

# 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  $\text{PKE} = (\text{KG}, \text{Enc}, \text{Dec})$  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.  $(pk, sk) \leftarrow \text{KG}$
2.  $prk \leftarrow \{0,1\}^L$
3.  $pk := pk$
4.  $sk := (prk, sk)$

$\text{Encaps}(pk = pk)$ :

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

$\text{Decaps}(sk = (prk, sk), c = c)$ :

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

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

KeyGen:

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

Encaps( $pk = pk$ ):

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

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

- **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.  $(pk, sk) \leftarrow KG$
2.  $prk \leftarrow \{0,1\}^L$
3.  $pk := pk$
4.  $sk := (prk, sk)$

Encaps( $pk = pk$ ):

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

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

- **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:

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

Encaps( $pk = pk$ ):

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

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

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

KeyGen:

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

Encaps( $pk = pk$ ):

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

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 **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

$\text{Decaps}(sk = (prk, sk), c = c)$ :

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

$\text{Decaps}(sk = (prk, sk), c = c)$ :

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

$\text{CT-Decaps}(sk = (prk, sk), c = c)$ :

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

$\text{Decaps}(sk = (prk, sk), c = c)$ :

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  $H(pk, prk, c)$

# 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

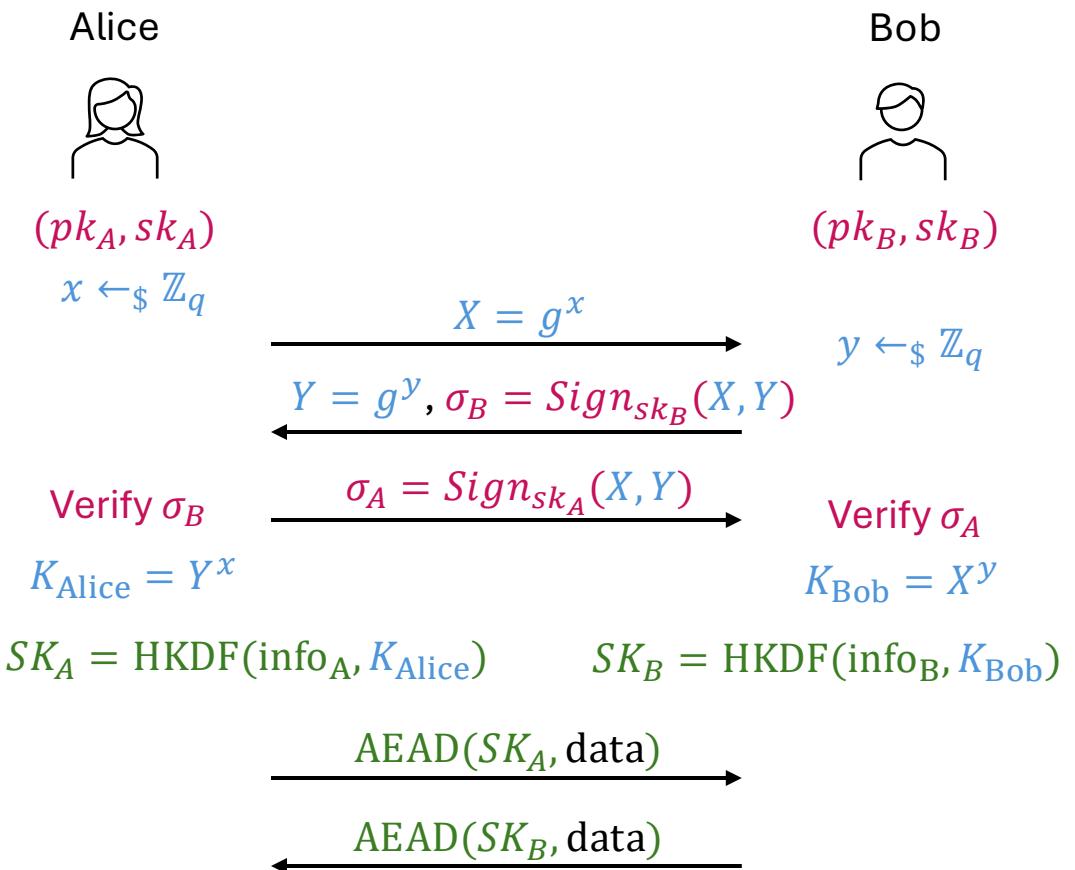
- Post-quantum secure KEM
  - Design a IND-CPA PKE scheme first
  - Use the FO transform (or its variants) to get an IND-CCA KEM scheme
- Post-quantum secure Signature
  - More complex structures...

