

Cryptography Engineering

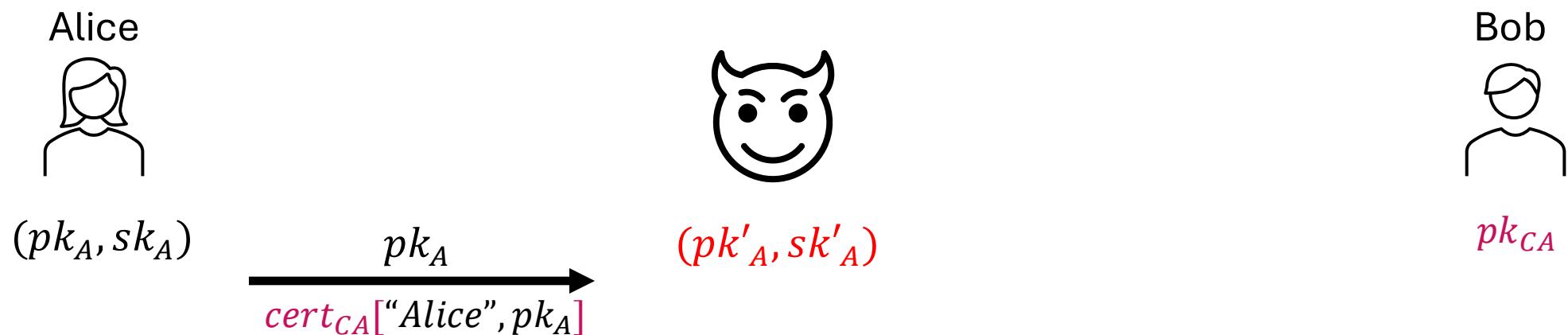
- Lecture 3 (Nov 05, 2024)
- Today's notes:
 - Signed Diffie-Hellman Key Exchange (SigDH) Protocol
 - TLS handshake and HTTPS protocol
 - Case study: ECDSA
- Today's coding tasks:
 - Implement a toy example of TLS handshake
- First homework set

Code Review

- Some useful notes:
 - Crate x25519_dalek

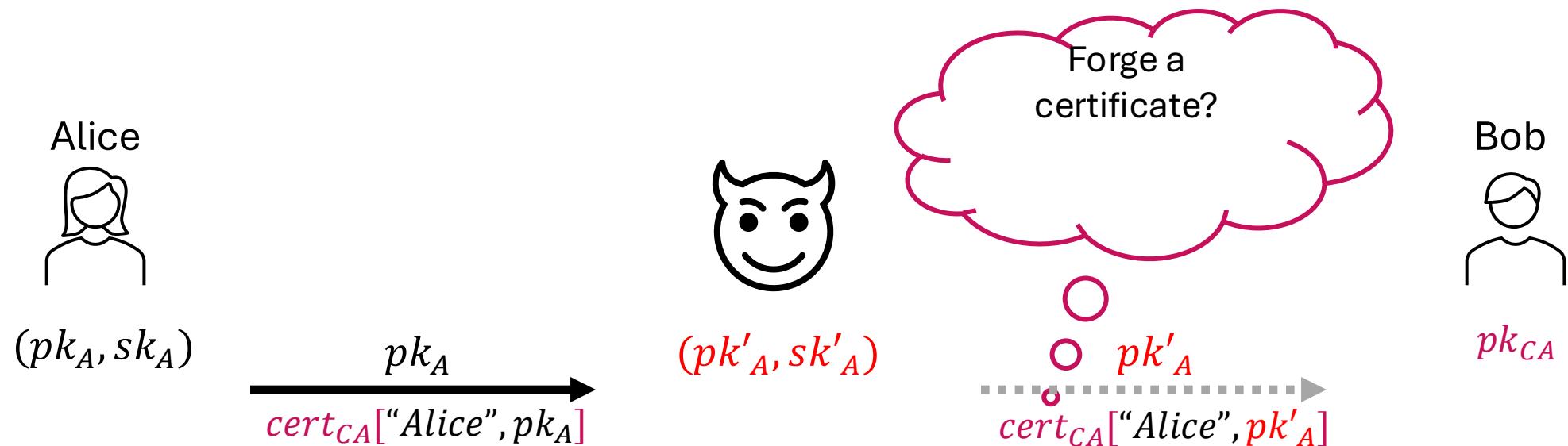
Prevent MitM using signature/certificate

- Transporting **(malicious)** public keys **(with signature/certificate)**
 - (Note that a certificate binds a public key with the identity of owner)



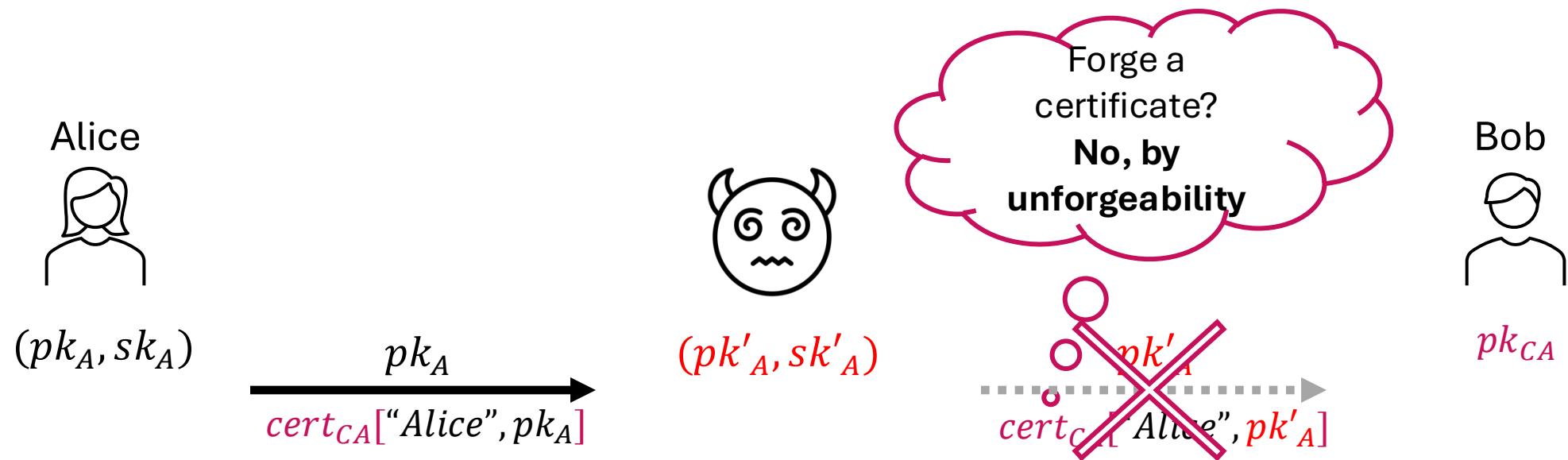
Prevent MitM using signature/certificate

- Transporting (**malicious**) public keys (**with signature/certificate**)
 - (Note that a certificate binds a public key with the identity of owner)



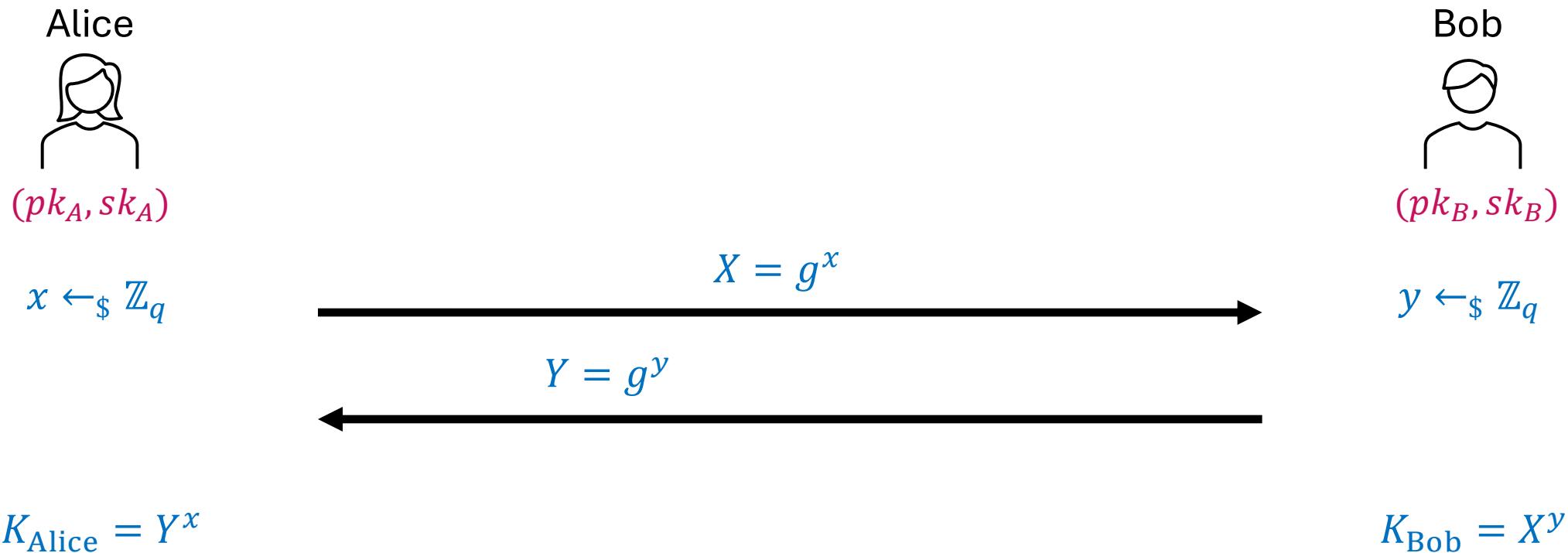
Prevent MitM using signature/certificate

- Transporting (**malicious**) public keys (**with signature/certificate**)
 - (Note that a certificate binds a public key with the identity of owner)



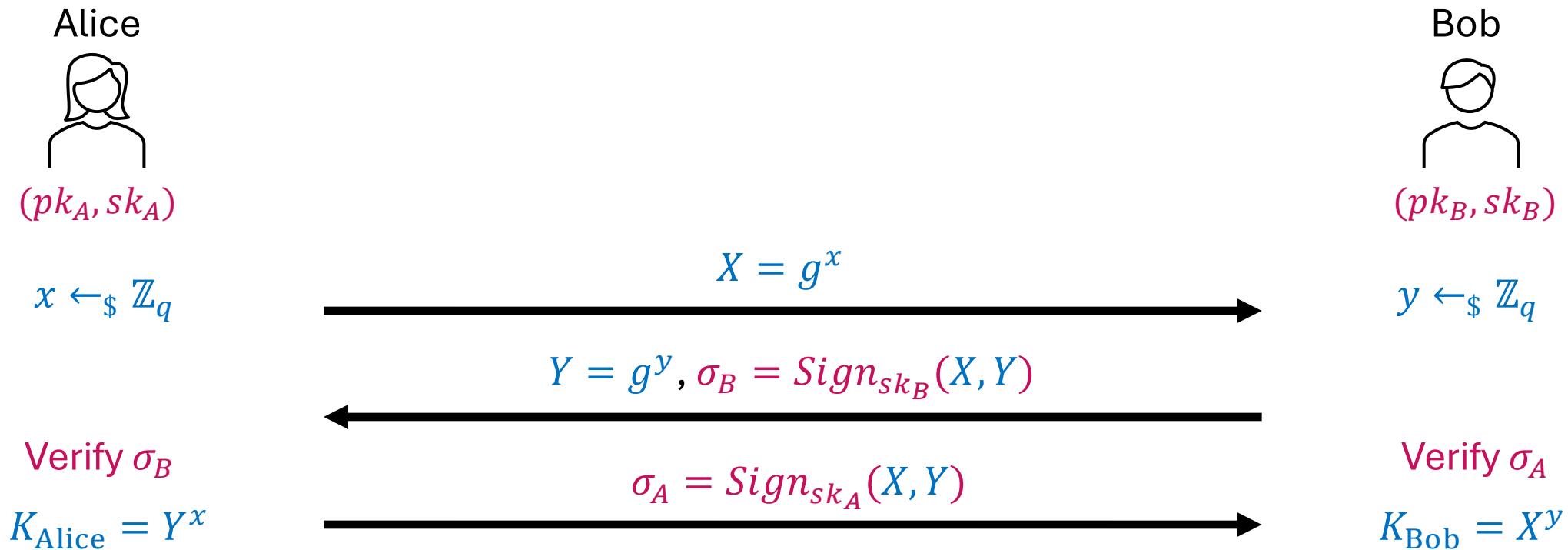
Signed Diffie-Hellman Protocol (Simplified)

