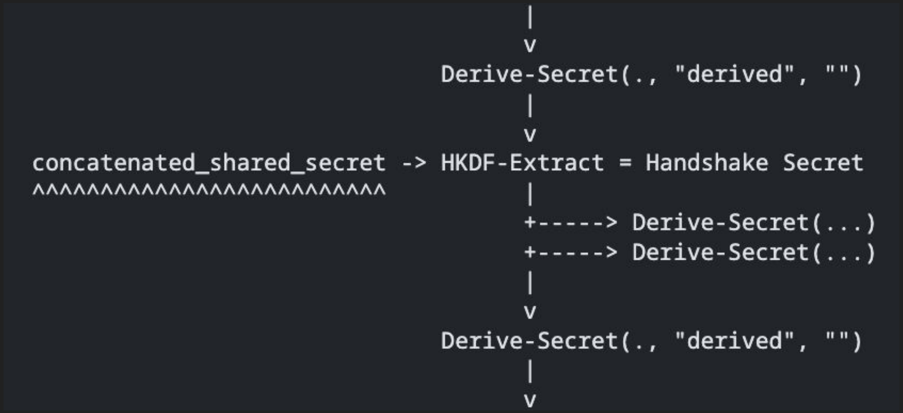
TLS 1.3 hybrid key agreement[1]



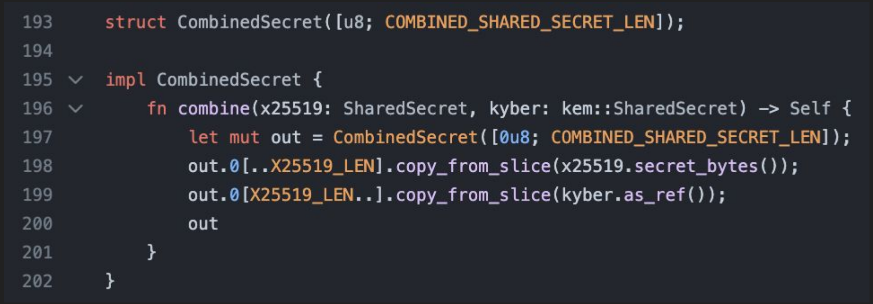
```
In other words, the shared secret is calculated as

   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

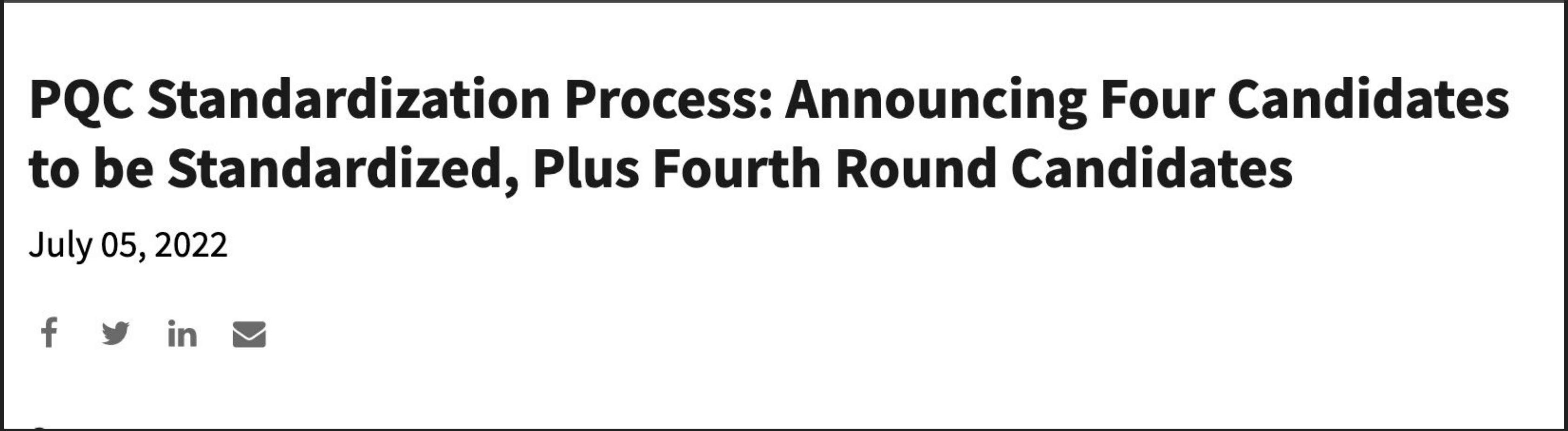
Be like TLS 1.3: hash in everything

Key Agreement
Signatures
‘Fancy Crypto’

Key Agreement

Signatures

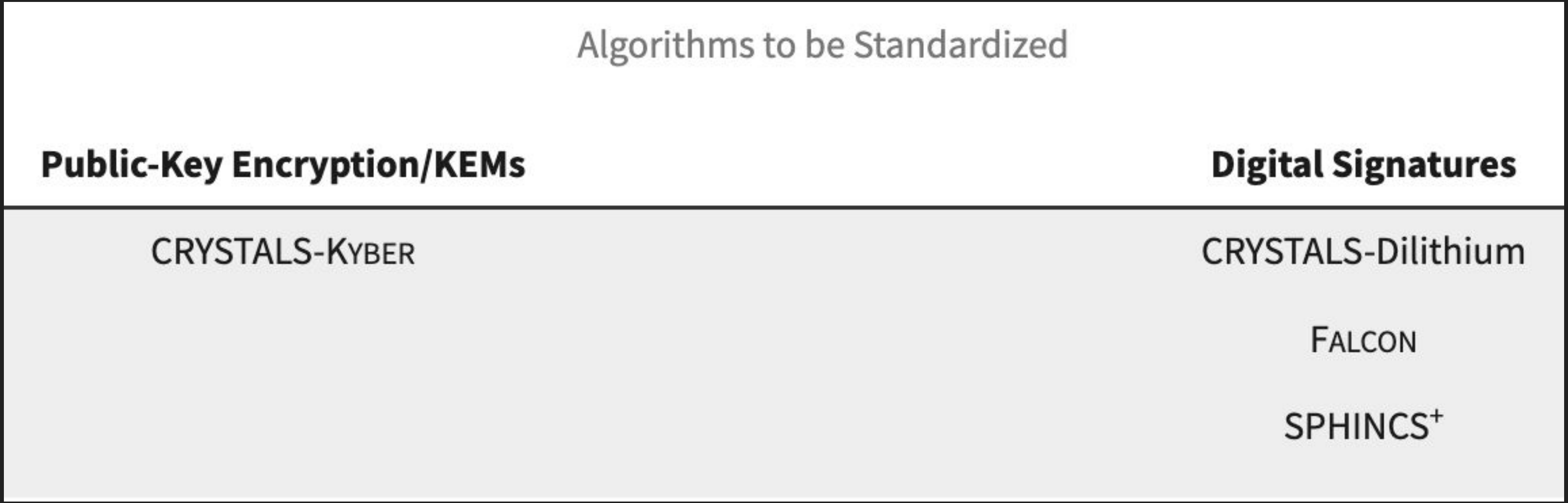
‘Fancy Crypto’



PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates

July 05, 2022

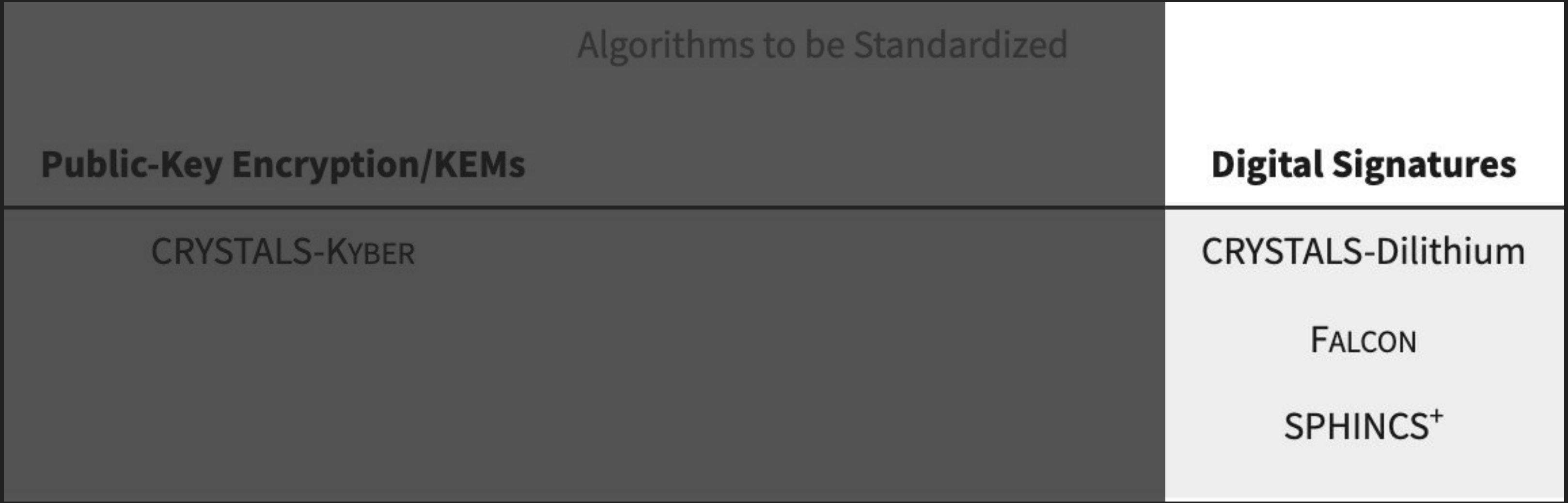
https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4



Algorithms to be Standardized

