Cryptography Engineering

- Lecture 6 (Nov 26, 2025)
- Today's notes:
    - Fujisaki-Okamoto transformation – 2
    - Authentication via KEM
    - PQ-TLS: KEM + Sign
    - KEM-TLS

Fujisaki-Okamoto Transformation

- Let PKE = (KG, Enc, Dec) be a public-key encryption scheme
- Let H, G be two hash functions (Quick question: How to instantiate them using SHA256)
- We construct an FOKEM scheme based on PKE

```
KeyGen:
1. $(pk, sk) \leftarrow$ KG
2. $prk \leftarrow \{0,1\}^L$
3. $pk \coloneqq pk$
4. $sk \coloneqq (prk, sk)$
```

```
Encaps($pk = pk$):
1. $m \leftarrow_{\$}$ MsgSpace
2. $r \coloneqq G(pk, m)$
   // randomness for PKE
3. $c \coloneqq$ Enc$(pk, m; r)$
4. $K \coloneqq H(pk, m, c)$
5. $c \coloneqq c$
6. return $(c, K)$
```

```
Decaps($sk = (prk, sk), c = c$):
1. $m =$ Dec$(sk, c)$
2. $r \coloneqq G(pk, m)$
   // Recover randomness
3. $c' \coloneqq$ Enc$(pk, m; r)$
   // Re-encryption check
4. If $c == c'$: return $H(pk, m, c)$
5. Else: return $H(pk, prk, c)$
```

Fujisaki-Okamoto Transformation

- **Implicit rejection:** Return a pseudorandom key (rather than returning a rejection symbol).

```
KeyGen:
$(pk, sk) \leftarrow \mathsf{KG}$
$prk \leftarrow \{0,1\}^L$
$pk := pk$
$sk := (prk, sk)$
```

```
Encaps($pk = pk$):
$m \leftarrow_{\$} \mathsf{MsgSpace}$
$r := G(pk, m)$
  // randomness for PKE
$c := \mathsf{Enc}(pk, m; r)$
$K := H(pk, m, c)$
$c := c$
return $(c, K)$
```

```
Decaps($sk = (prk, sk), c = c$):
$m = \mathsf{Dec}(sk, c)$
$r := G(pk, m)$
  // Recover randomness
$c' := \mathsf{Enc}(pk, m; r)$
  // Re-encryption check
If $c == c'$: return $H(pk, m, c)$
Else: return $H(pk, prk, c)$
```

Fujisaki-Okamoto Transformation

- **Implicit rejection:** Return a pseudorandom key (rather than returning a rejection symbol).
  - Quick question: Why the returning key (if the re-enc check fails) is pseudorandom?

```
KeyGen:
1. $(pk, sk) \leftarrow \text{KG}$
2. $prk \leftarrow \{0,1\}^L$
3. $pk \coloneqq pk$
4. $sk \coloneqq (prk, sk)$
```

```
Encaps($pk = pk$):
1. $m \leftarrow_\$ \text{MsgSpace}$
2. $r \coloneqq G(pk, m)$
   // randomness for PKE
3. $c \coloneqq \text{Enc}(pk, m; r)$
4. $K \coloneqq H(pk, m, c)$
5. $c \coloneqq c$
6. return $(c, K)$
```

```
Decaps($sk = (prk, sk), c = c$):
1. $m = \text{Dec}(sk, c)$
2. $r \coloneqq G(pk, m)$
   // Recover randomness
3. $c' \coloneqq \text{Enc}(pk, m; r)$
   // Re-encryption check
4. If $c == c'$: return $H(pk, m, c)$
5. Else: return $H(pk, prk, c)$
```

Fujisaki-Okamoto Transformation

- **Implicit rejection:** Return a pseudorandom key (rather than returning a rejection symbol).
  - Quick question: Why the returning key (if the re-enc check fails) is pseudorandom?
  - (1) prk is random and secret => H(pk, prk, c) also looks like random
  - (2) The same input to Decaps => The same output

```
KeyGen:
$(pk, sk) \leftarrow$ KG
$prk \leftarrow \{0,1\}^L$
$pk := pk$
$sk := (prk, sk)$
```

```
Encaps$(pk = pk)$:
$m \leftarrow_\$$ MsgSpace
$r := G(pk, m)$
  // randomness for PKE
$c := \mathsf{Enc}(pk, m; r)$
$K := H(pk, m, c)$
$c := c$
return $(c, K)$
```

```
Decaps$(sk = (prk, sk), c = c)$:
$m = \mathsf{Dec}(sk, c)$
$r := G(pk, m)$
  // Recover randomness
$c' := \mathsf{Enc}(pk, m; r)$
  // Re-encryption check
If $c == c'$: return $H(pk, m, c)$
Else: return $H(pk, prk, c)$
```

Fujisaki-Okamoto Transformation

- **Explicit rejection:** Return a rejection symbol (or throw an exception)

```
KeyGen:
1. $(pk, sk) \leftarrow$ KG
~~2. $prk \leftarrow \{0,1\}^L$~~
3. $pk \coloneqq pk$
4. $sk \coloneqq (~~prk,~~ sk)$
```

```
Encaps($pk = pk$):
1. $m \leftarrow_{\$}$ MsgSpace
2. $r \coloneqq G(pk, m)$
   // randomness for PKE
3. $c \coloneqq$ Enc$(pk, m; r)$
4. $K \coloneqq H(pk, m, c)$
5. $c \coloneqq c$
6. return $(c, K)$
```

```
Decaps($sk = (~~prk,~~ sk), c = c$):
1. $m =$ Dec$(sk, c)$
2. $r \coloneqq G(pk, m)$
   // Recover randomness
3. $c' \coloneqq$ Enc$(pk, m; r)$
   // Re-encryption check
4. If $c == c'$: return $H(pk, m, c)$
5. Else: return **REJECT**
```

Fujisaki-Okamoto Transformation

- Constant-time comparison
  - Regular equality checks may leak timing information
  - E.g., early-return behavior can reveal where the first differing bit occurs

```
Decaps($sk = (prk, sk), c = c$):
1. $m = \text{Dec}(sk, c)$
2. $r := G(pk, m)$
   // Recover randomness
3. $c' := \text{Enc}(pk, m; r)$
   // Re-encryption check
4. If $c == c'$: return $H(pk, m, c)$
5. Else: return $H(pk, prk, c)$
```

Fujisaki-Okamoto Transformation

- Constant-time comparison
  - Regular equality checks may leak timing information
  - E.g., early-return behavior can reveal where the first differing bit occurs

```
Decaps($sk = (prk, sk), c = c$):
1. $m = \text{Dec}(sk, c)$
2. $r \coloneqq G(pk, m)$
   // Recover randomness
3. $c' \coloneqq \text{Enc}(pk, m; r)$
   // Re-encryption check
4. If $c == c'$: return $H(pk, m, c)$
5. Else: return $H(pk, prk, c)$
```

Fujisaki-Okamoto Transformation

- Constant-time comparison
  - Regular equality checks may leak timing information
  - E.g., early-return behavior can reveal where the first differing bit occurs

```
CT-Decaps($sk = (prk, sk), c = c$):
1. $m = \mathrm{Dec}(sk, c)$
2. $r \coloneqq G(pk, m)$
3. $c' \coloneqq \mathrm{Enc}(pk, m; r)$
4. $K_0 = H(pk, m, c)$
5. $K_1 = H(pk, prk, c)$
6. $b =$ constant-time-eq($c', c$)
7. $K =$ constant-time-select($b, K_0, K_1$)
   // $K = (1 - b) \cdot K_0 \oplus b \cdot K_1$
8. return $K$
```

```
Decaps($sk = (prk, sk), c = c$):
1. $m = \mathrm{Dec}(sk, c)$
2. $r \coloneqq G(pk, m)$
   // Recover randomness
3. $c' \coloneqq \mathrm{Enc}(pk, m; r)$
   // Re-encryption check
4. If $c == c'$: return $H(pk, m, c)$
5. Else: return $H(pk, prk, c)$
```

Post-quantum Cryptography

- Most widely deployed cryptosystems rely on the hardness of Diffie–Hellman and Factoring (RSA).
- However, these problems are no longer considered hard in the presence of large-scale quantum computers.

- **Post-quantum cryptography:**
  - Cryptographic algorithms run in classical computers
  - Security against adversaries with quantum computers

- NIST PQC standardized algorithms:
  - KEM schemes: ML-KEM (Crystal-Kyber)
  - Signature schemes: ML-DSA (Crystal-Dilithium)

Post-quantum Cryptography

- Post-quantum secure KEM
  - Design a IND-CPA PKE scheme first
  - Use the FO transform (or its variants) to get an IND-CCA KEM scheme

- Post-quantum secure Signature
  - More complex structures…

Post-quantum TLS

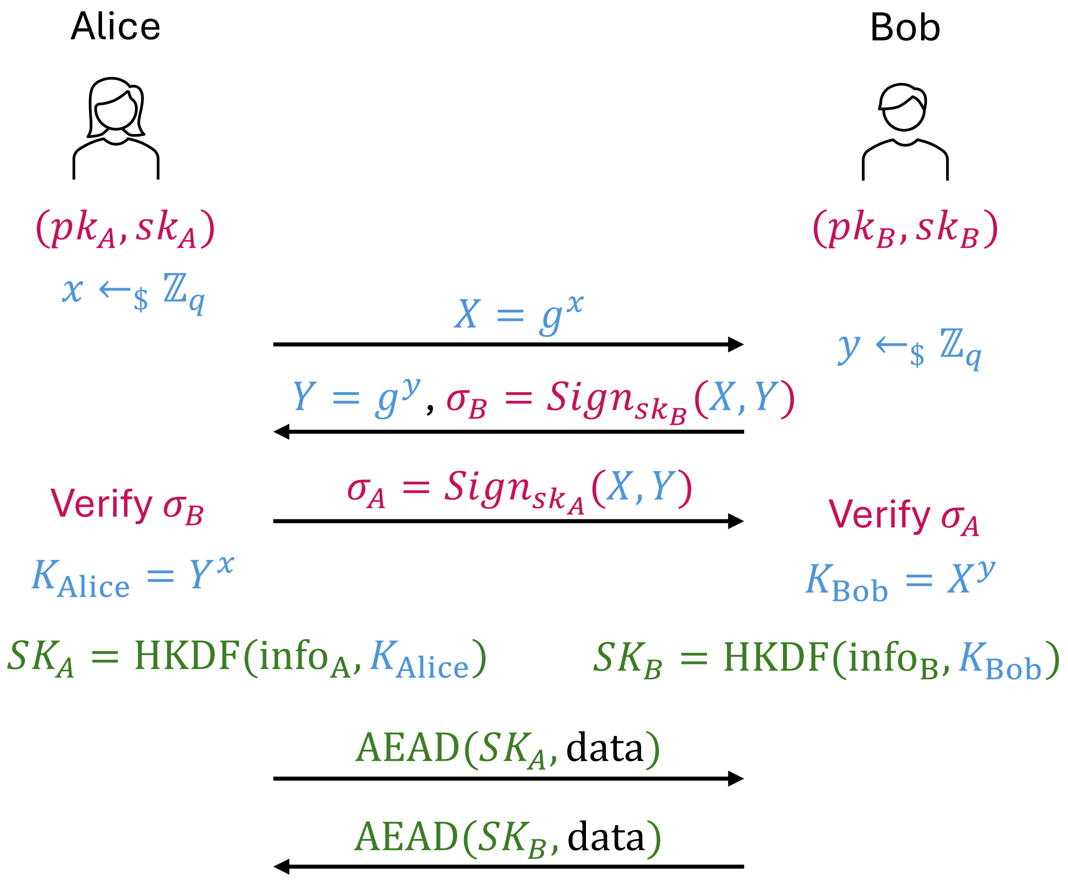
- Post-quantum secure KEM
  - Design a IND-CPA PKE scheme first
  - Use the FO transform (or its variants) to get an IND-CCA KEM scheme

