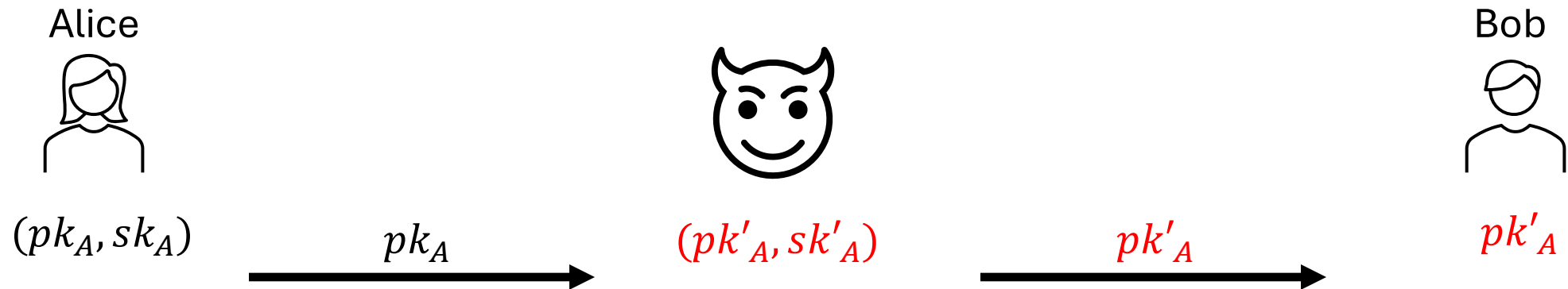


Cryptography Engineering

- Lecture 3 (Nov 06, 2024)
- Today's notes:
 - Signed Diffie-Hellman Key Exchange (SigDH) Protocol
 - TLS handshake and HTTPS protocol
- Today's coding tasks (and homework):
 - Play with HKDF
 - Implement a toy example of TLS handshake

Prevent MitM using signature/certificate

- Transporting **(malicious)** public keys



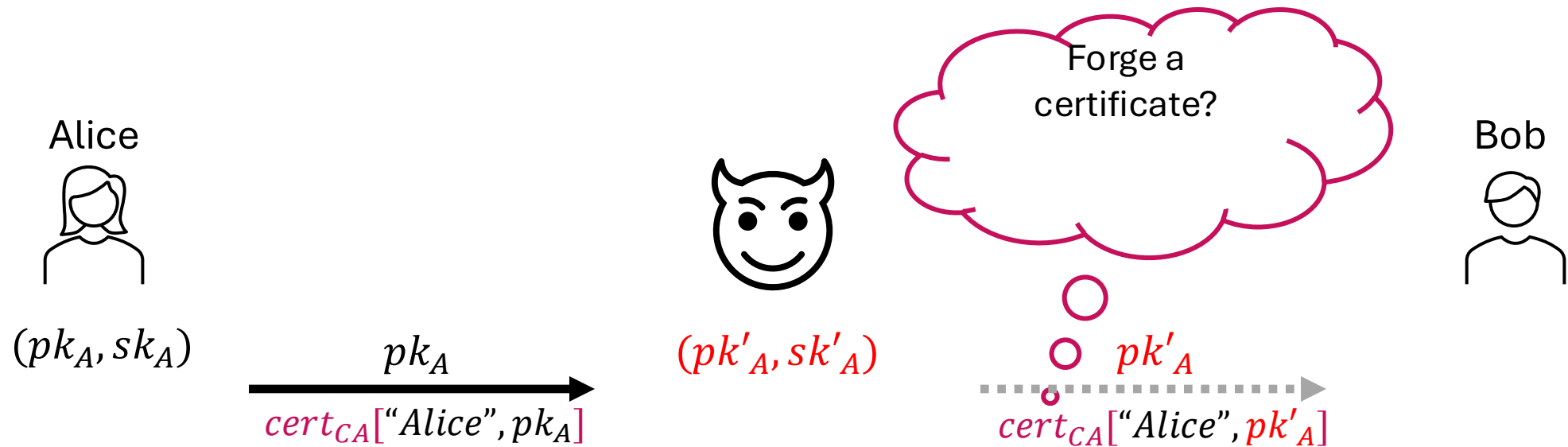
Prevent MitM using signature/certificate

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



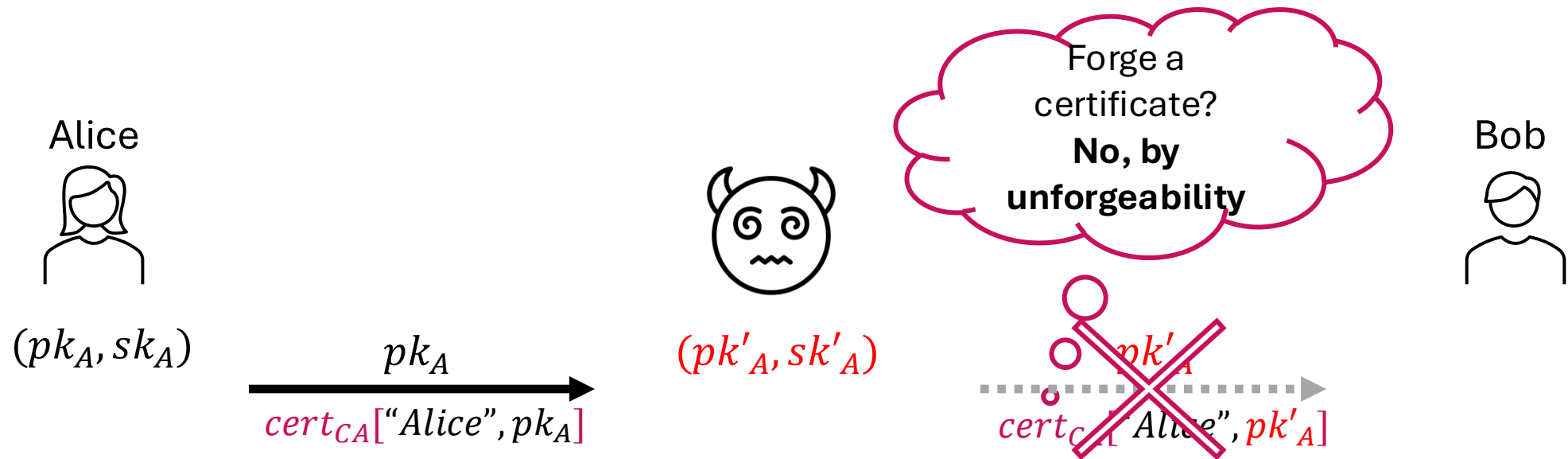
Prevent MitM using signature/certificate

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



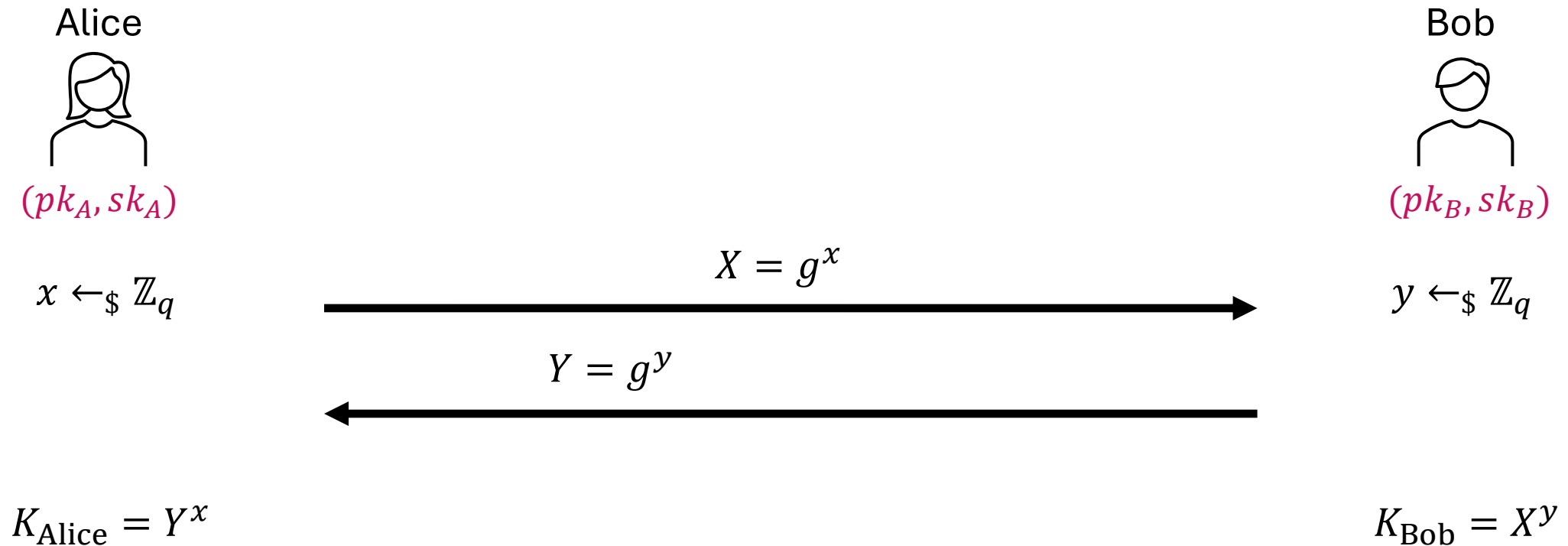
Prevent MitM using signature/certificate

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



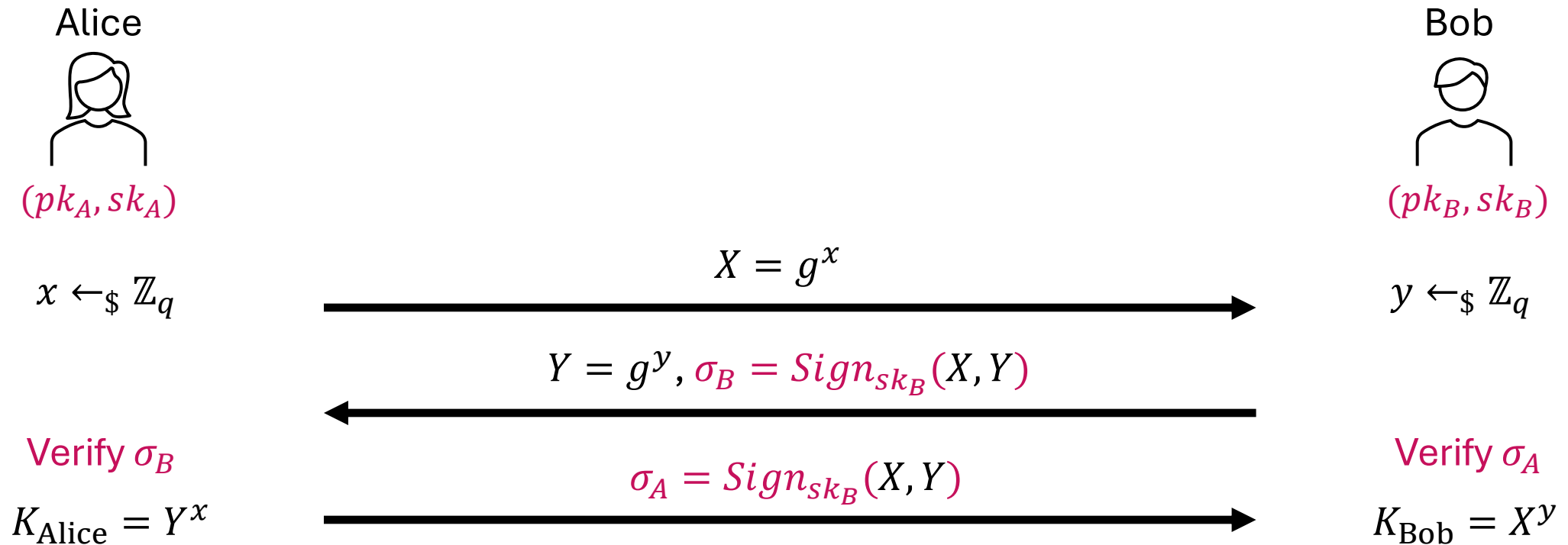
Signed Diffie-Hellman Protocol (Simplified)