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

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



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

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

Key Agreement

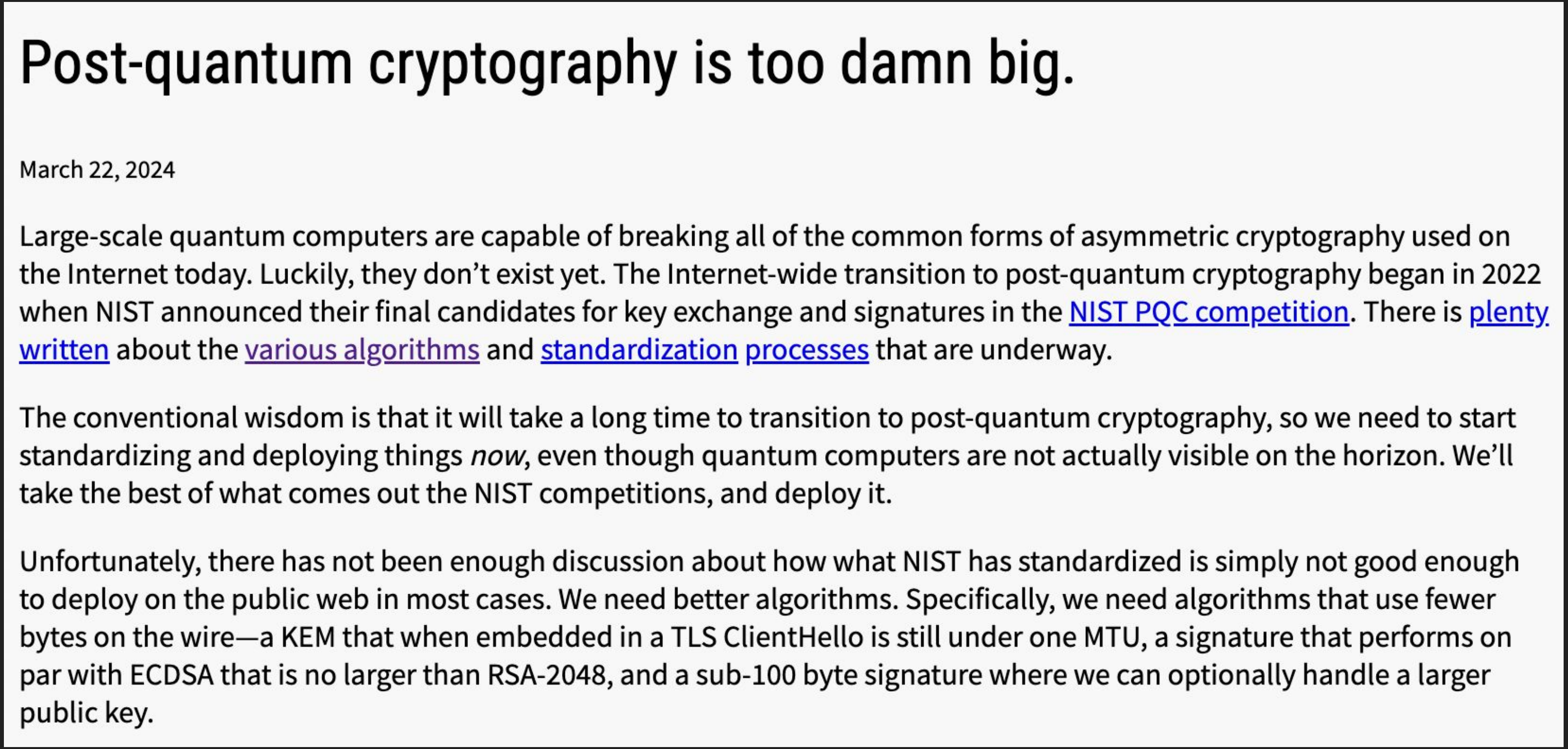
Signatures

‘Fancy Crypto’

Key Agreement

Signatures

‘Fancy Crypto’



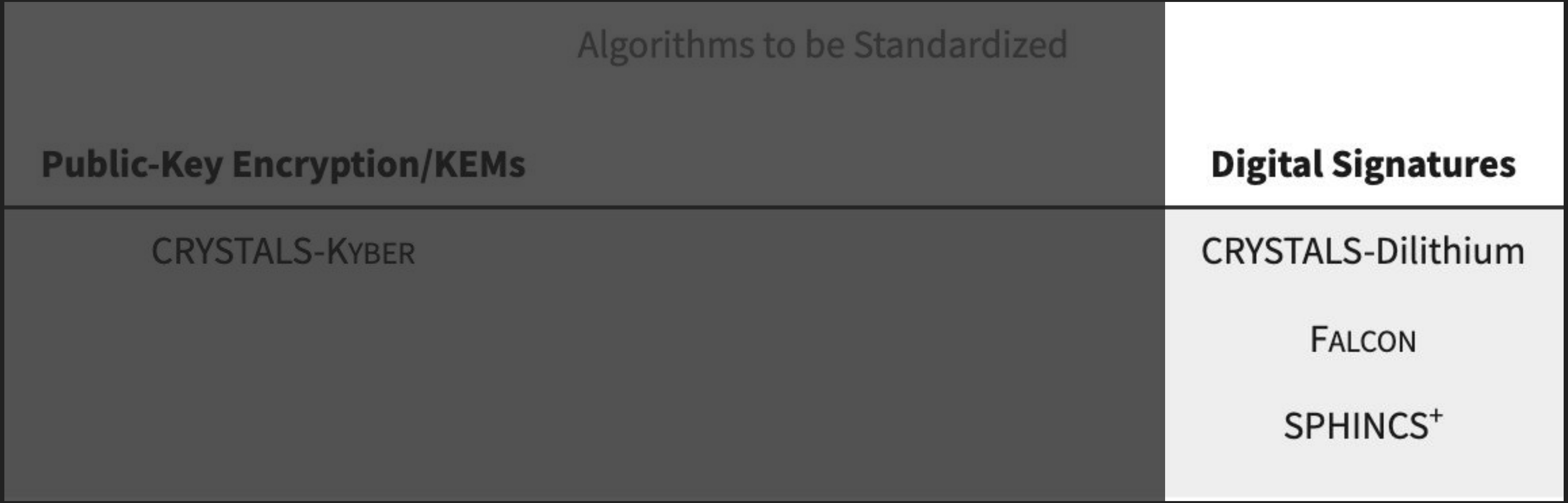
Post-quantum cryptography is too damn big.

March 22, 2024

Large-scale quantum computers are capable of breaking all of the common forms of asymmetric cryptography used on the Internet today. Luckily, they don't exist