- Post-quantum secure Signature
  - More complex structures…

- TLS 1.3 is based on DH problems, so it is not post-quantum secure.
  - **Can we have post-quantum TLS?**

Post-quantum TLS

- The signed DH protocol



Alice

Bob

(pk_A, sk_A)

(pk_B, sk_B)

$x \leftarrow_\$ \mathbb{Z}_q$

$X = g^x$

$y \leftarrow_\$ \mathbb{Z}_q$

$Y = g^y, \sigma_B = Sign_{sk_B}(X, Y)$

Verify σ_B

$\sigma_A = Sign_{sk_A}(X, Y)$

Verify σ_A

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

$SK_A = \text{HKDF}(\text{info}_\text{A}, K_{\text{Alice}})$

$SK_B = \text{HKDF}(\text{info}_\text{B}, K_{\text{Bob}})$

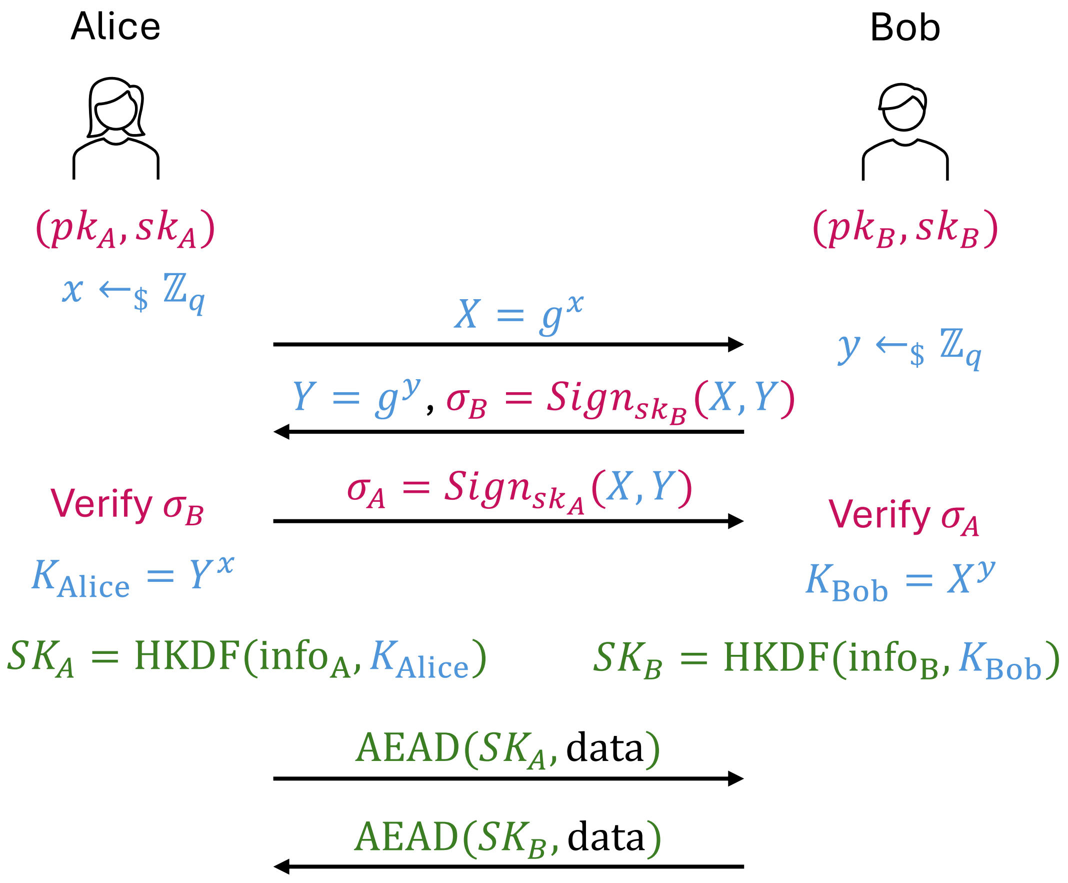
$\text{AEAD}(SK_A, \text{data})$

$\text{AEAD}(SK_B, \text{data})$

Post-quantum TLS

- The signed DH protocol

- To make it post-quantum secure, which parts should we change?



Alice

Bob

(pk_A, sk_A)

(pk_B, sk_B)

$x \leftarrow_\$ \mathbb{Z}_q$

$X = g^x$

$y \leftarrow_\$ \mathbb{Z}_q$

$Y = g^y, \sigma_B = Sign_{sk_B}(X, Y)$

Verify σ_B

$\sigma_A = Sign_{sk_A}(X, Y)$

Verify σ_A

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

$SK_A = \text{HKDF}(\text{info}_\text{A}, K_{\text{Alice}})$

$SK_B = \text{HKDF}(\text{info}_\text{B}, K_{\text{Bob}})$

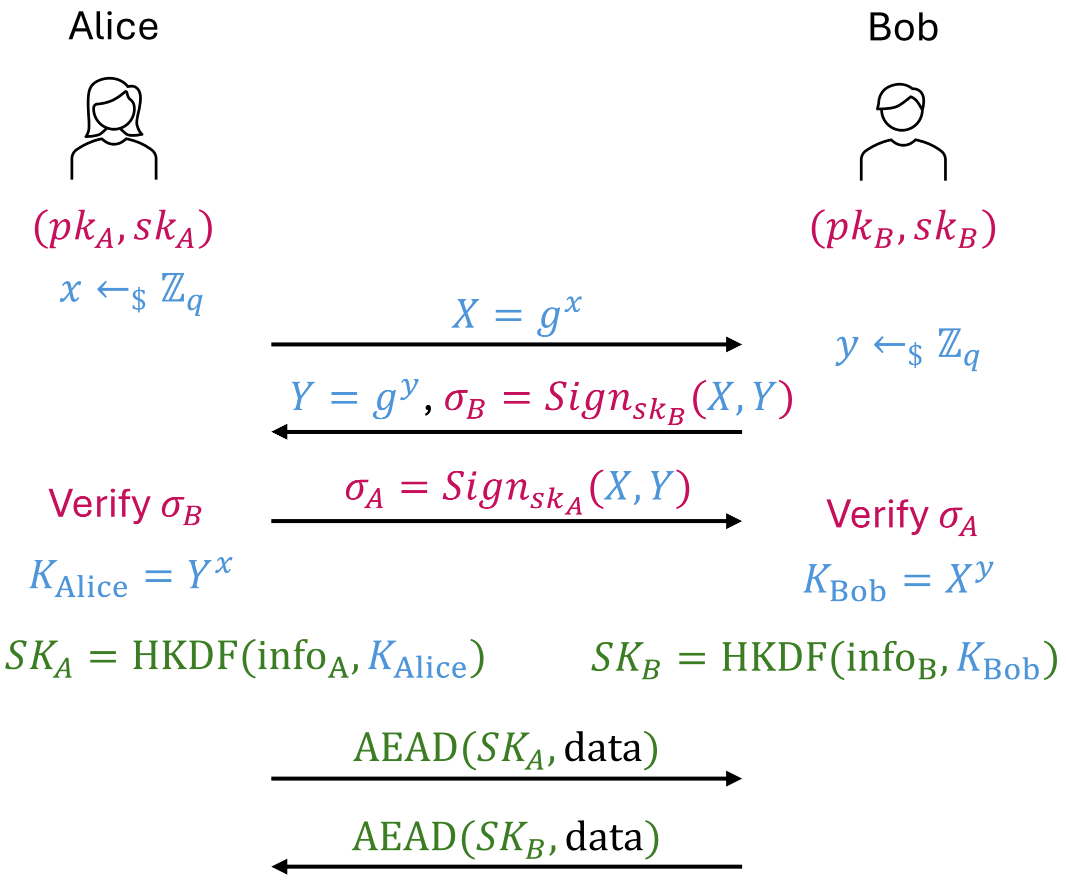
$\text{AEAD}(SK_A, \text{data})$

$\text{AEAD}(SK_B, \text{data})$

Post-quantum TLS

- The signed DH protocol

- To make it post-quantum secure, which parts should we change?
  - Symmetric algorithms remain secure
  - **The DHKE and the signature scheme (e.g., ECDSA, RSA) are not post-quantum secure**



Alice

Bob

(pk_A, sk_A)

(pk_B, sk_B)

$x \leftarrow_{\$} \mathbb{Z}_q$

$X = g^x$

$y \leftarrow_{\$} \mathbb{Z}_q$

$Y = g^y, \sigma_B = Sign_{sk_B}(X, Y)$

Verify σ_B

$\sigma_A = Sign_{sk_A}(X, Y)$

Verify σ_A

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

$SK_A = \text{HKDF}(\text{info}_\text{A}, K_{\text{Alice}})$

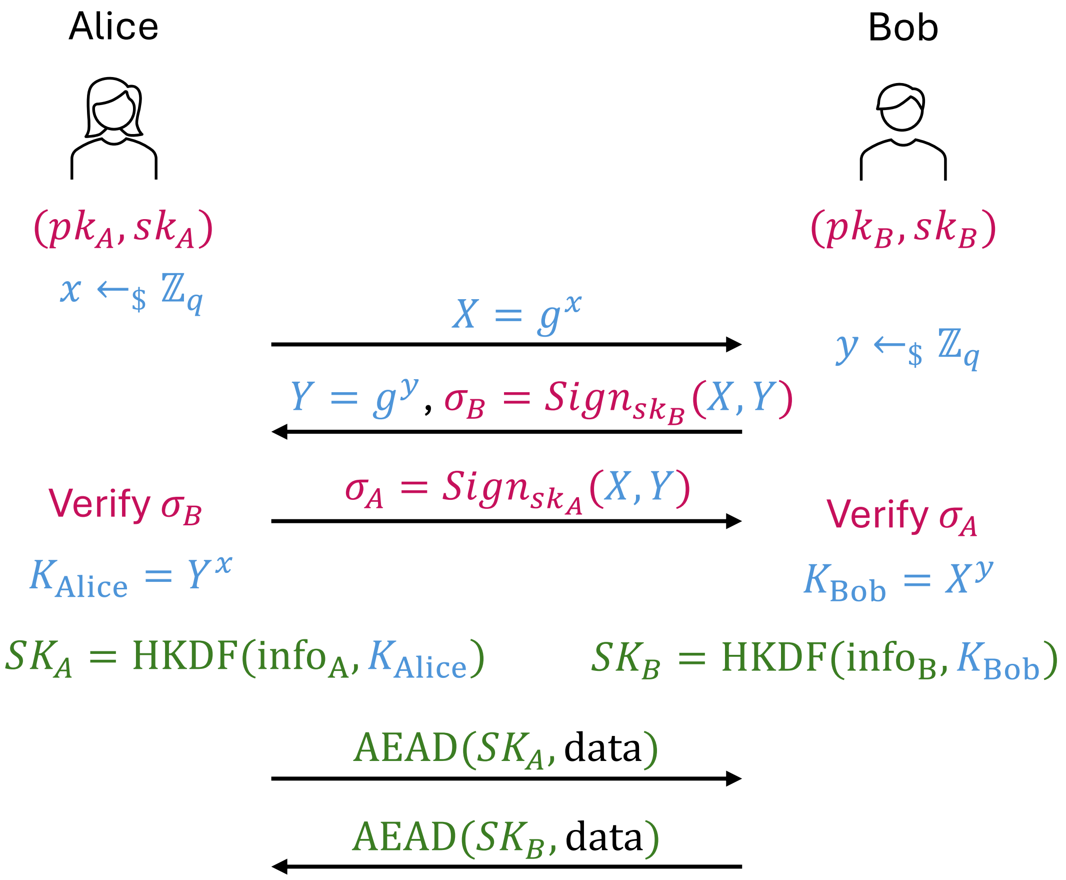
$SK_B = \text{HKDF}(\text{info}_\text{B}, K_{\text{Bob}})$