- Use **signature** to avoid MitM attacks



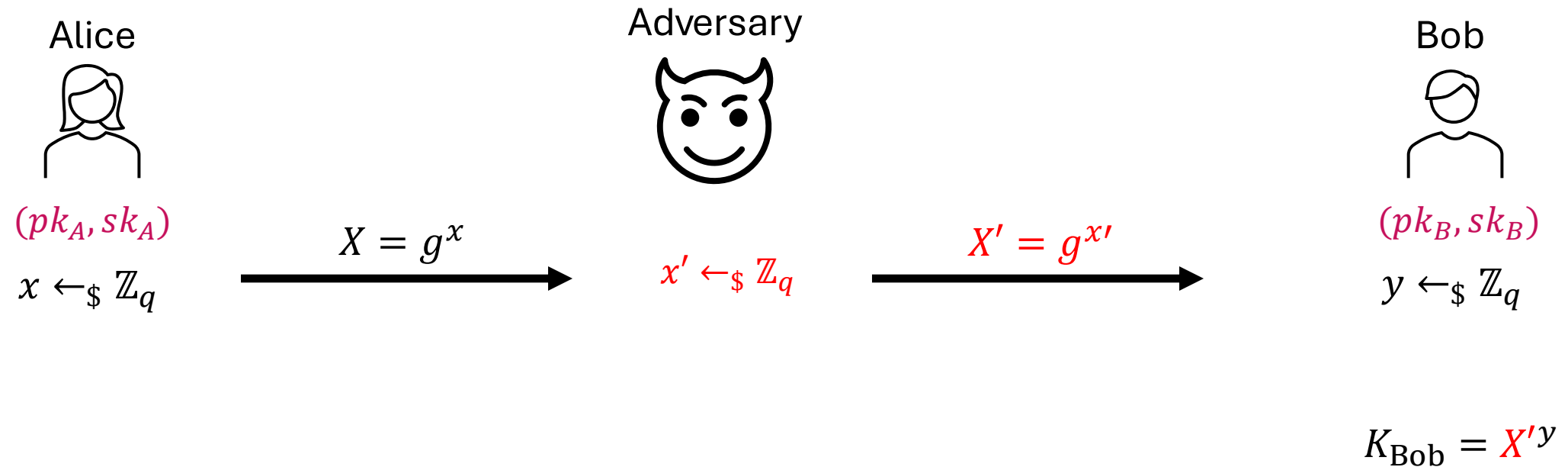
Signed Diffie-Hellman Protocol (Simplified)

- Use **signature** to avoid MitM attacks



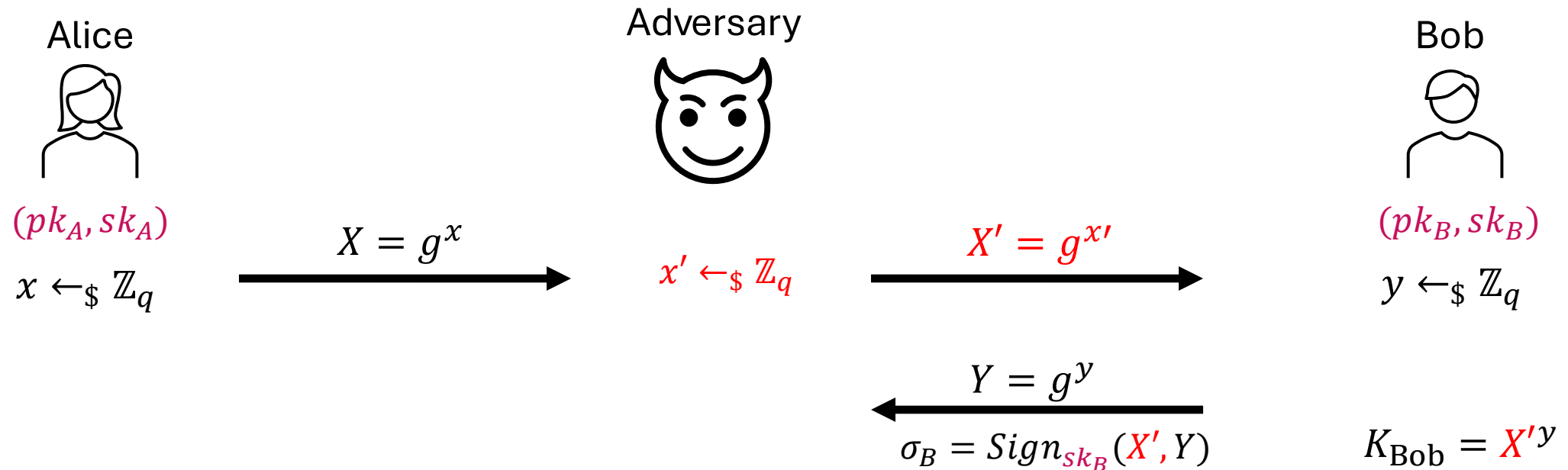
Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



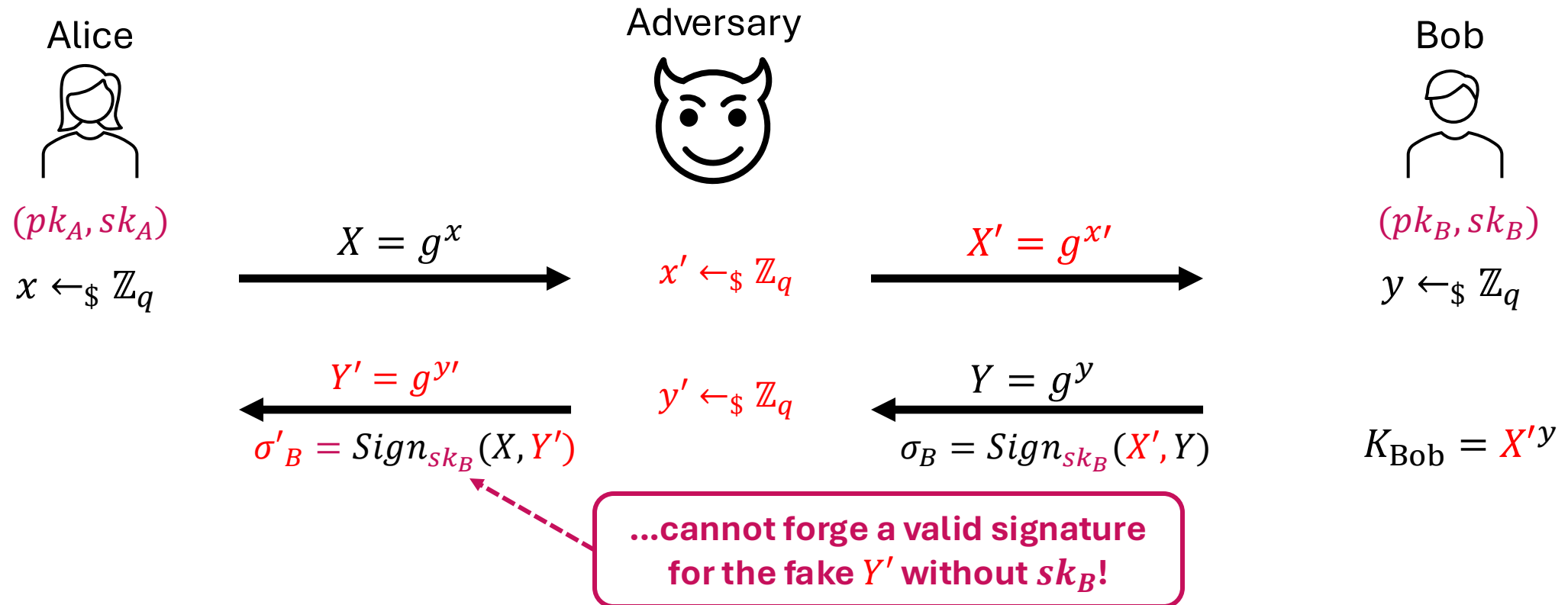
Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



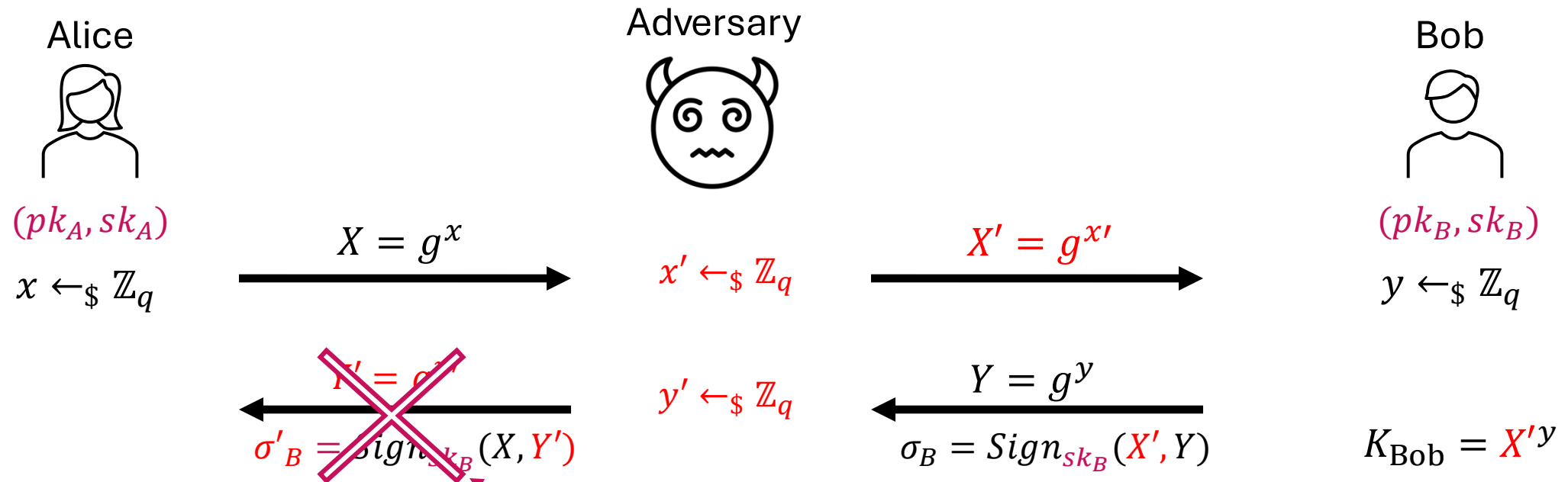
Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