yet. The Internet-wide transition to post-quantum cryptography began in 2022 when NIST announced their final candidates for key exchange and signatures in the NIST PQC competition. There is plenty written about the various algorithms and standardization processes that are underway.

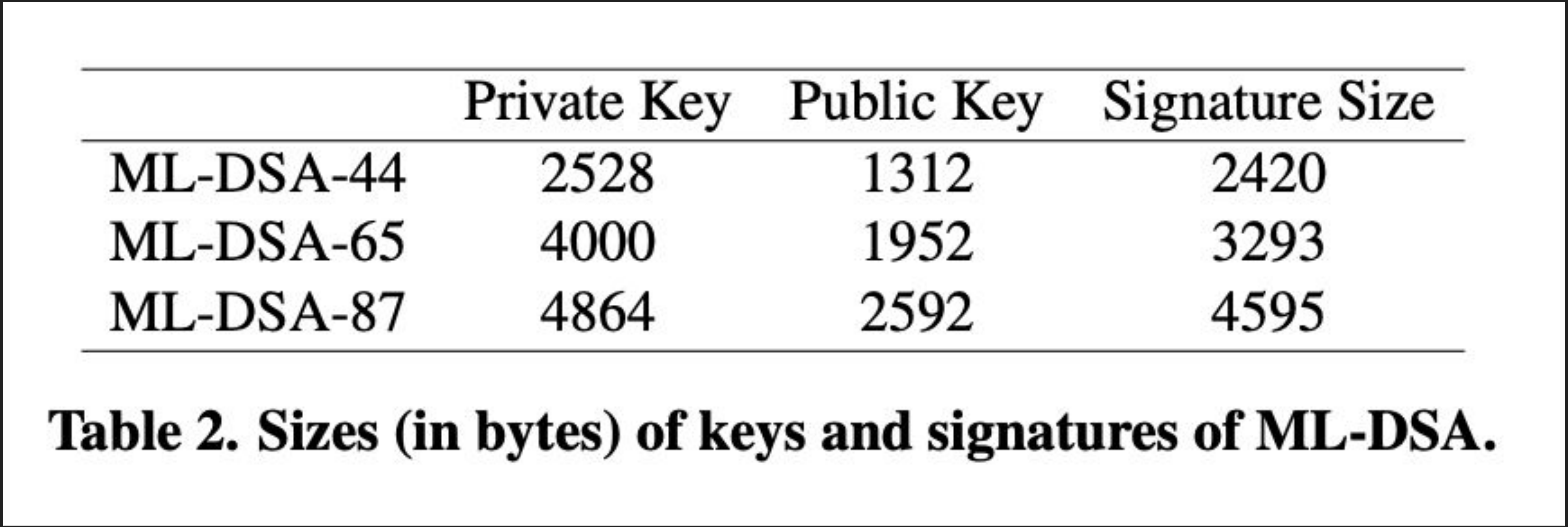
The conventional wisdom is that it will take a long time to transition to post-quantum cryptography, so we need to start standardizing and deploying things *now*, even though quantum computers are not actually visible on the horizon. We'll take the best of what comes out the NIST competitions, and deploy it.