# 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?**

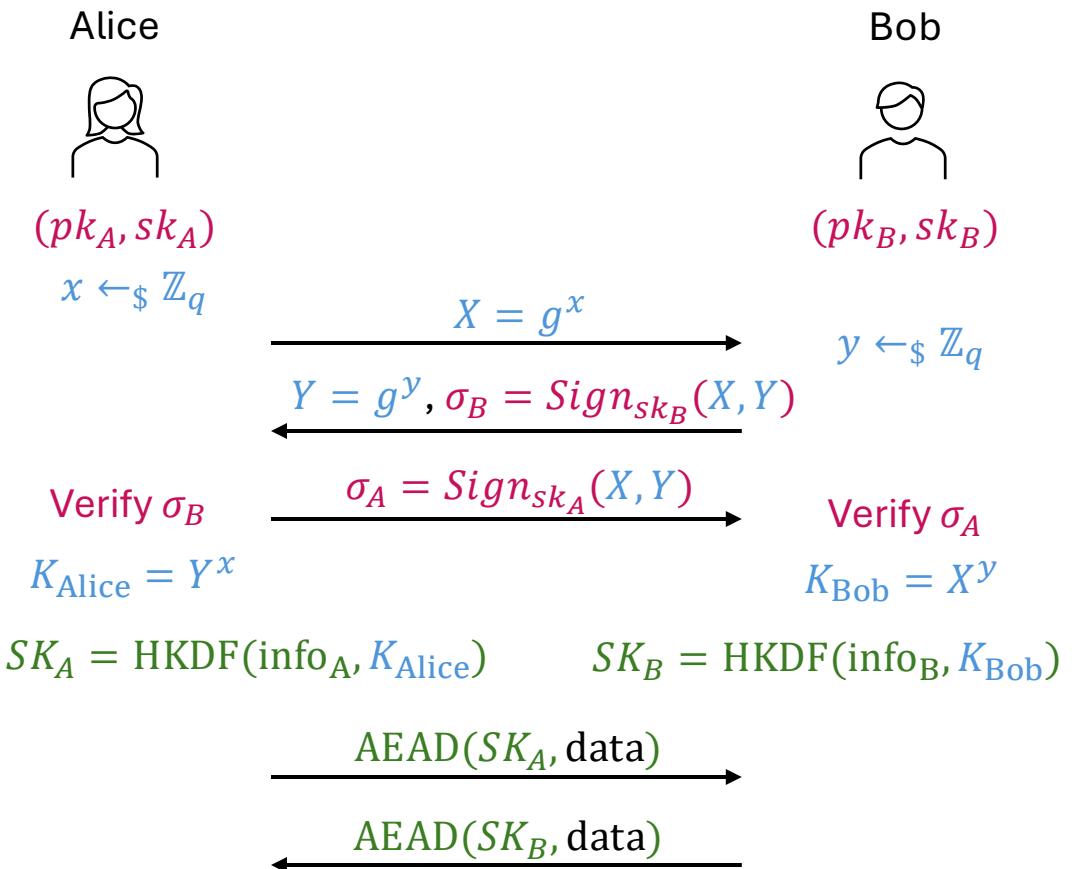
# Post-quantum TLS

- The signed DH protocol



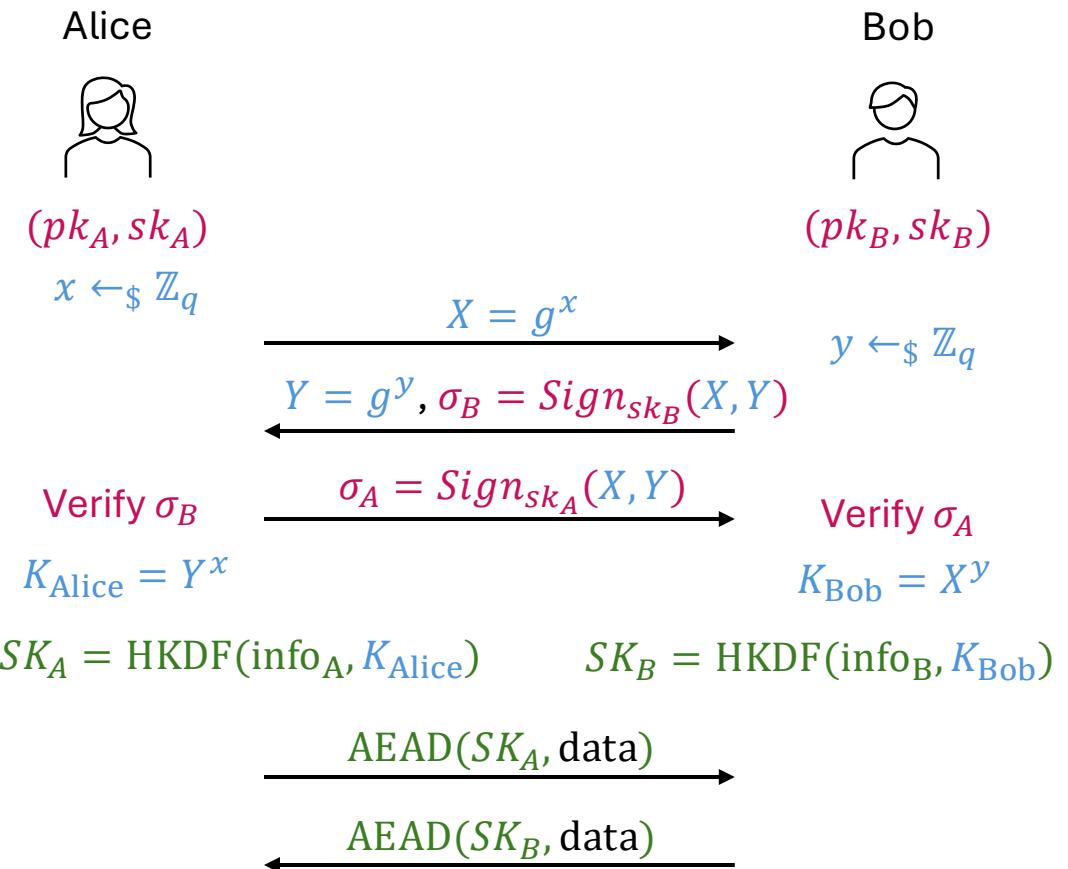
# Post-quantum TLS

- The signed DH protocol
- To make it post-quantum secure, which parts should we change?



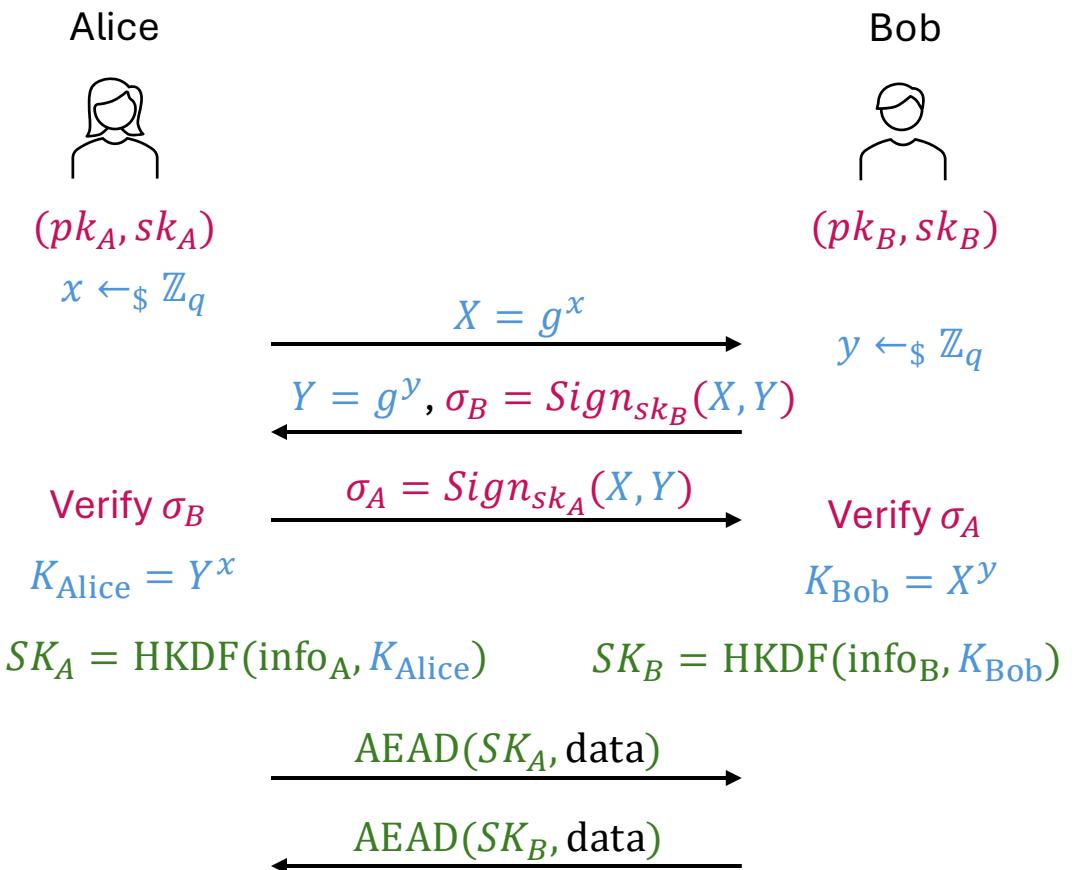
# Post-quantum TLS

- 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**



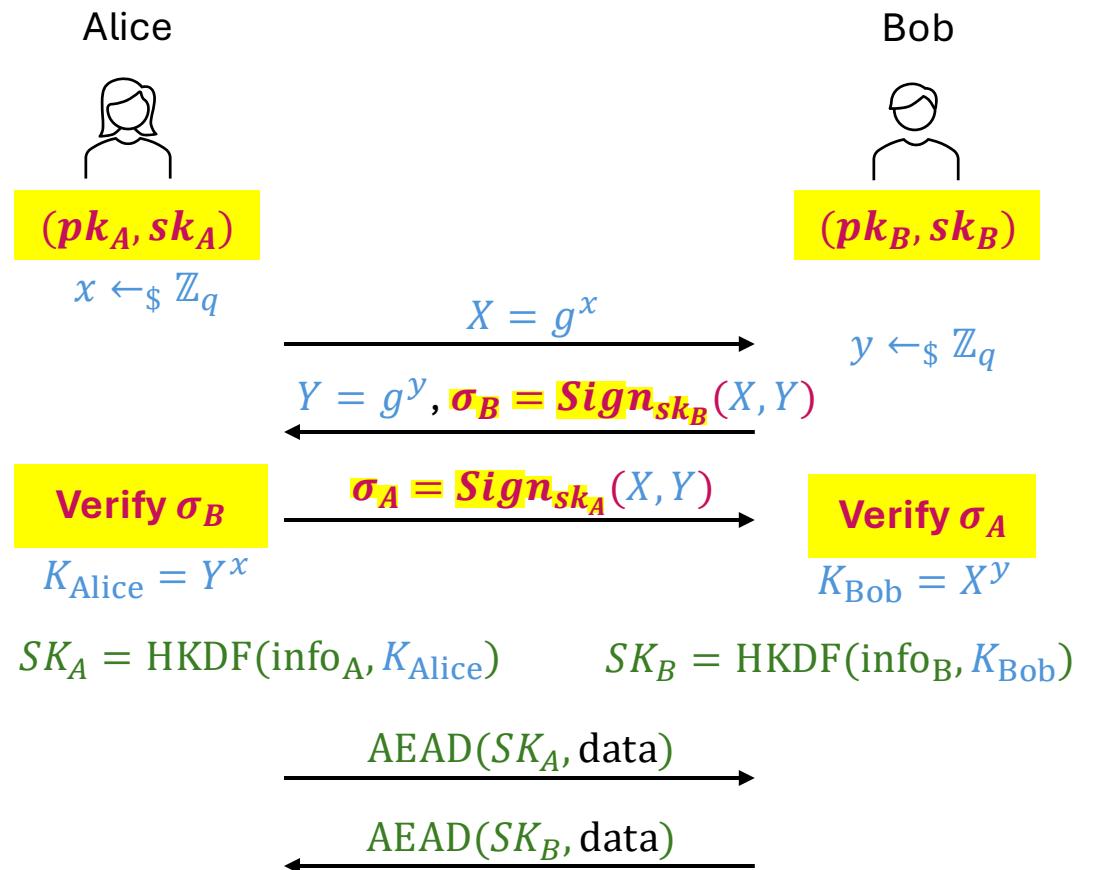
# Post-quantum TLS

- 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)



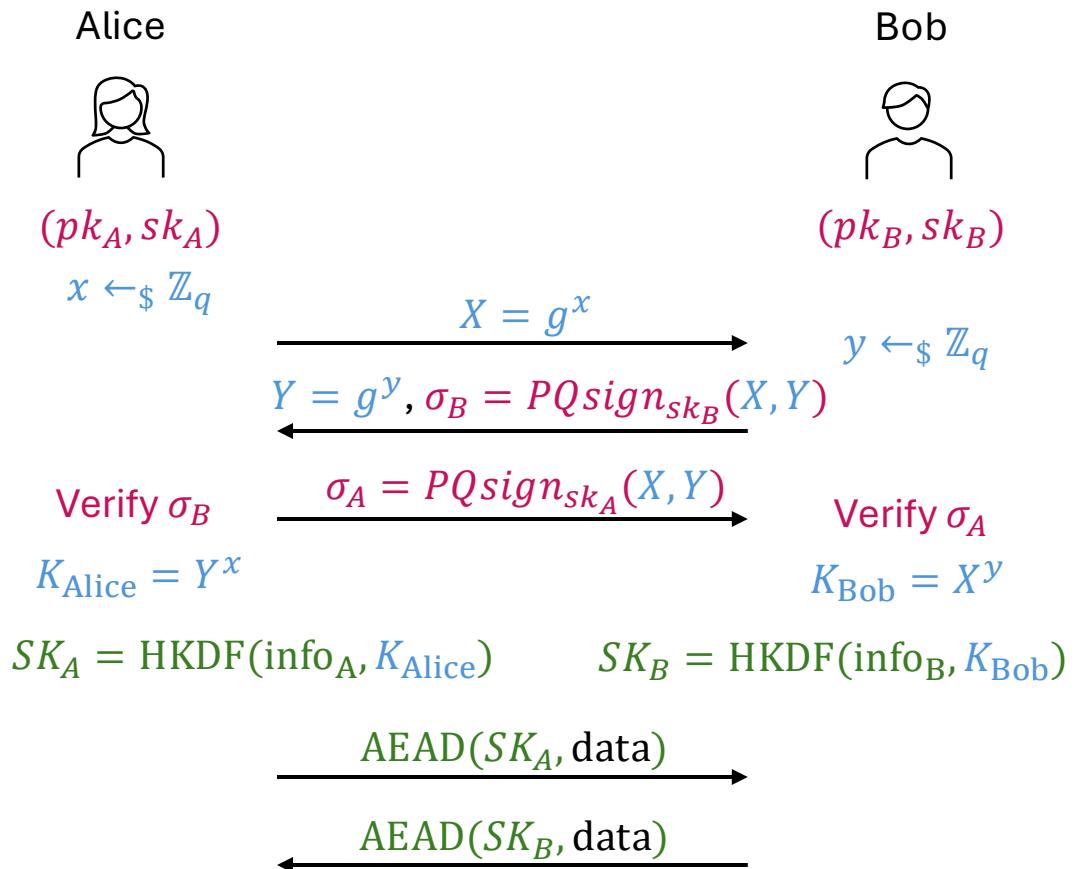
# Post-quantum TLS

- 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)



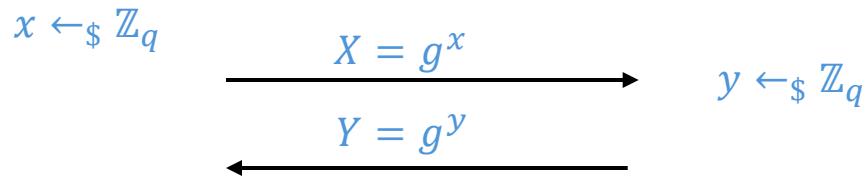
# Post-quantum TLS

- The DH + PQsign protocol
- How can we replace the DH part?



# Post-quantum TLS

- DH key exchange

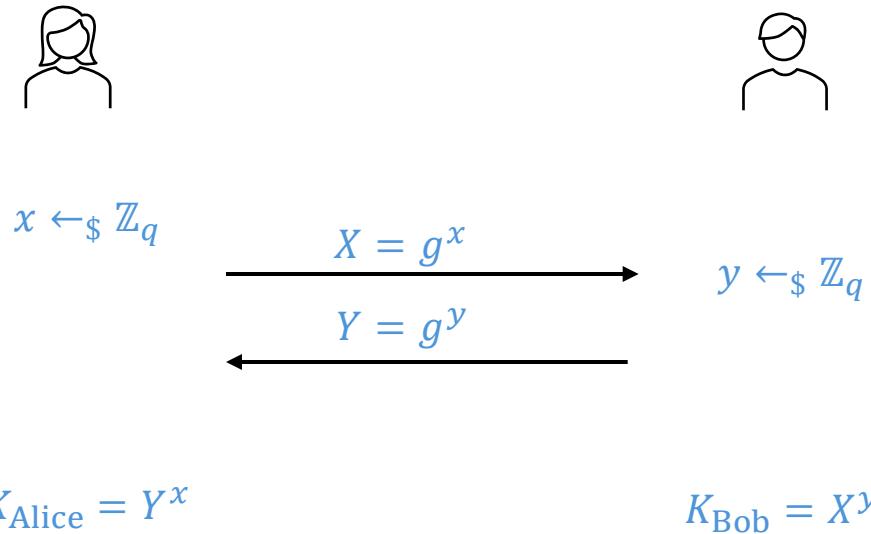


$$K_{\text{Alice}} = Y^x$$

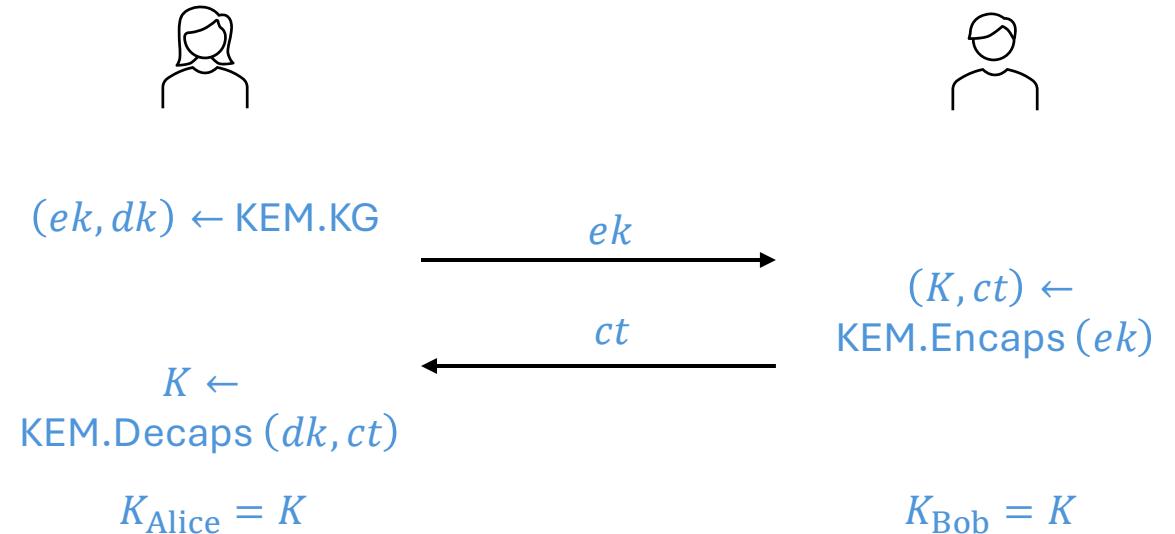
$$K_{\text{Bob}} = X^y$$

# Post-quantum TLS

- DH key exchange

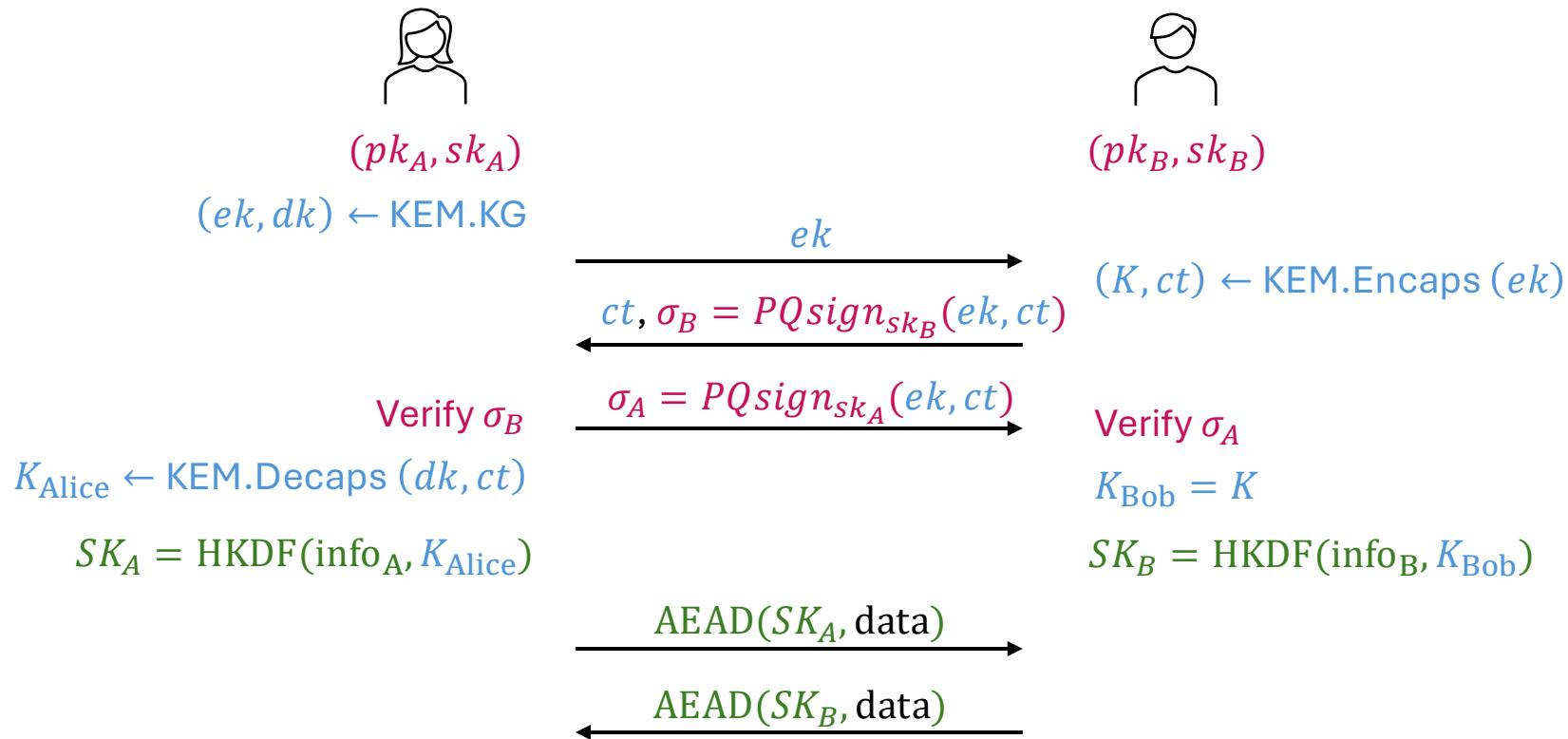


- KEM-based key exchange



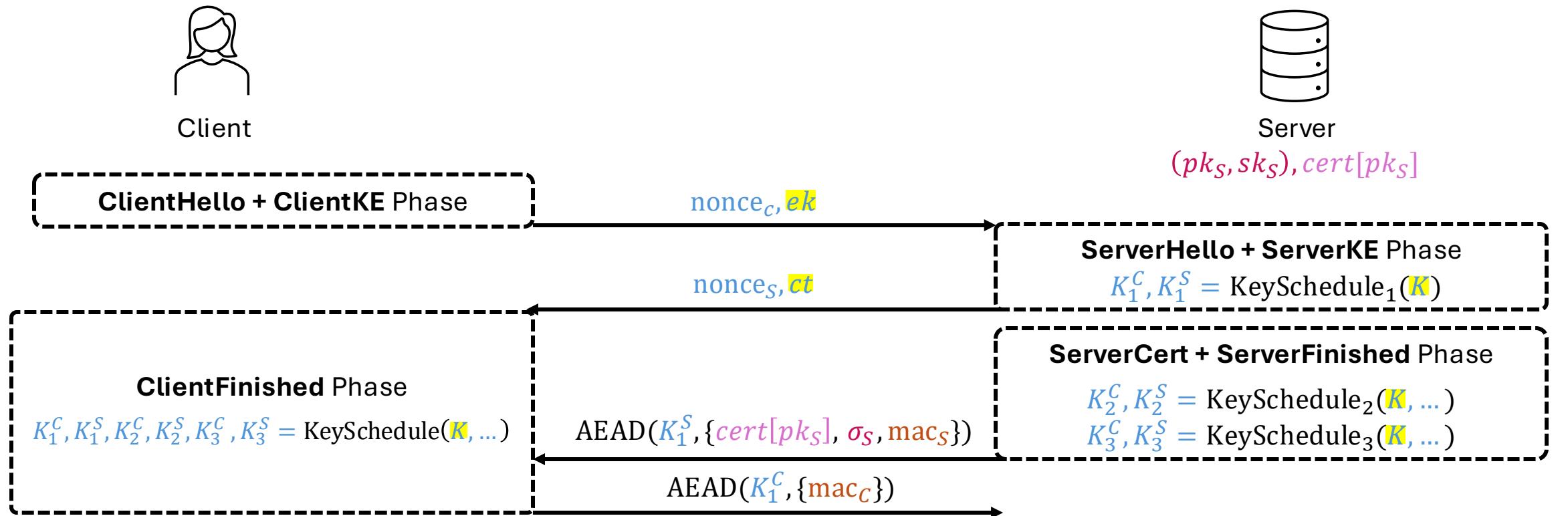
# Post-quantum TLS

- PQKEM + PQSign = PQLS (demo)



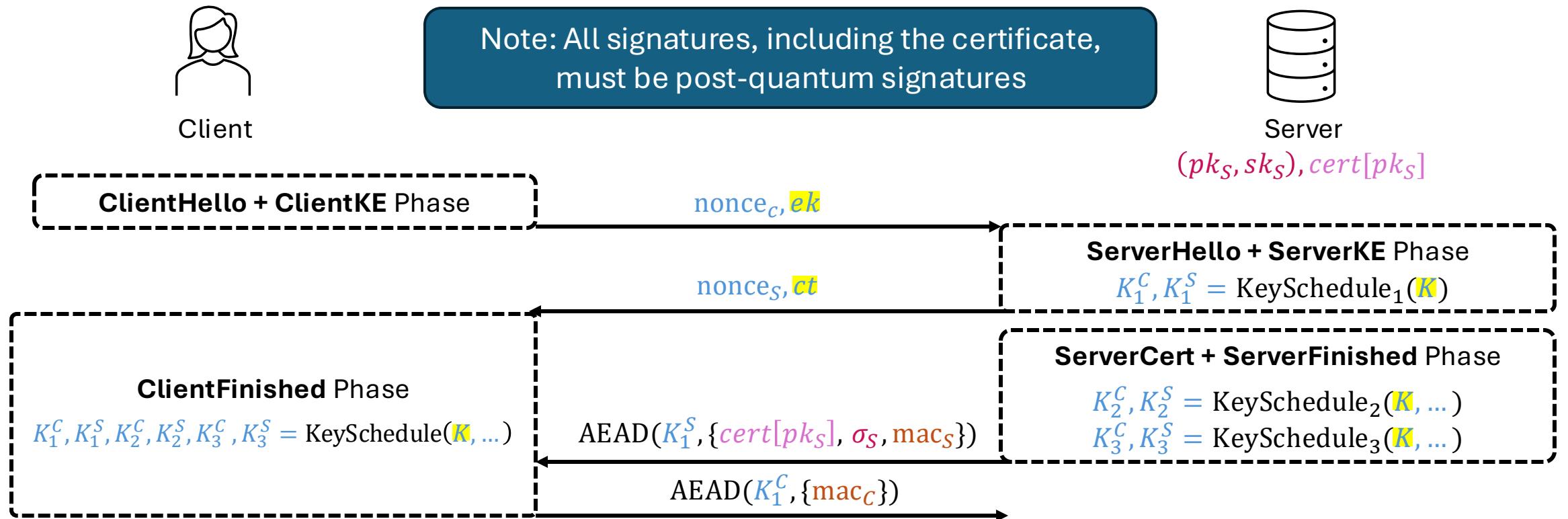
# Post-quantum TLS

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



# Post-quantum TLS

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



# 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

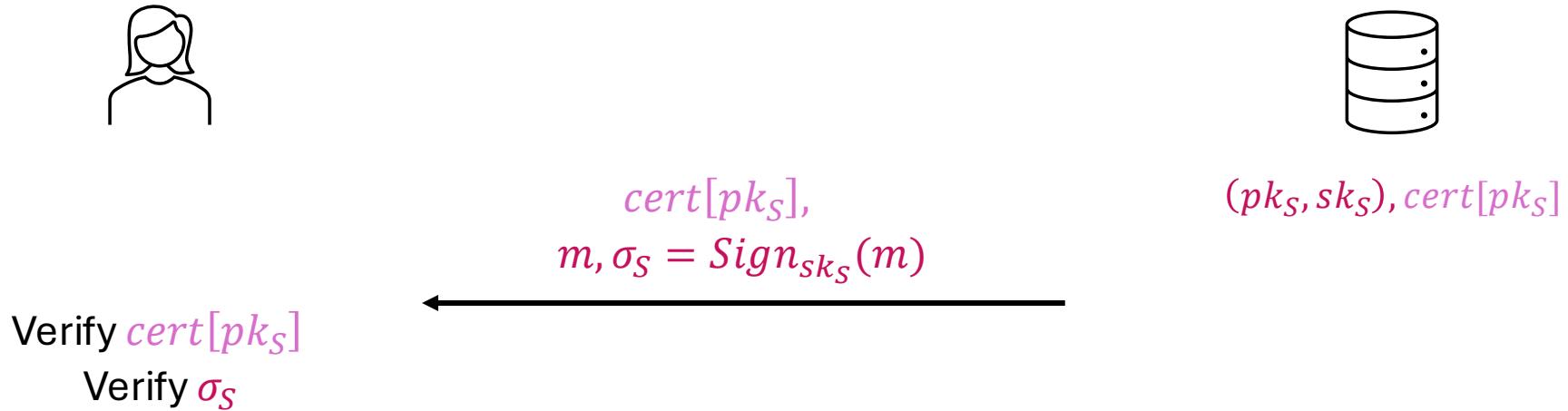
- In post-quantum cryptography, signature schemes are generally less efficient than KEM
- In TLS 1.3, we use signature to
  - (1) Certify the server's long-term public key (signed by the CA) ...one-time price
  - (2) **Authenticate the serverKE message (signed by the server)** ...needed in every session