- Use **signature** to avoid MitM attacks



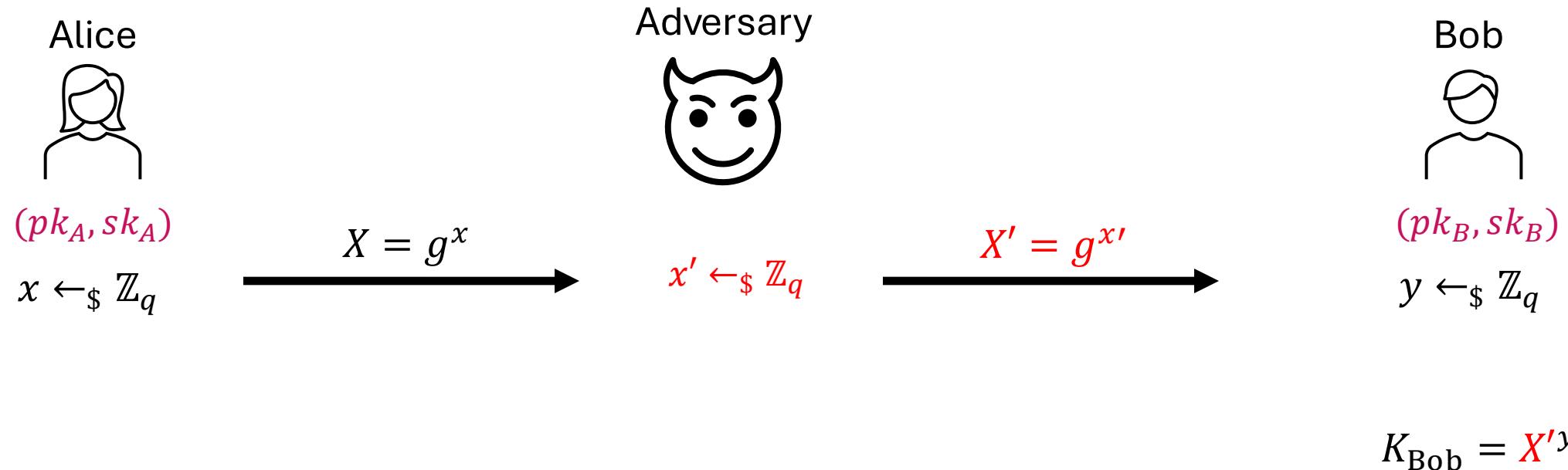
Signed Diffie-Hellman Protocol (Simplified)

- Use **signature** to avoid MitM attacks



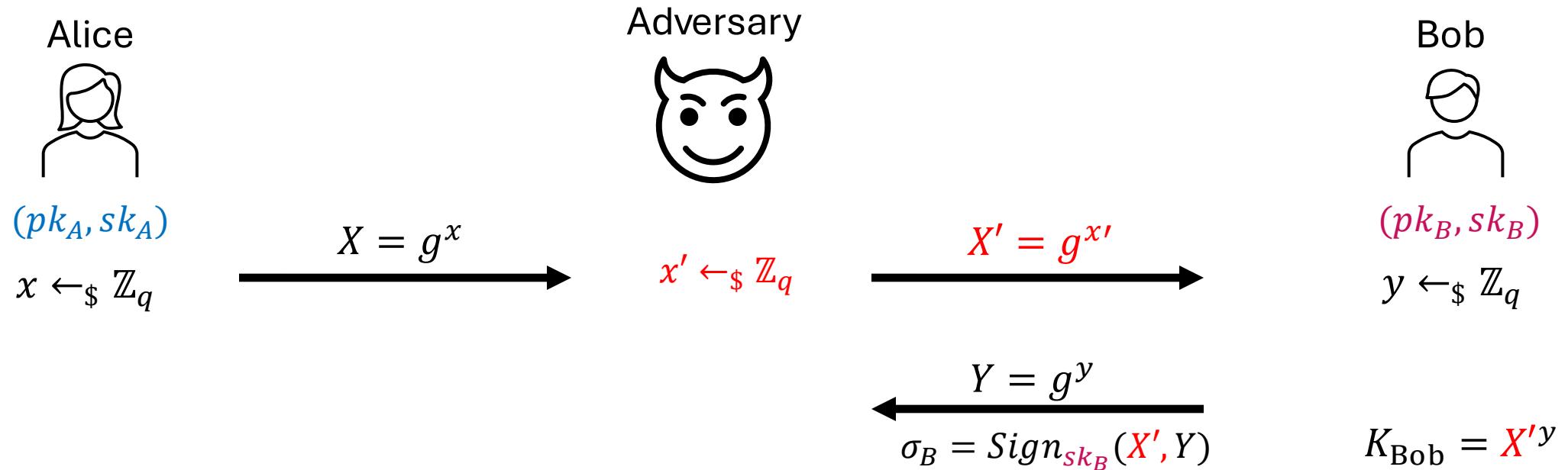
Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



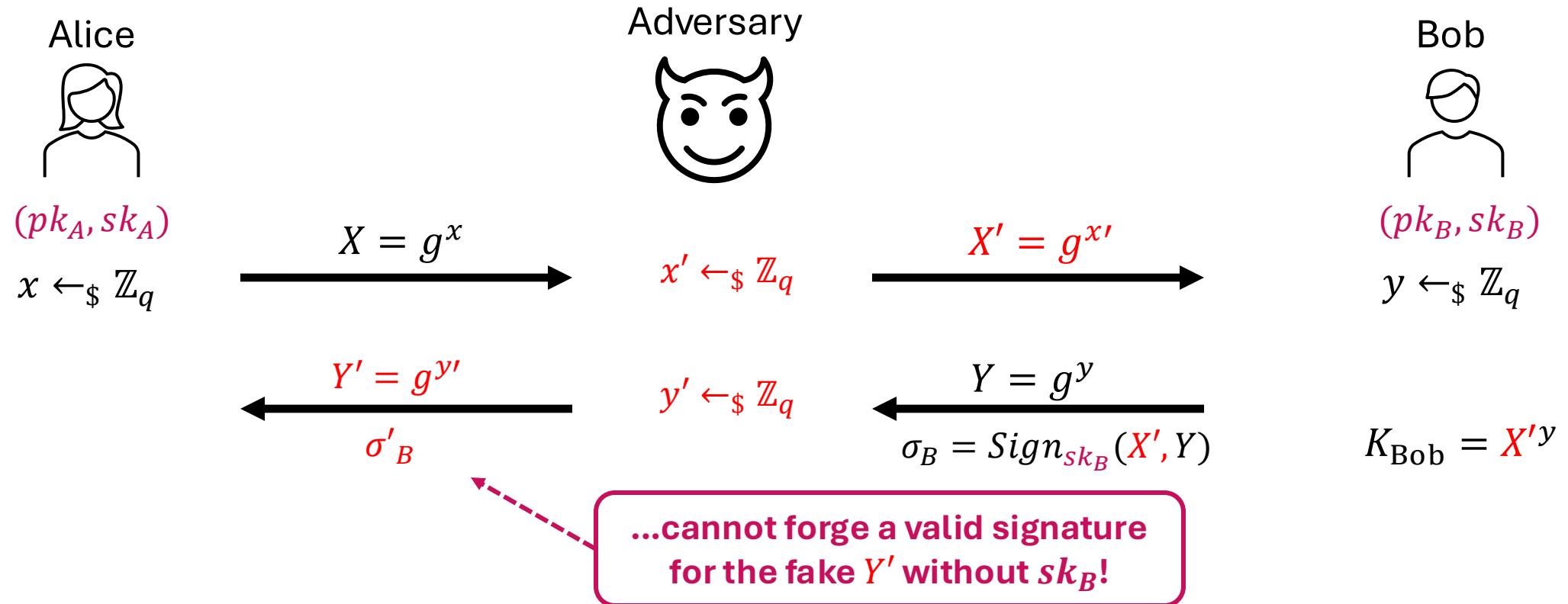
Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



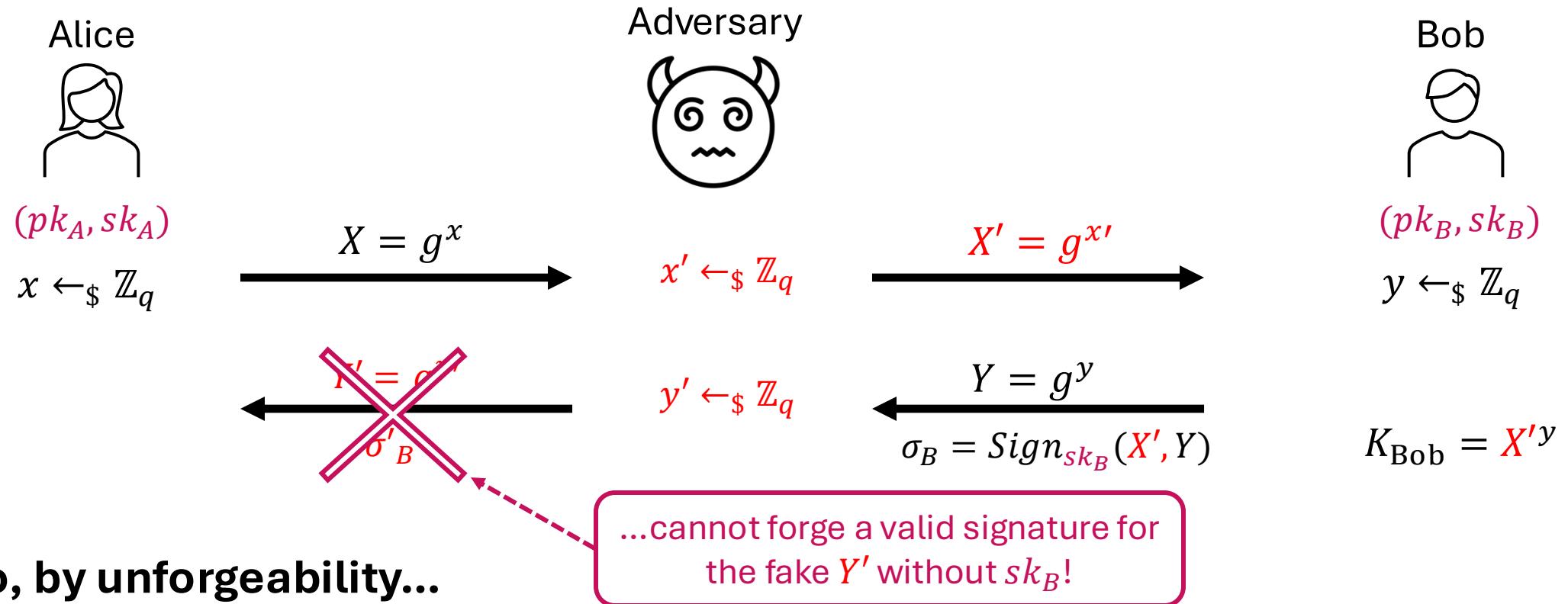
Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?

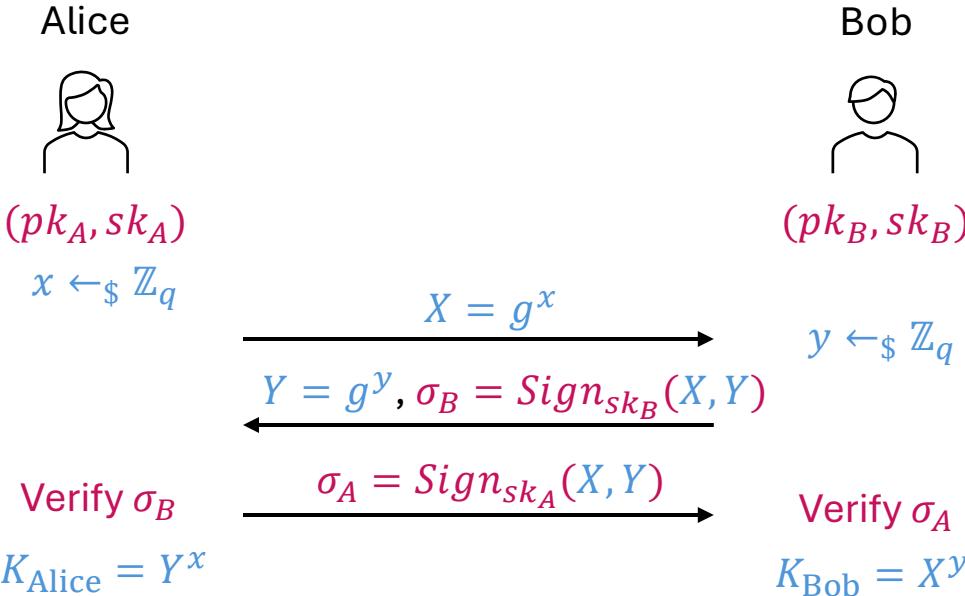


Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?