Signed Diffie-Hellman Protocol

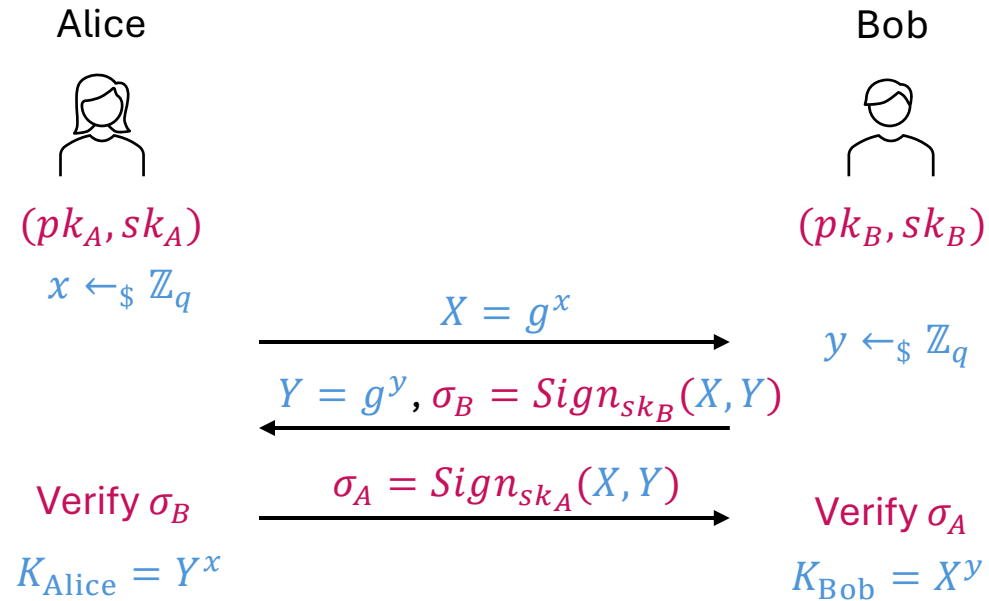
- Can we launch a MitM attack on SigDH?



- No, by unforgeability...

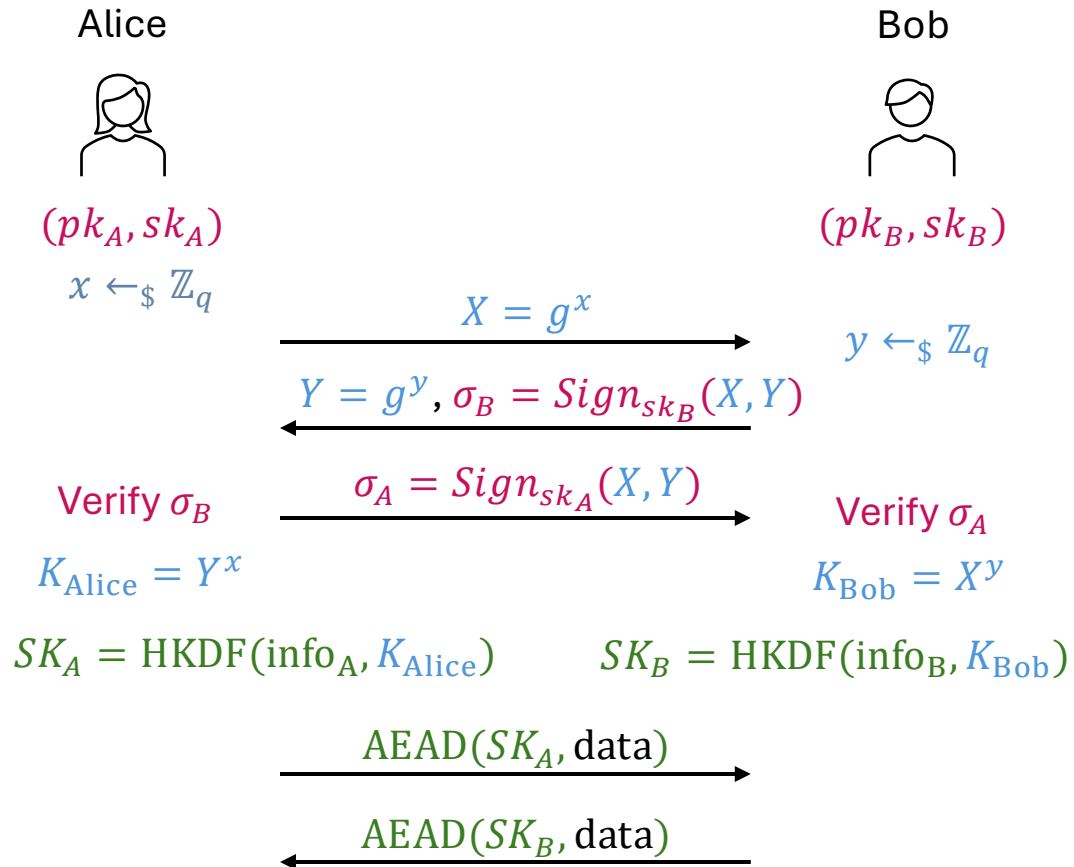
...cannot forge a valid signature for the fake Y' without sk_B !

Signed Diffie-Hellman Protocol



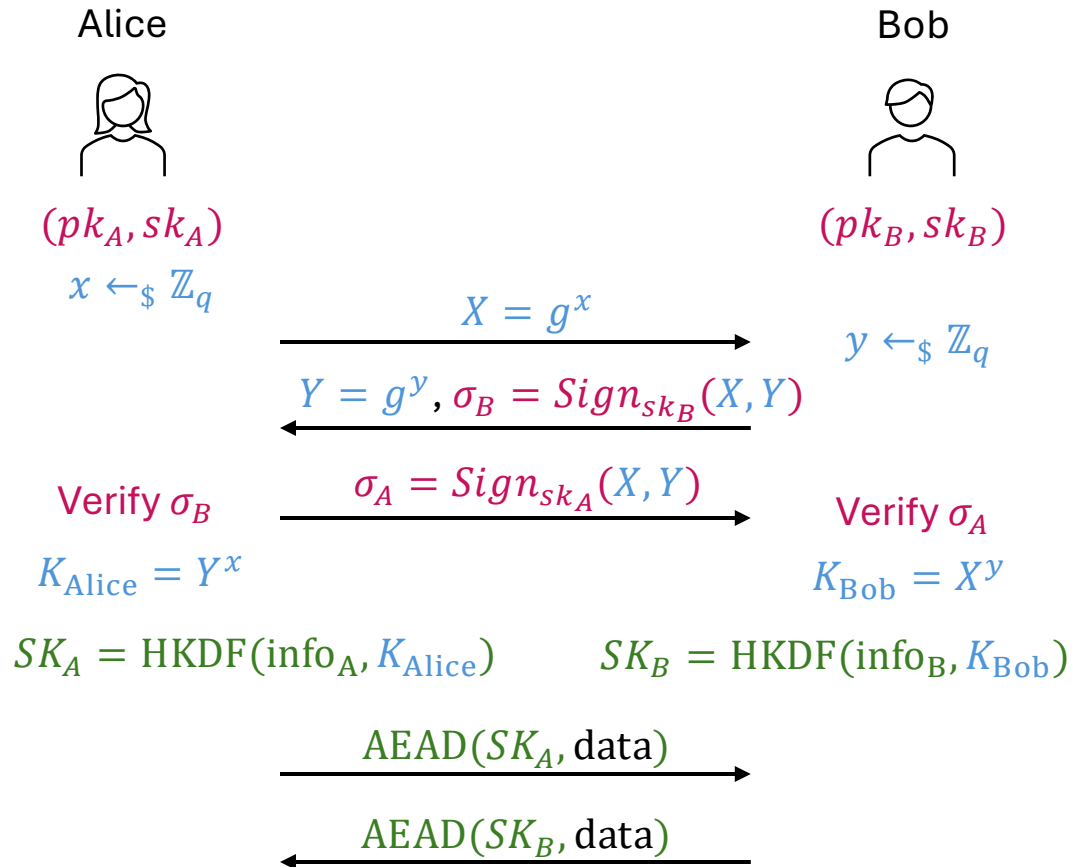
- SigDH
 - Add signature to avoid MitM
 - **Authenticated** Key Exchange

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**...
 - Note: Cryptographic algorithms “in textbooks” often contrasts with their real-world implementation

TLS Handshake Protocol

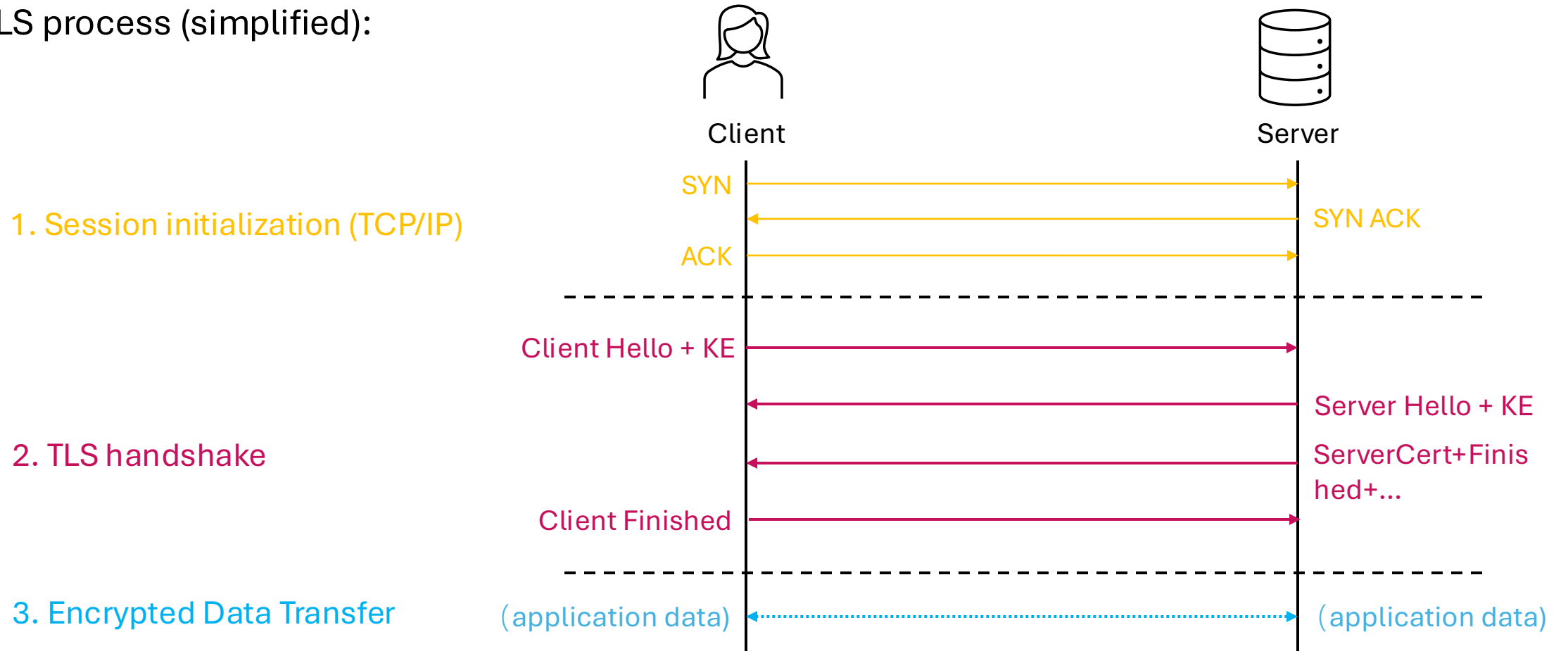
- Transport Layer Security (TLS) Protocol
 - Designed to provide communications security over an open network
 - Used in HTTPS , Email protocols, OpenVPN, MySQL-over-TLS, ...