$\text{AEAD}(SK_A, \text{data})$

$\text{AEAD}(SK_B, \text{data})$

Post-quantum TLS

- The signed DH protocol
- To make it post-quantum secure, which parts should we change?
  - Symmetric algorithms remain secure
  - The DHKE and the signature scheme (e.g., ECDSA, RSA) are not post-quantum secure
- We first replace the signature with ML-DSA (or other post-quantum secure signature schemes)



Alice

Bob

(pk_A, sk_A)

(pk_B, sk_B)

$x \leftarrow_\$ \mathbb{Z}_q$

$X = g^x$

$y \leftarrow_\$ \mathbb{Z}_q$

$Y = g^y, \sigma_B = Sign_{sk_B}(X, Y)$

Verify σ_B

$\sigma_A = Sign_{sk_A}(X, Y)$

Verify σ_A

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

$SK_A = \text{HKDF}(\text{info}_\text{A}, K_{\text{Alice}})$

$SK_B = \text{HKDF}(\text{info}_\text{B}, K_{\text{Bob}})$

$\text{AEAD}(SK_A, \text{data})$

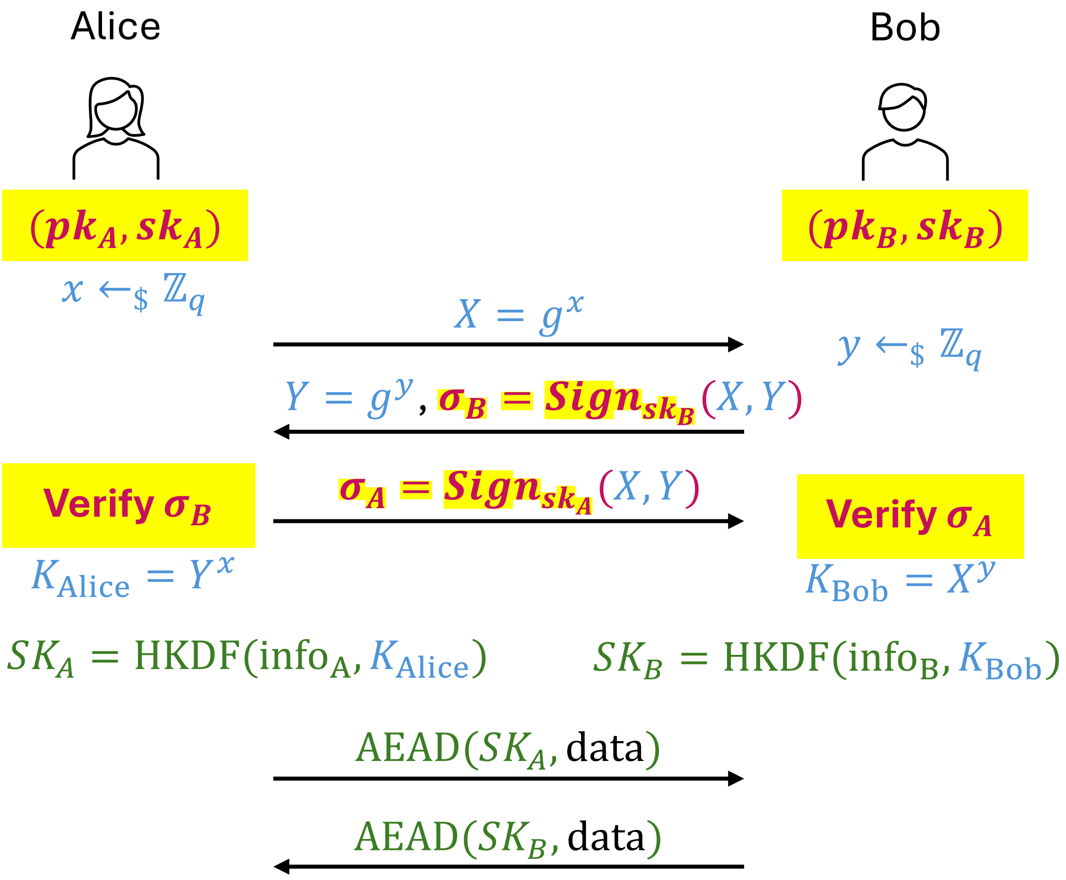
$\text{AEAD}(SK_B, \text{data})$

Post-quantum TLS

- The signed DH protocol

- To make it post-quantum secure, which parts should we change?
  - Symmetric algorithms remain secure
  - The DHKE and the signature scheme (e.g., ECDSA, RSA) are not post-quantum secure

- We first replace the signature part with **ML-DSA** (or other post-quantum secure signature schemes)



Alice

Bob

(pk_A, sk_A)

(pk_B, sk_B)

$x \leftarrow_\$ \mathbb{Z}_q$

$X = g^x$

$y \leftarrow_\$ \mathbb{Z}_q$

$Y = g^y, \sigma_B = Sign_{sk_B}(X, Y)$

Verify σ_B

$\sigma_A = Sign_{sk_A}(X, Y)$

Verify σ_A

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

$SK_A = \text{HKDF}(\text{info}_\text{A}, K_{\text{Alice}})$

$SK_B = \text{HKDF}(\text{info}_\text{B}, K_{\text{Bob}})$

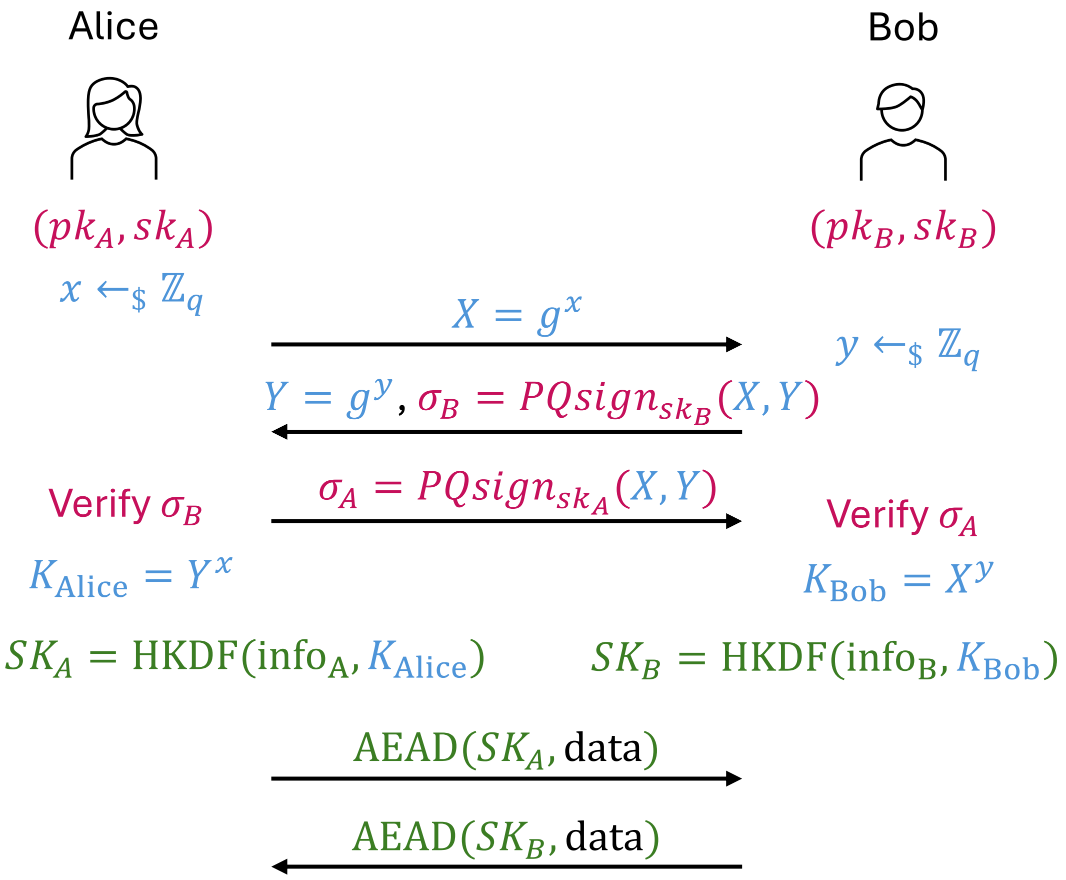
$\text{AEAD}(SK_A, \text{data})$

$\text{AEAD}(SK_B, \text{data})$

Post-quantum TLS

- The DH + PQsign protocol

- How can we replace the DH part?



Alice

Bob

(pk_A, sk_A)

(pk_B, sk_B)

$x \leftarrow_\$ \mathbb{Z}_q$

$X = g^x$

$y \leftarrow_\$ \mathbb{Z}_q$

$Y = g^y, \sigma_B = PQsign_{sk_B}(X, Y)$

Verify σ_B

$\sigma_A = PQsign_{sk_A}(X, Y)$

Verify σ_A

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

$SK_A = \text{HKDF}(\text{info}_\text{A}, K_{\text{Alice}})$

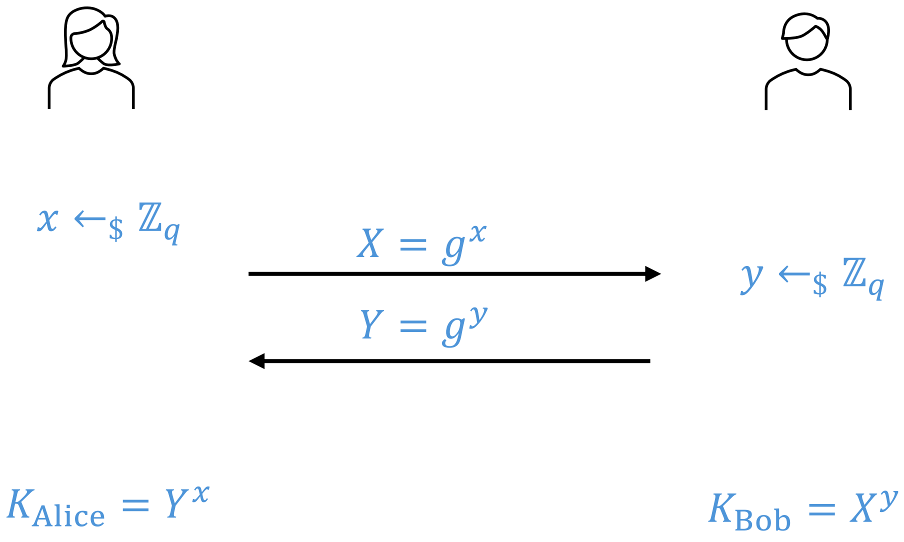
$SK_B = \text{HKDF}(\text{info}_\text{B}, K_{\text{Bob}})$