Unfortunately, there has not been enough discussion about how what NIST has standardized is simply not good enough to deploy on the public web in most cases. We need better algorithms. Specifically, we need algorithms that use fewer bytes on the wire—a KEM that when embedded in a TLS ClientHello is still under one MTU, a signature that performs on par with ECDSA that is no larger than RSA-2048, and a sub-100 byte signature where we can optionally handle a larger public key.



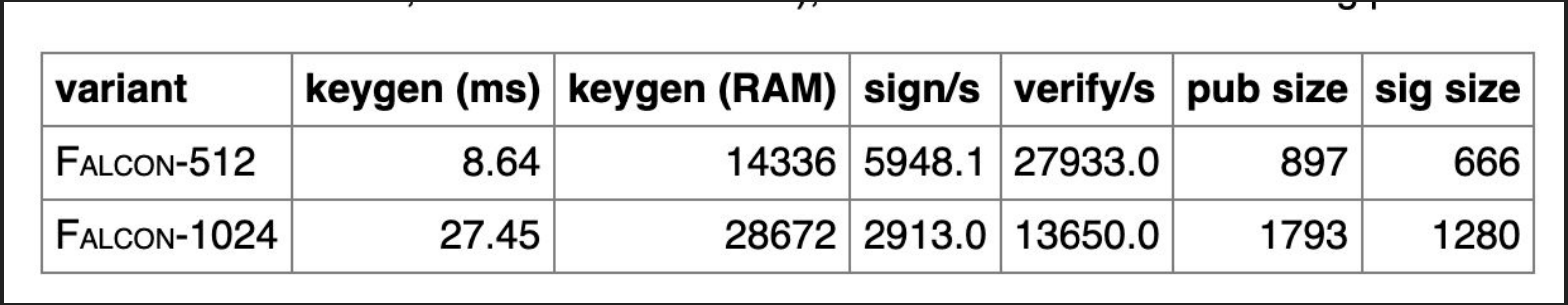
Algorithms to be Standardized	
Public-Key Encryption/KEMs	**Digital Signatures**
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS$^+$

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



	Private Key	Public Key	Signature Size
ML-DSA-44	2528	1312	2420
ML-DSA-65	4000	1952	3293
ML-DSA-87	4864	2592	4595

Table 2. Sizes (in bytes) of keys and signatures of ML-DSA.



variant	keygen (ms)	keygen (RAM)	sign/s	verify/s	pub size	sig size
FALCON-512	8.64	14336	5948.1	27933.0	897	666
FALCON-1024	27.45	28672	2913.0	13650.0	1793	1280

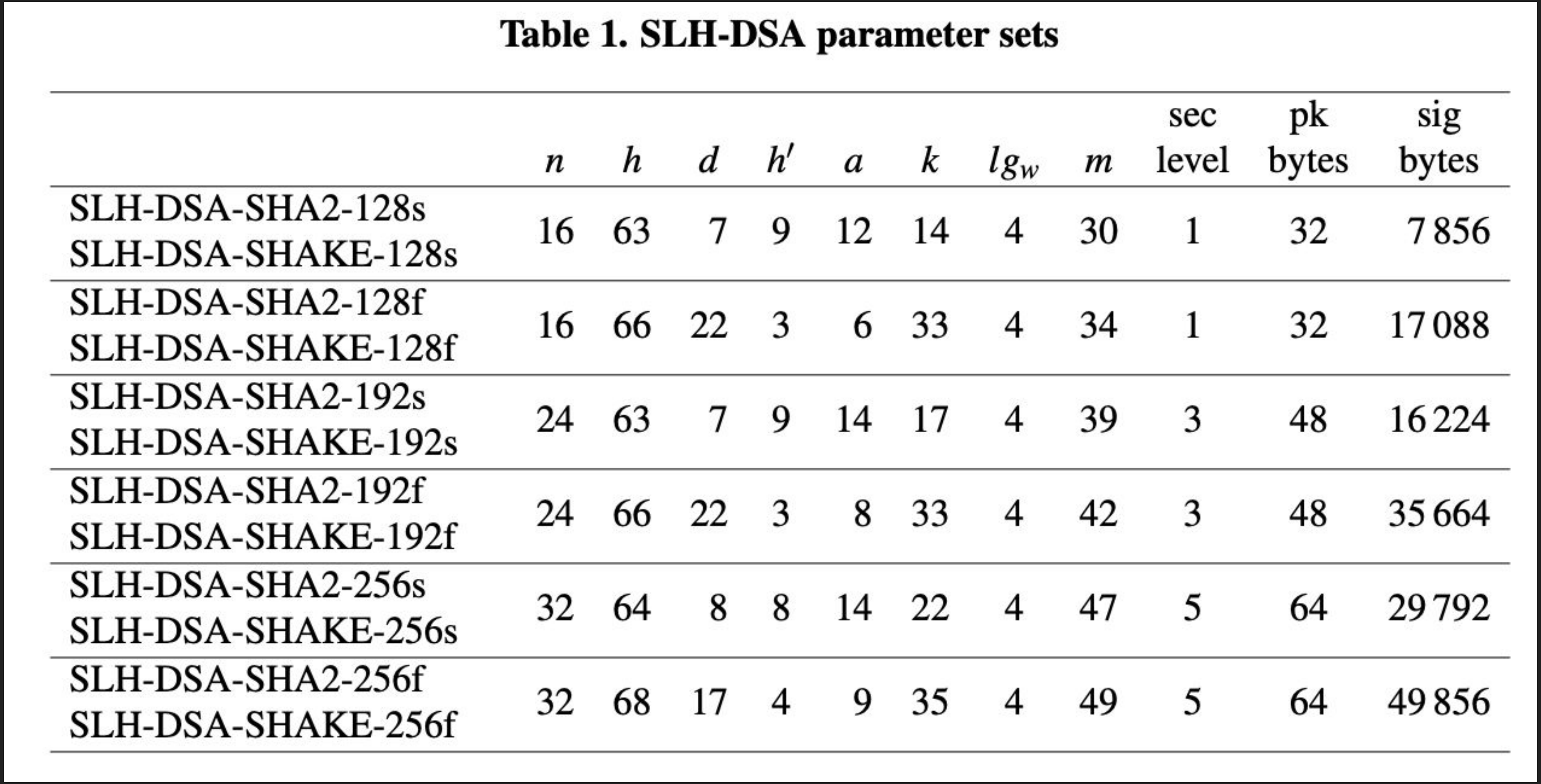
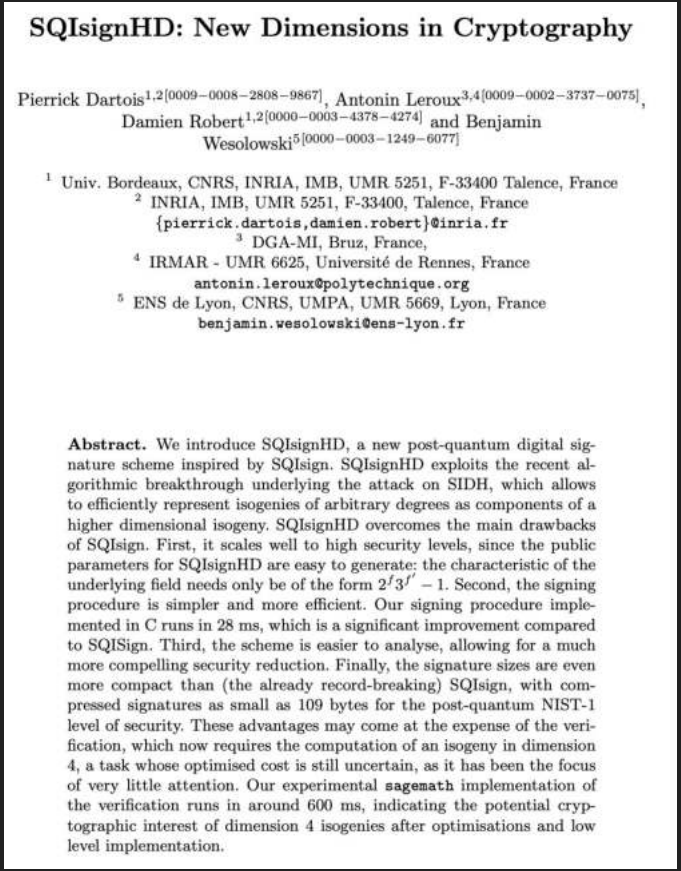


Table 1. SLH-DSA parameter sets

	n	h	d	h'	a	k	lg_w	m	sec level	pk bytes	sig bytes