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

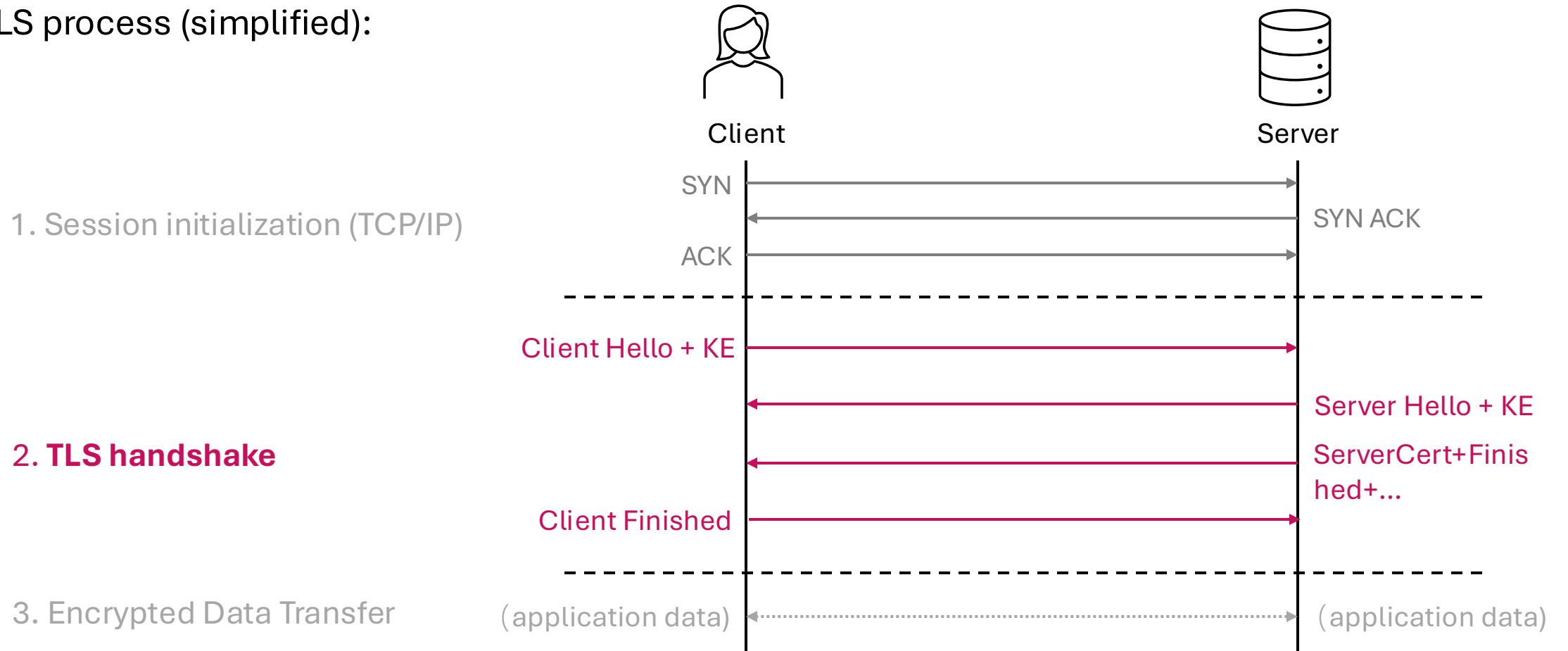
TLS Handshake Protocol

- TLS process (simplified):



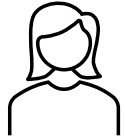
TLS Handshake Protocol

- TLS process (simplified):