$\text{AEAD}(SK_A, \text{data})$

$\text{AEAD}(SK_B, \text{data})$

Post-quantum TLS

- DH key exchange



$x \leftarrow_\$ \mathbb{Z}_q$

$X = g^x$

$y \leftarrow_\$ \mathbb{Z}_q$

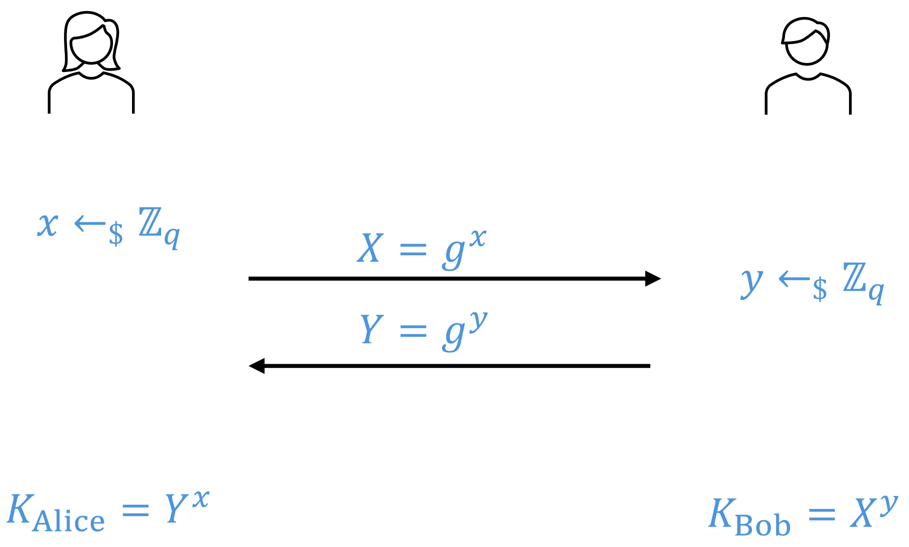
$Y = g^y$

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

Post-quantum TLS

- DH key exchange



$x \leftarrow_{\$} \mathbb{Z}_q$

$X = g^x$

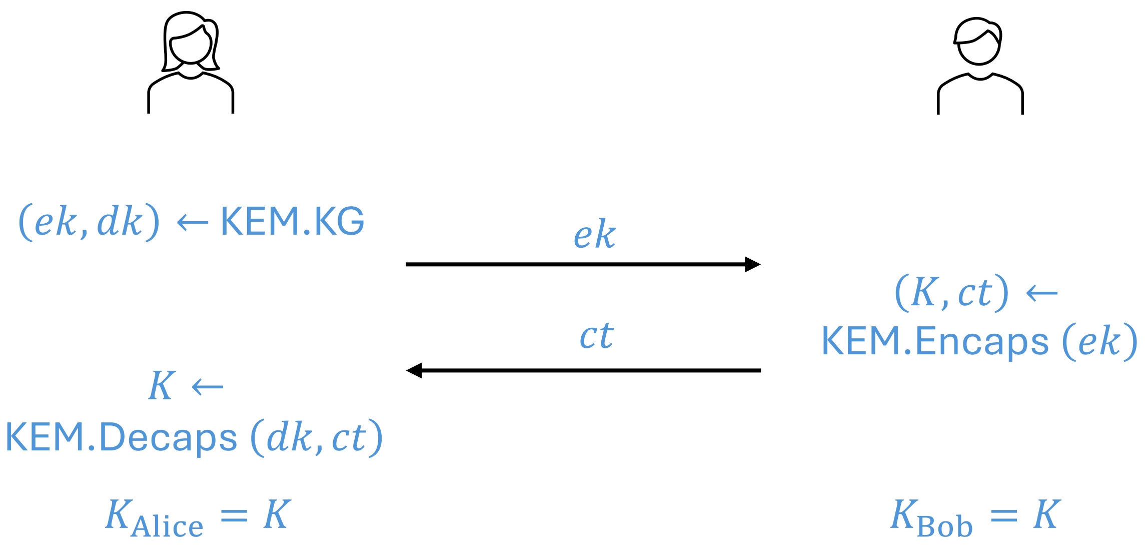
$y \leftarrow_{\$} \mathbb{Z}_q$

$Y = g^y$

$K_{\text{Alice}} = Y^x$

$K_{\text{Bob}} = X^y$

- KEM-based key exchange



$(ek, dk) \leftarrow \text{KEM.KG}$

ek

$(K, ct) \leftarrow \text{KEM.Encaps}\,(ek)$

ct

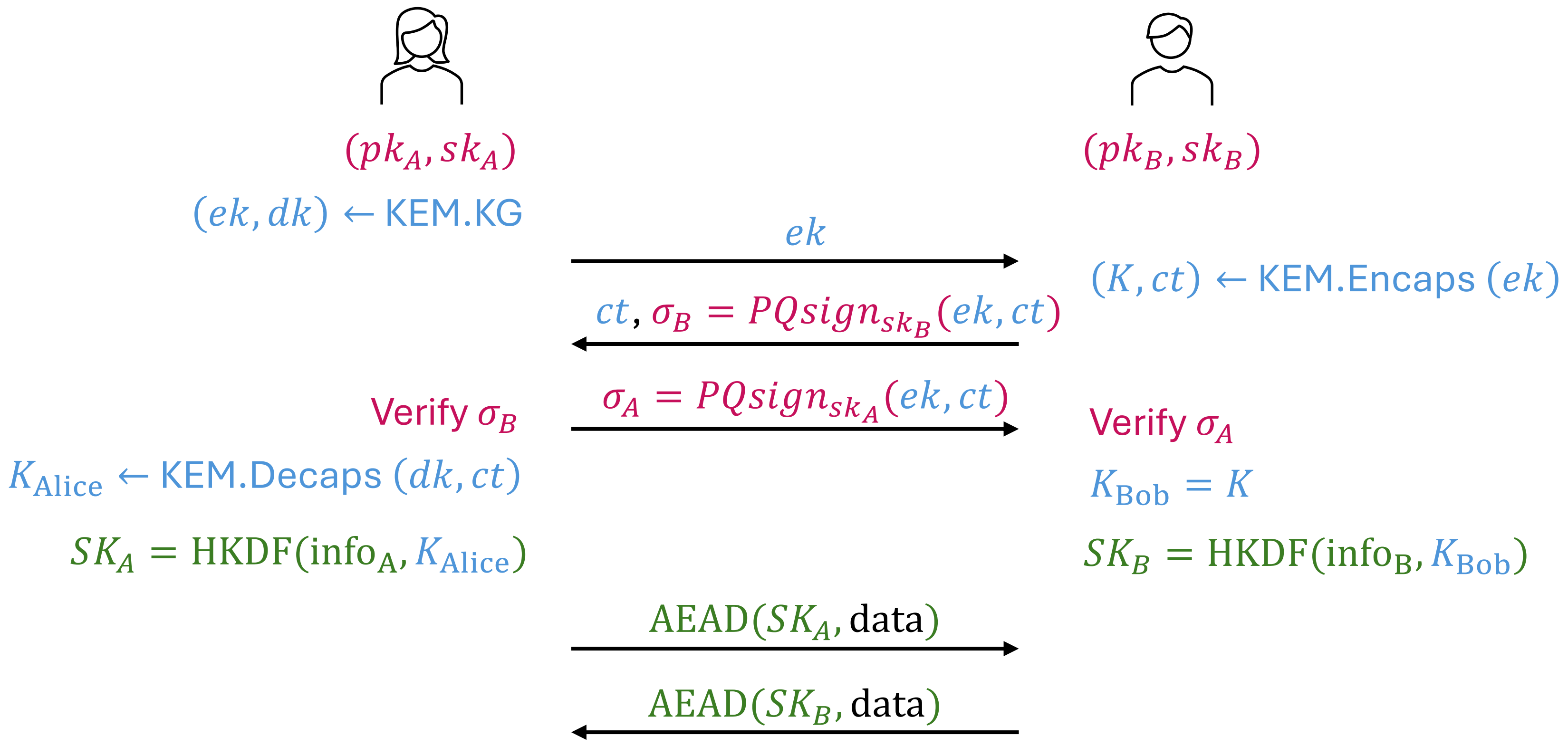
$K \leftarrow \text{KEM.Decaps}\,(dk, ct)$

$K_{\text{Alice}} = K$

$K_{\text{Bob}} = K$

Post-quantum TLS

- PQKEM + PQSign = PQTLS (demo)



(pk_A, sk_A)

(pk_B, sk_B)

$(ek, dk) \leftarrow$ KEM.KG

ek →

$(K, ct) \leftarrow$ KEM.Encaps (ek)

← $ct, \sigma_B = PQsign_{sk_B}(ek, ct)$

Verify σ_B

$\sigma_A = PQsign_{sk_A}(ek, ct)$ →

Verify σ_A

$K_{\mathrm{Alice}} \leftarrow$ KEM.Decaps (dk, ct)

$K_{\mathrm{Bob}} = K$

$SK_A = \mathrm{HKDF}(\mathrm{info}_\mathrm{A}, K_{\mathrm{Alice}})$

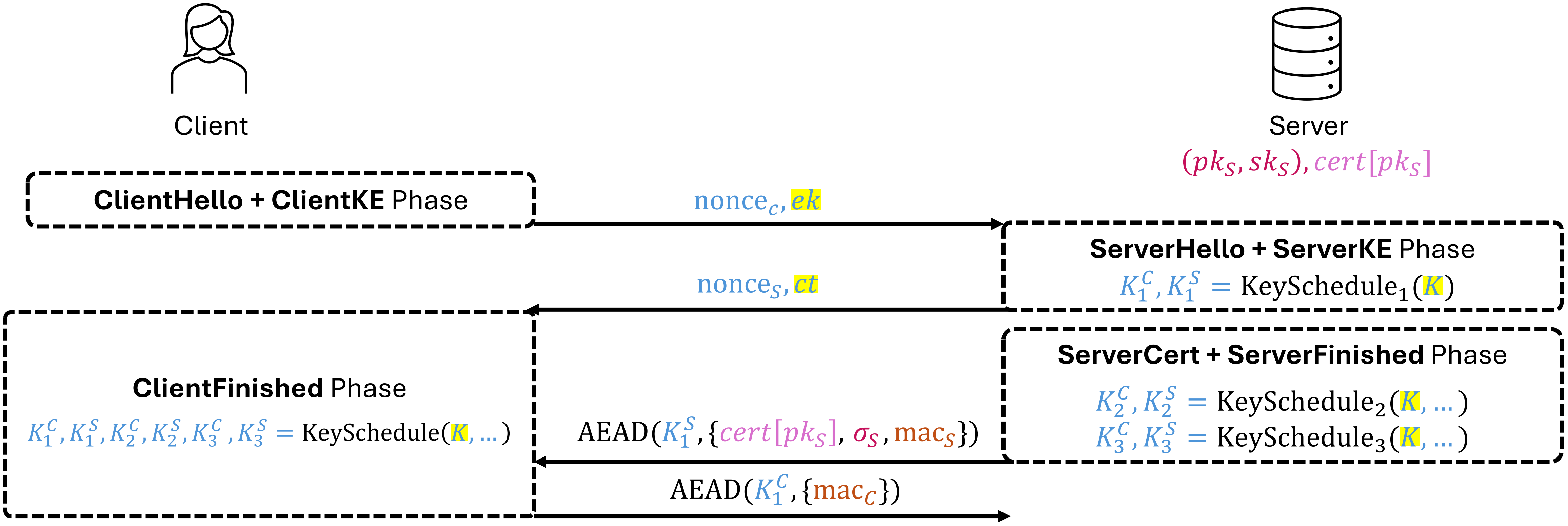
$SK_B = \mathrm{HKDF}(\mathrm{info}_\mathrm{B}, K_{\mathrm{Bob}})$

$\mathrm{AEAD}(SK_A, \mathrm{data})$ →

← $\mathrm{AEAD}(SK_B, \mathrm{data})$

Post-quantum TLS

- Integrate the PQTLS into the TLS 1.3 framework (Homework 1)