SLH-DSA-SHA2-128s SLH-DSA-SHAKE-128s	16	63	7	9	12	14	4	30	1	32	7 856
SLH-DSA-SHA2-128f SLH-DSA-SHAKE-128f	16	66	22	3	6	33	4	34	1	32	17 088
SLH-DSA-SHA2-192s SLH-DSA-SHAKE-192s	24	63	7	9	14	17	4	39	3	48	16 224
SLH-DSA-SHA2-192f SLH-DSA-SHAKE-192f	24	66	22	3	8	33	4	42	3	48	35 664
SLH-DSA-SHA2-256s SLH-DSA-SHAKE-256s	32	64	8	8	14	22	4	47	5	64	29 792
SLH-DSA-SHA2-256f SLH-DSA-SHAKE-256f	32	68	17	4	9	35	4	49	5	64	49 856



SQIsignHD: New Dimensions in Cryptography

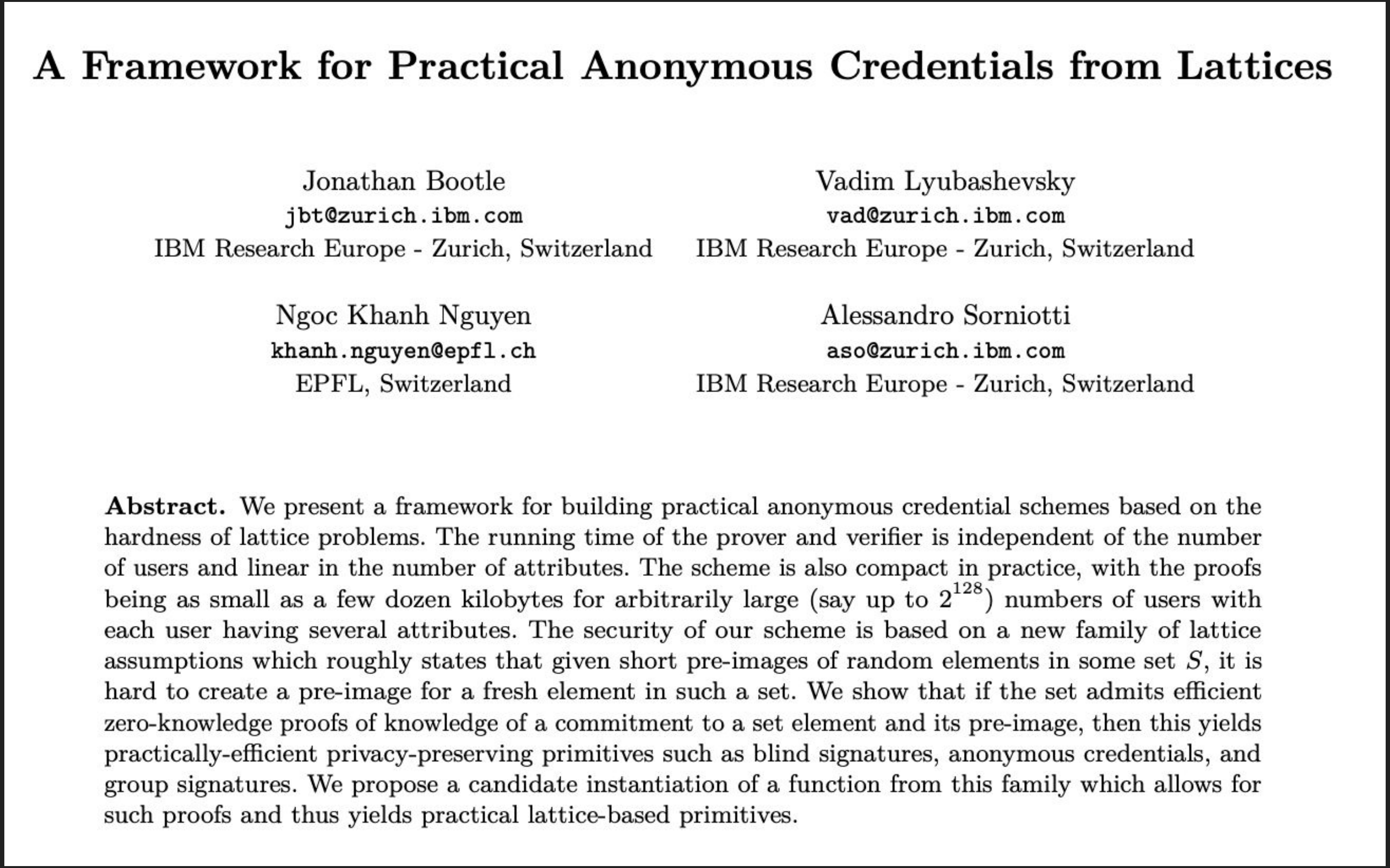
Pierrick Dartois[1,2][0009-0008-2808-9867], Antonin Leroux[3,4][0009-0002-3737-0075], Damien Robert[1,2][0000-0003-4378-4274] and Benjamin Wesolowski[5][0000-0003-1249-6077]

[1] Univ. Bordeaux, CNRS, INRIA, IMB, UMR 5251, F-33400 Talence, France
[2] INRIA, IMB, UMR 5251, F-33400, Talence, France
{pierrick.dartois,damien.robert}@inria.fr
[3] DGA-MI, Bruz, France,
[4] IRMAR - UMR 6625, Université de Rennes, France
antonin.leroux@polytechnique.org
[5] ENS de Lyon, CNRS, UMPA, UMR 5669, Lyon, France
benjamin.wesolowski@ens-lyon.fr

Abstract. We introduce SQIsignHD, a new post-quantum digital signature scheme inspired by SQIsign. SQIsignHD exploits the recent algorithmic breakthrough underlying the attack on SIDH, which allows to efficiently represent isogenies of arbitrary degrees as components of a higher dimensional isogeny. SQIsignHD overcomes the main drawbacks of SQIsign. First, it scales well to high security levels, since the public parameters for SQIsignHD are easy to generate: the characteristic of the underlying field needs only be of the form $2^f 3^{f'} - 1$. Second, the signing procedure is simpler and more efficient. Our signing procedure implemented in C runs in 28 ms, which is a significant improvement compared to SQISign. Third, the scheme is easier to analyse, allowing for a much more compelling security reduction. Finally, the signature sizes are even more compact than (the already record-breaking) SQIsign, with compressed signatures as small as 109 bytes for the post-quantum NIST-1 level of security. These advantages may come at the expense of the verification, which now requires the computation of an isogeny in dimension 4, a task whose optimised cost is still uncertain, as it has been the focus of very little attention. Our experimental `sagemath` implementation of the verification runs in around 600 ms, indicating the potential cryptographic interest of dimension 4 isogenies after optimisations and low level implementation.