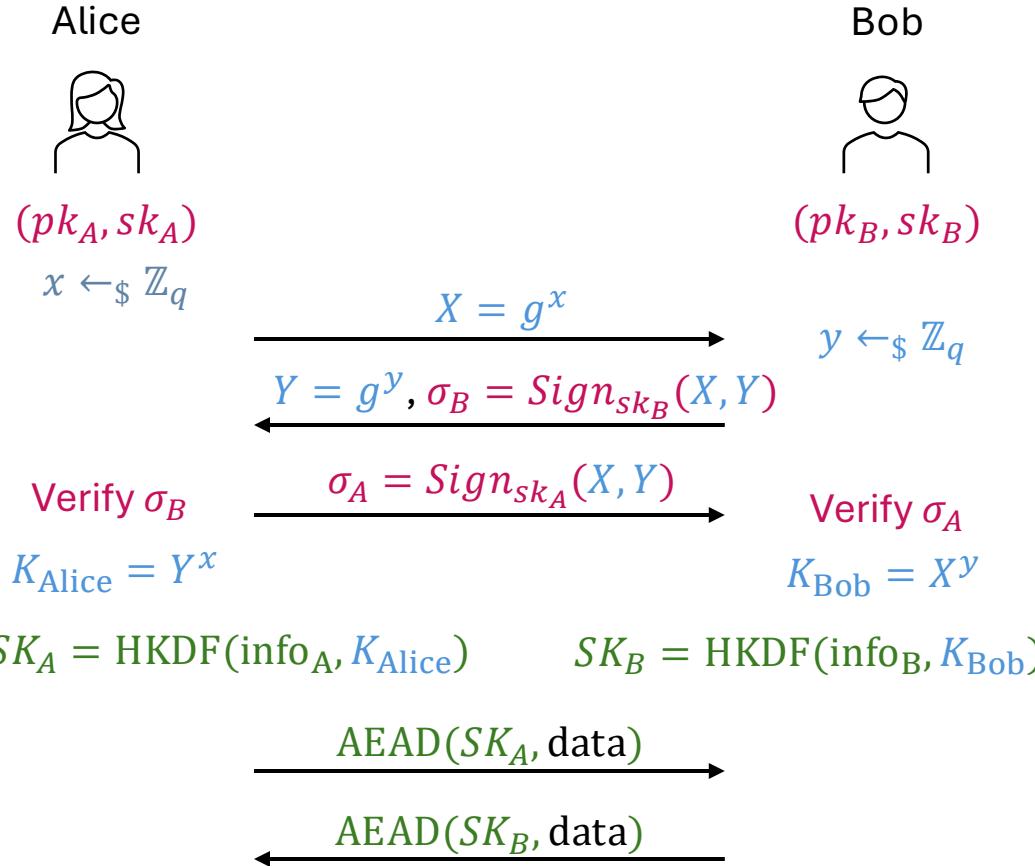


Signed Diffie-Hellman Protocol



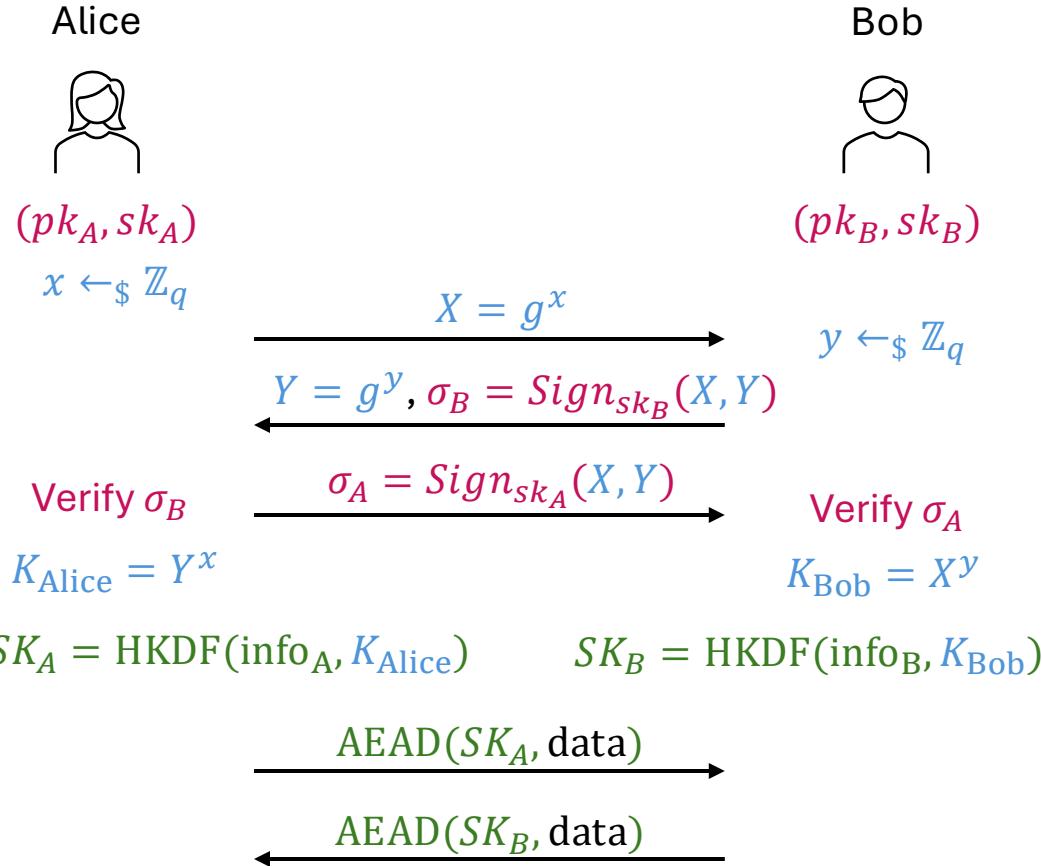
- SigDH
 - Add signature to avoid MitM
 - **Authenticated Key Exchange**

Signed Diffie-Hellman Protocol



- SigDH
 - Add signature to avoid MitM
 - **Authenticated** Key Exchange
- In practice: ECDH + ECDSA + HKDF/HMAC + AEAD ...

Signed Diffie-Hellman Protocol



- SigDH
 - Add signature to avoid MitM
 - **Authenticated** Key Exchange
- In practice: ECDH + ECDSA + HKDF/HMAC + AEAD ...
- Important Application: **TLS handshake protocol...**

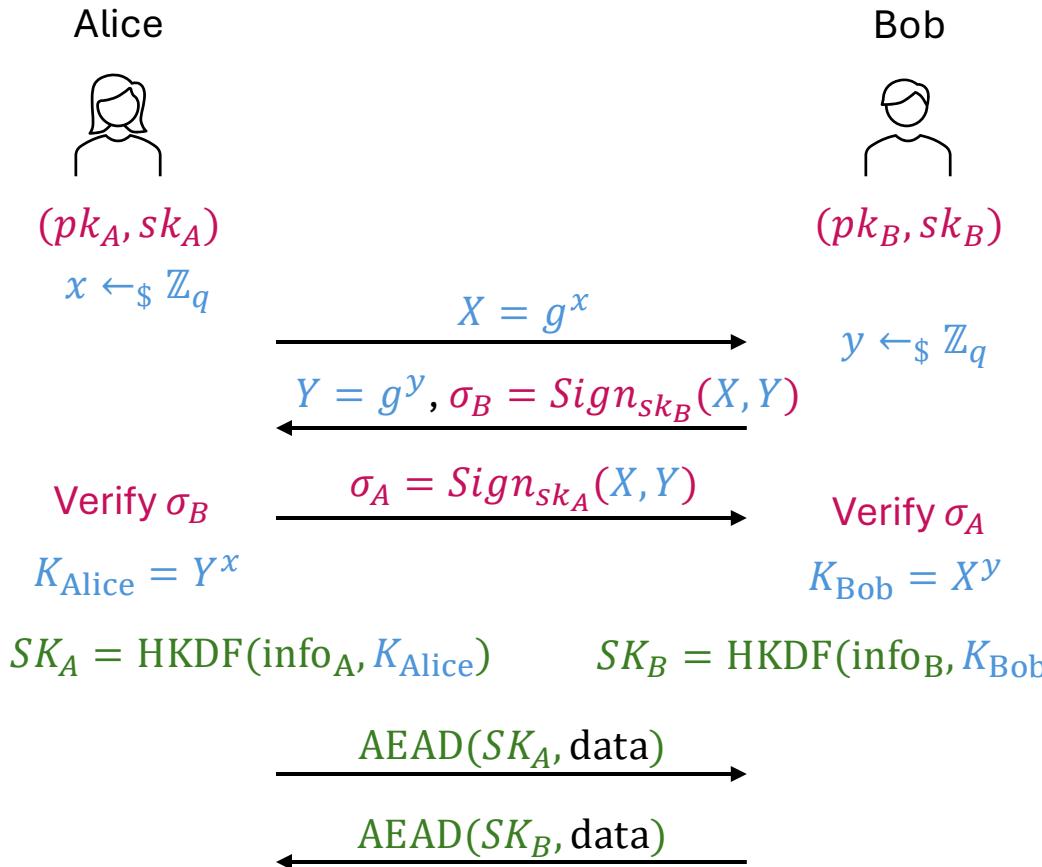
TLS Handshake Protocol

- Transport Layer Security (TLS) Protocol
 - Designed to provide communications security over an open network
 - Used in HTTPS  , Email protocols, OpenVPN, MySQL-over-TLS, ...

TLS Handshake Protocol

- Transport Layer Security (TLS) Protocol
 - Designed to provide communications security over an open network
 - Used in HTTPS  , Email protocols, OpenVPN, MySQL-over-TLS, ...
- TLS process (simplified):
 1. Session initialization (TCP/IP)
 2. **TLS handshake**
 3. **Encrypted Data Transfer**
 4. Session end
- In this lecture, we mainly consider **the client-server setting**
 - **Server Authentication Only:** A client normally does not have static public-private key pair and certificates

SigDH in TLS (Simplified)



SigDH in TLS (Simplified)

Client



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

$$X = g^x$$

Server



$$(pk_S, sk_S)$$

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

Verify σ_{CA}, σ_B

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

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

$$\sigma_{CA} = \text{cert}_{CA}[pk_S]$$

$$Y = g^y, \sigma_B = \text{Sign}_{sk_S}(X, Y)$$

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

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

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

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

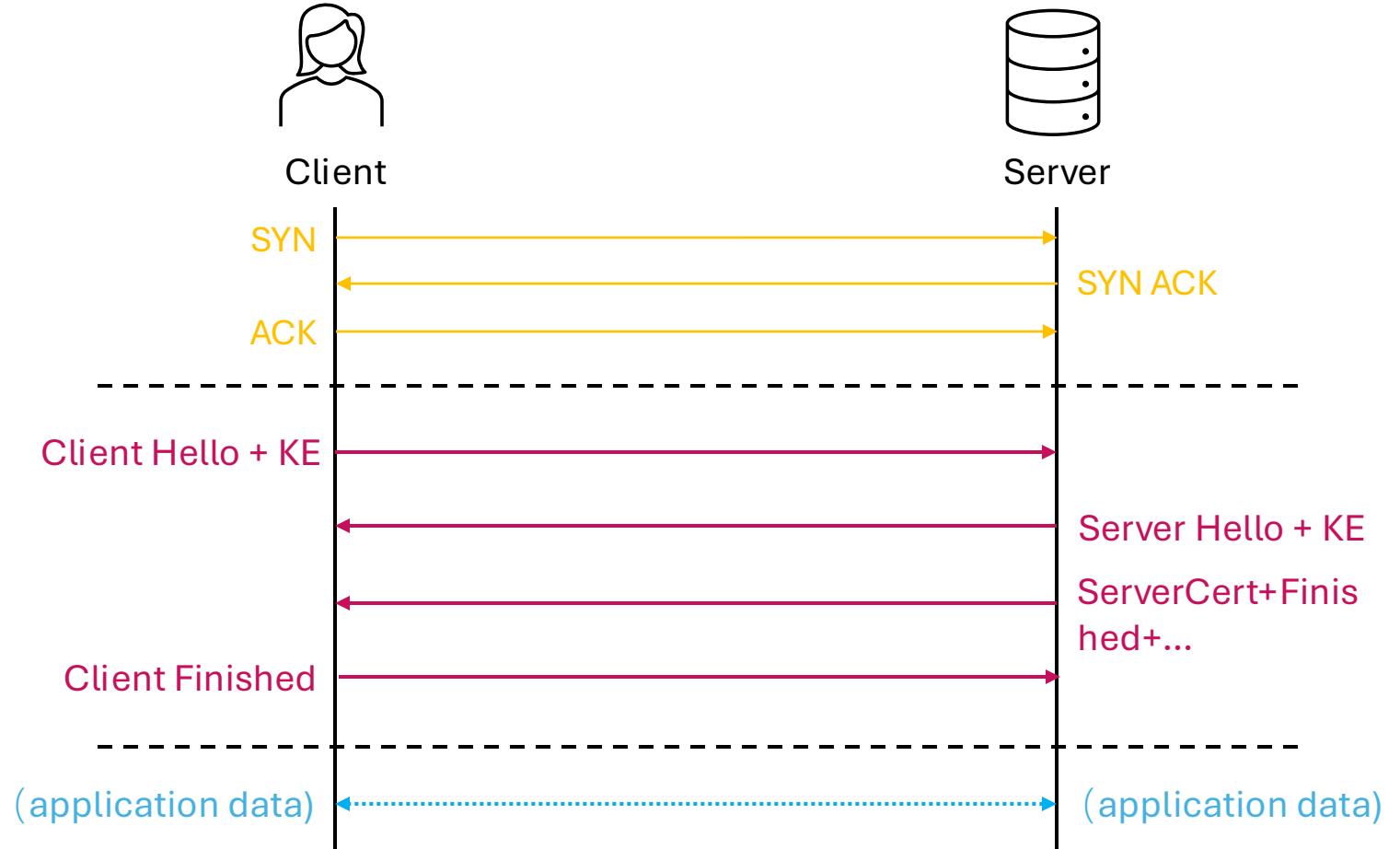
TLS Handshake Protocol

- TLS process (simplified):