TLS Handshake Protocol

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



Client

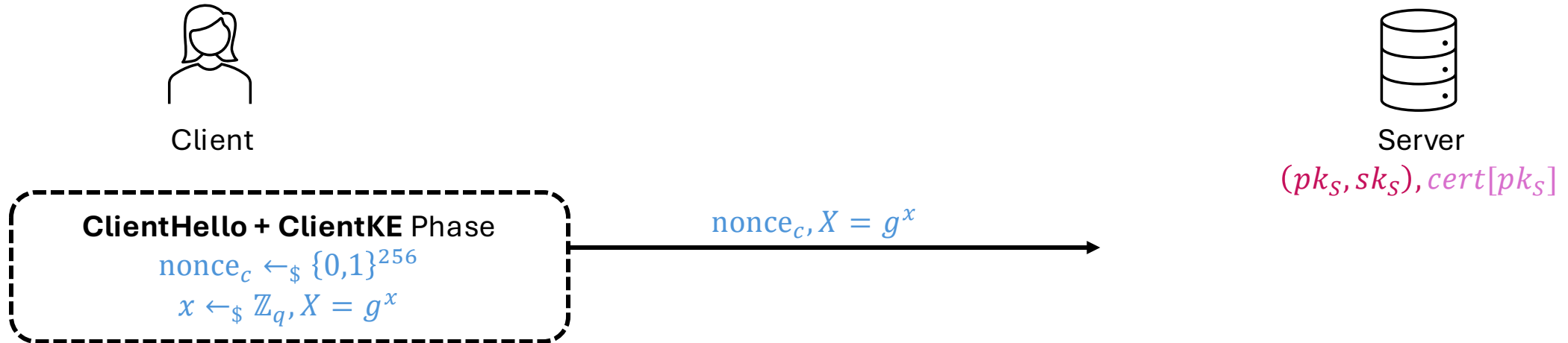


Server

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

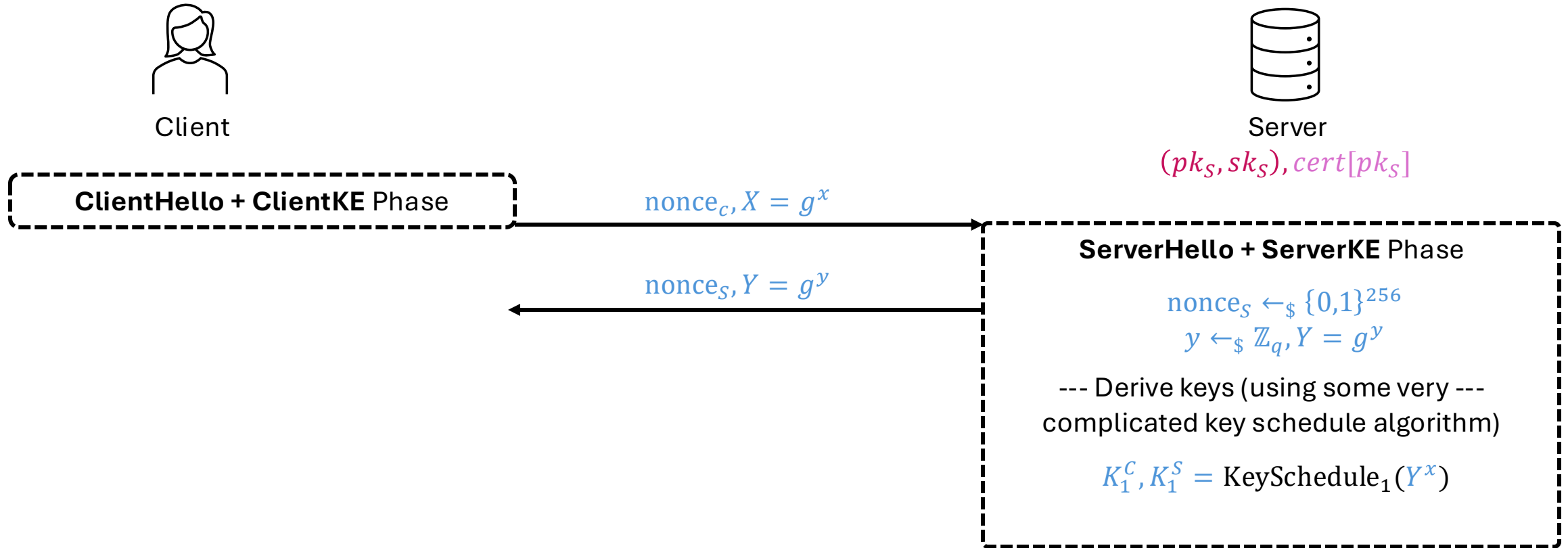
TLS Handshake Protocol

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



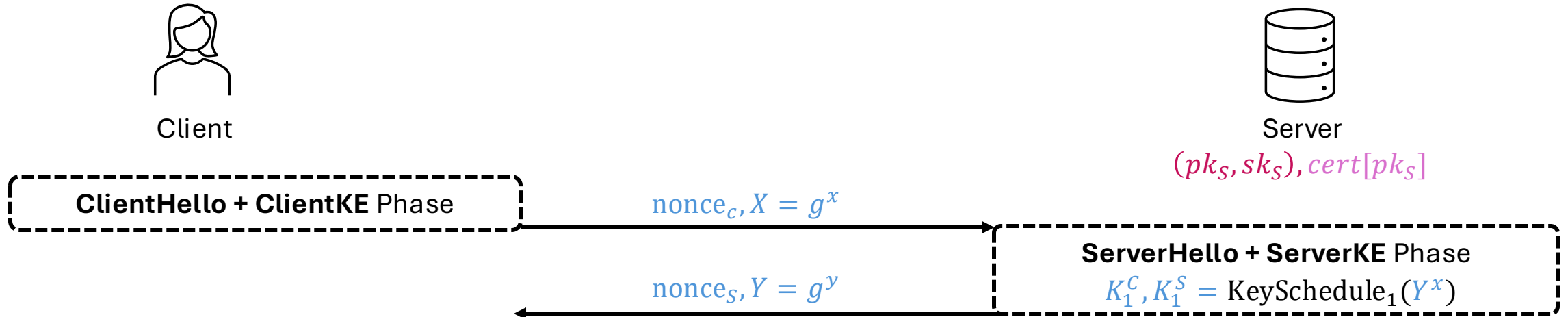
TLS Handshake Protocol

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



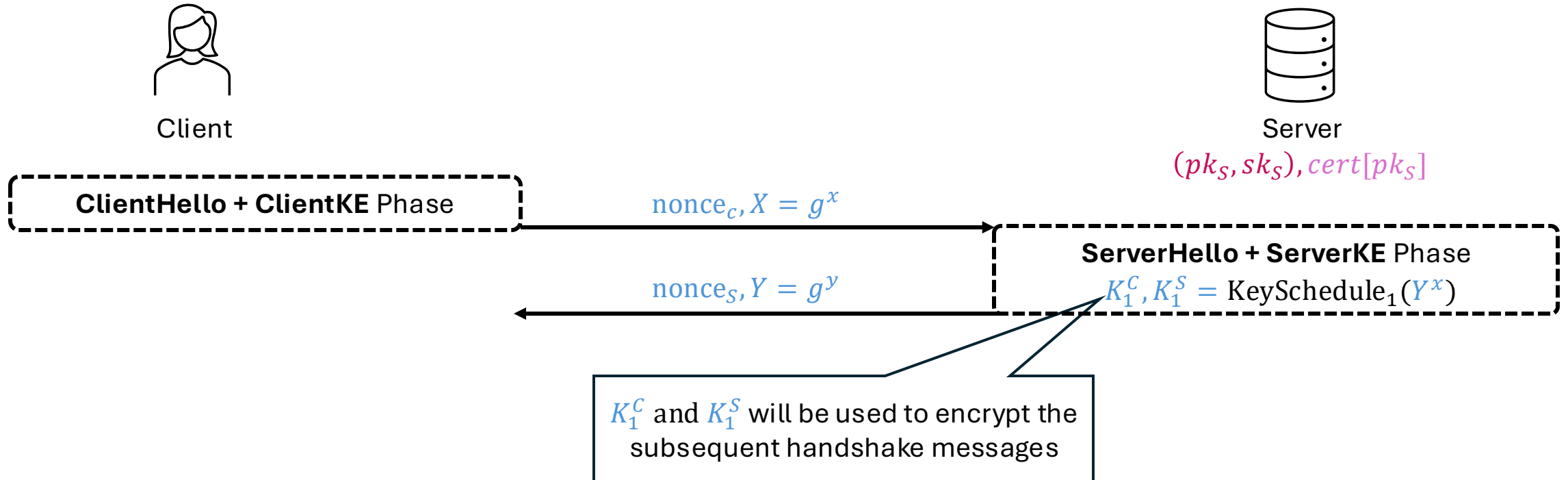
TLS Handshake Protocol

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



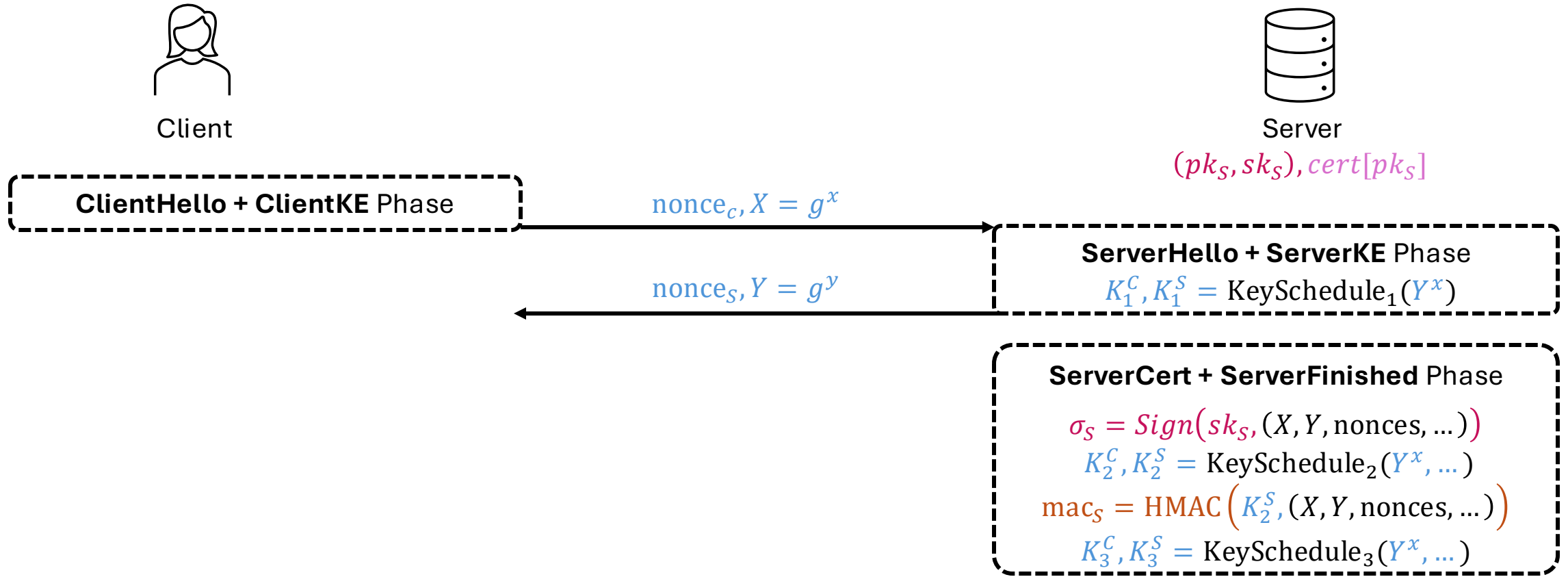
TLS Handshake Protocol

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



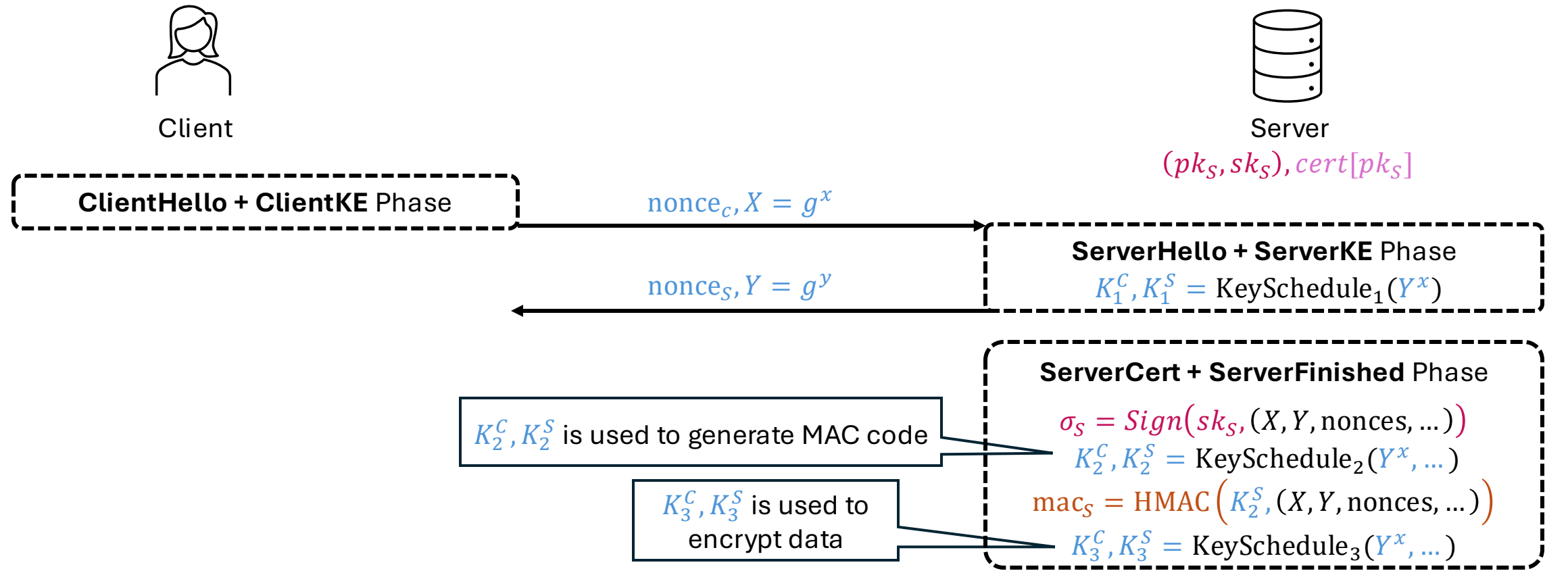
TLS Handshake Protocol

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



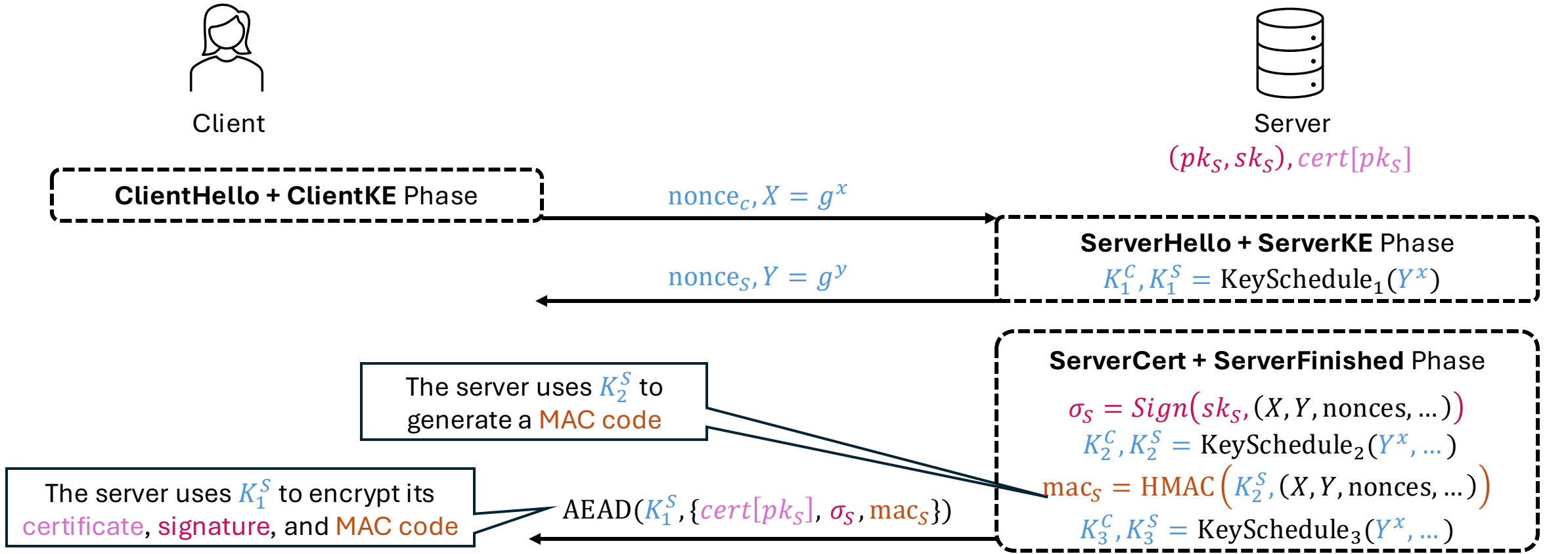