Key Agreement
Signatures
‘Fancy Crypto’

Key Agreement
Signatures
‘Fancy Crypto’



A Framework for Practical Anonymous Credentials from Lattices

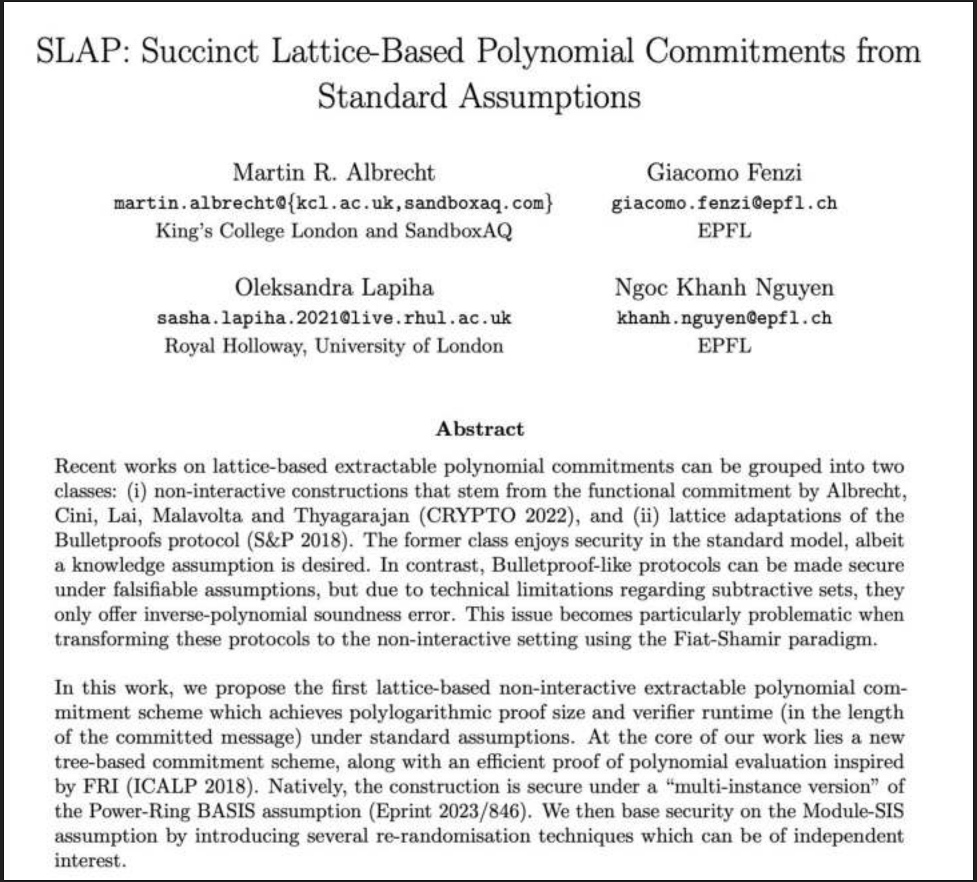
Jonathan Bootle
jbt@zurich.ibm.com
IBM Research Europe - Zurich, Switzerland

Vadim Lyubashevsky
vad@zurich.ibm.com
IBM Research Europe - Zurich, Switzerland

Ngoc Khanh Nguyen
khanh.nguyen@epfl.ch
EPFL, Switzerland

Alessandro Sorniotti
aso@zurich.ibm.com
IBM Research Europe - Zurich, Switzerland

Abstract. We present a framework for building practical anonymous credential schemes based on the hardness of lattice problems. The running time of the prover and verifier is independent of the number of users and linear in the number of attributes. The scheme is also compact in practice, with the proofs being as small as a few dozen kilobytes for arbitrarily large (say up to 2^{128}) numbers of users with each user having several attributes. The security of our scheme is based on a new family of lattice assumptions which roughly states that given short pre-images of random elements in some set S, it is hard to create a pre-image for a fresh element in such a set. We show that if the set admits efficient zero-knowledge proofs of knowledge of a commitment to a set element and its pre-image, then this yields practically-efficient privacy-preserving primitives such as blind signatures, anonymous credentials, and group signatures. We propose a candidate instantiation of a function from this family which allows for such proofs and thus yields practical lattice-based primitives.



SLAP: Succinct Lattice-Based Polynomial Commitments from Standard Assumptions

Martin R. Albrecht
martin.albrecht@{kcl.ac.uk,sandboxaq.com}
King's College London and SandboxAQ

Giacomo Fenzi
giacomo.fenzi@epfl.ch
EPFL

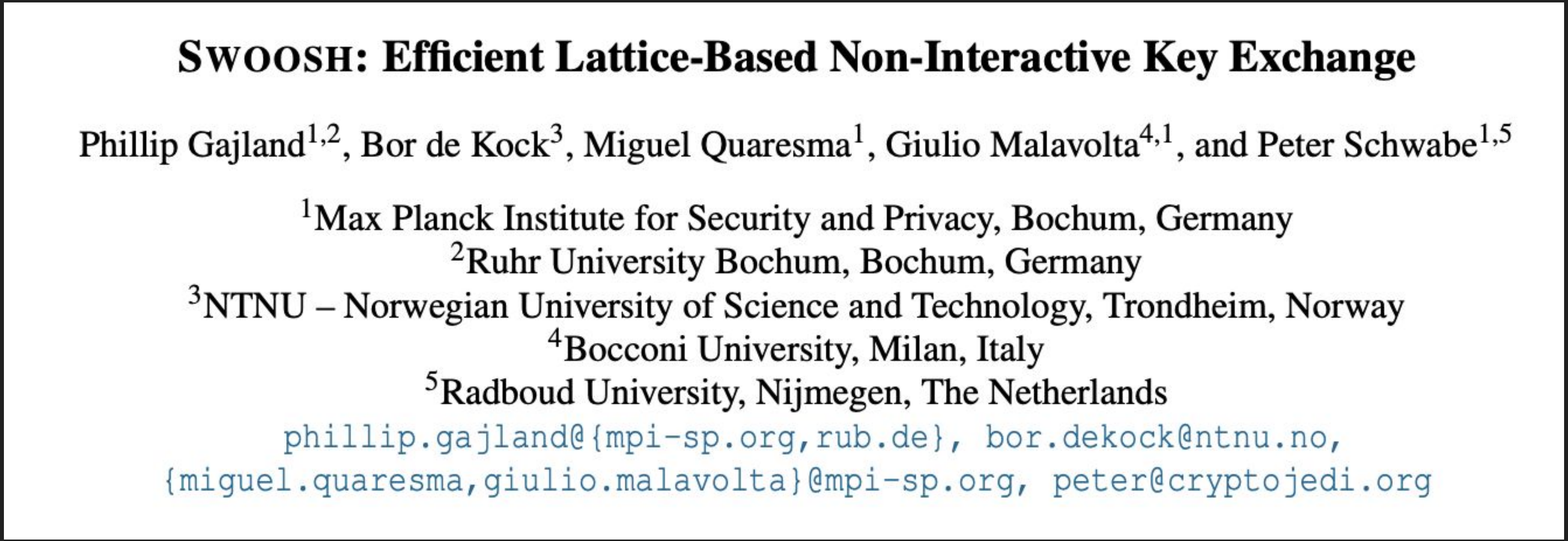
Oleksandra Lapiha
sasha.lapiha.2021@live.rhul.ac.uk
Royal Holloway, University of London

Ngoc Khanh Nguyen
khanh.nguyen@epfl.ch
EPFL

Abstract

Recent works on lattice-based extractable polynomial commitments can be grouped into two classes: (i) non-interactive constructions that stem from the functional commitment by Albrecht, Cini, Lai, Malavolta and Thyagarajan (CRYPTO 2022), and (ii) lattice adaptations of the Bulletproofs protocol (S&P 2018). The former class enjoys security in the standard model, albeit a knowledge assumption is desired. In contrast, Bulletproof-like protocols can be made secure under falsifiable assumptions, but due to technical limitations regarding subtractive sets, they only offer inverse-polynomial soundness error. This issue becomes particularly problematic when transforming these protocols to the non-interactive setting using the Fiat-Shamir paradigm.