1. Session initialization (TCP/IP)

2. TLS handshake

3. Encrypted Data Transfer



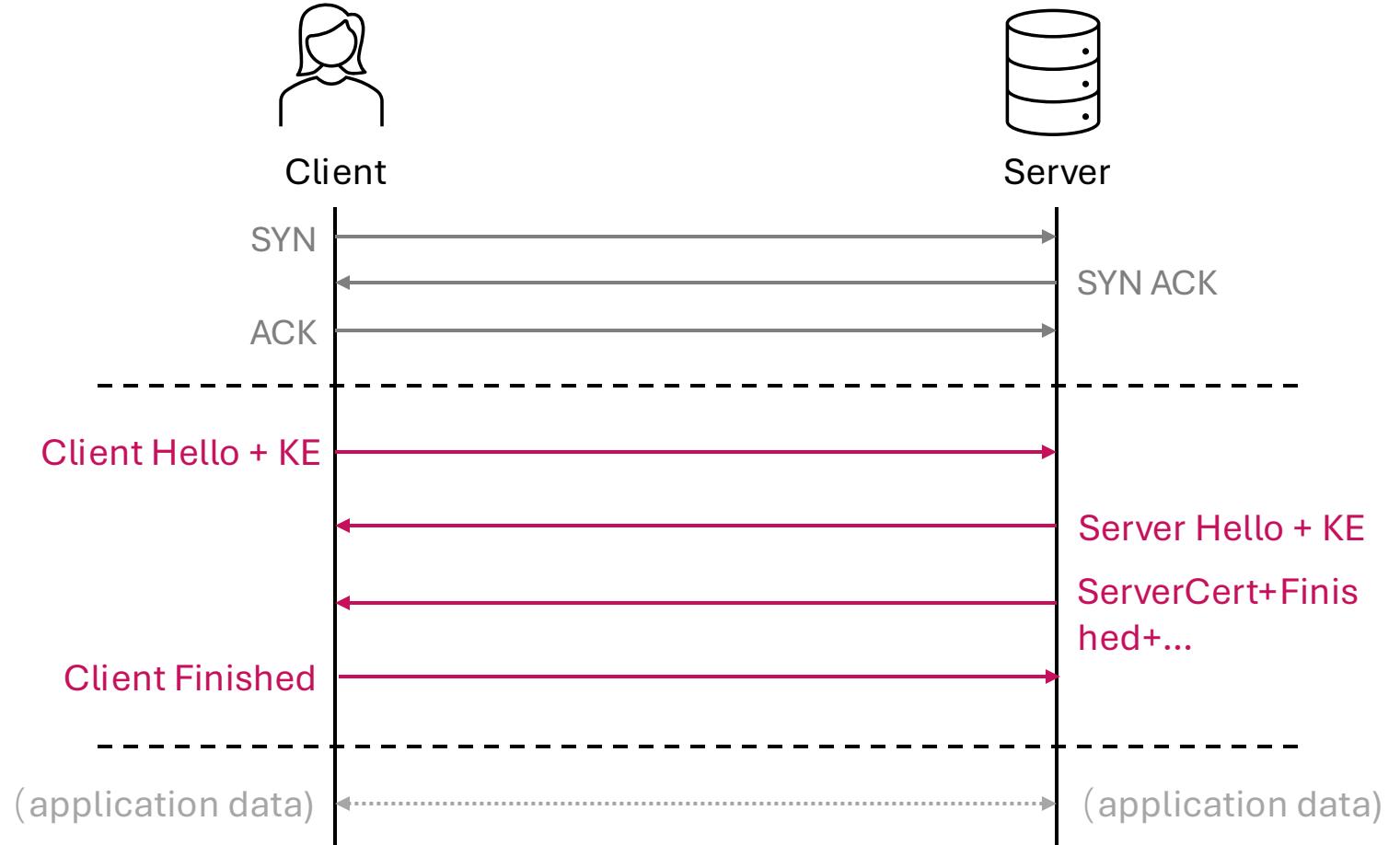
TLS Handshake Protocol

- TLS process (simplified):

1. Session initialization (TCP/IP)

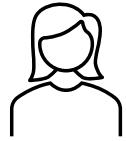
2. TLS handshake

3. Encrypted Data Transfer

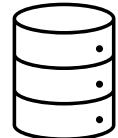


TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



Client

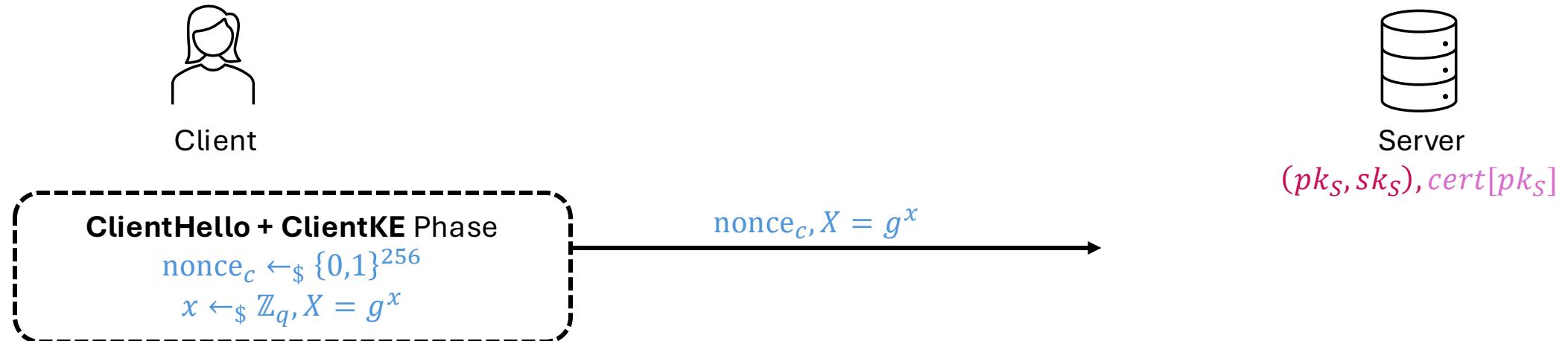


Server

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

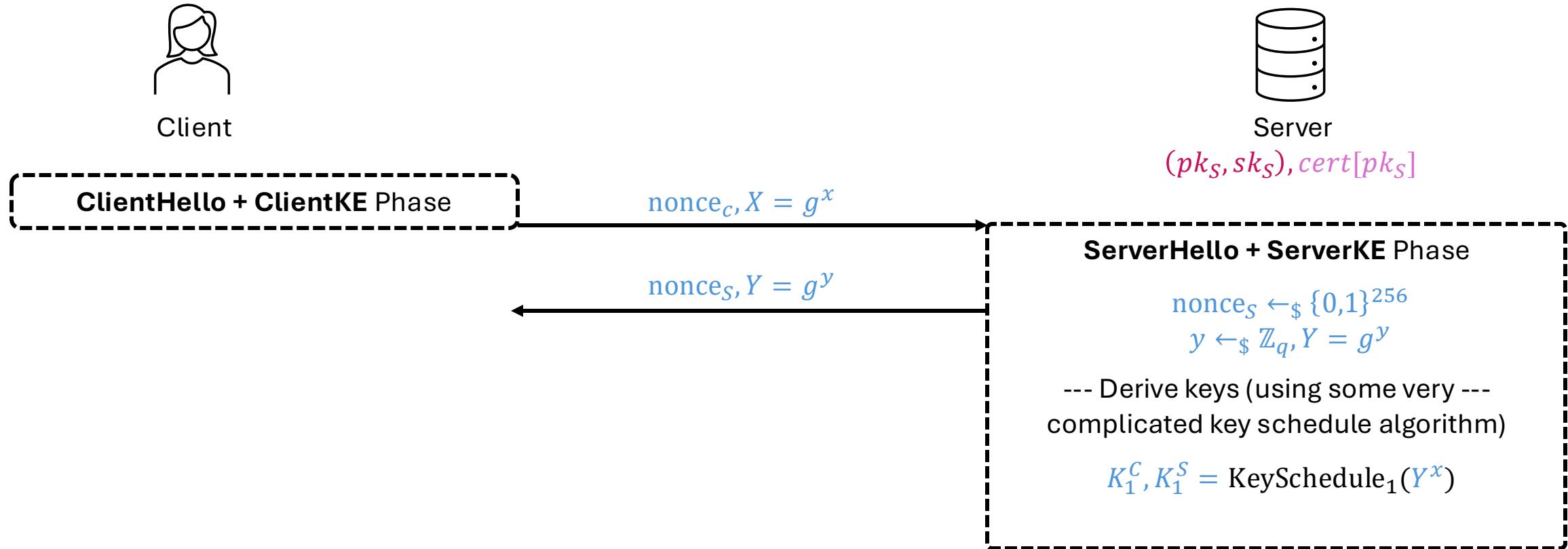
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



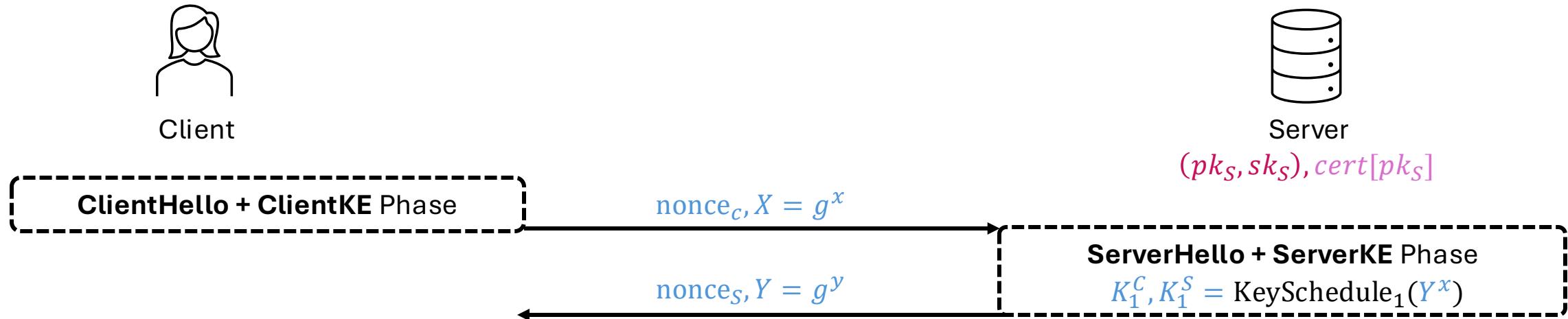
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



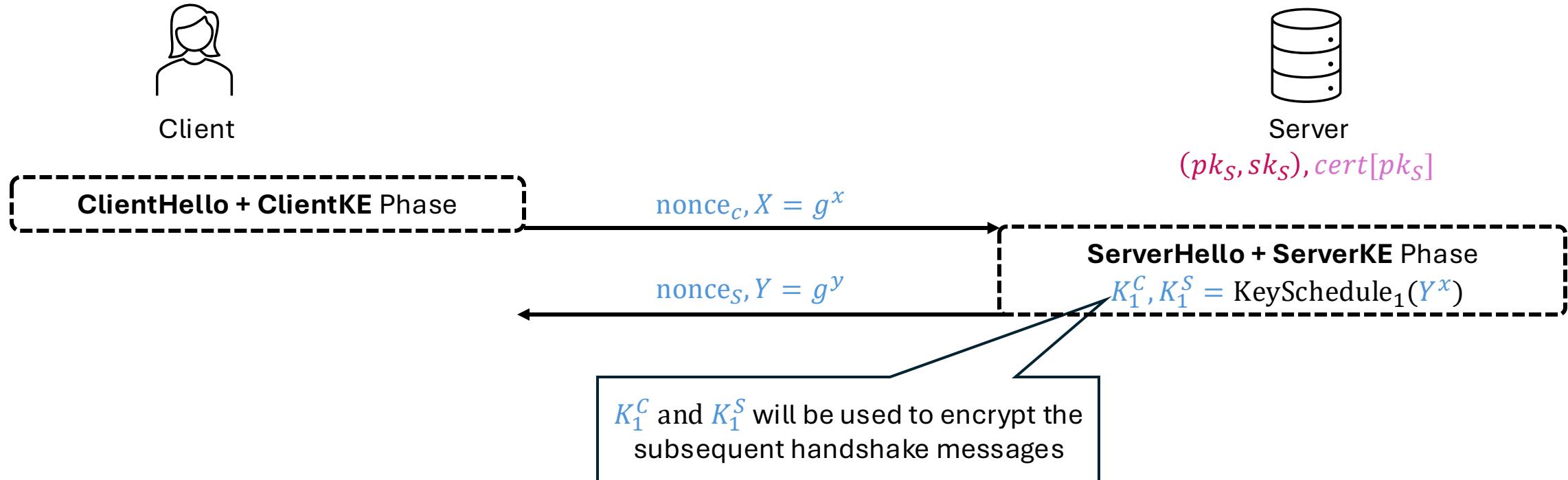
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



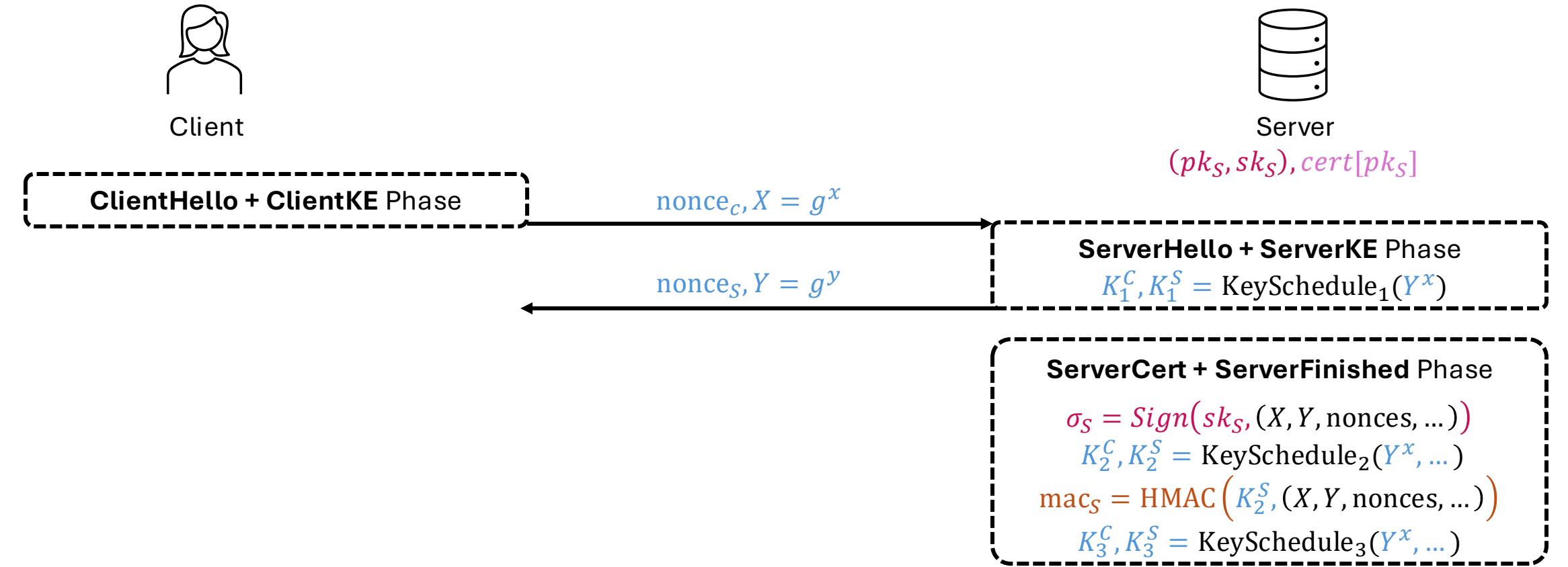
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