TLS Handshake Protocol

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



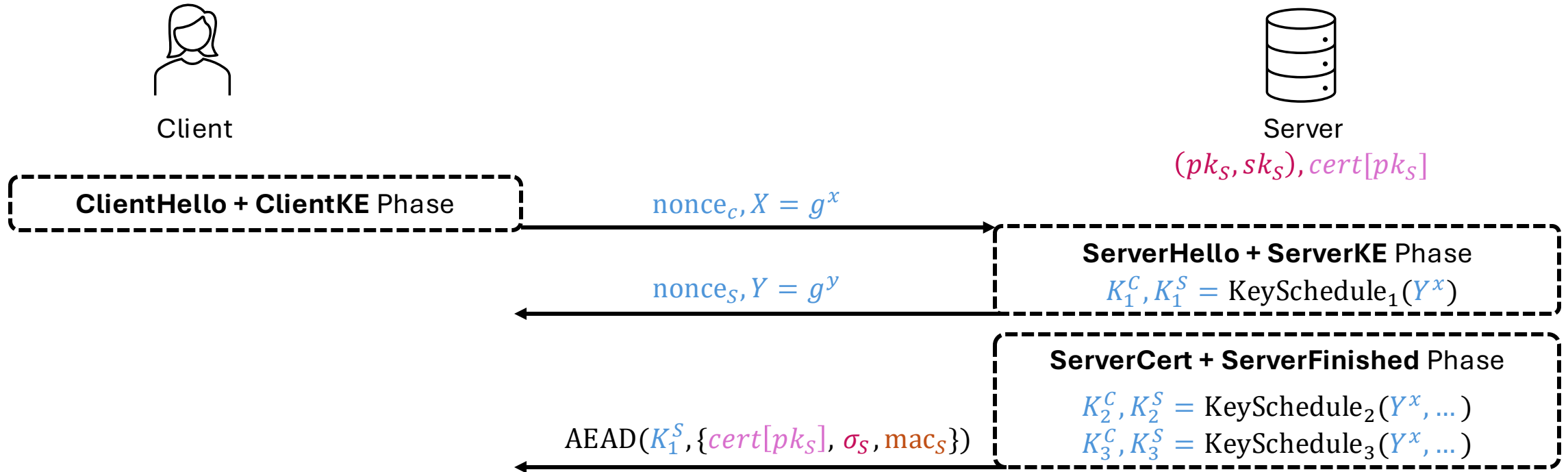
TLS Handshake Protocol

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



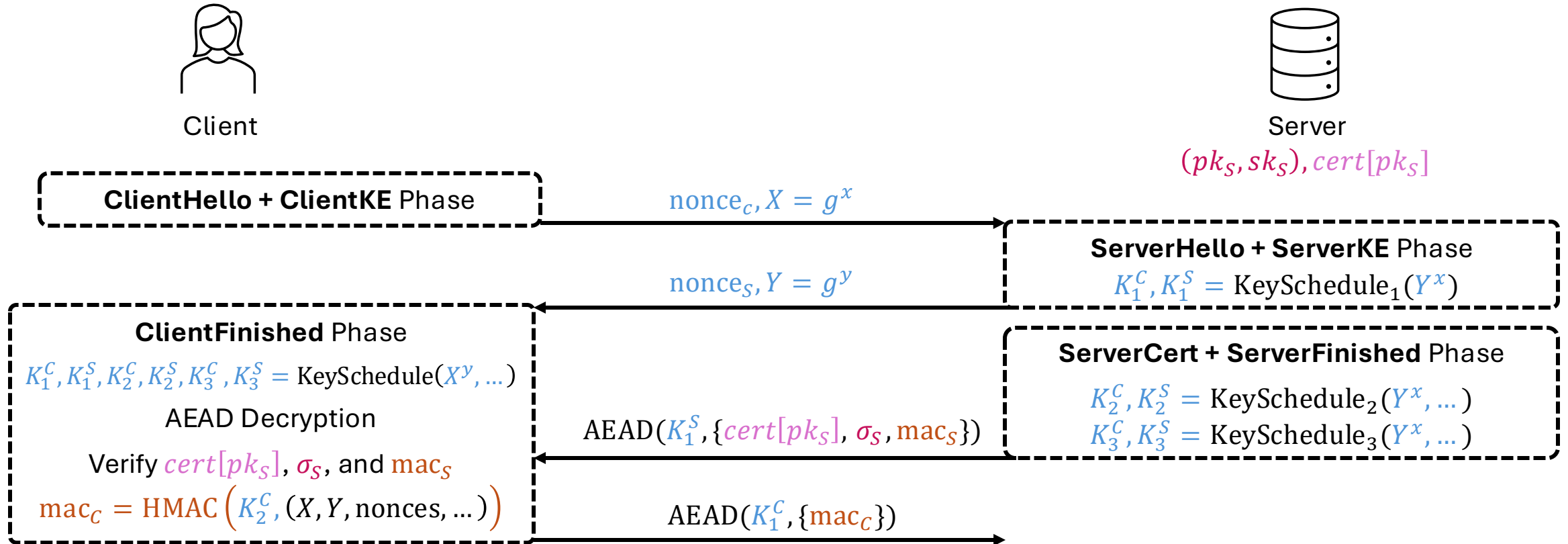
TLS Handshake Protocol

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



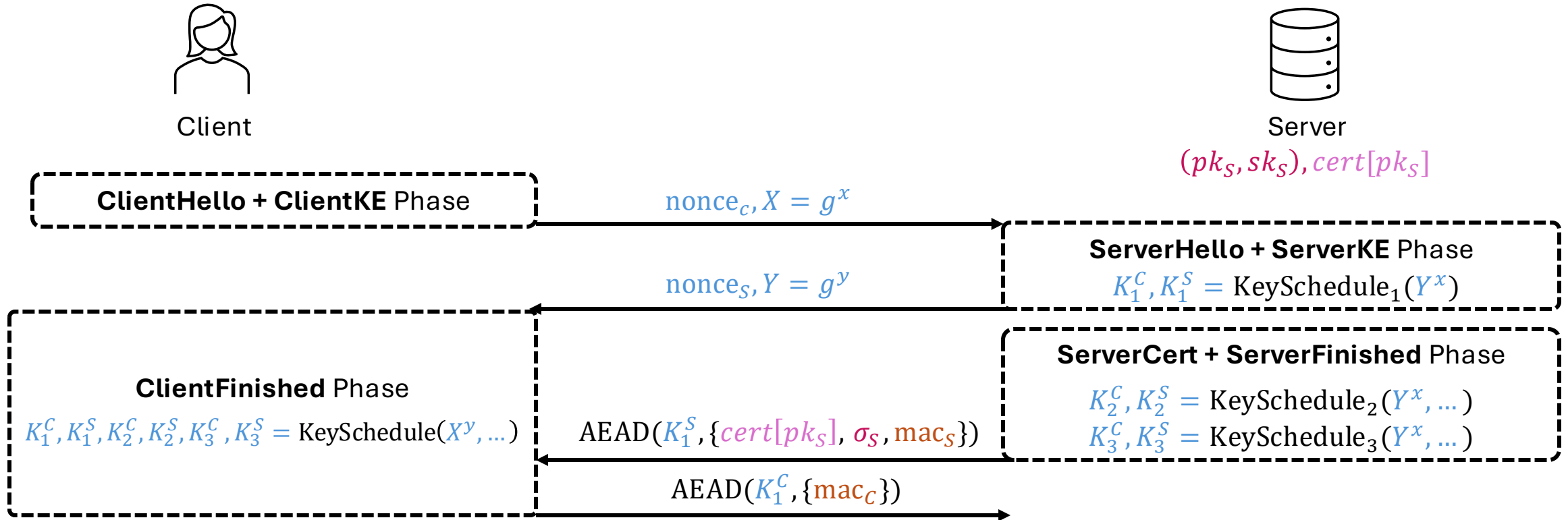
TLS Handshake Protocol

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



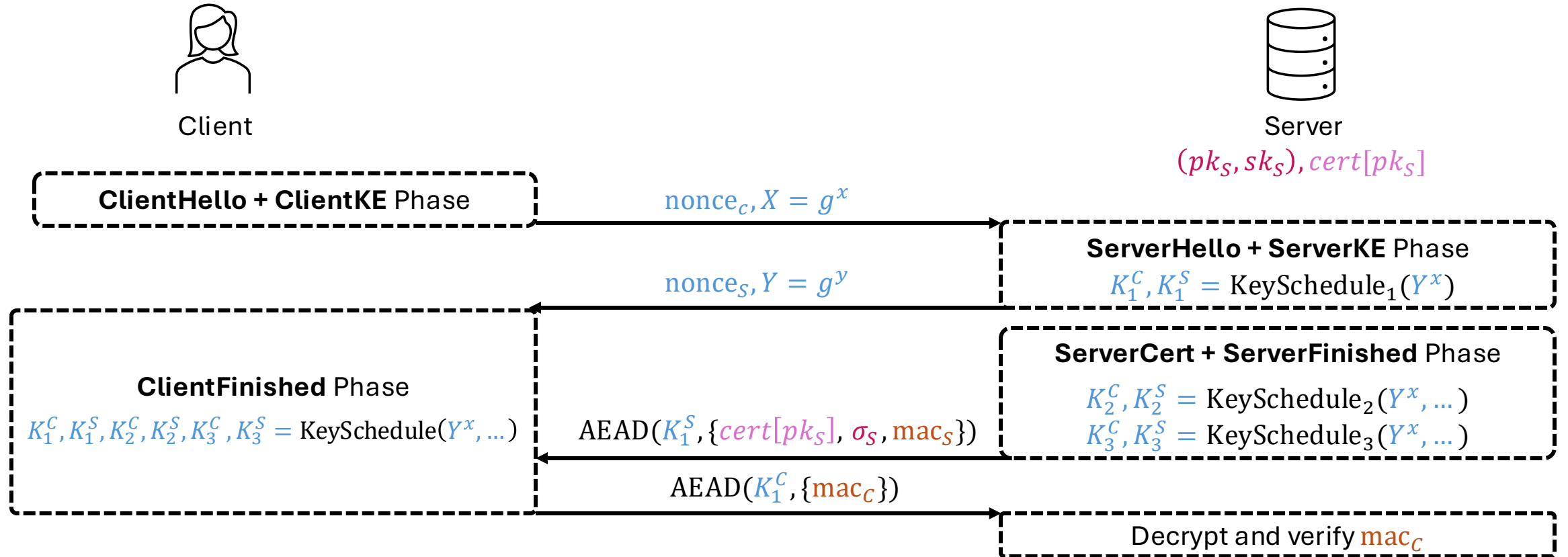
TLS Handshake Protocol

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



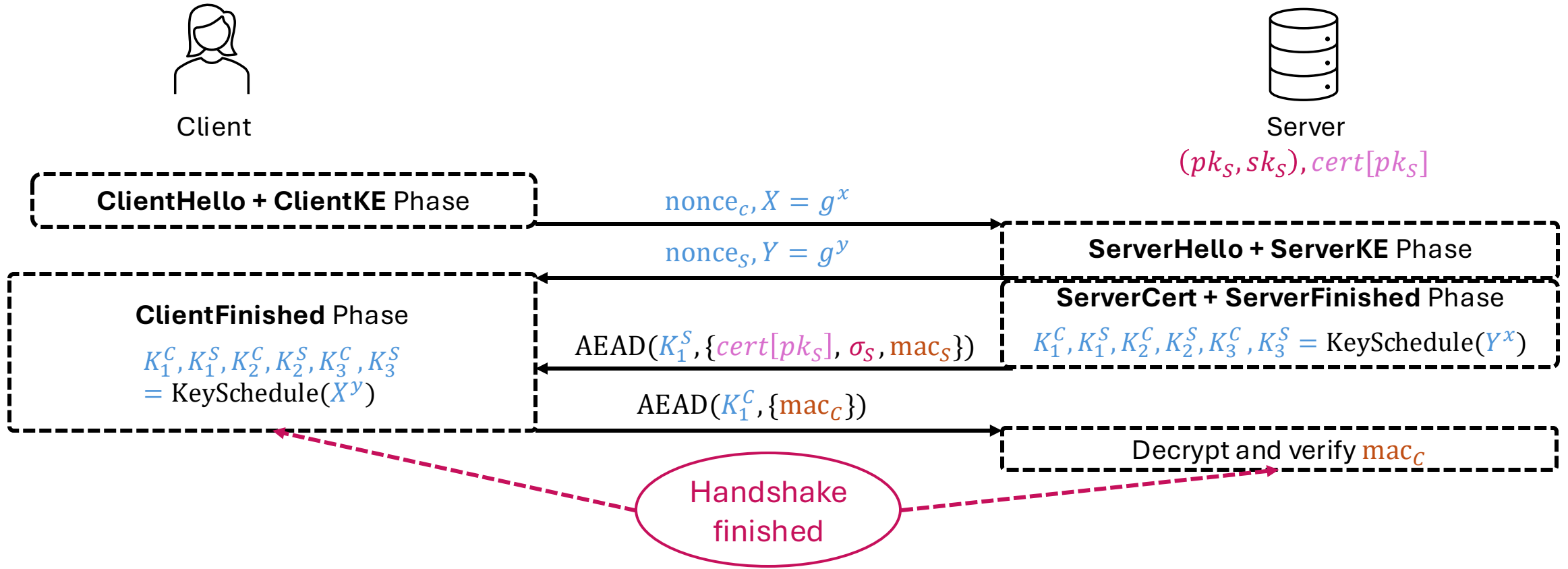
TLS Handshake Protocol

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



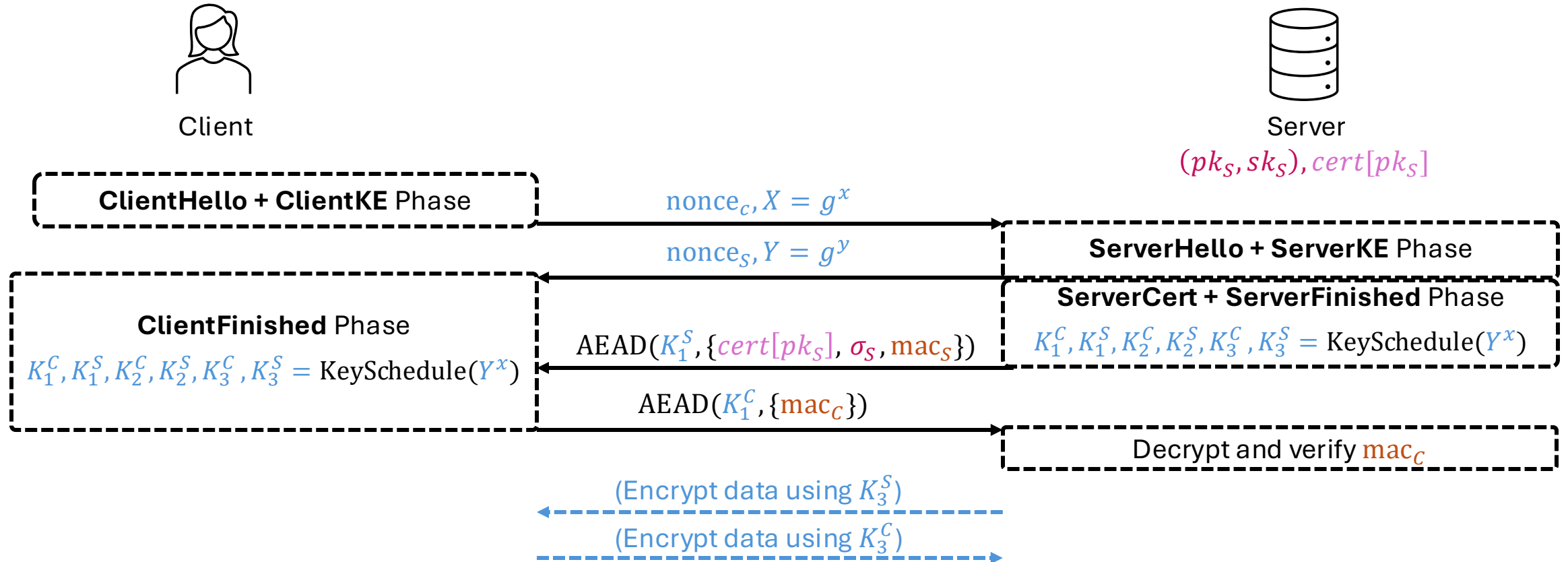
TLS Handshake Protocol

- 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)