In this work, we propose the first lattice-based non-interactive extractable polynomial commitment scheme which achieves polylogarithmic proof size and verifier runtime (in the length of the committed message) under standard assumptions. At the core of our work lies a new tree-based commitment scheme, along with an efficient proof of polynomial evaluation inspired by FRI (ICALP 2018). Natively, the construction is secure under a "multi-instance version" of the Power-Ring BASIS assumption (Eprint 2023/846). We then base security on the Module-SIS assumption by introducing several re-randomisation techniques which can be of independent interest.

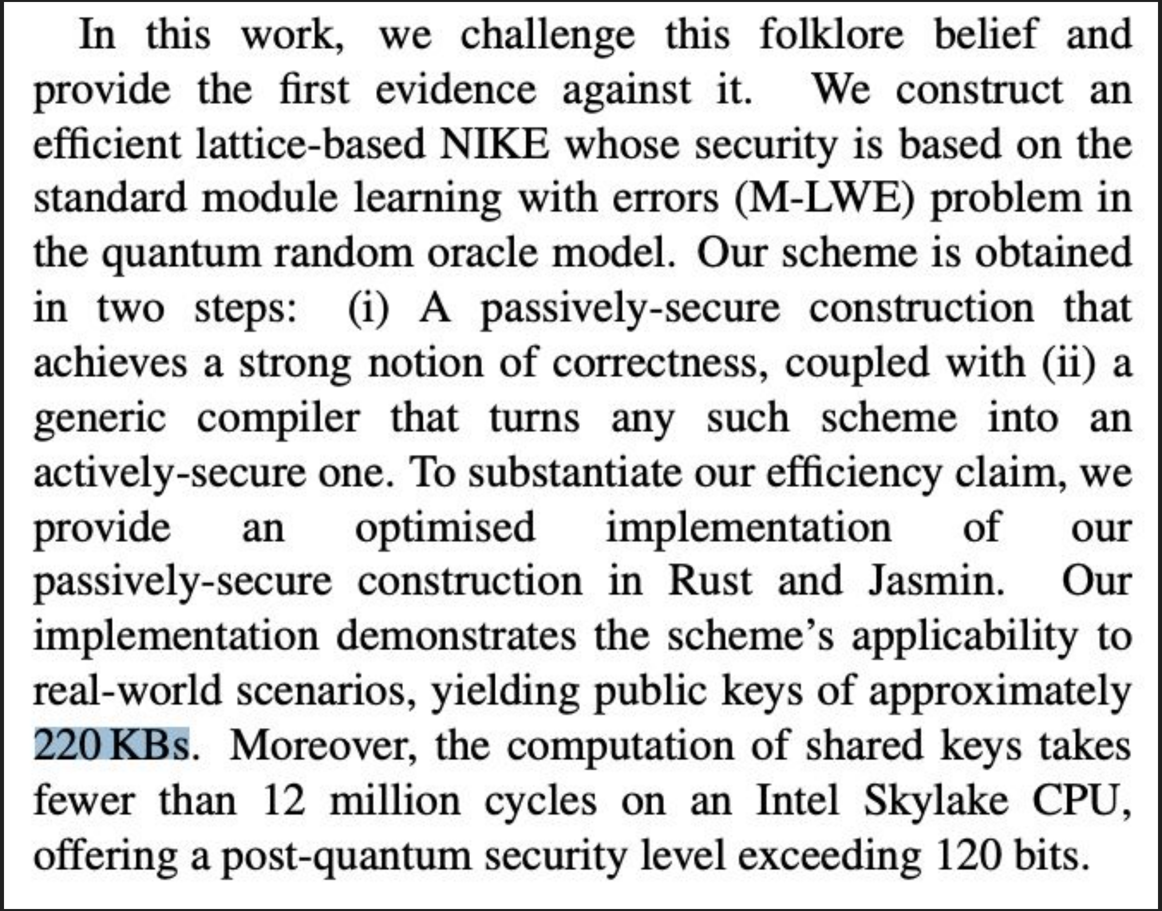


SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange

Phillip Gajland[1,2], Bor de Kock[3], Miguel Quaresma[1], Giulio Malavolta[4,1], and Peter Schwabe[1,5]

[1]Max Planck Institute for Security and Privacy, Bochum, Germany
[2]Ruhr University Bochum, Bochum, Germany
[3]NTNU – Norwegian University of Science and Technology, Trondheim, Norway
[4]Bocconi University, Milan, Italy
[5]Radboud University, Nijmegen, The Netherlands

phillip.gajland@{mpi-sp.org,rub.de}, bor.dekock@ntnu.no,
{miguel.quaresma,giulio.malavolta}@mpi-sp.org, peter@cryptojedi.org


In this work, we challenge this folklore belief and provide the first evidence against it. We construct an efficient lattice-based NIKE whose security is based on the standard module learning with errors (M-LWE) problem in the quantum random oracle model. Our scheme is obtained in two steps: (i) A passively-secure construction that achieves a strong notion of correctness, coupled with (ii) a generic compiler that turns any such scheme into an actively-secure one. To substantiate our efficiency claim, we provide an optimised implementation of our passively-secure construction in Rust and Jasmin. Our implementation demonstrates the scheme's applicability to real-world scenarios, yielding public keys of approximately 220 KBs. Moreover, the computation of shared keys takes fewer than 12 million cycles on an Intel Skylake CPU, offering a post-quantum security level exceeding 120 bits.

Key Agreement
Signatures
‘Fancy Crypto’

Questions?

Going Post-Quantum

Deirdre Connolly - SandboxAQ