Client

Server

$(pk_S, sk_S), cert[pk_S]$

ClientHello + ClientKE Phase

nonce_C, ek

ServerHello + ServerKE Phase

$K_1^C, K_1^S = \text{KeySchedule}_1(K)$

nonce_S, ct

ServerCert + ServerFinished Phase

$K_2^C, K_2^S = \text{KeySchedule}_2(K, \dots)$

$K_3^C, K_3^S = \text{KeySchedule}_3(K, \dots)$

$\text{AEAD}(K_1^S, \{cert[pk_S], \sigma_S, \text{mac}_S\})$

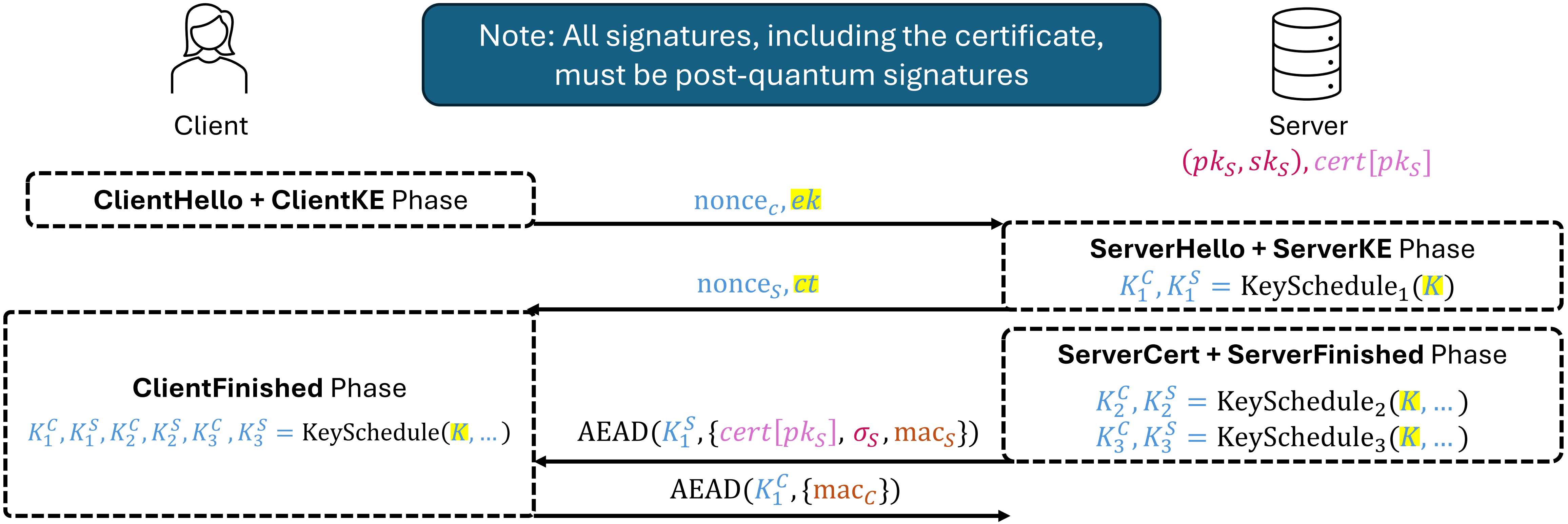
ClientFinished Phase

$K_1^C, K_1^S, K_2^C, K_2^S, K_3^C, K_3^S = \text{KeySchedule}(K, \dots)$

$\text{AEAD}(K_1^C, \{\text{mac}_C\})$

Post-quantum TLS

- Integrate the PQTLS into the TLS 1.3 framework (Homework 1)



Note: All signatures, including the certificate, must be post-quantum signatures

Client

Server

$(pk_S, sk_S), cert[pk_S]$

ClientHello + ClientKE Phase

nonce_C, ek

ServerHello + ServerKE Phase

$K_1^C, K_1^S = \text{KeySchedule}_1(K)$

nonce_S, ct

ServerCert + ServerFinished Phase

$K_2^C, K_2^S = \text{KeySchedule}_2(K, \dots)$

$K_3^C, K_3^S = \text{KeySchedule}_3(K, \dots)$

ClientFinished Phase

$K_1^C, K_1^S, K_2^C, K_2^S, K_3^C, K_3^S = \text{KeySchedule}(K, \dots)$

$\text{AEAD}(K_1^S, \{cert[pk_S], \sigma_S, \text{mac}_S\})$

$\text{AEAD}(K_1^C, \{\text{mac}_C\})$

Post-quantum TLS based on KEM

- In post-quantum cryptography, signature schemes are generally less efficient than KEM

- In TLS 1.3, we use signature to
  - (1) Certify the server’s public key (signed by the CA)
  - (2) Authenticate the serverKE message (signed by the server)

Post-quantum TLS based on KEM

- In post-quantum cryptography, signature schemes are generally less efficient than KEM

- In TLS 1.3, we use signature to
  - (1) Certify the server's long-term public key (signed by the CA) ...one-time price
  - (2) **Authenticate the serverKE message (signed by the server) ...needed in every session**

Post-quantum TLS based on KEM

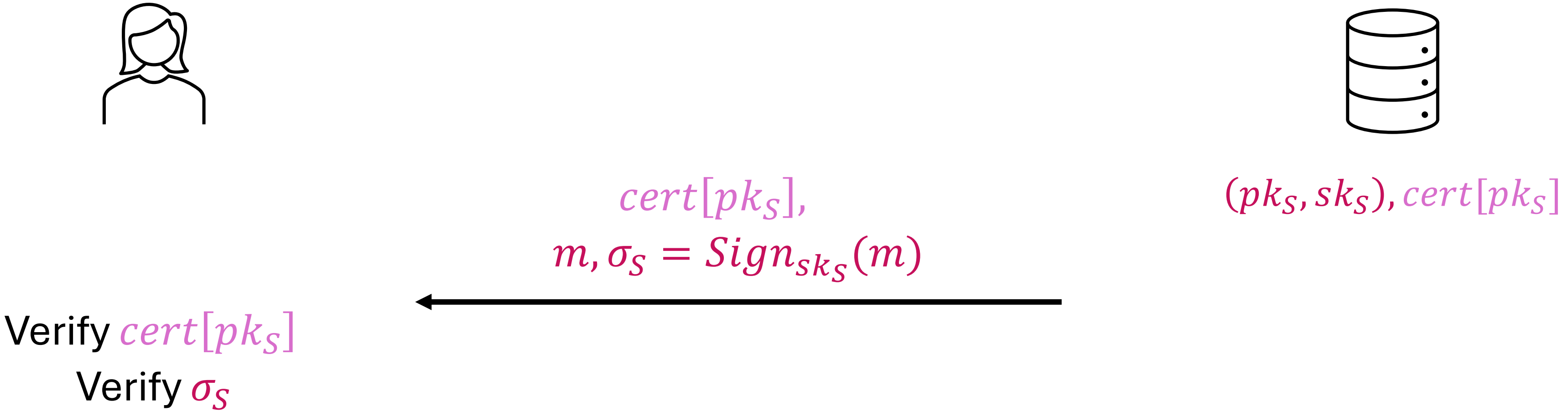
- In post-quantum cryptography, signature schemes are generally less efficient than KEM

- In TLS 1.3, we use signature to
  - (1) Certify the server's public key (signed by the CA) ...one-time price
  - (2) Authenticate the serverKE message (signed by the server) ...needed in every session

- **Can we replace the second part with KEM?**

Post-quantum TLS based on KEM

- Use KEM to authenticate messages



$(pk_S, sk_S), cert[pk_S]$

$cert[pk_S]$,
$m, \sigma_S = Sign_{sk_S}(m)$

Verify $cert[pk_S]$
Verify σ_S

Only the sk owner can generate the signature
=> If σ_S verifies successfully, then m is from the server

Post-quantum TLS based on KEM

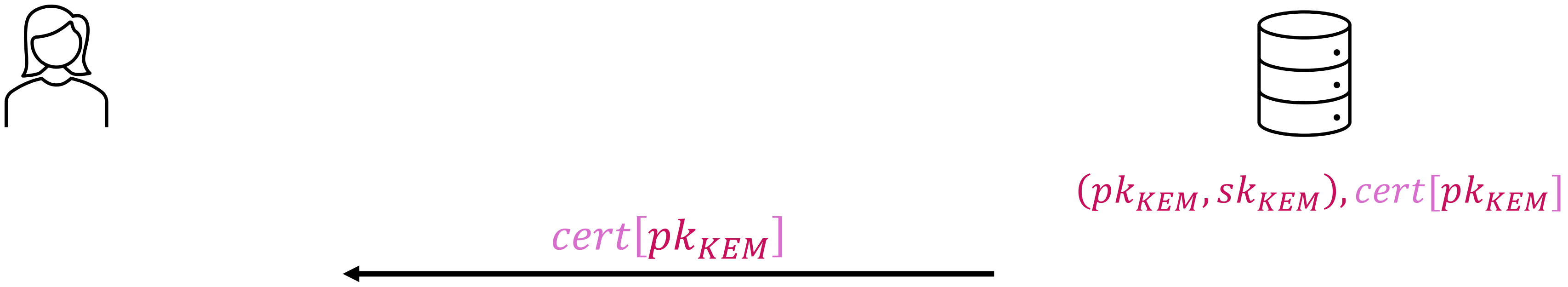
- Use KEM to authenticate messages

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

When using KEM:
having the sk ⇔ Decrypt ciphertexts

Post-quantum TLS based on KEM

- Use KEM to authenticate messages

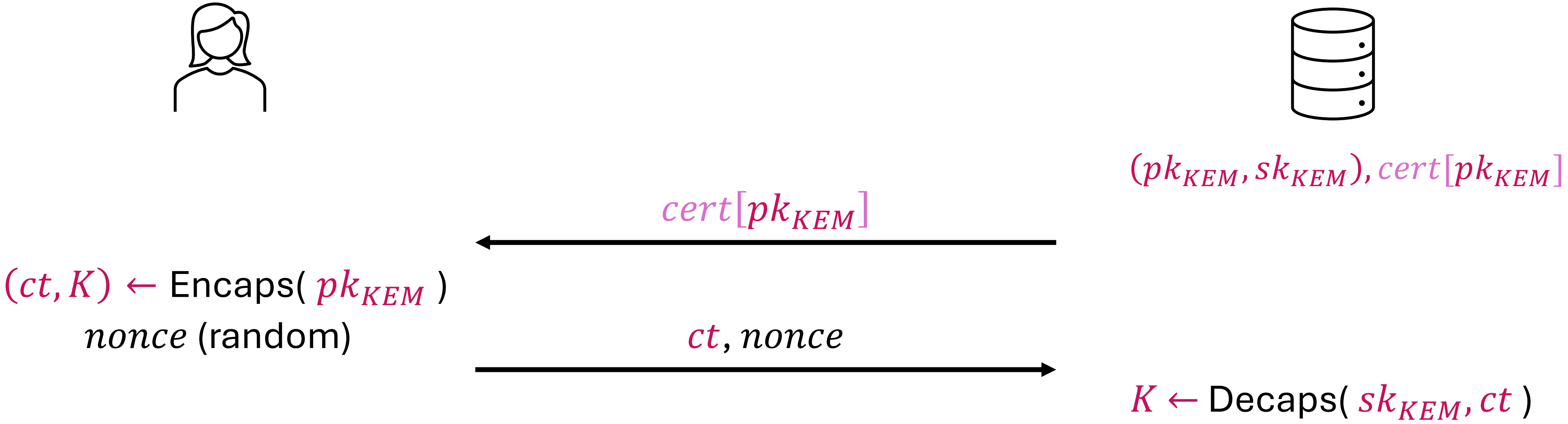


$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

$cert[pk_{KEM}]$

Post-quantum TLS based on KEM

- Use KEM to authenticate messages



$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

$cert[pk_{KEM}]$

$(ct, K) \leftarrow$ Encaps(pk_{KEM})