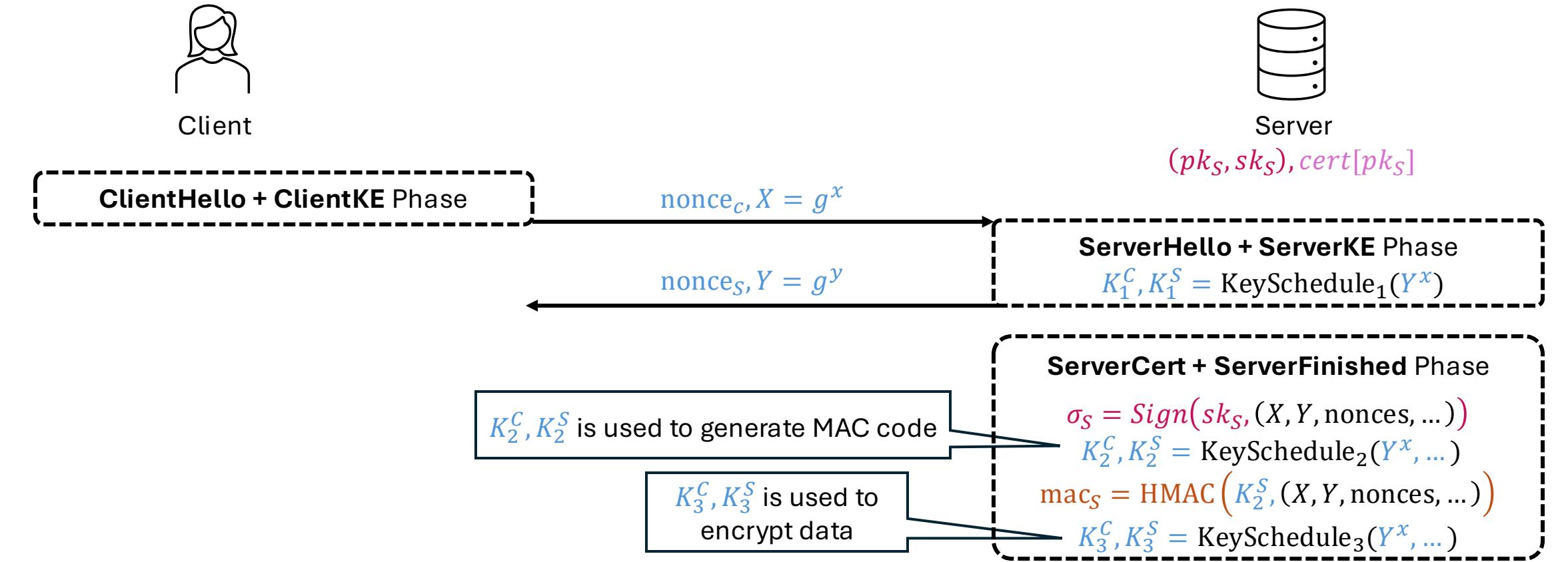
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



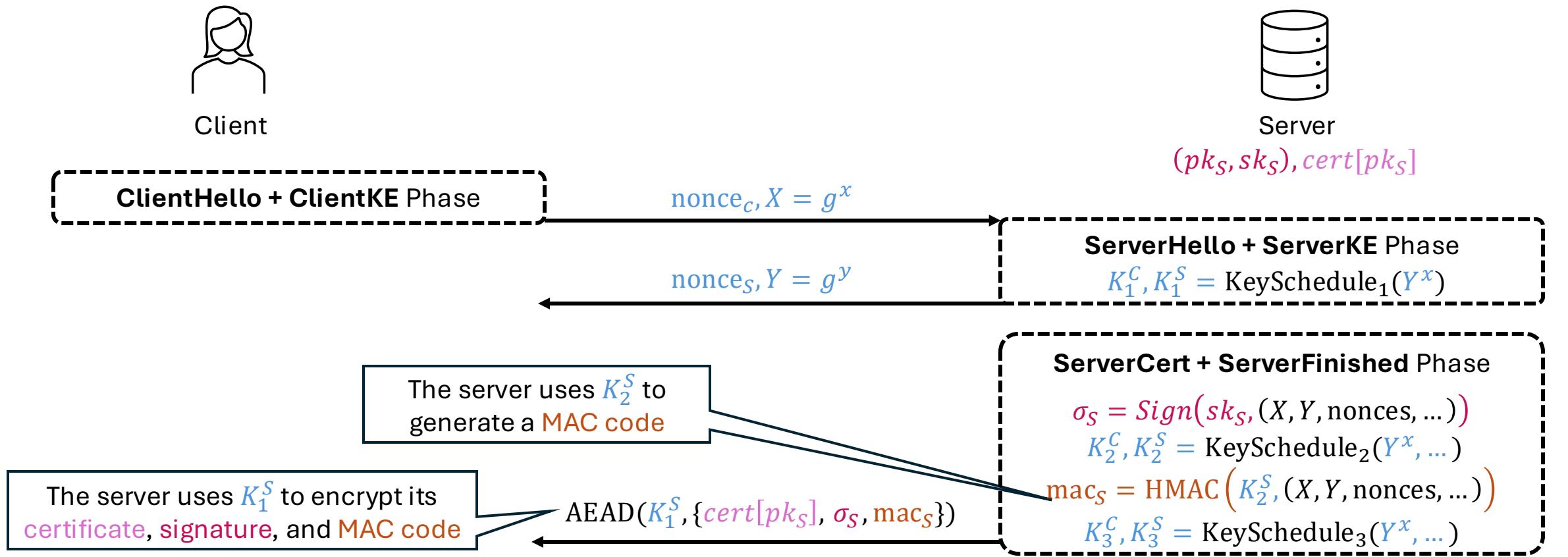
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



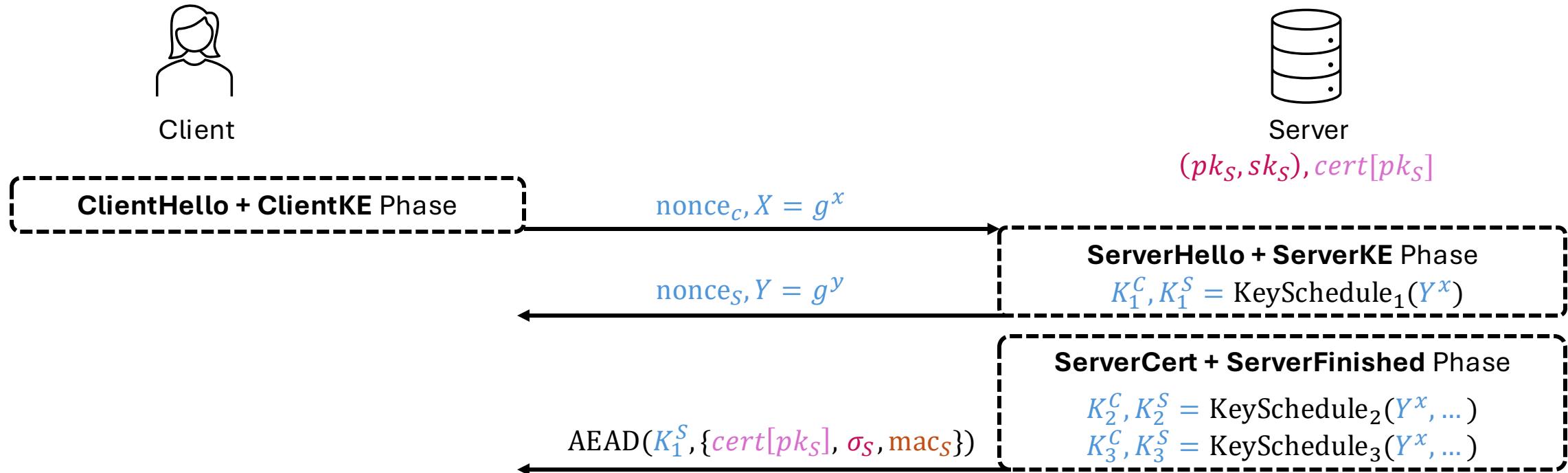
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



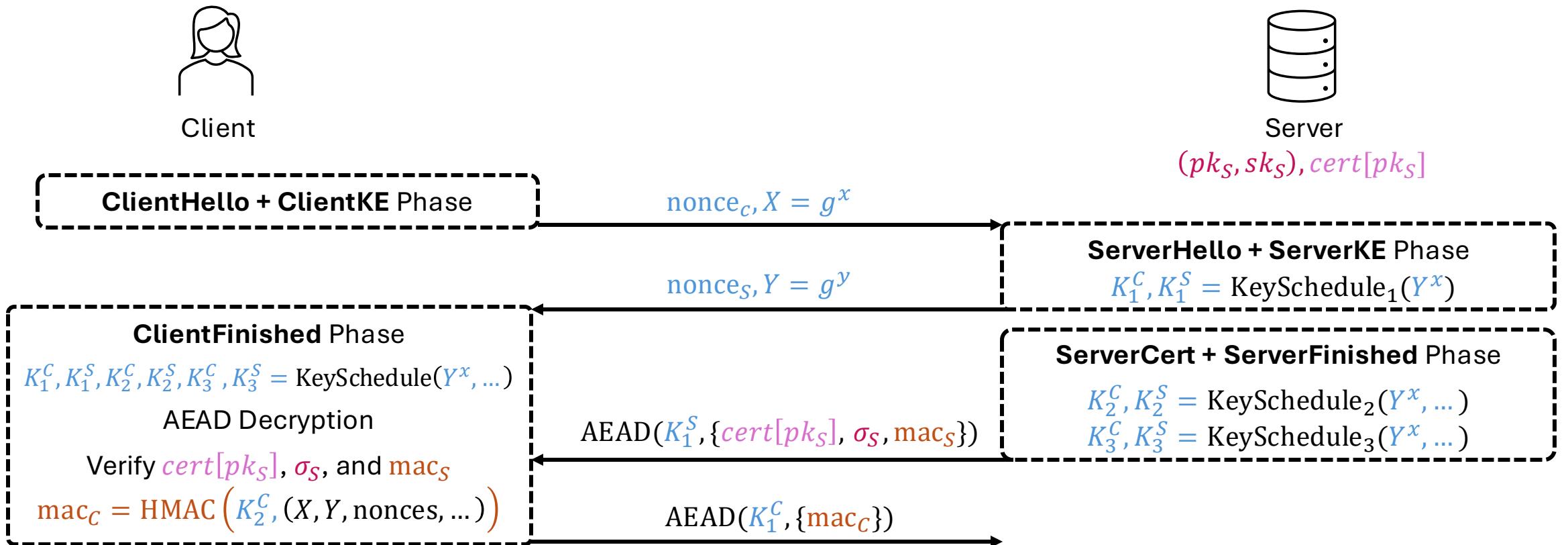
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



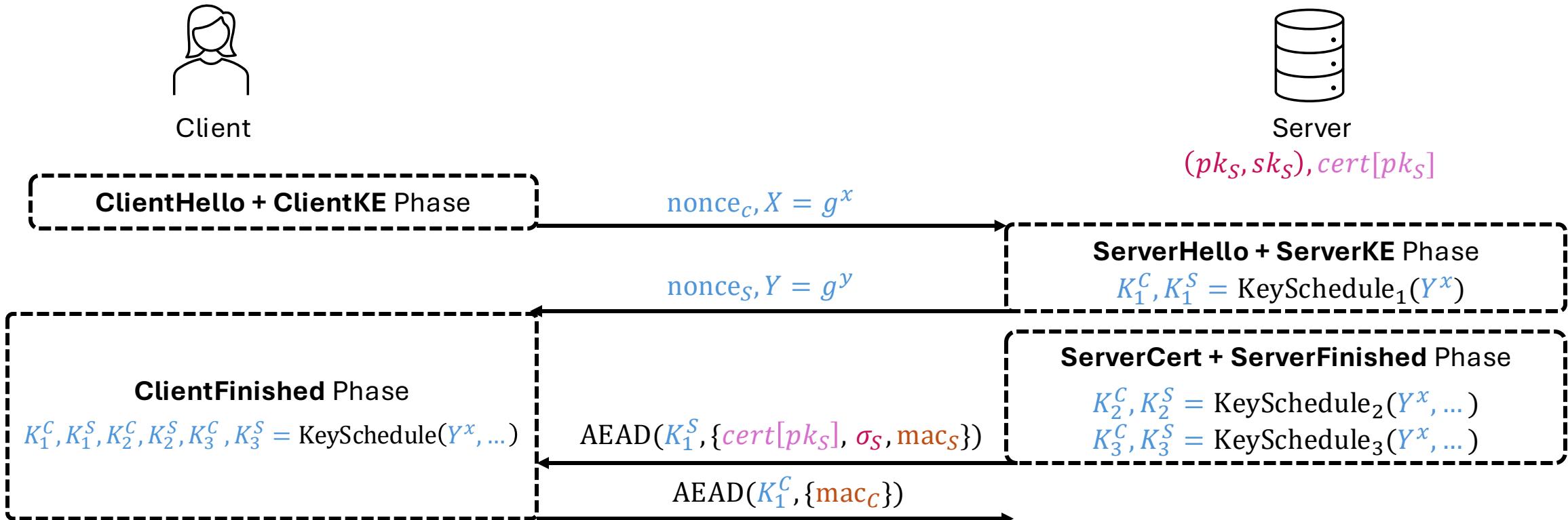
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