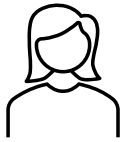


Case Study: HTTPs = HTTP over TLS

HTTP

V.S.

HTTPs



Client
(Browser)



<http://www.google.com>

Port: 80



Client
(Browser)



<https://www.google.com>

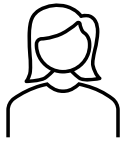
Port: 443

Case Study: HTTPs = HTTP over TLS

HTTP

V.S.

HTTPs



Client
(Browser)



<http://www.google.com>

Port: 80



Client
(Browser)



<https://www.google.com>

Port: 443

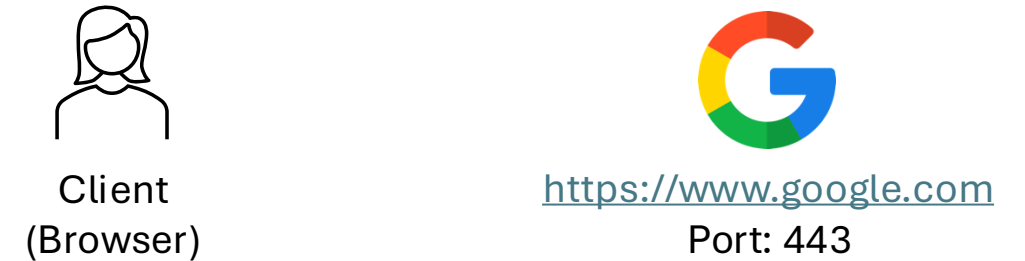
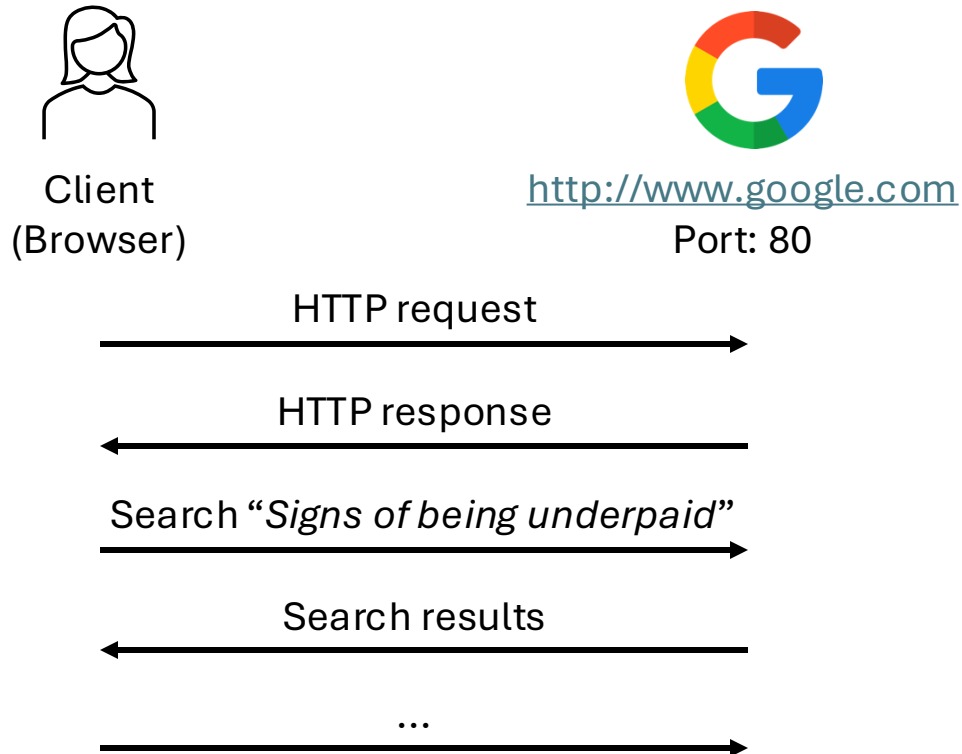
Just an example. You probably
cannot access
<http://www.google.com>
because your browser or Google
enforces HTTPs connections.

Case Study: HTTPs = HTTP over TLS

HTTP

V.S.

HTTPs

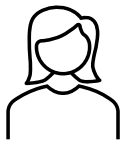


Case Study: HTTPs = HTTP over TLS

HTTP

V.S.

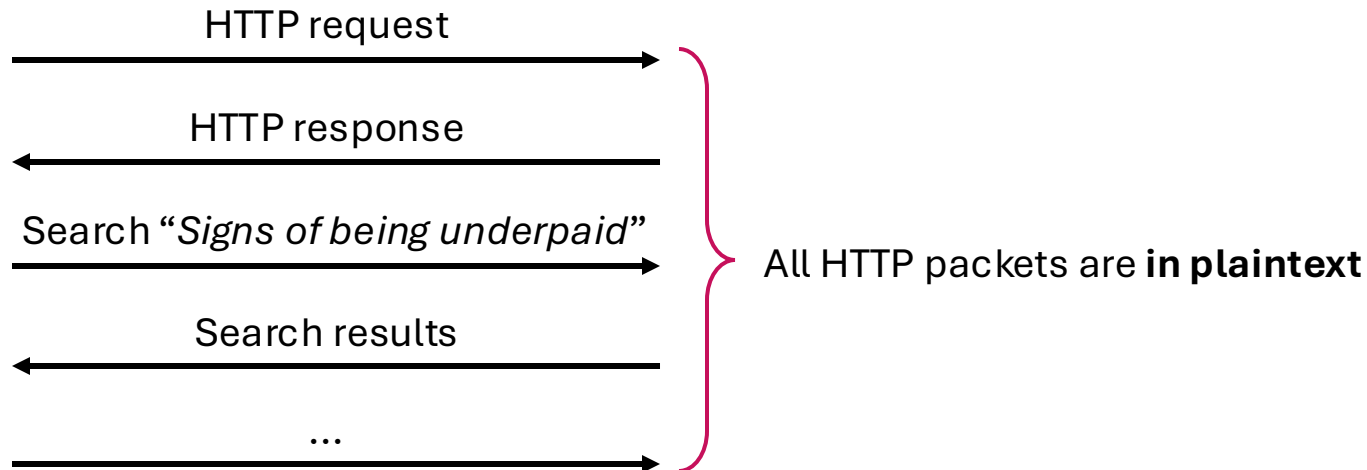
HTTPs


Client
(Browser)