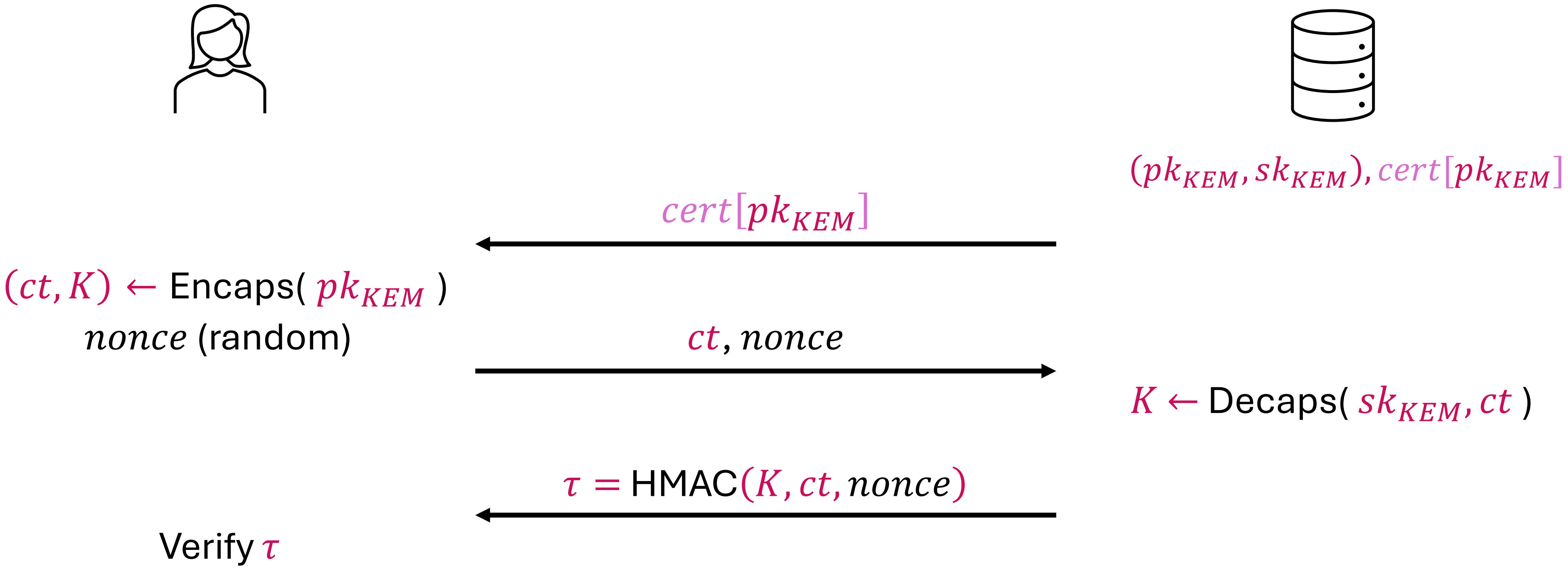
$nonce$ (random)

$ct, nonce$

$K \leftarrow$ Decaps(sk_{KEM}, ct)

Post-quantum TLS based on KEM

- Use KEM to authenticate messages



$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

$cert[pk_{KEM}]$

$(ct, K) \leftarrow$ Encaps(pk_{KEM})

$nonce$ (random)

$ct, nonce$

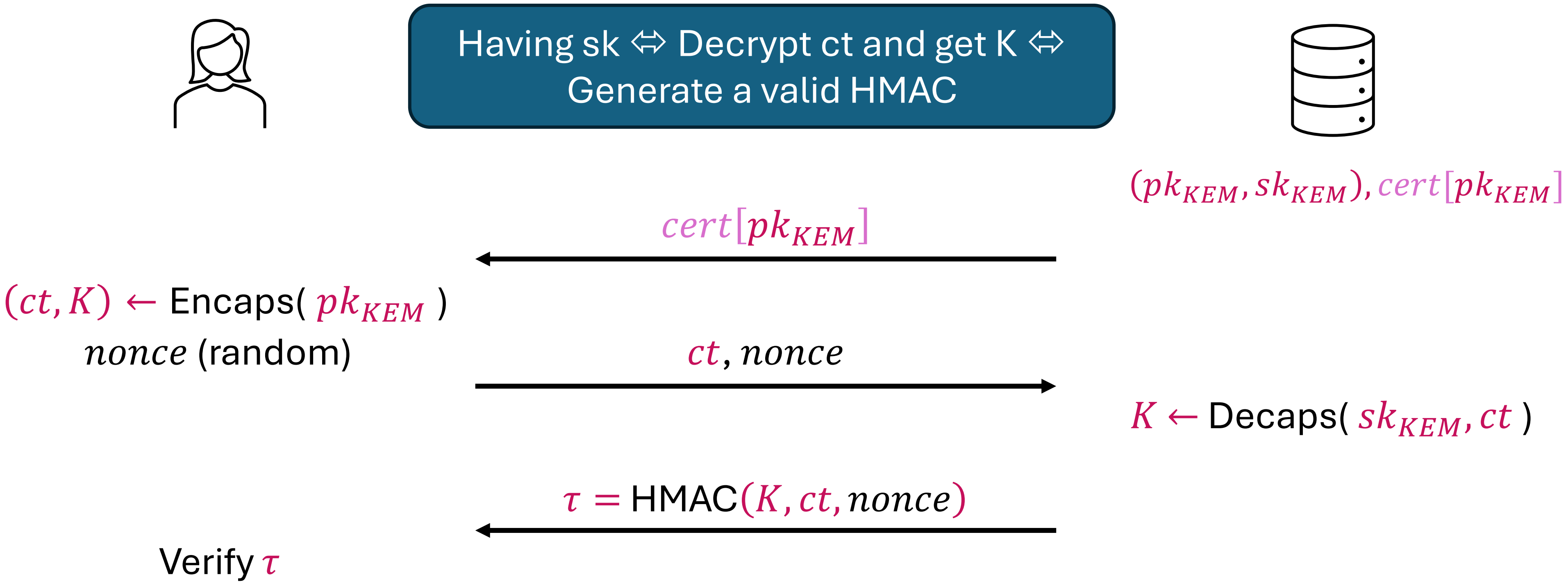
$K \leftarrow$ Decaps(sk_{KEM}, ct)

$\tau =$ HMAC$(K, ct, nonce)$

Verify τ

Post-quantum TLS based on KEM

- Use KEM to authenticate messages



Having sk ⇔ Decrypt ct and get K ⇔ Generate a valid HMAC

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

$cert[pk_{KEM}]$

$(ct, K) \leftarrow$ Encaps(pk_{KEM})

$nonce$ (random)

$ct, nonce$

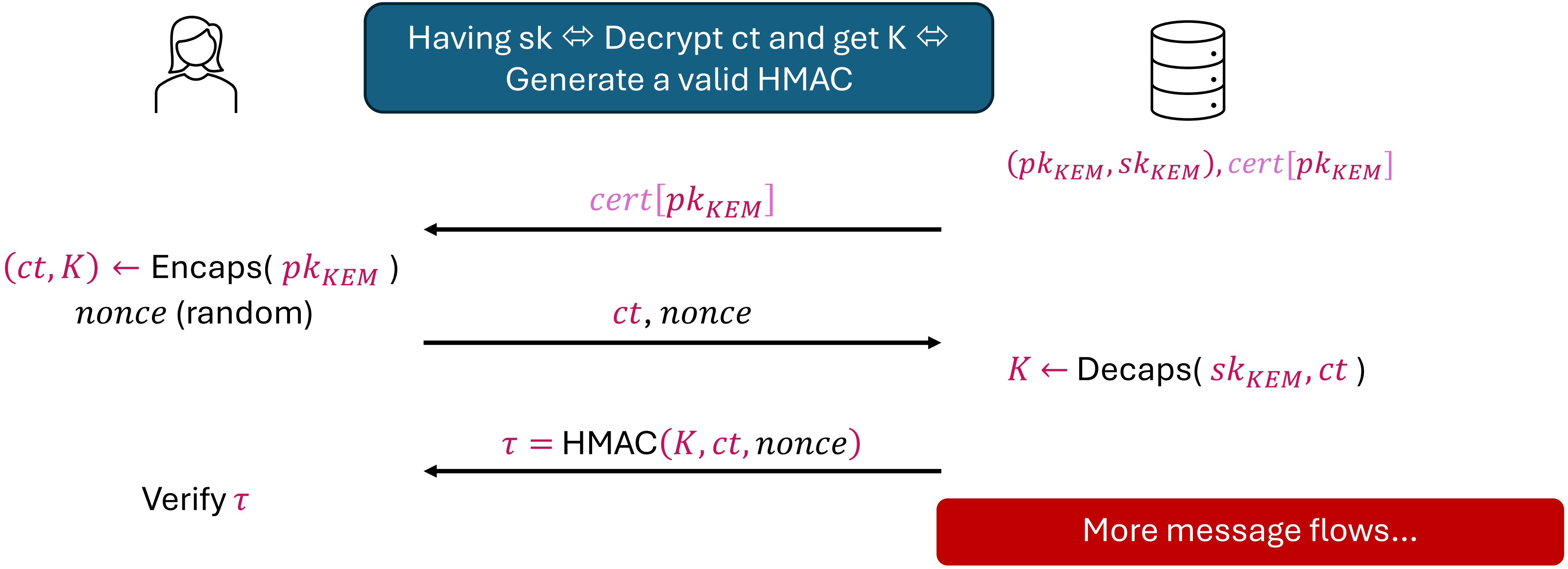
$K \leftarrow$ Decaps(sk_{KEM}, ct)

$\tau =$ HMAC$(K, ct, nonce)$

Verify τ

Post-quantum TLS based on KEM

- Use KEM to authenticate messages



Having sk ⇔ Decrypt ct and get K ⇔ Generate a valid HMAC

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

$cert[pk_{KEM}]$

$(ct, K) \leftarrow$ Encaps(pk_{KEM})

$nonce$ (random)

$ct, nonce$

$K \leftarrow$ Decaps(sk_{KEM}, ct)