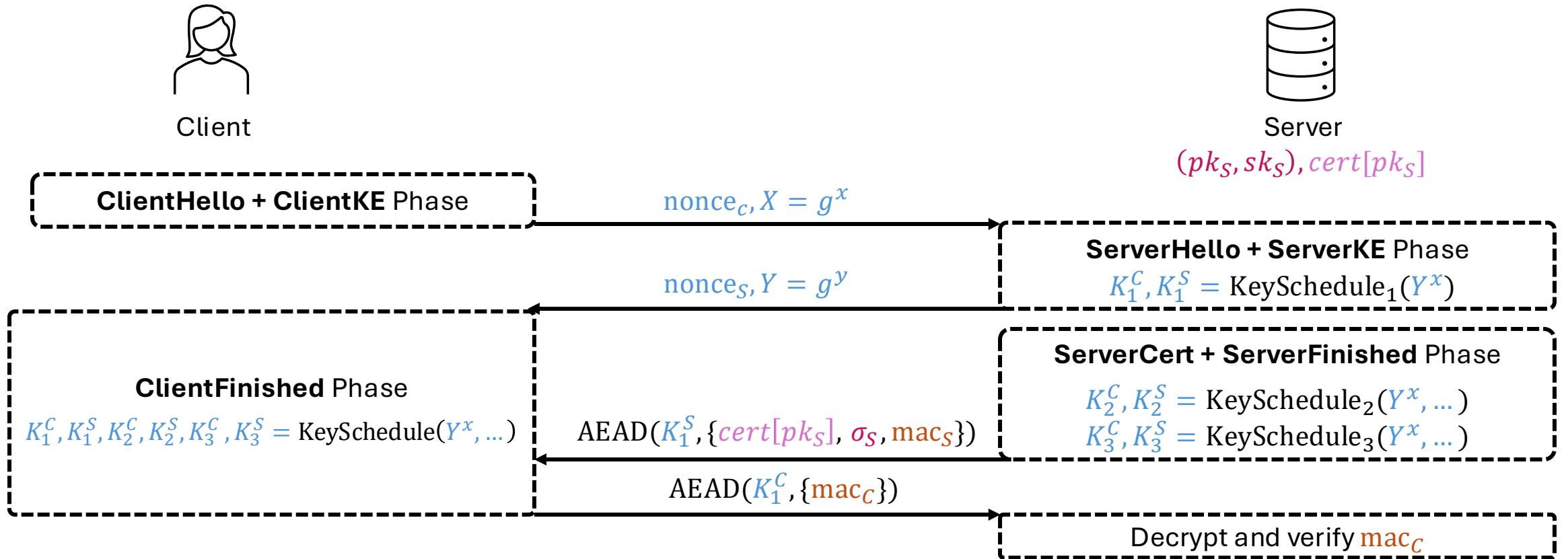
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



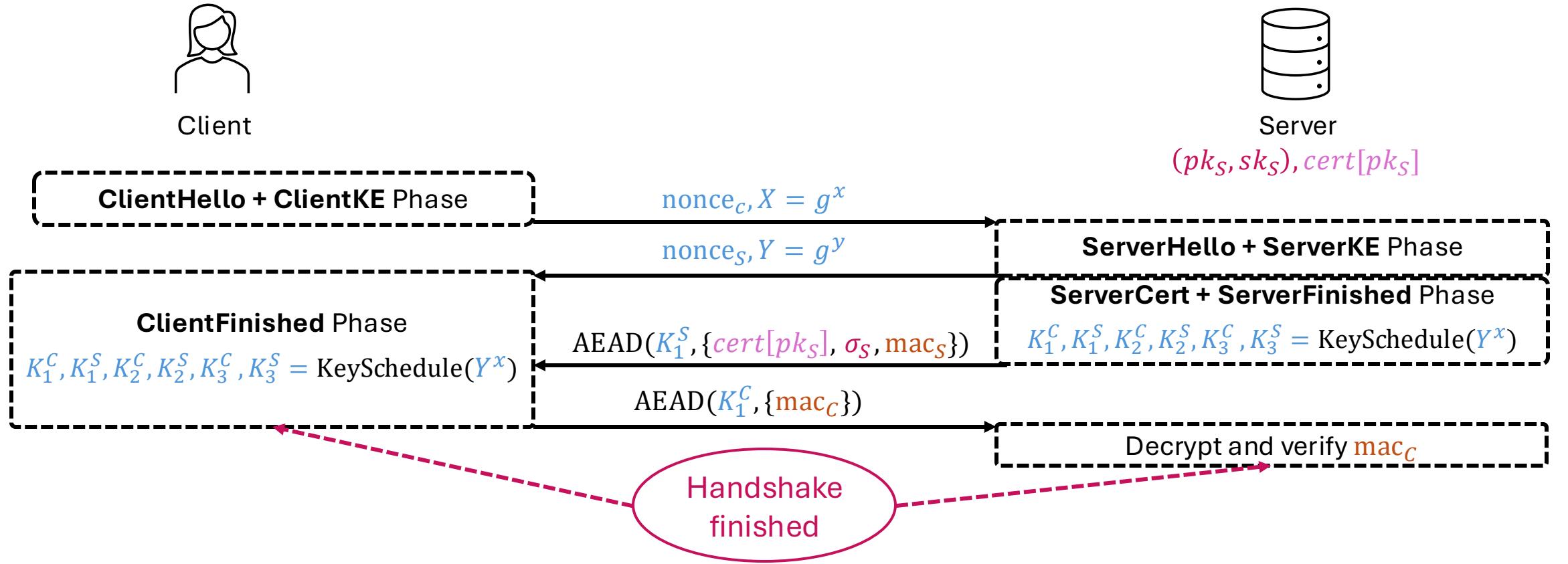
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



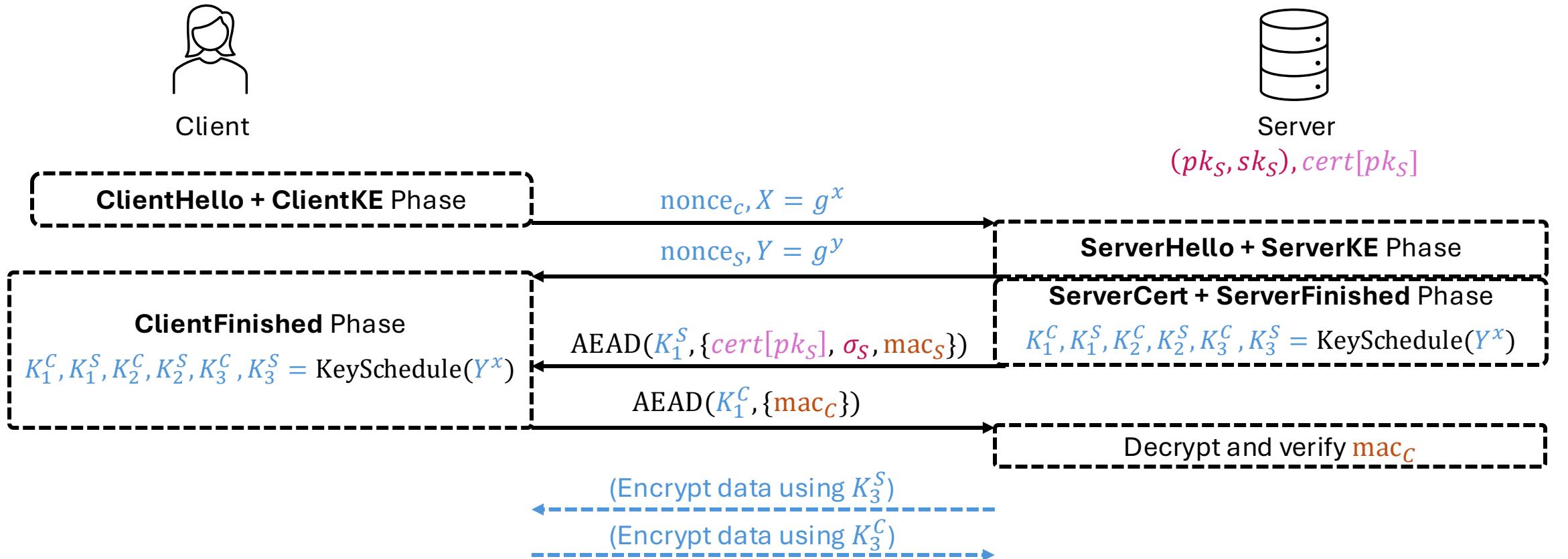
TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



TLS Handshake Protocol

- TLS 1.3 handshake protocol (Simplified description, we ignore the TLS key schedule)



Case Study: HTTPs = HTTP over TLS

HTTP



Client
(Browser)

v.S.



<http://www.google.com>
Port: 80

HTTPs

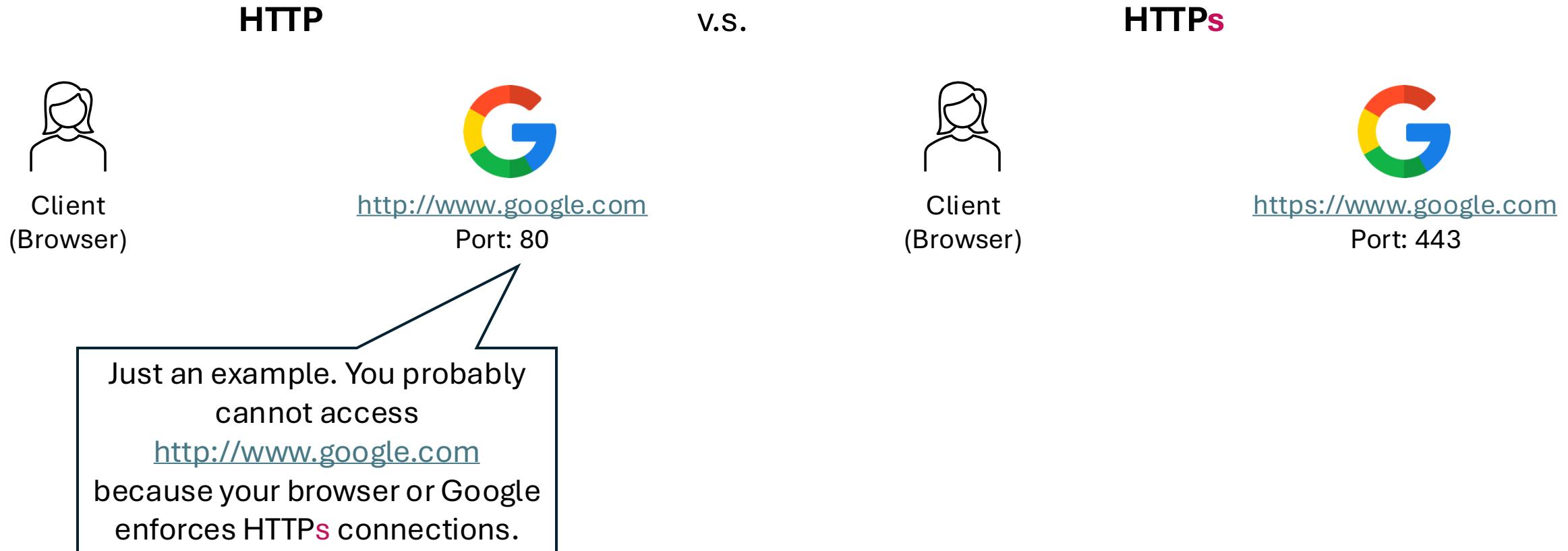


Client
(Browser)



<https://www.google.com>
Port: 443

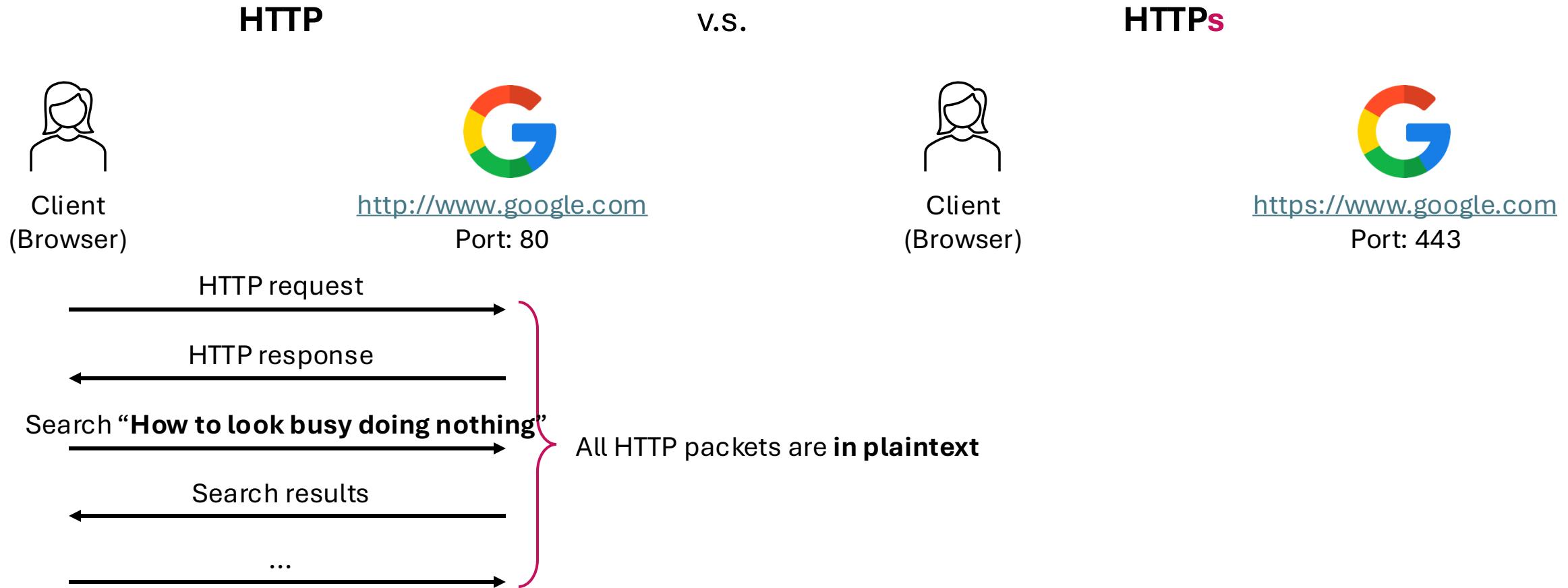
Case Study: HTTPs = HTTP over TLS



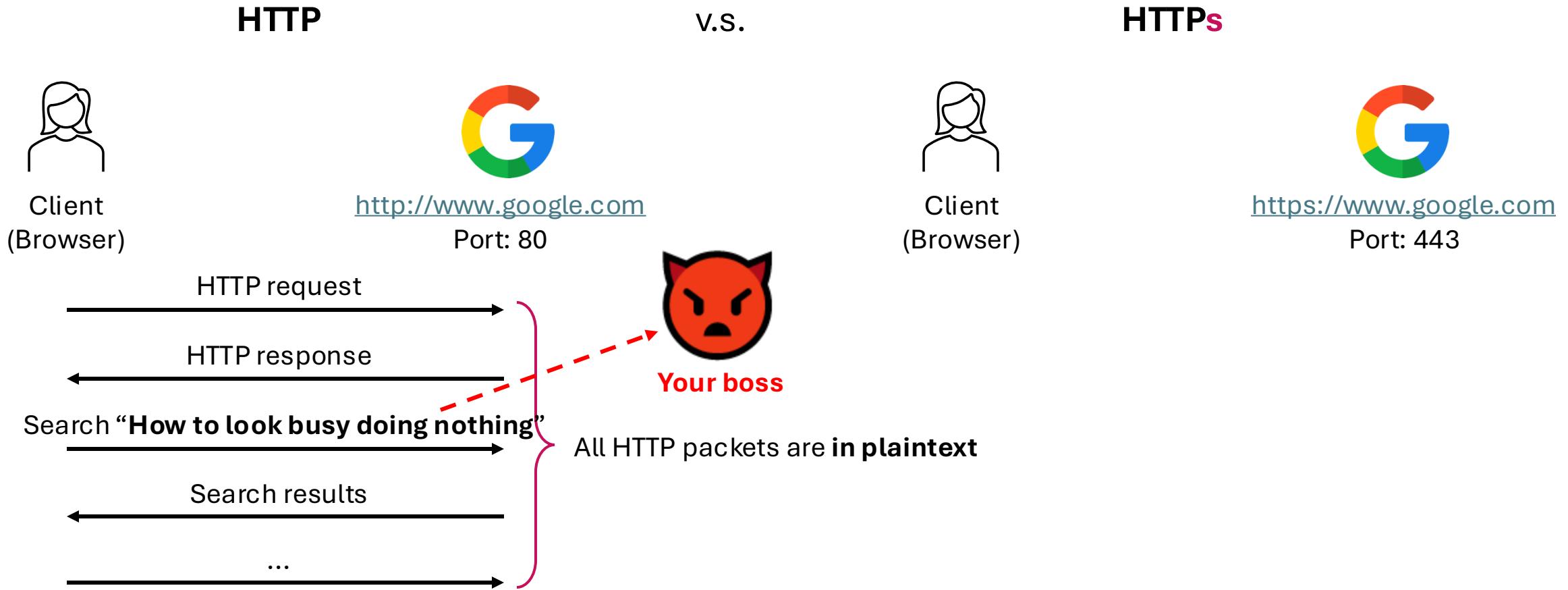
Case Study: HTTPs = HTTP over TLS



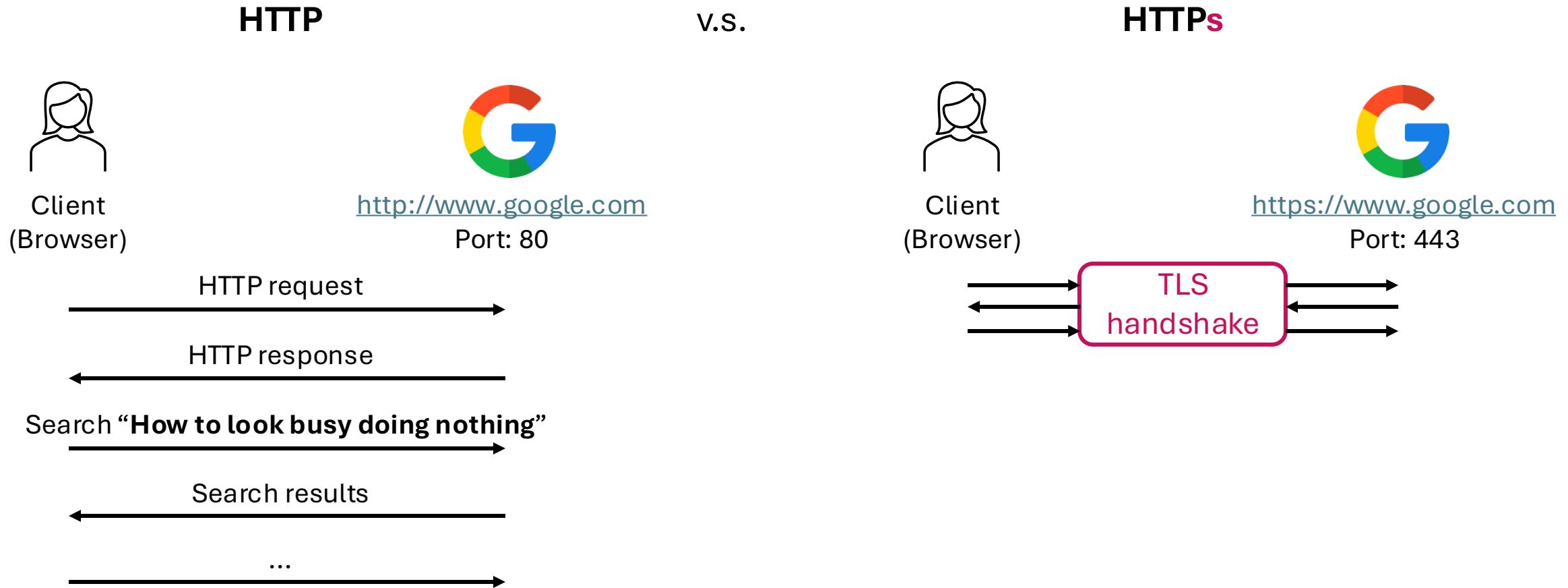
Case Study: HTTPs = HTTP over TLS



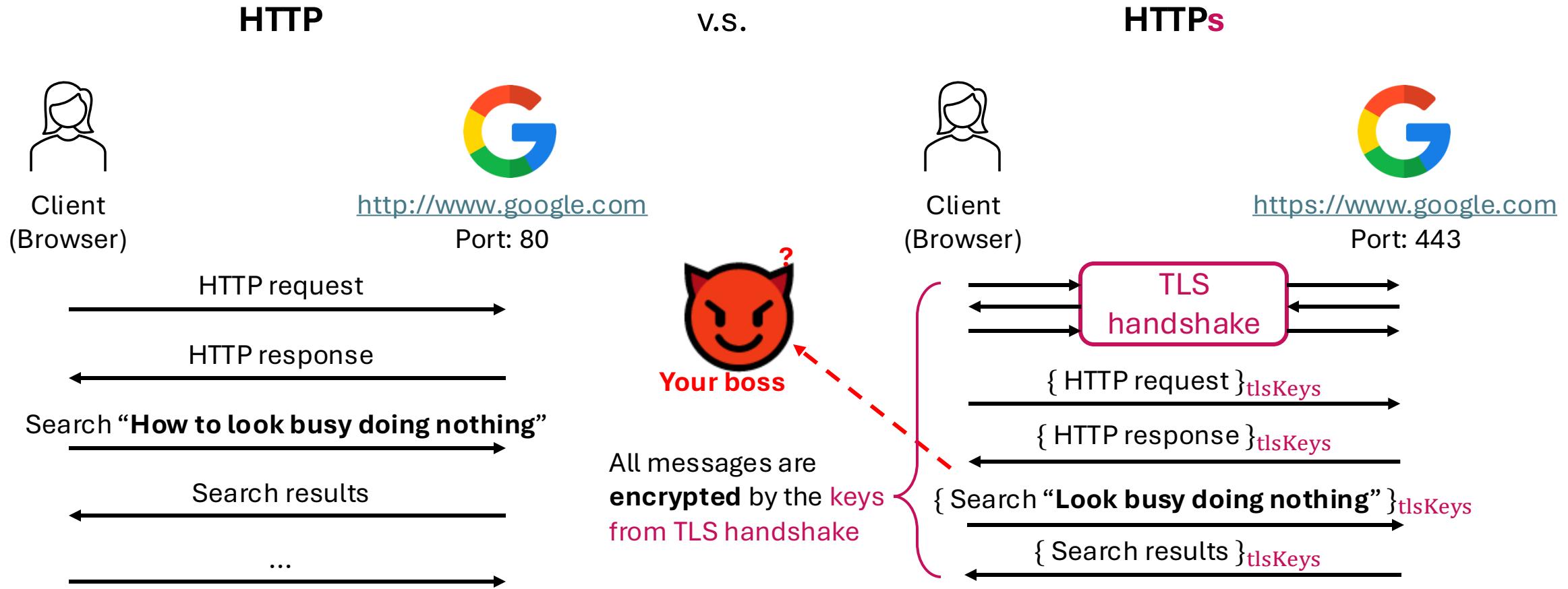
Case Study: HTTPs = HTTP over TLS



Case Study: HTTPs = HTTP over TLS



Case Study: HTTPs = HTTP over TLS



Homework

- Task 1: Implement the signed DH Hellman protocol.
- Task 2: Implement the tweaked TLS handshake protocol (using the KeySchedule algorithms in the next slide)
- Task 3: Implement the randomness reuse attack on ECDSA
- **Deadline: Nov 28th, 2025**
- Example code:
 - DHKE demo (from the second lecture)
 - Signature demo (using Ed25519)
 - HKDF-Extract-Expand demo
 - ECDSA demo

Homework

- Implement the tweaked TLS handshake protocol
 - Use the simplified key schedule algorithm:

KeySchedule₁(g^{xy}):

1. HS = DeriveHS(g^{xy})
2. K_1^C = HKDF.Expand(HS, SHA256("ClientKE"))
3. K_1^S = HKDF.Expand(HS, SHA256("ServerKE"))
4. **return** K_1^C, K_1^S

KeySchedule₂($\text{nonce}_C, X, \text{nonce}_S, Y, g^{xy}$):

1. HS = DeriveHS(g^{xy})
2. ClientKC = SHA256($\text{nonce}_C \parallel X \parallel \text{nonce}_S \parallel Y \parallel$ "ClientKC") // " \parallel " is the concatenation operation
3. ServerKC = SHA256($\text{nonce}_C \parallel X \parallel \text{nonce}_S \parallel Y \parallel$ "ServerKC")
4. K_2^C = HKDF.Expand(HS, ClientKC)
5. K_2^S = HKDF.Expand(HS, ServerKC)
6. **return** K_2^C, K_2^S

- ... (next page)

Warning: This key schedule scheme may not be secure. If you want to use TLS in real-world applications, please follow the TLS 1.3 standard

Homework

Warning: This key schedule scheme may not be secure. If you want to use TLS in real-world applications, please follow the TLS 1.3 standard

KeySchedule₃($\text{nonce}_C, X, \text{nonce}_S, Y, g^{xy}, \sigma, \text{cert}[pk_S], \text{mac}_S$):

1. HS = DeriveHS(g^{xy})
2. dHS = HKDF.Expand(HS, SHA256("DerivedHS"))
3. MS = HKDF.Extract(dHS, 0) // 0 = zeros (bytes) of length 32
2. ClientSKH = SHA256($\text{nonce}_C \parallel X \parallel \text{nonce}_S \parallel Y \parallel \sigma \parallel \text{cert}[pk_S] \parallel \text{mac}_S \parallel \text{"ClientEncK"}$)
3. ServerSKH = SHA256($\text{nonce}_C \parallel X \parallel \text{nonce}_S \parallel Y \parallel \sigma \parallel \text{cert}[pk_S] \parallel \text{mac}_S \parallel \text{"ServerEncK"}$)
2. K_3^C = HKDF.Expand(MS, ClientSKH)
3. K_3^S = HKDF.Expand(MS, ServerSKH)
4. **return** K_3^C, K_3^S

- **How to compute the signature/MAC code:**

For server: $\sigma = \text{Sign}(sk_S, \text{SHA256}(\text{nonce}_C \parallel X \parallel \text{nonce}_S \parallel Y \parallel \text{cert}[pk_S]))$ // Use DSA with SHA256 and P256

For server: $\text{mac}_S = \text{HMAC}\left(K_2^S, \text{SHA256}(\text{nonce}_C \parallel X \parallel \text{nonce}_S \parallel Y \parallel \sigma \parallel \text{cert}[pk_S] \parallel \text{"ServerMAC"})\right)$

For client: $\text{mac}_C = \text{HMAC}\left(K_2^C, \text{SHA256}(\text{nonce}_C \parallel X \parallel \text{nonce}_S \parallel Y \parallel \sigma \parallel \text{cert}[pk_S] \parallel \text{"ClientMAC"})\right)$

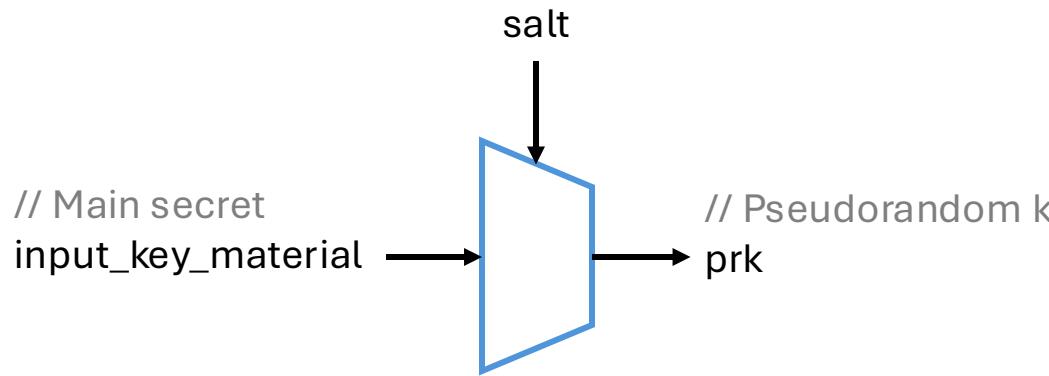
- **How to verify HMAC:** To verify if mac is the *valid* HMAC code of M with respect to the key K,
Just check: $\text{mac} =? \text{HMAC}(K, M)$

Further Reading

- RFC 8446 - TLS 1.3: <https://datatracker.ietf.org/doc/html/rfc8446>
- RFC 2818 - HTTP over TLS: <https://datatracker.ietf.org/doc/html/rfc2818>
- Felix Günther's lecture notes on TLS 1.3: https://www.felixguenther.info/teaching/2019-tls-seminar/2019-tls-seminar_02-21_TLS13-intro-MSKE-MKC.pdf
- Cryptography analysis of TLS 1.3 handshake: <https://eprint.iacr.org/2020/1044>

Case study: HKDF.(Extract, Expand)

// Optional, may add some extra secret information



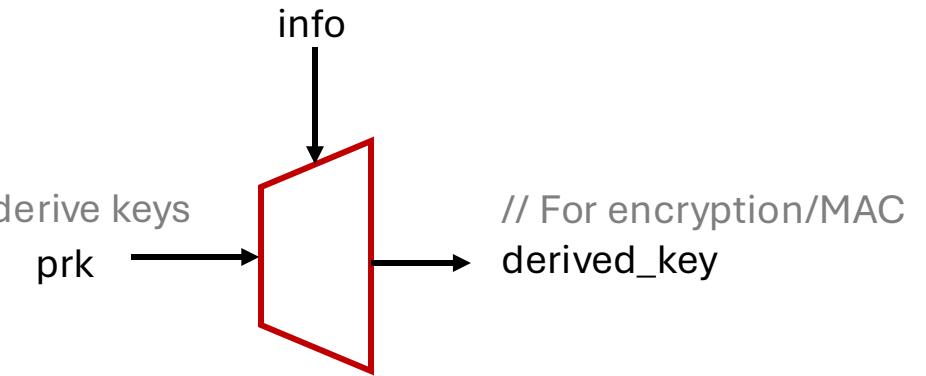
// Main secret
input_key_material

salt

prk

prk = HKDF.Extract(ikm, salt)

// Optional, make the derived key context-specific



prk

info

// For encryption/MAC
derived_key

derived_key = HKDF.Expand(prk, info)

Case Study: ECDSA

- ECDSA (Elliptic Curve Digital Signature Algorithm): DSA based on Elliptic Curve
- ECDSA (based on EC) vs DSA (based on Module Integer Groups)
- Why do we prefer Elliptic Curve Groups over Module Integer Groups?
 - **Stronger:** For example, a 256-bit elliptic curve key offers comparable security to a 3072-bit RSA key...
 - **Shorter:** Smaller key size => shorter ciphertext/signature, reducing bandwidth usage...
 - **Faster:** Smaller key size => faster computations and lower computation overhead...

Case Study: ECDSA

- A quick background on Elliptic Curve Groups
- An elliptic curve E is a plane curve which consists of the points satisfying the equation:

$$E: y^2 = x^3 + ax + b$$

- In Elliptic-Curve Cryptography (ECC), we use ECs over finite fields.

Case Study: ECDSA

- ECDSA (Elliptic Curve Digital Signature Algorithm): DSA based on Elliptic Curve
- Public parameter (publicly known): $(\text{CURVE}, \mathbb{G}, g, p)$
 - CURVE: Tell the users what *the elliptic curve and equations* are being used.
 - (\mathbb{G}, g, p) : A subgroup \mathbb{G} over CURVE with a **large prime** order p . The base point (generator) g generates \mathbb{G} .
- Key Generation:
 - $sk = d \leftarrow_{\$} \mathbb{Z}_p^*$ // ($= \{1, 2, \dots, p-1\}$, here “*” means that we exclude zero)
 - $pk = d \circ g$ // “ \circ ” is the “exponential operator” of Elliptic Curve, just like g^d . and you cannot recover d given $d \circ g$

Case Study: ECDSA

- Signing algorithm ($sk = d$: secret key, m : message):

- 1) $e' = \text{Hash}(m)$ // “Compress” the message and get its digest
- 2) $e = \lceil \log_2 p \rceil$ leftmost bits of e' // Truncate some bits to fit in the format
- 3) $k \leftarrow_{\$} \mathbb{Z}_p^*$
- 4) $(x, y) = k \circ g$ // g is the base point
- 5) $r = x \pmod p$ // Now r is an integer module p . Given x , y is determined.
- 6) Assert $[x \pmod p \neq 0]$ // Make sure we do not get a “trivial point”
- 7) $s = k^{-1} \cdot (e + r \cdot d) \pmod p$ // Signing
- 8) return (r, s)

Case Study: ECDSA

- Verification algorithm (pk : public key, m : message, (r, s) : signature):
 - 1) $e' = \text{Hash}(m)$ // “Compress” the message and get its digest
 - 2) $e = \lceil \log_2 p \rceil$ leftmost bits of e' // Truncate some bits to fit in the format
 - 3) $u_1 = e \cdot s^{-1} \pmod{p}$
 - 4) $u_2 = r \cdot s^{-1} \pmod{p}$
 - 5) $(x, y) = u_1 \circ g + u_2 \circ pk$ // Recalculate the point
 - 6) Accept this signature if $x \equiv r \pmod{p}$. Otherwise, reject.

Case Study: ECDSA

- Verification algorithm (pk : public key, m : message, (r, s) : signature):
 - 1) $e' = \text{Hash}(m)$ // “Compress” the message and get its digest
 - 2) $e = \lceil \log_2 p \rceil$ leftmost bits of e' // Truncate some bits to fit in the format
 - 3) $u_1 = e \cdot s^{-1} \pmod{p}$
 - 4) $u_2 = r \cdot s^{-1} \pmod{p}$
 - 5) $(x, y) = u_1 \circ g + u_2 \circ pk$ // Recalculate the point
 - 6) Accept this signature if $x \equiv r \pmod{p}$. Otherwise, reject.
- You can prove that the verification algorithm works correctly.
- ECDSA has unforgeability if the *Discrete Logarithm Problem* over the elliptic curve is hard.

Case Study: ECDSA

- **Do not reuse nonce** in DSA(/Schnorr/SM2/...) !
- In DSA, nonce (the k value) reuse => private key recovery => break the unforgeability

Case Study: ECDSA

- **Do not reuse nonce** in DSA(/Schnorr/SM2/...) !
- In DSA, nonce (the k value) reuse => private key recovery => break the unforgeability

Two DSA signatures of different messages: $m_1, \sigma_1 = (r, s_1)$ with nonce k (Same value r if k is the same)
 $m_2, \sigma_2 = (r, s_2)$ with nonce k

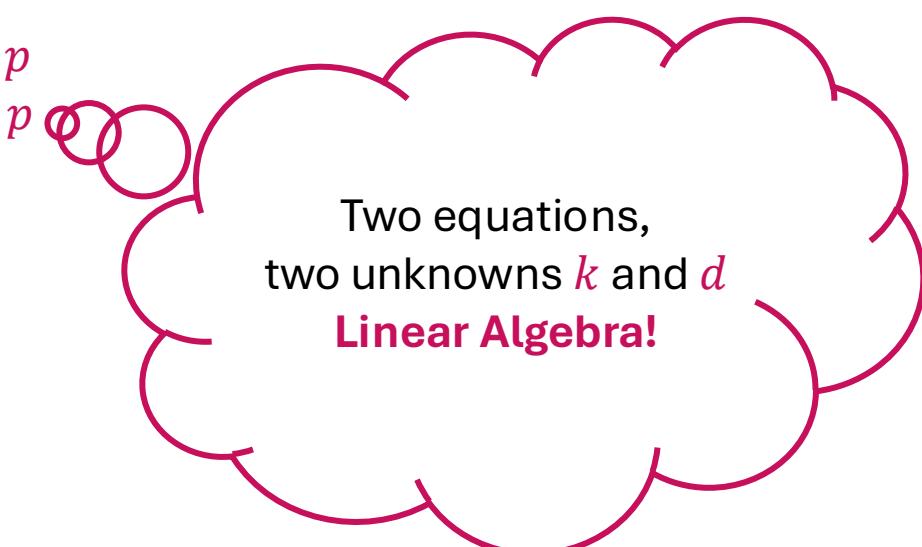
By DSA construction: $s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \pmod{p}$
 $s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \pmod{p}$

Case Study: ECDSA

- **Do not reuse nonce** in DSA(/Schnorr/SM2/...) !
- In DSA, nonce (the k value) reuse => private key recovery => break the unforgeability

Two DSA signatures
of different messages: $m_1, \sigma_1 = (r, s_1)$ with nonce k
 $m_2, \sigma_2 = (r, s_2)$ with nonce k (Same value r if k is the same)

By DSA construction: $s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \pmod{p}$
 $s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \pmod{p}$



Two equations,
two unknowns k and d
Linear Algebra!

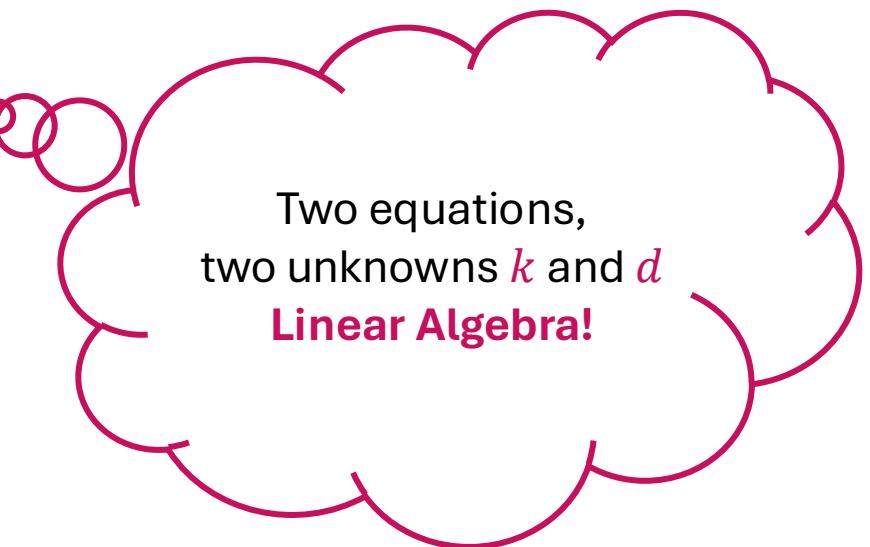
Case Study: ECDSA

- **Do not reuse nonce** in DSA(/Schnorr/SM2/...) !
- In DSA, nonce (the k value) reuse => private key recovery => break the unforgeability

Two DSA signatures
of different messages: $m_1, \sigma_1 = (r, s_1)$ with nonce k
 $m_2, \sigma_2 = (r, s_2)$ with nonce k (Same value r if k is the same)

By DSA construction: $s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \pmod{p}$
 $s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \pmod{p}$

By Linear Algebra: $\begin{bmatrix} s_1 & r \\ s_2 & r \end{bmatrix} \begin{bmatrix} k \\ d \end{bmatrix} = \begin{bmatrix} H(m_1) \\ H(m_2) \end{bmatrix} \pmod{p}$



Two equations,
two unknowns k and d
Linear Algebra!

Case Study: ECDSA

- **Do not reuse nonce** in DSA(/Schnorr/SM2/...) !
- In DSA, nonce (the k value) reuse => private key recovery => break the unforgeability

Two DSA signatures of different messages: $m_1, \sigma_1 = (r, s_1)$ with nonce k (Same value r if k is the same)
 $m_2, \sigma_2 = (r, s_2)$ with nonce k

By DSA construction: $s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \pmod{p}$
 $s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \pmod{p}$

By Linear Algebra:
$$\begin{bmatrix} k \\ d \end{bmatrix} = \begin{bmatrix} (s_1 - s_2)^{-1} \cdot (H(m_1) - H(m_2)) \\ r^{-1} \cdot (s_1 \cdot k - H(m_1)) \end{bmatrix} \pmod{p}$$

Case Study: ECDSA

- **Do not reuse nonce** in DSA(/Schnorr/SM2/...) !
- In DSA, nonce (the k value) reuse => private key recovery => break the unforgeability

Two DSA signatures of different messages: $m_1, \sigma_1 = (r, s_1)$ with nonce k (Same value r if k is the same)
 $m_2, \sigma_2 = (r, s_2)$ with nonce k

By DSA construction: $s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \pmod{p}$
 $s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \pmod{p}$

By Linear Algebra:
$$\begin{bmatrix} k \\ d \end{bmatrix} = \begin{bmatrix} (s_1 - s_2)^{-1} \cdot (H(m_1) - H(m_2)) \\ r^{-1} \cdot (s_1 \cdot k - H(m_1)) \end{bmatrix} \pmod{p}$$

- Real-world event: Hacking the PlayStation 3 (2010-2011)...
 - A typical example of: Provable secure in the theoretical world, but wrong implementation in the real world.