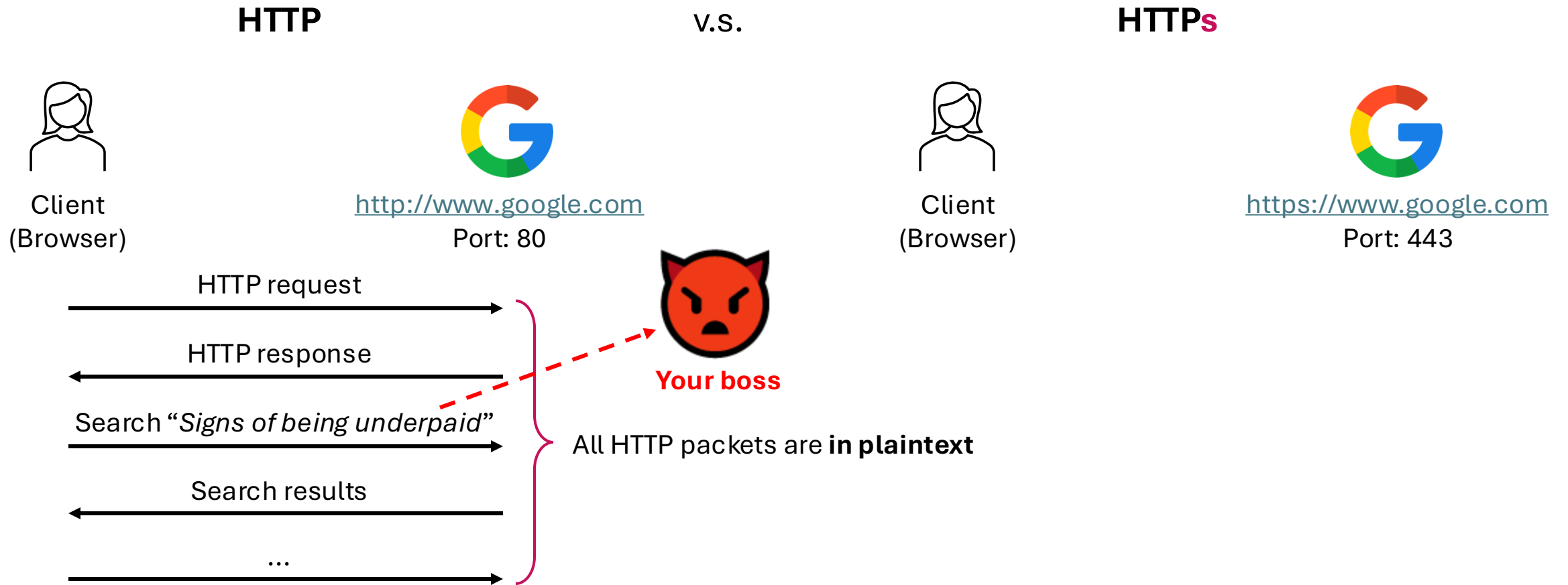

<http://www.google.com>
Port: 80


Client
(Browser)


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



Case Study: HTTPs = HTTP over TLS

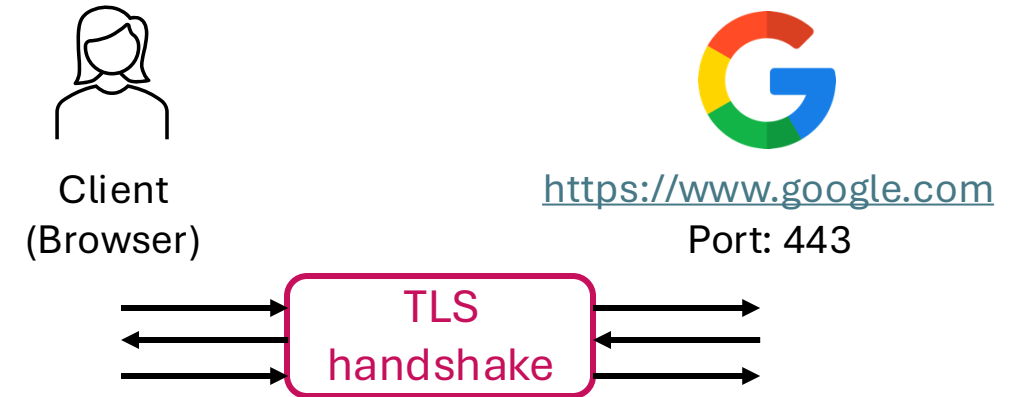
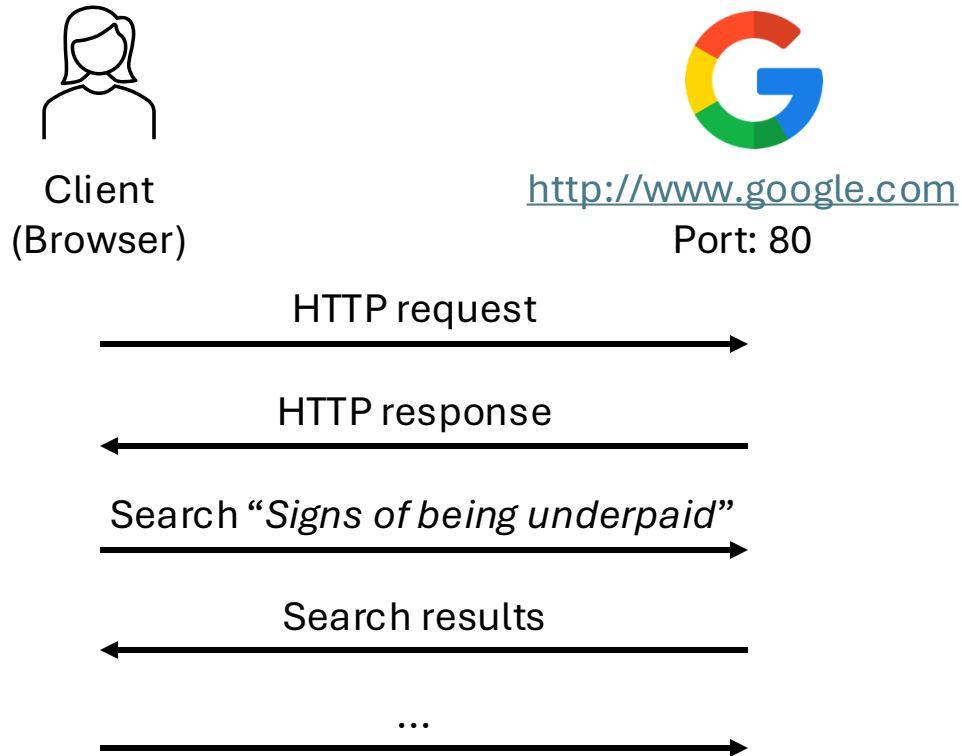


Case Study: HTTPs = HTTP over TLS

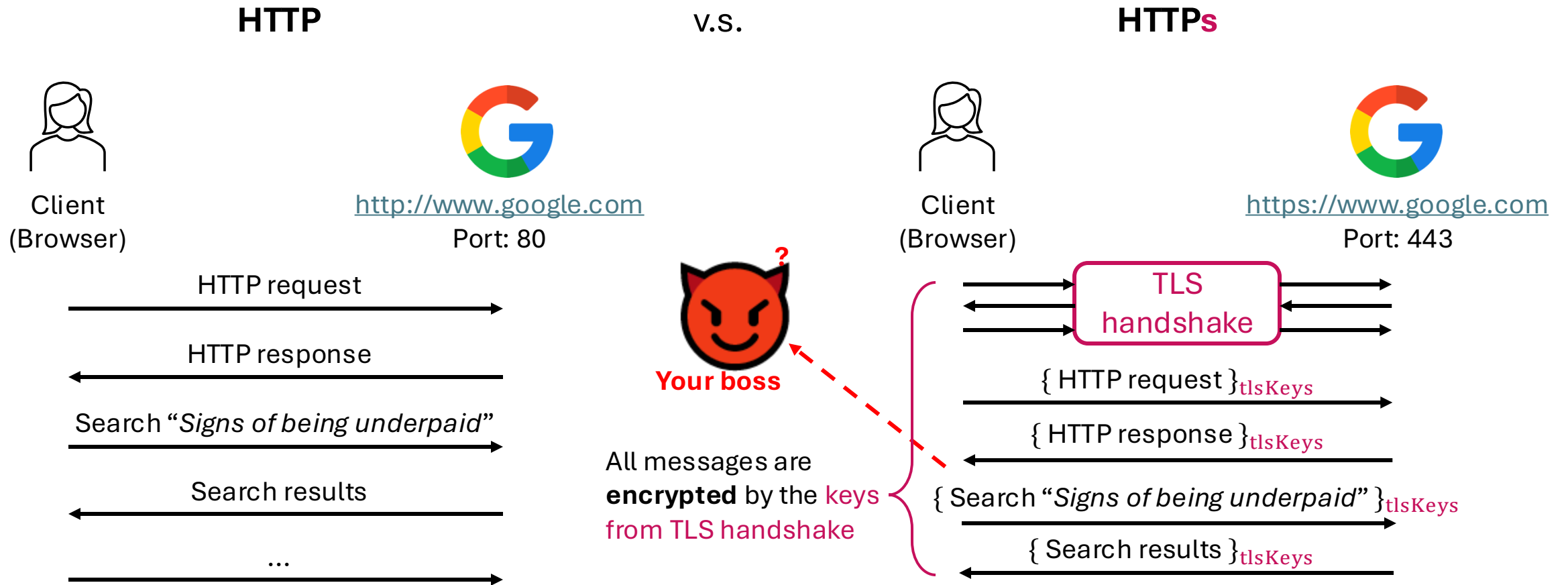
HTTP

V.S.

HTTPs

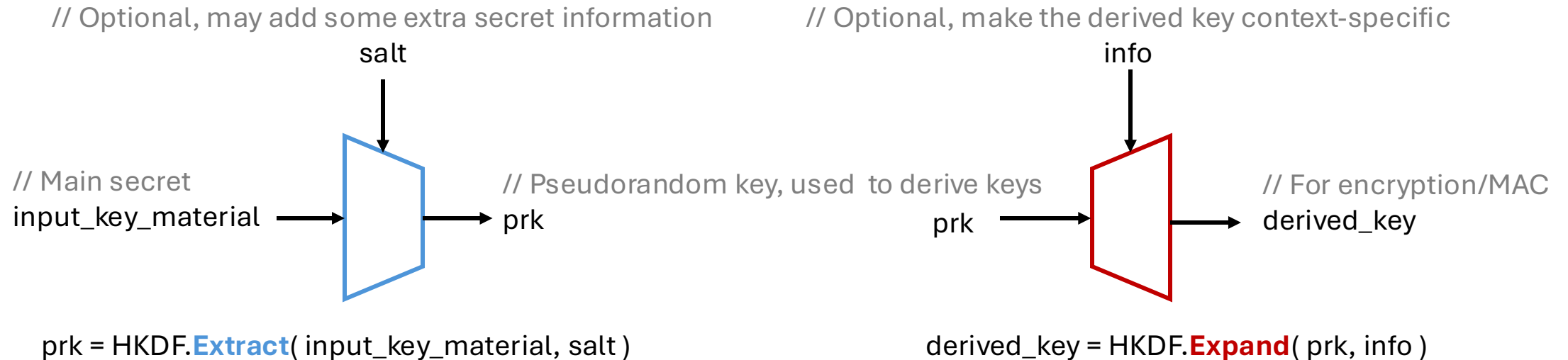


Case Study: HTTPs = HTTP over TLS



Coding Tasks

1. Run the example code “HKDF.py”. Play with it and learn how to derive keys from a secret.



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

- Implement the tweaked TLS handshake protocol (in the Client-Server setting using sockets)

- 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

DeriveHS(g^{xy}):

1. ES = HKDF.Extract(0, 0) // 0 = zeros (bytes) of length 32
2. dES = HKDF.Expand(ES, SHA256("DerivedES"))
3. HS = HKDF.Extract(dES, SHA256(g^{xy}))
4. **return** HS

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

1. HS = DeriveHS(g^{xy})
2. ClientKC = SHA256($\text{nonce}_C || X || \text{nonce}_S || Y || \text{"ClientKC"}$) // "||" is the concatenation operation
3. ServerKC = SHA256($\text{nonce}_C || X || \text{nonce}_S || Y || \text{"ServerKC"}$)
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)

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

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

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

Homework

- **Bonus:** Implement the same protocol, but this time use SHA3-512 as the hash function (for HKDF, HMAC, and the key schedule) and P-521 as the elliptic curve for key exchange. This should allow you to derive a key with 512 bits (64 bytes).

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>