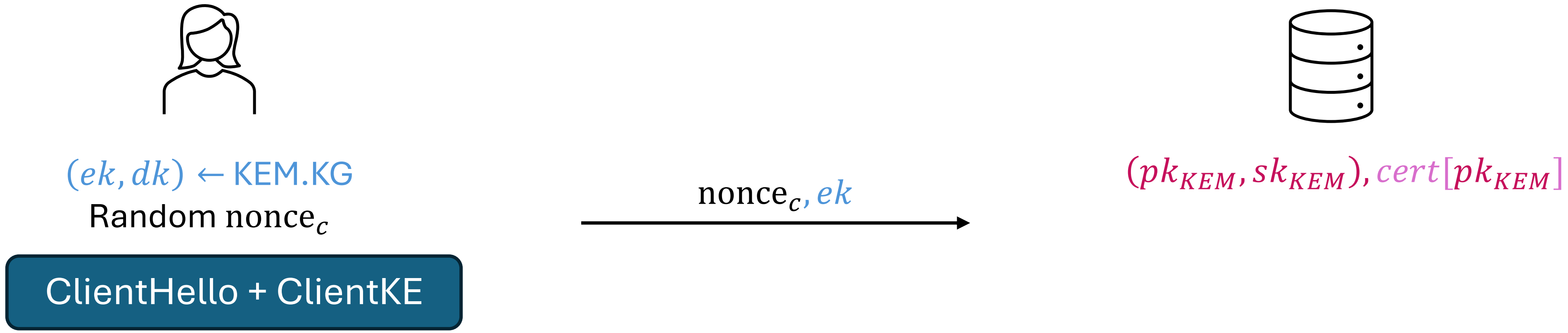
$\tau =$ HMAC$(K, ct, nonce)$

Verify τ

More message flows…

KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



$(ek, dk) \leftarrow$ KEM.KG

Random nonce_c

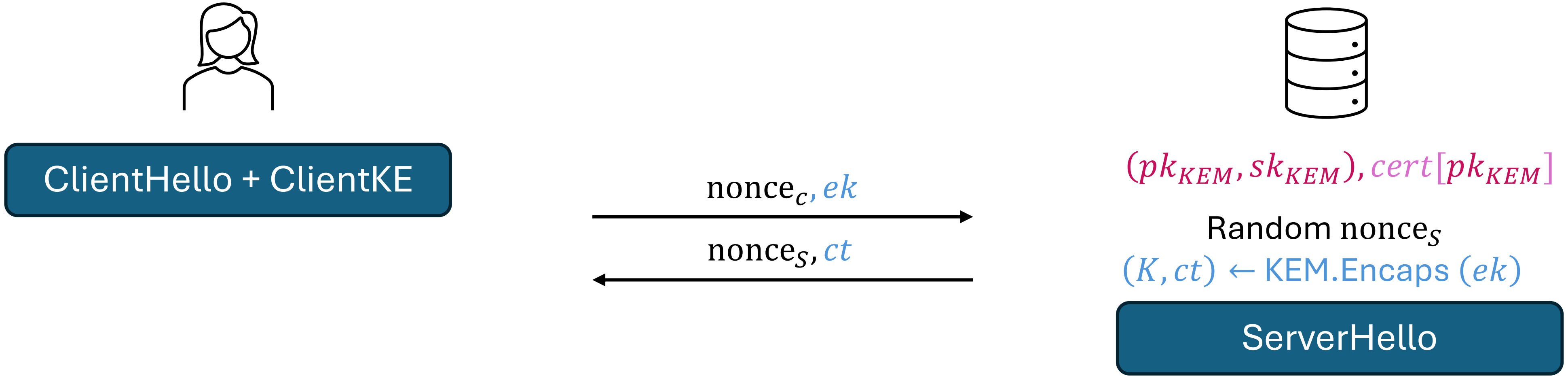
ClientHello + ClientKE

nonce_c, ek

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



ClientHello + ClientKE

nonce_c, ek

nonce_S, ct

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

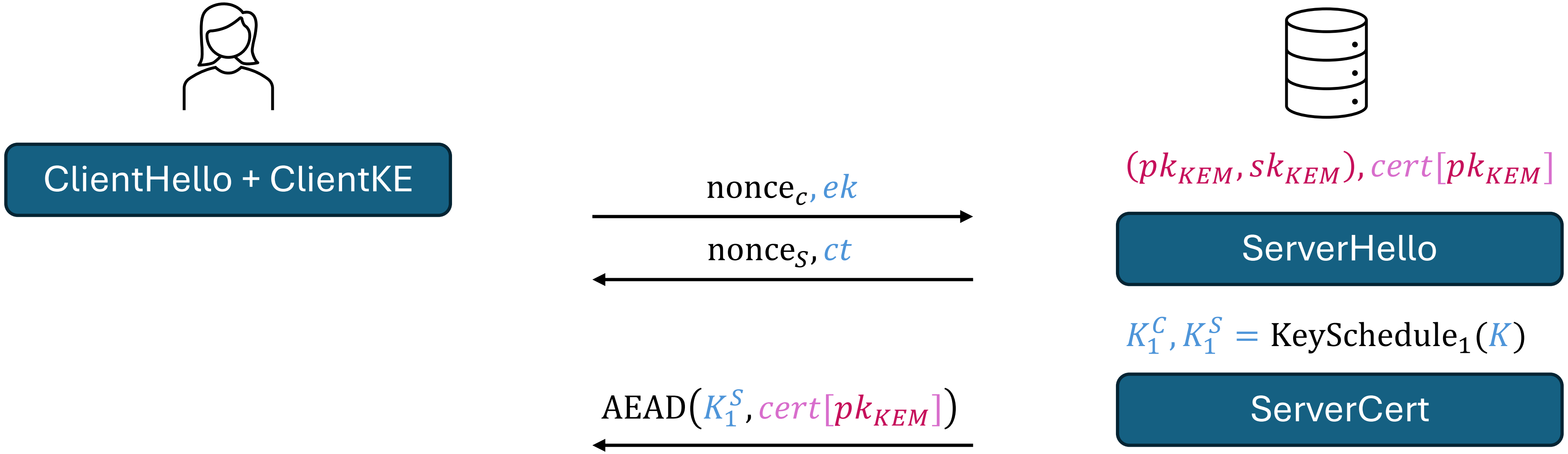
Random nonce_S

$(K, ct) \leftarrow \text{KEM.Encaps}\ (ek)$

ServerHello

KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



ClientHello + ClientKE

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

nonce_c, ek

ServerHello

nonce_S, ct

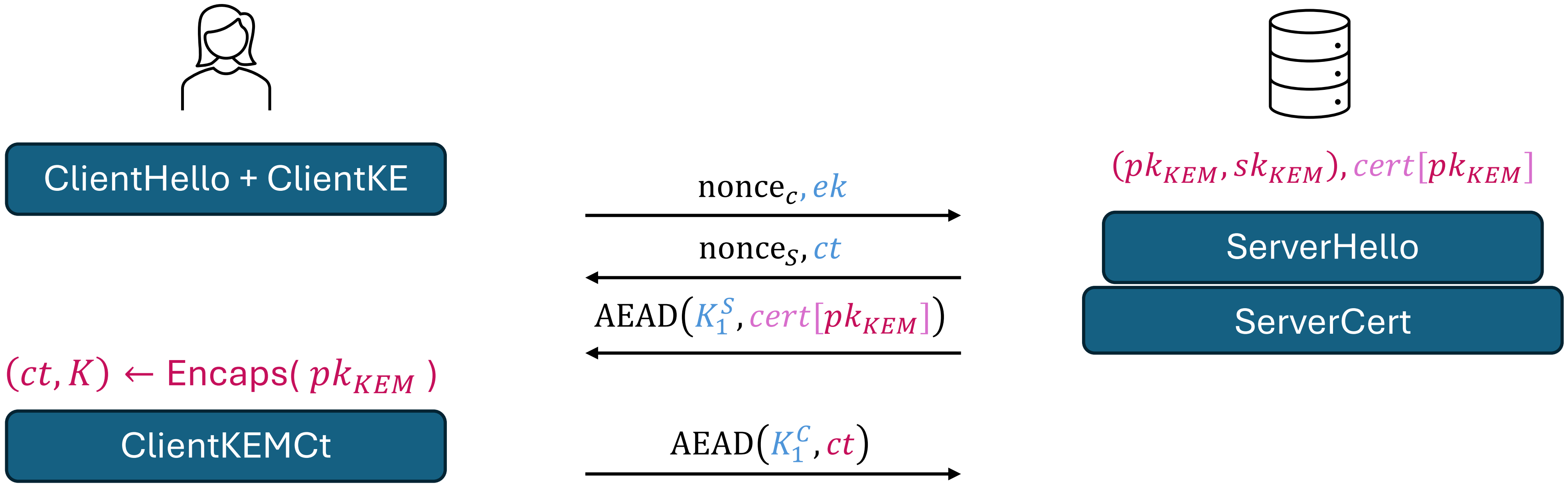
$K_1^C, K_1^S = \text{KeySchedule}_1(K)$

ServerCert

$\text{AEAD}(K_1^S, cert[pk_{KEM}])$

KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

ClientHello + ClientKE

nonce_c, ek

ServerHello

nonce_S, ct

ServerCert

$\text{AEAD}(K_1^S, cert[pk_{KEM}])$

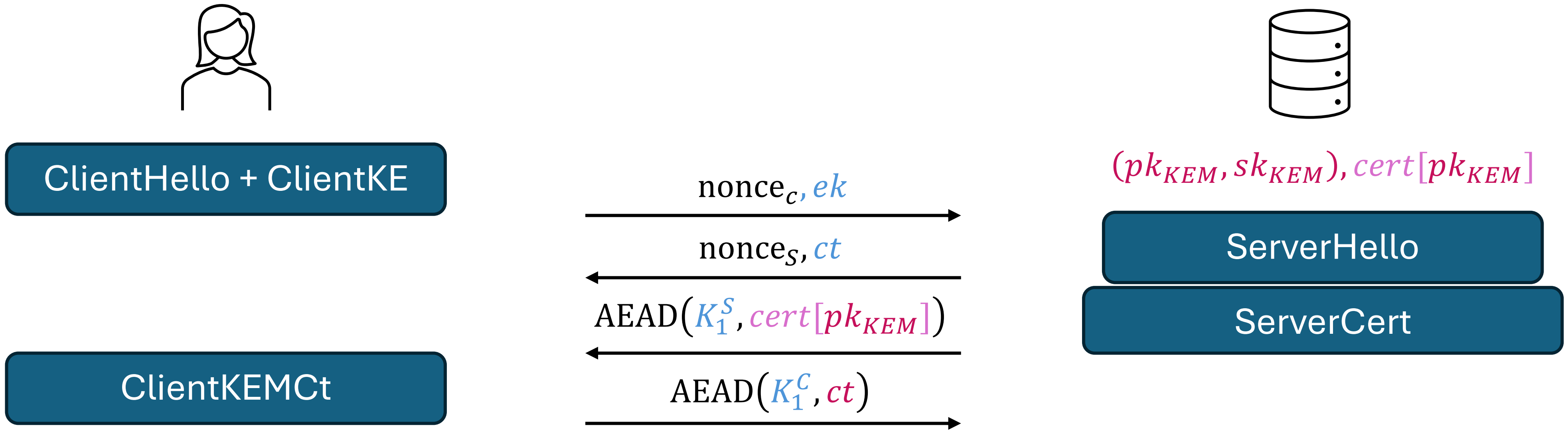
$(ct, K) \leftarrow$ Encaps(pk_{KEM})