# 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

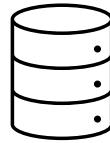
- Use KEM to authenticate messages



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

# Post-quantum TLS based on KEM

- Use KEM to authenticate messages

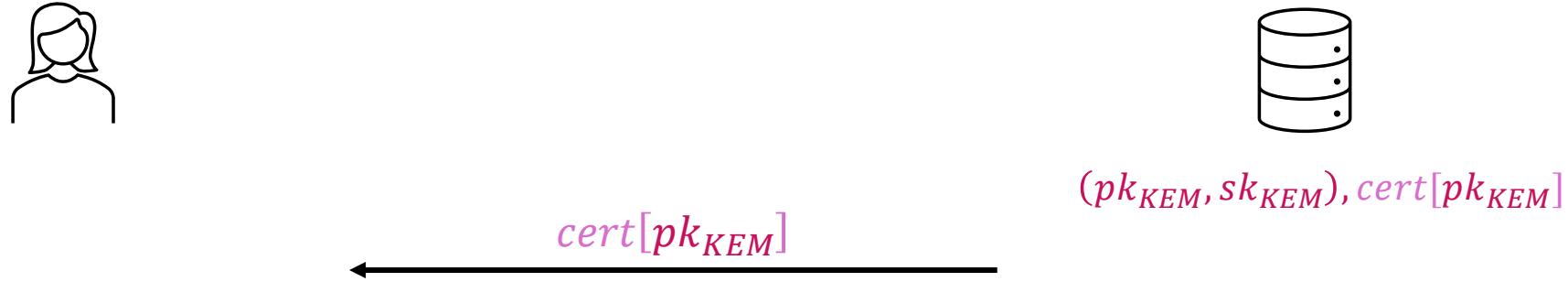


$(pk_{KEM}, sk_{KEM}), cert[pk_{KEM}]$

When using KEM:  
having the  $sk \Leftrightarrow$  Decrypt ciphertexts

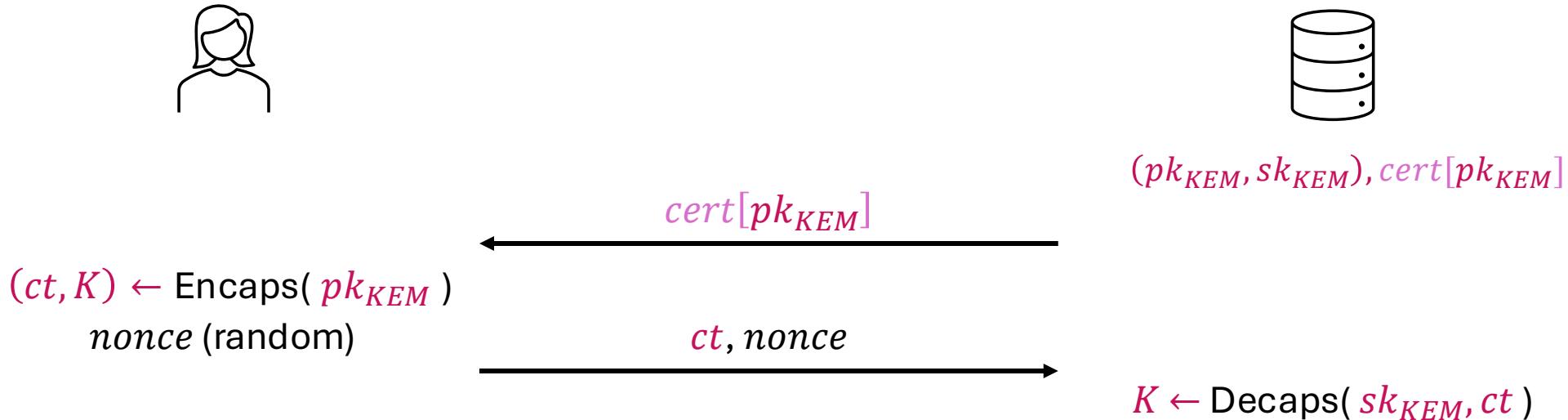
# Post-quantum TLS based on KEM

- Use KEM to authenticate messages



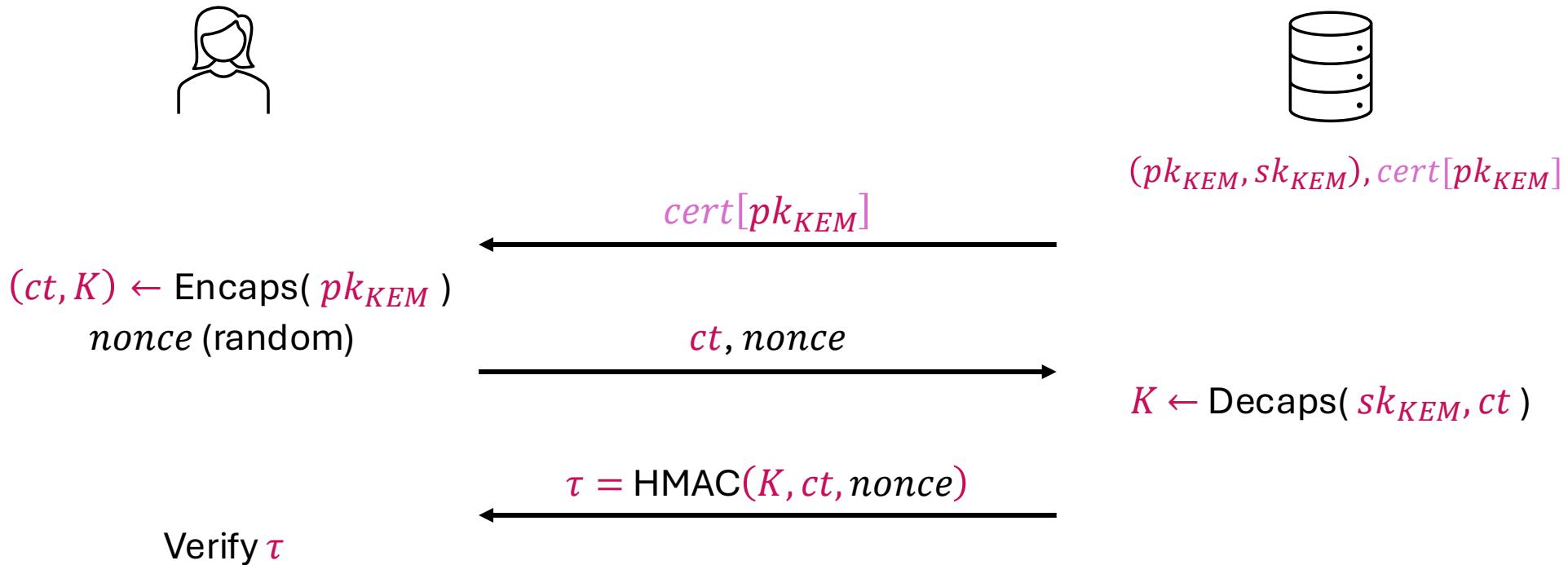
# Post-quantum TLS based on KEM

- Use KEM to authenticate messages



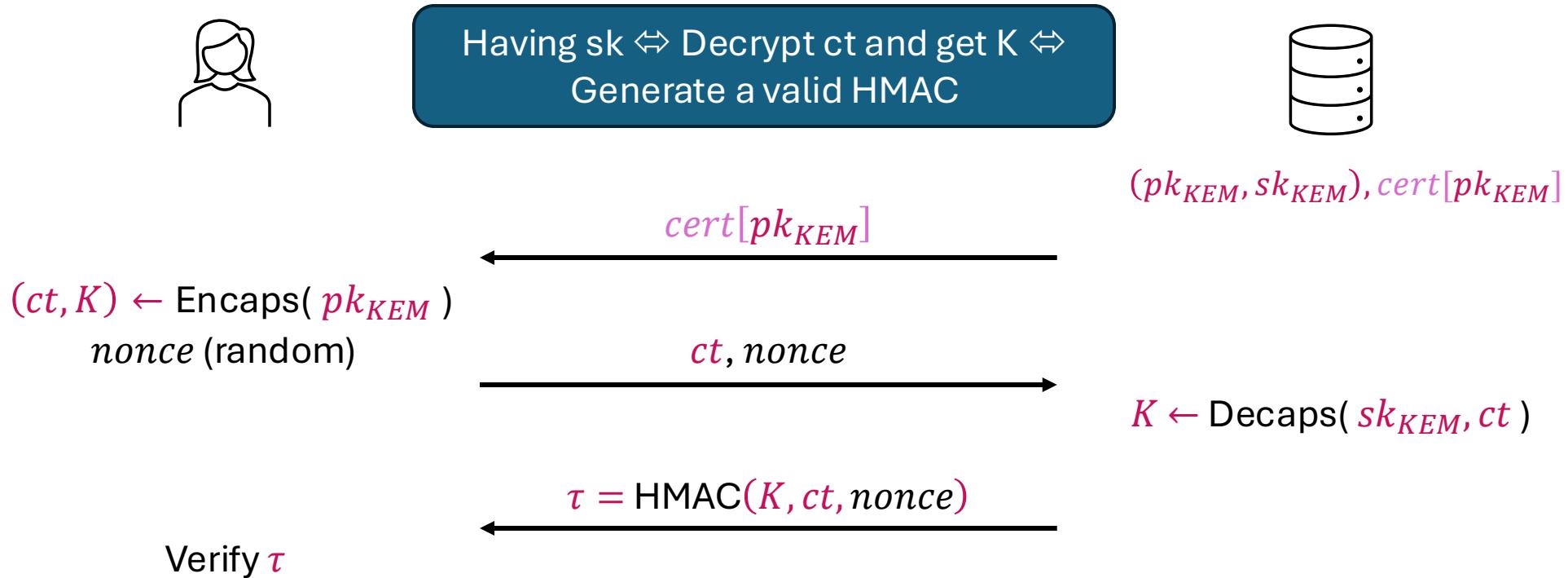
# Post-quantum TLS based on KEM

- Use KEM to authenticate messages



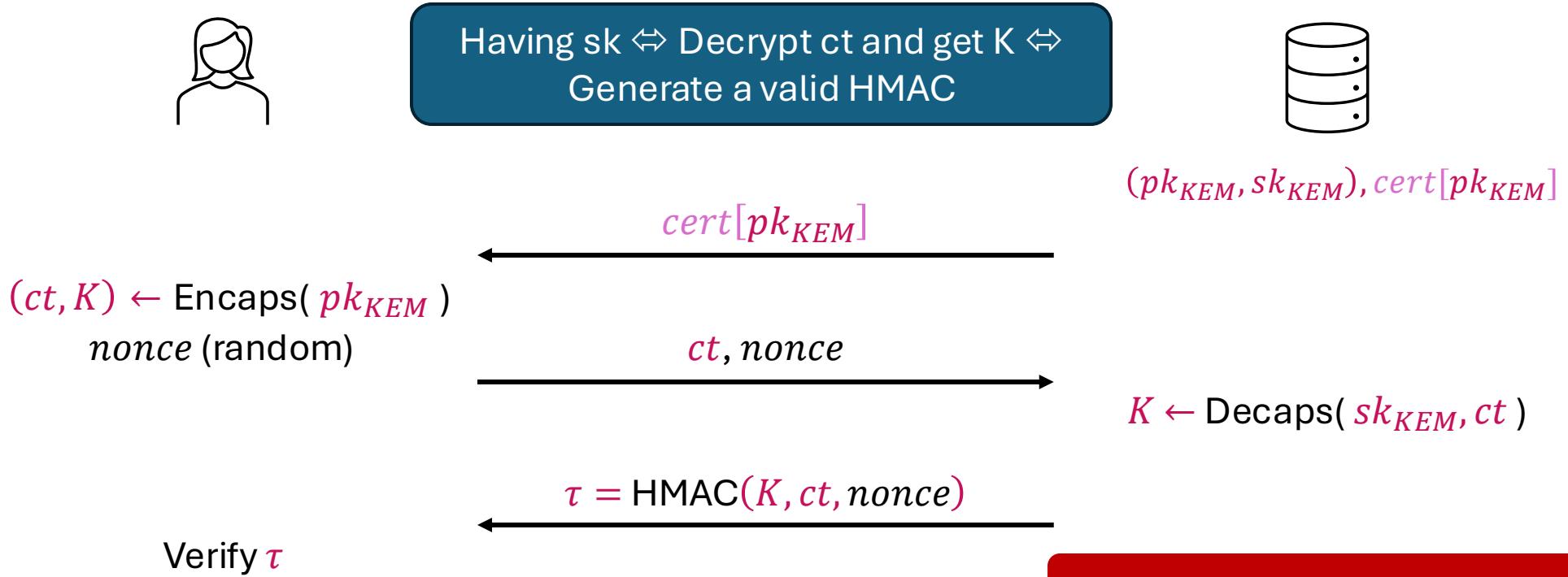
# Post-quantum TLS based on KEM

- Use KEM to authenticate messages



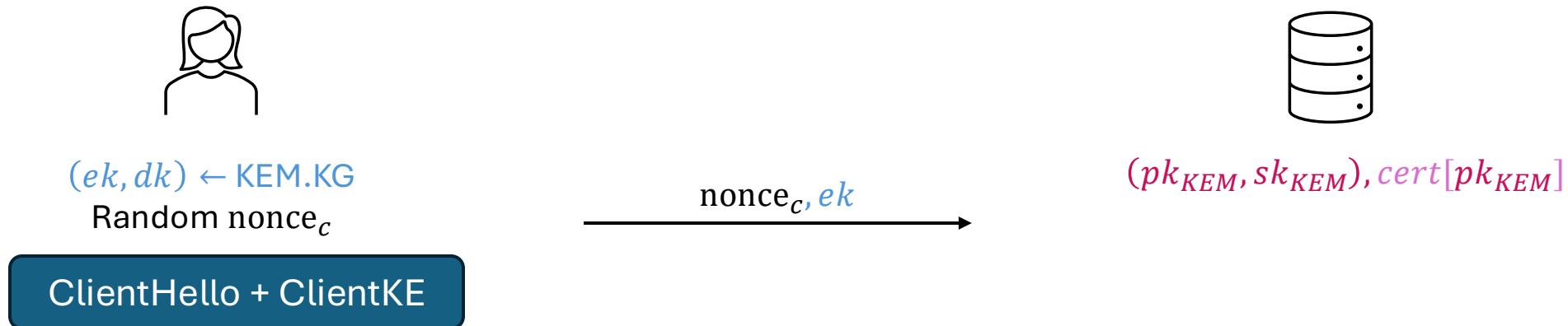
# Post-quantum TLS based on KEM

- Use KEM to authenticate messages



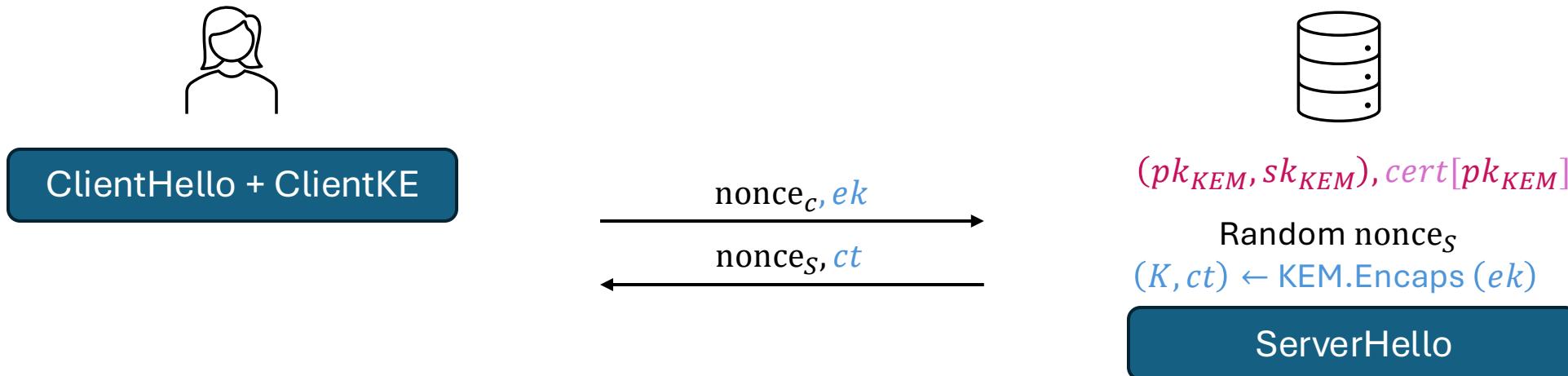
# KEM-TLS

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



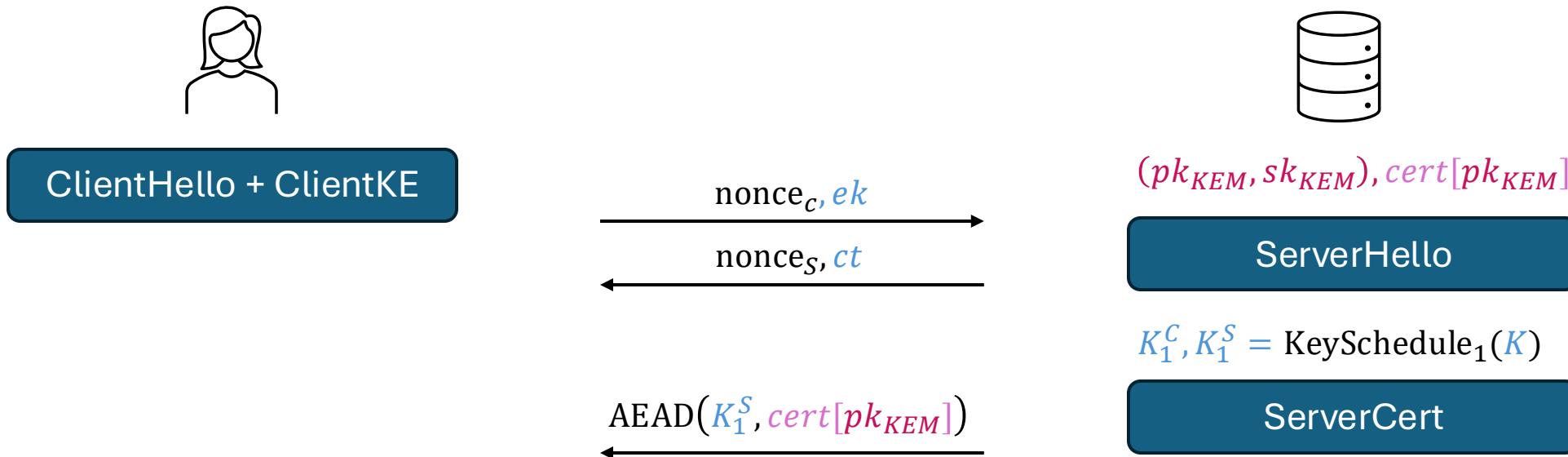
# KEM-TLS

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



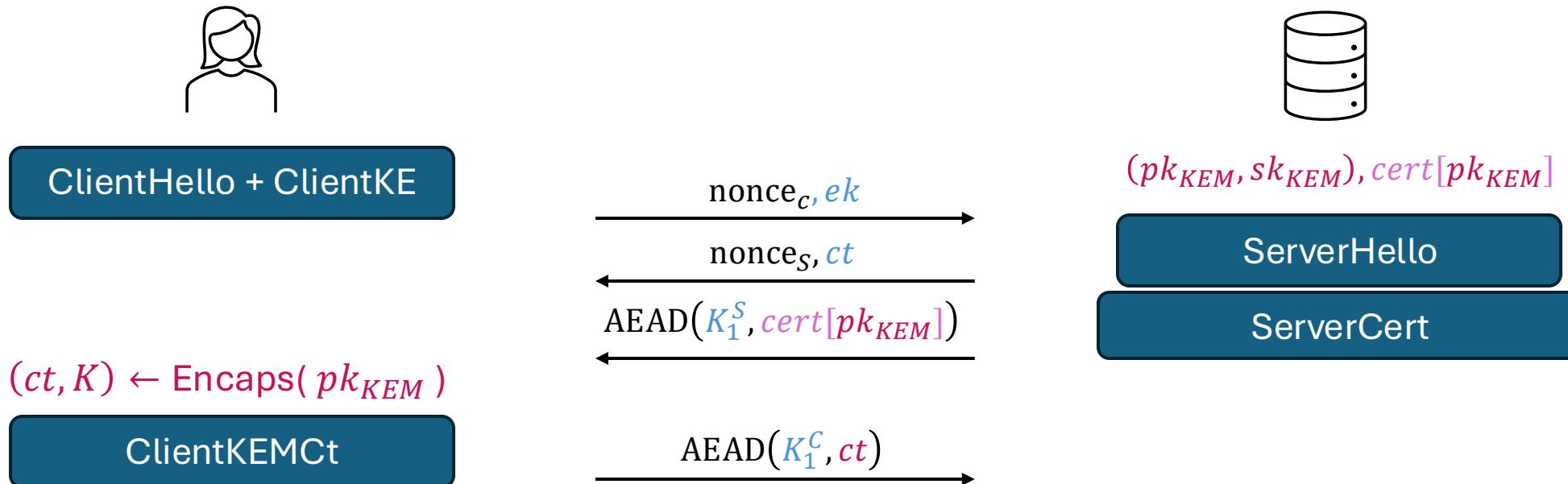
# KEM-TLS

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



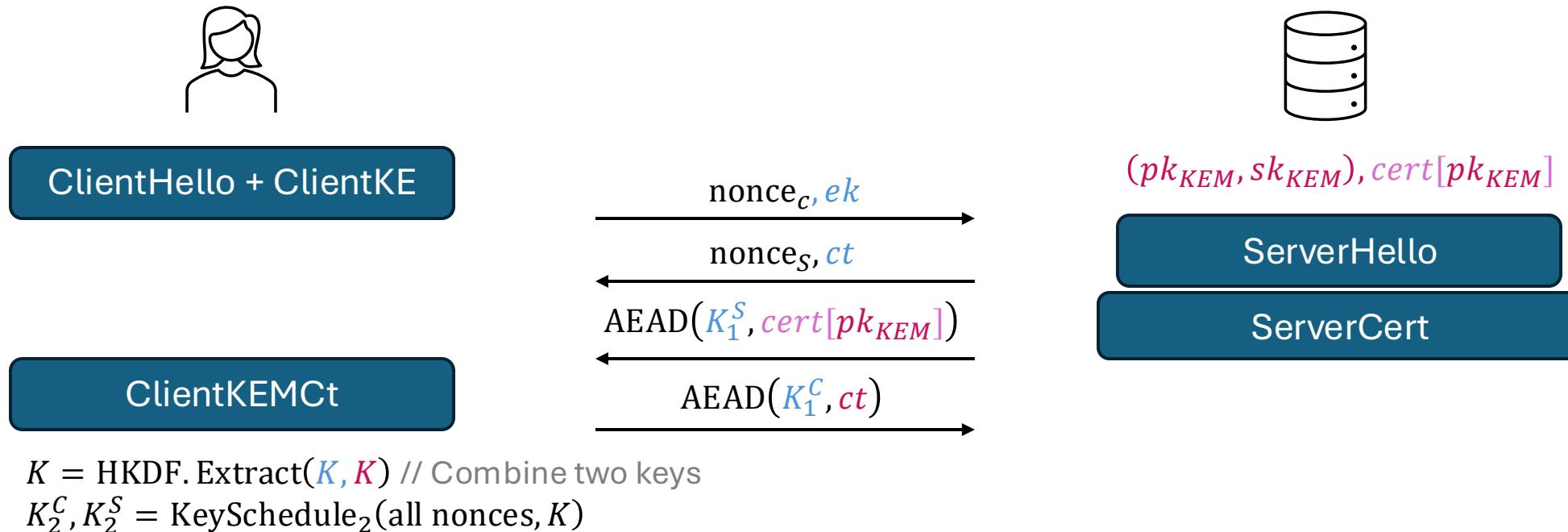
# KEM-TLS

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



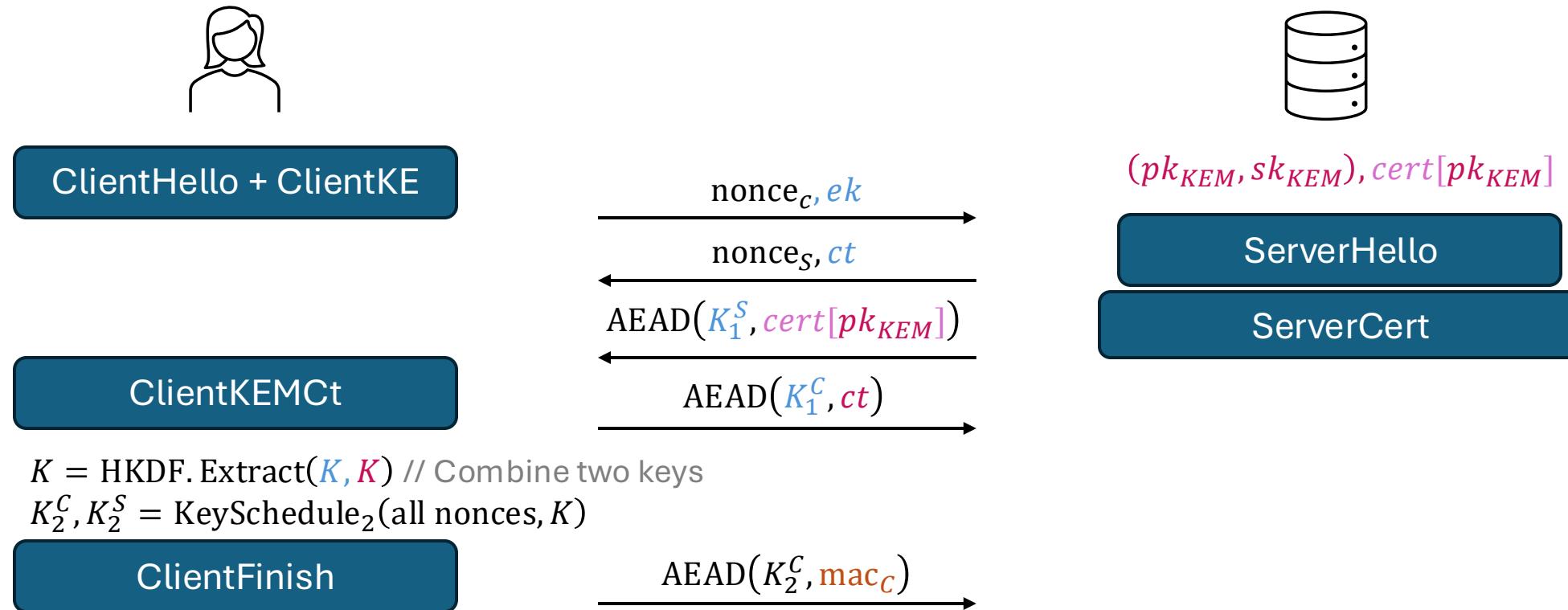
# KEM-TLS

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



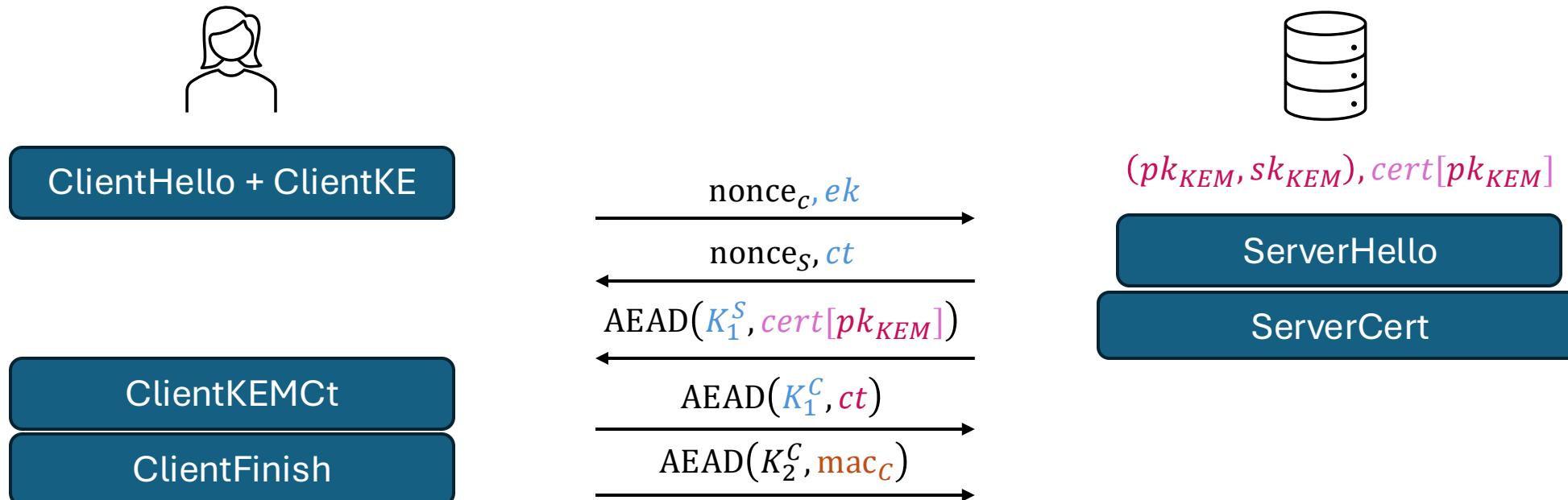
# KEM-TLS

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



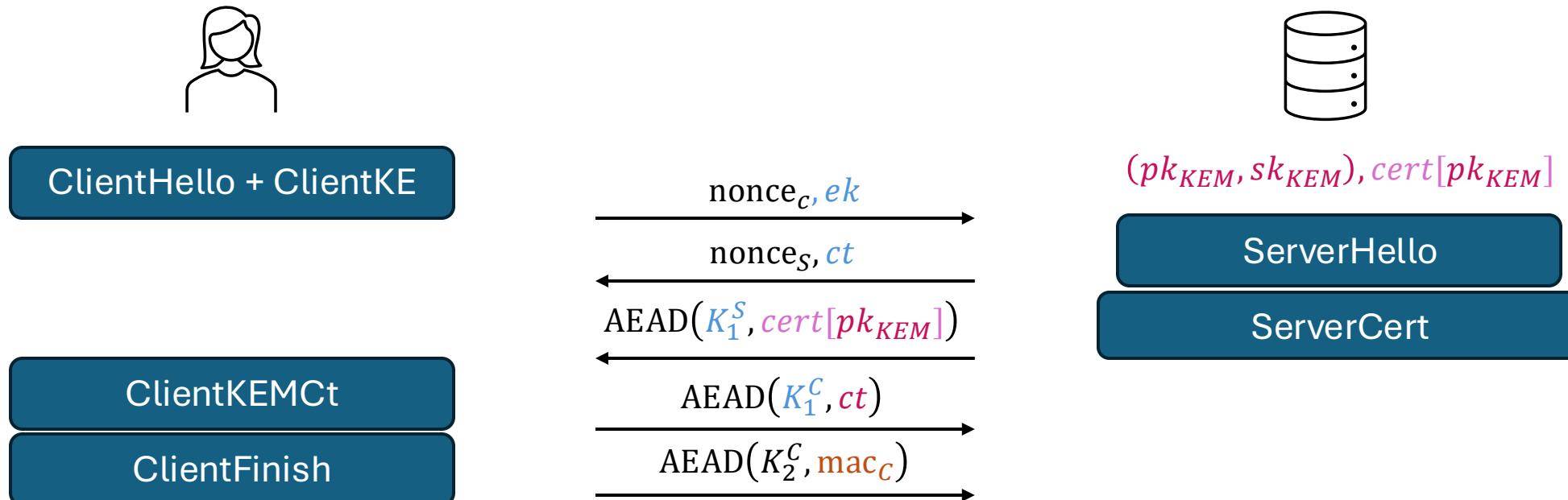
# KEM-TLS

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



# KEM-TLS

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

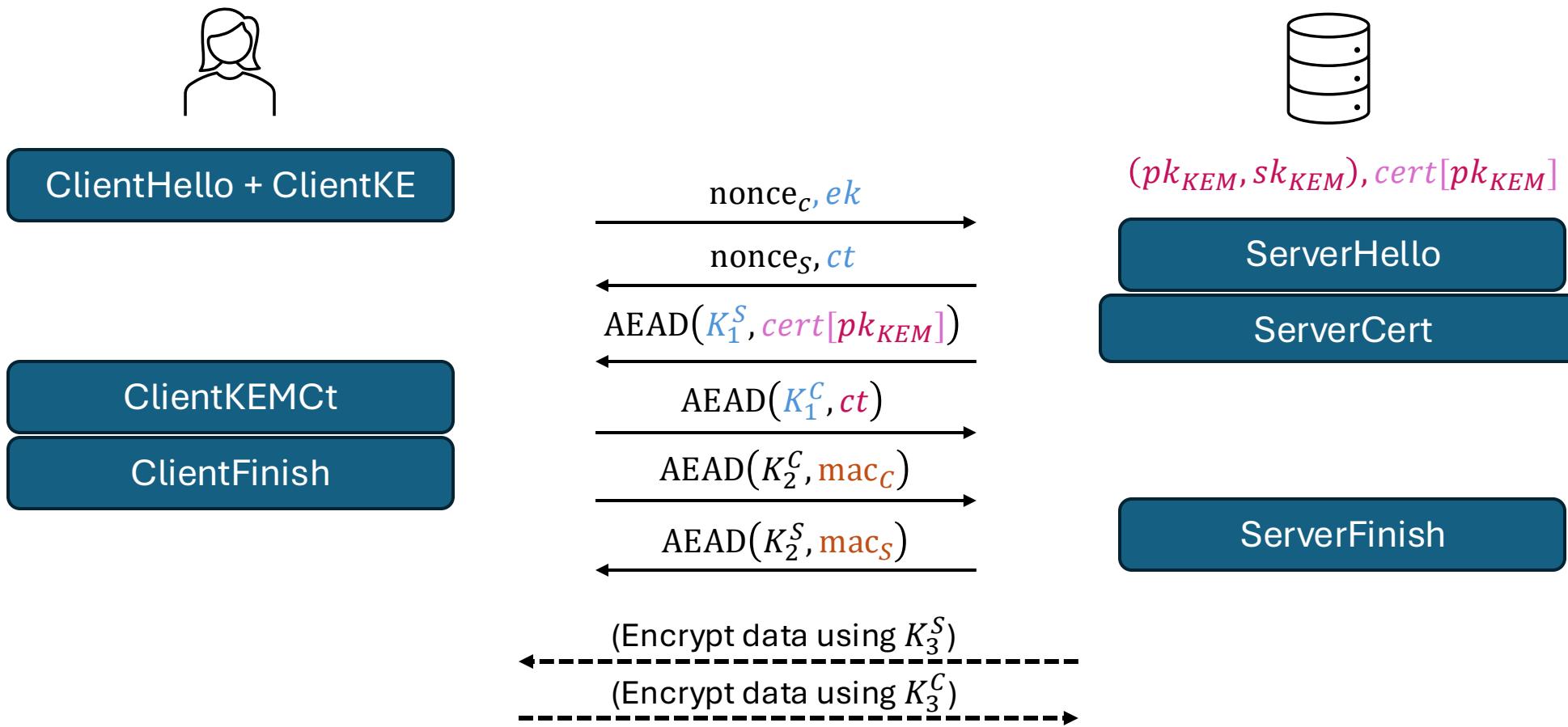


Decrypt  $ct$

$K = \text{HKDF.Extract}(K_1, K) // \text{Combine two keys}$

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

# KEM-TLS



# 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.
- Rust:
  - [https://docs.rs/ml-kem/latest/ml\\_kem/](https://docs.rs/ml-kem/latest/ml_kem/)
  - [https://docs.rs/ml-dsa/latest/ml\\_dsa/](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>