ClientKEMCt

$\text{AEAD}(K_1^C, ct)$

KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



ClientHello + ClientKE

nonce_c, ek

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

ServerHello

ServerCert

nonce_S, ct

$\text{AEAD}(K_1^S, cert[pk_{KEM}])$

ClientKEMCt

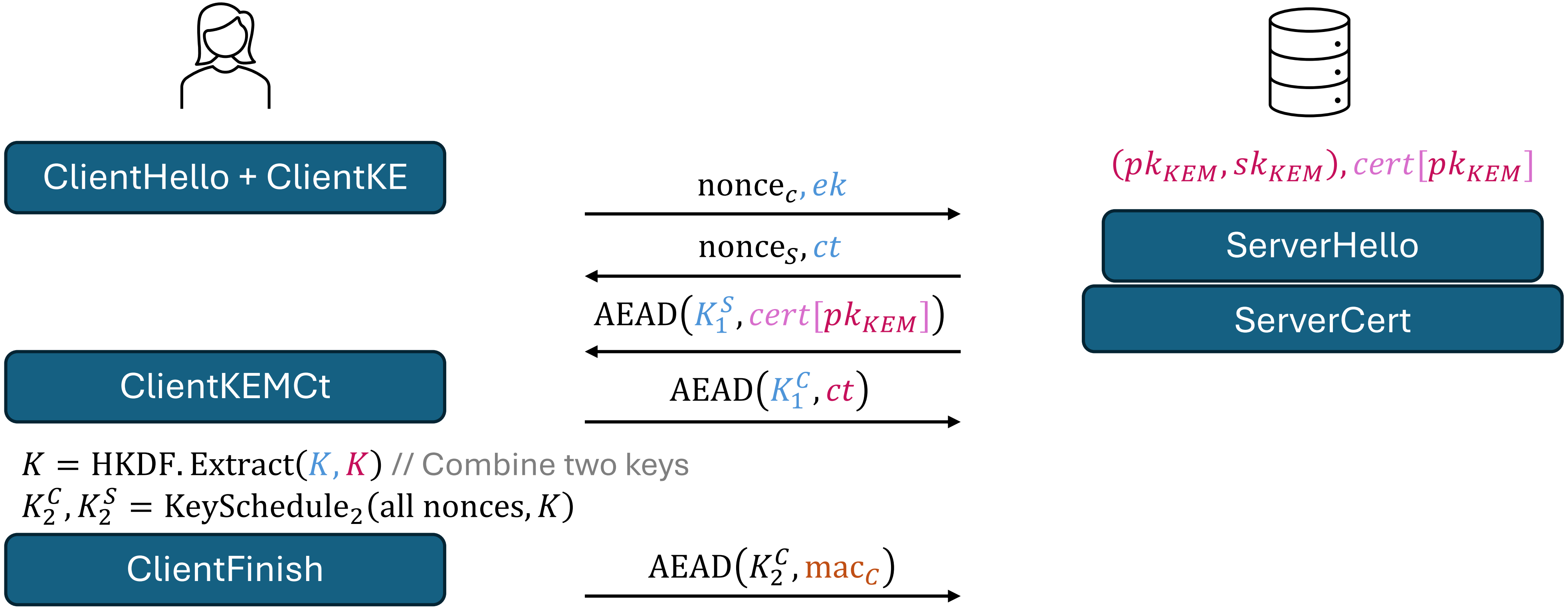
$\text{AEAD}(K_1^C, ct)$

$K = \text{HKDF.Extract}(K, K)$ // Combine two keys

$K_2^C, K_2^S = \text{KeySchedule}_2(\text{all nonces}, K)$

KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



ClientHello + ClientKE

nonce_c, ek

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

ServerHello

ServerCert

nonce_S, ct

$\text{AEAD}(K_1^S, cert[pk_{KEM}])$

ClientKEMCt

$\text{AEAD}(K_1^C, ct)$

$K = \text{HKDF.Extract}(K, K)$ // Combine two keys

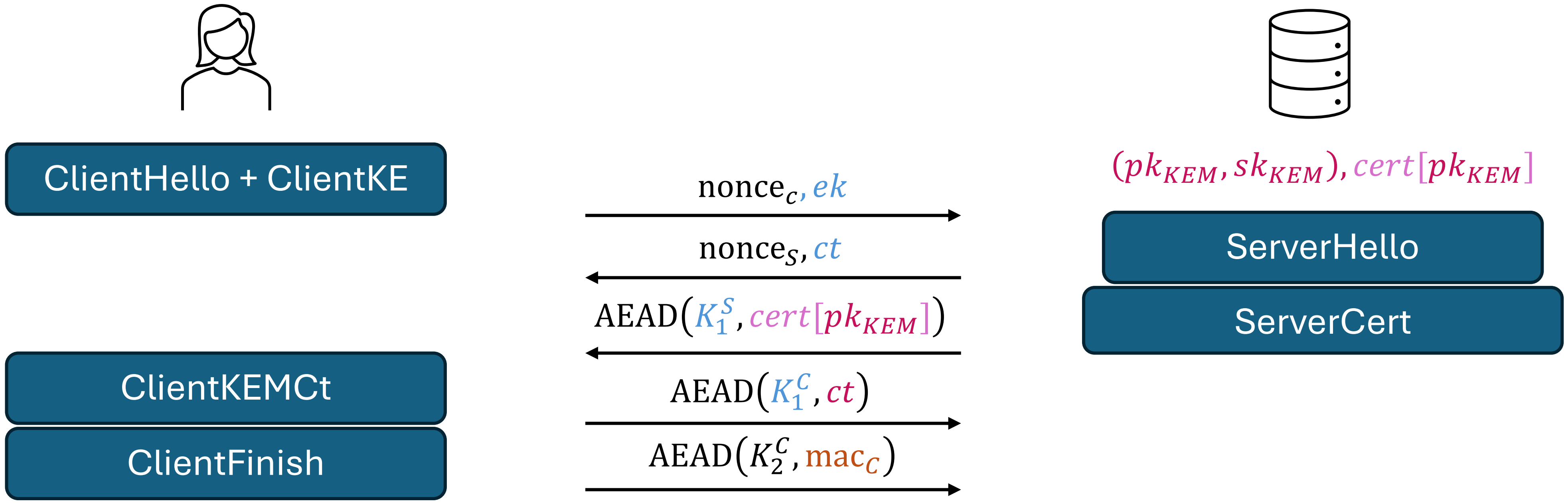
$K_2^C, K_2^S = \text{KeySchedule}_2(\text{all nonces}, K)$

ClientFinish

$\text{AEAD}(K_2^C, \text{mac}_C)$

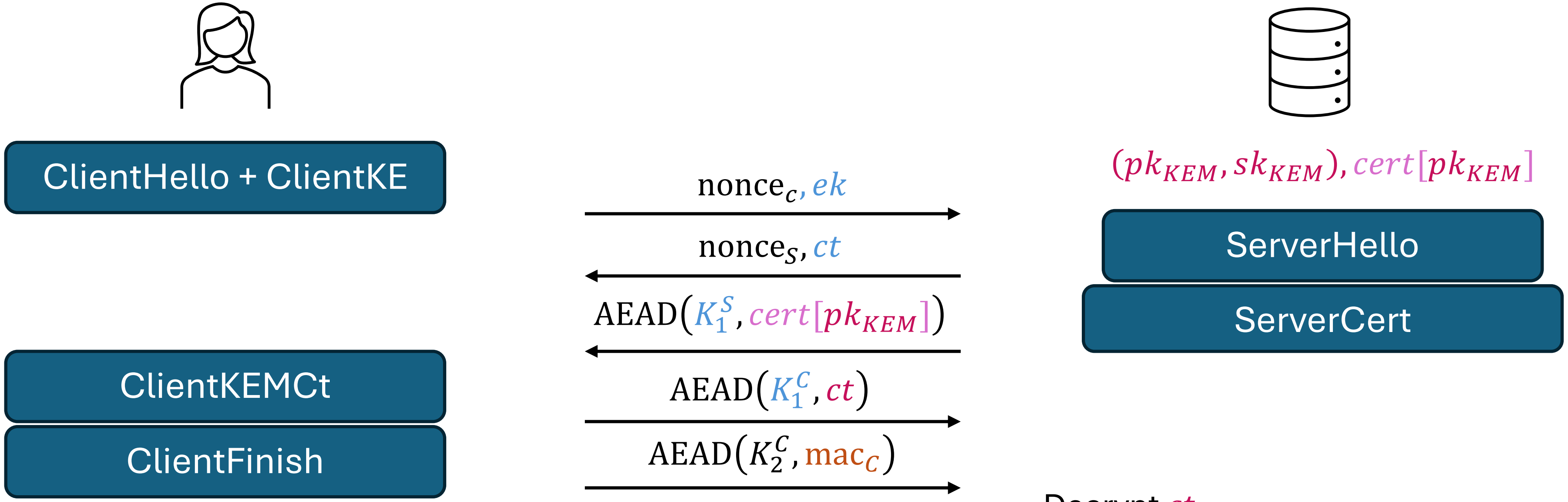
KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



KEM-TLS

- **KEM-TLS (demo):** Integrate the KEM authentication into the TLS framework



ClientHello + ClientKE

nonce_C, ek

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

ServerHello

ServerCert

nonce_S, ct

$\text{AEAD}(K_1^S, cert[pk_{KEM}])$

ClientKEMCt

ClientFinish

$\text{AEAD}(K_1^C, ct)$

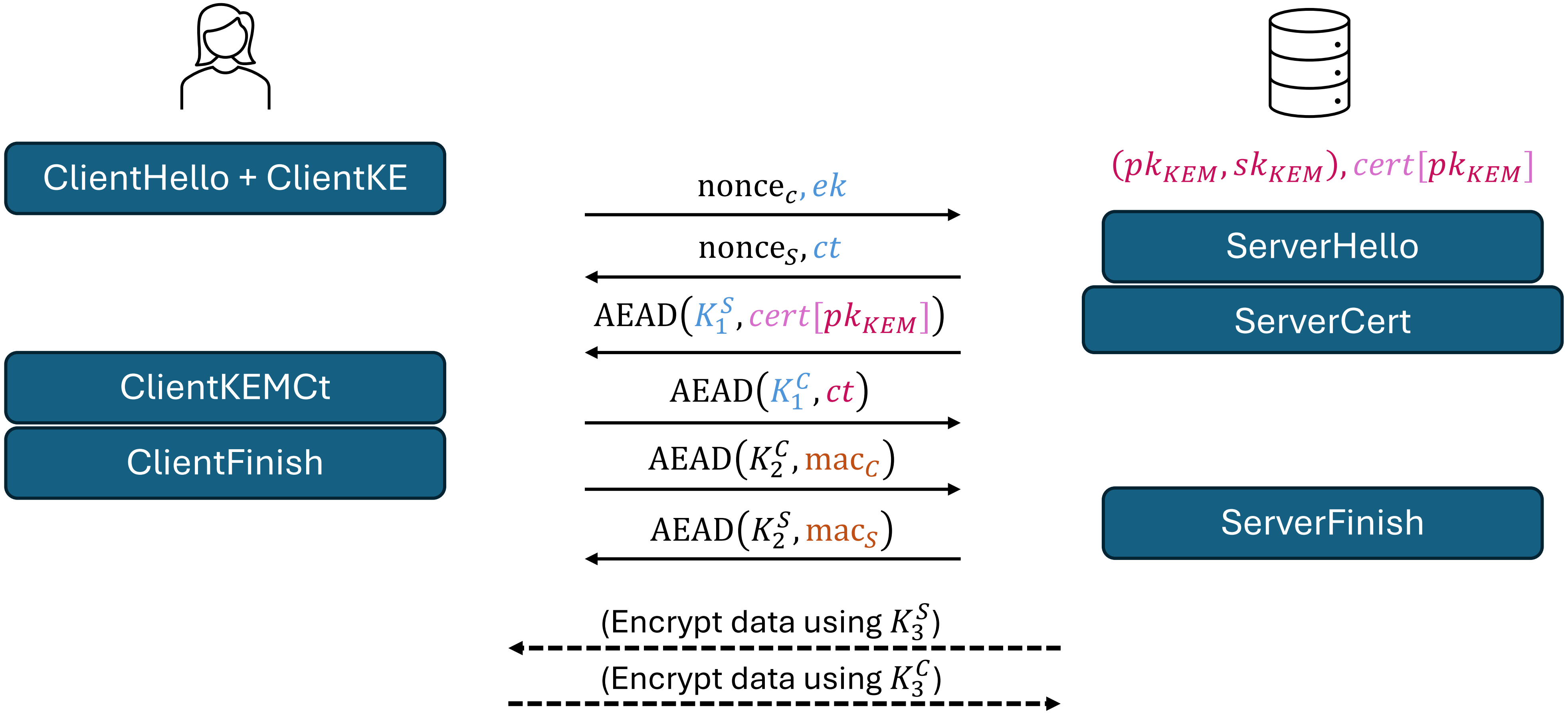
$\text{AEAD}(K_2^C, \text{mac}_C)$

Decrypt ct

$K = \text{HKDF.Extract}(K, K)$ // Combine two keys

$K_2^C, K_2^S = \text{KeySchedule}_2(\text{all nonces}, K)$

KEM-TLS



ClientHello + ClientKE

$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

nonce_c, ek

ServerHello

nonce_S, ct

ServerCert

$\text{AEAD}(K_1^S, cert[pk_{KEM}])$

ClientKEMCt

$\text{AEAD}(K_1^C, ct)$

ClientFinish

$\text{AEAD}(K_2^C, \text{mac}_C)$

ServerFinish

$\text{AEAD}(K_2^S, \text{mac}_S)$

(Encrypt data using K_3^S)

(Encrypt data using K_3^C)

Homework

- 1. Find Kyber (or ML-KEM) and Dilithium (or ML-DSA) implementations in your programming language. Then implement the PQ-TLS protocol.
- 2. Find Kyber (or ML-KEM) implementations in your programming language, then implement the KEM-TLS protocol.
- 3. Compare the efficiency between your PQ-TLS and KEM-TLS implementations.
- **DDL: Jan 09th, 2026 at 23:59**
- Rust:
  - https://docs.rs/ml-kem/latest/ml_kem/
  - https://docs.rs/ml-dsa/latest/ml_dsa/
- Python:
  - https://github.com/GiacomoPope/kyber-py
  - https://github.com/GiacomoPope/dilithium-py

Reference

- PQ-TLS: https://cic.iacr.org/p/1/2/6
- KEM-TLS: https://kemtls.org/
- Crystal-Kyber and ML-KEM: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf
- Crystal-Dilithium and ML-DSA: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf