Cryptography Engineering

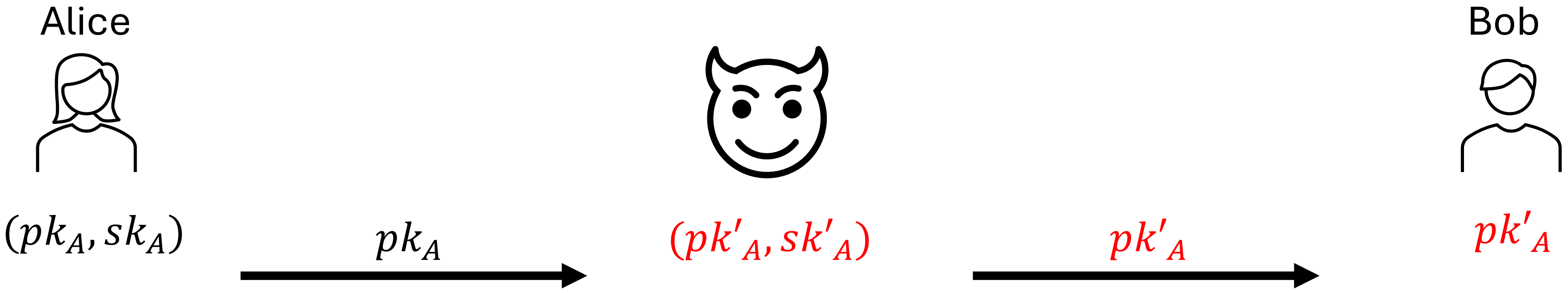
- Lecture 3 (Nov 06, 2024)

- 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



Alice

Bob

(pk_A, sk_A)

pk_A

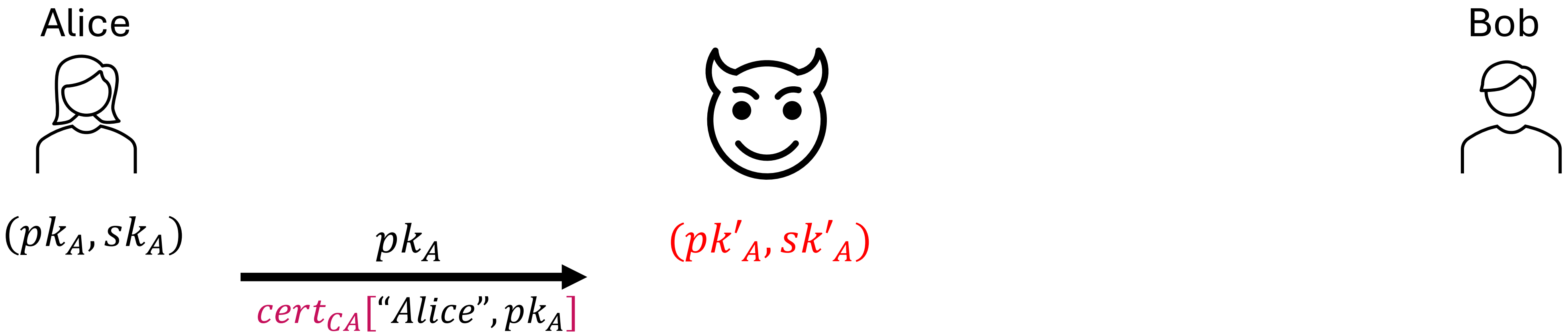
(pk'_A, sk'_A)

pk'_A

pk'_A

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)



Alice

Bob

(pk_A, sk_A)

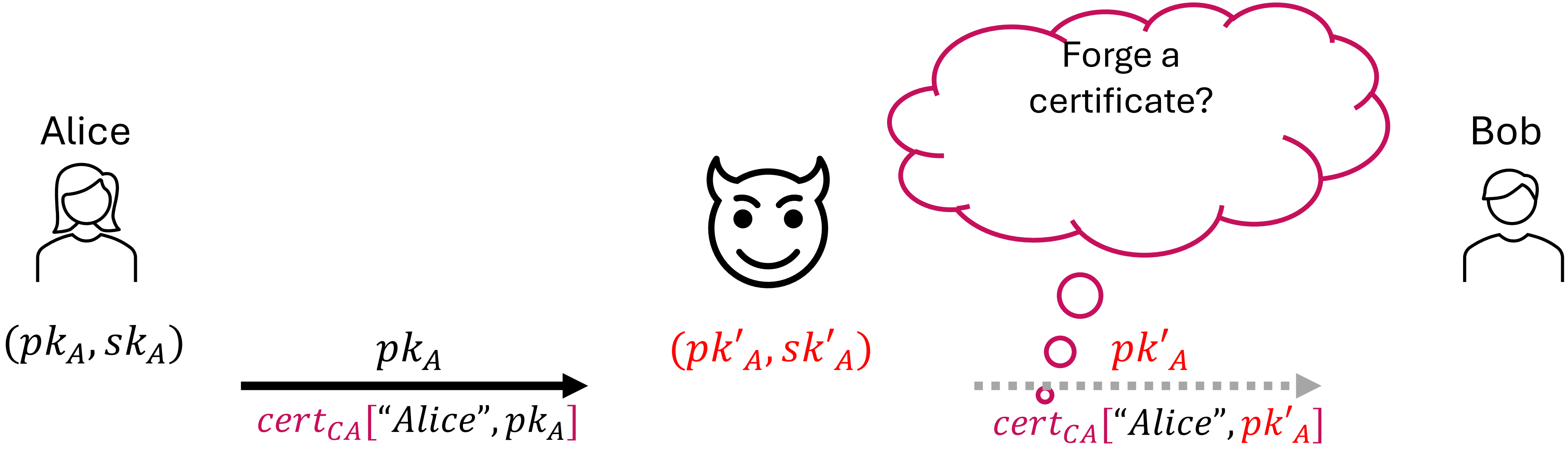
pk_A

$cert_{CA}[\text{“}Alice\text{”}, pk_A]$

(pk'_A, sk'_A)

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)



Alice

(pk_A, sk_A)

pk_A

$cert_{CA}[\text{"Alice"}, pk_A]$

(pk'_A, sk'_A)

Forge a certificate?

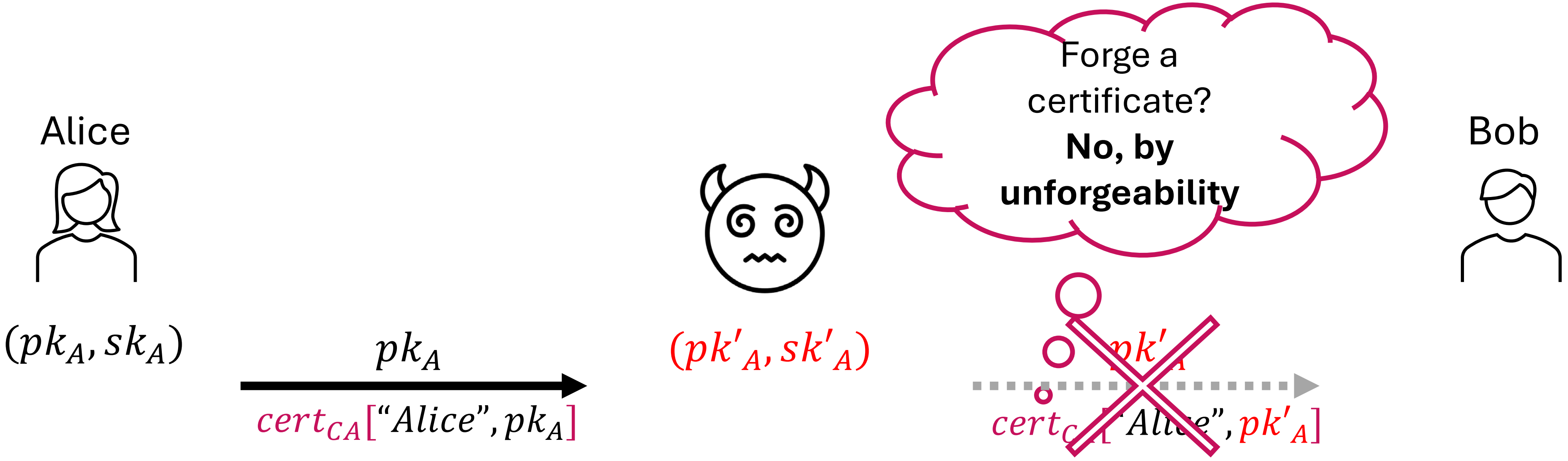
pk'_A

$cert_{CA}[\text{"Alice"}, pk'_A]$

Bob

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)



Alice

(pk_A, sk_A)

pk_A

$cert_{CA}[\text{“Alice”}, pk_A]$

(pk'_A, sk'_A)

Forge a certificate?
No, by unforgeability

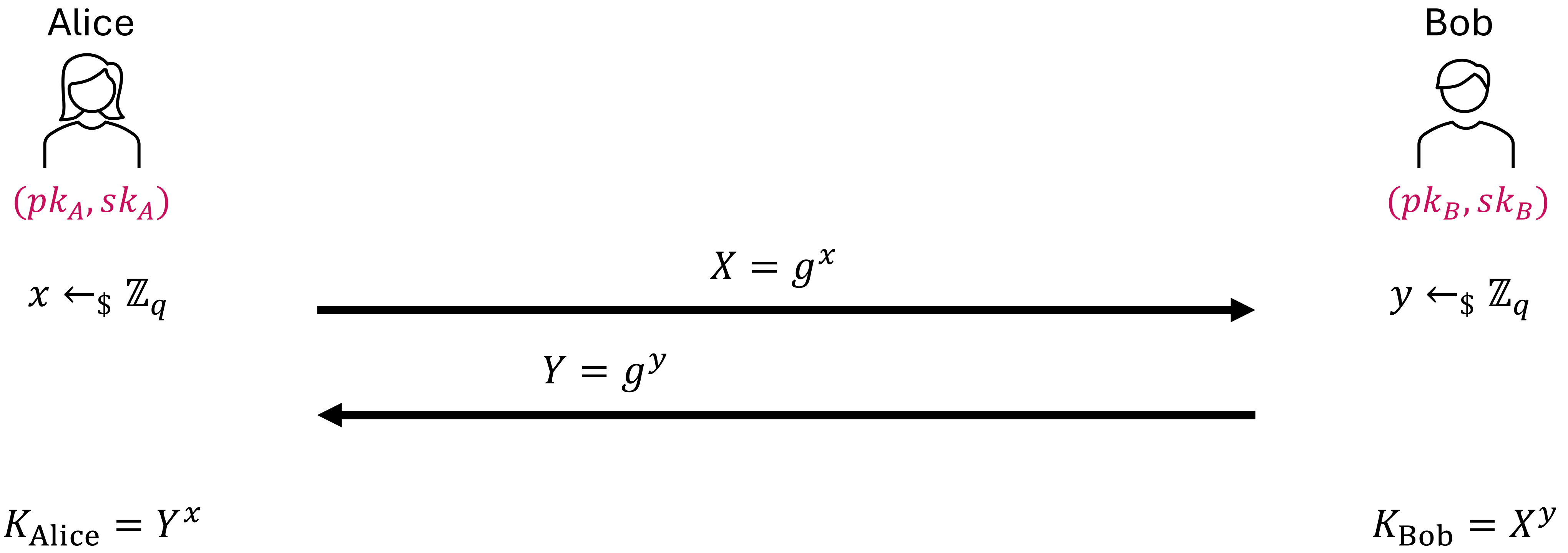
~~pk'_A~~

~~$cert_{CA}[\text{“Alice”}, pk'_A]$~~

Bob

Signed Diffie-Hellman Protocol (Simplified)

- Use signature to avoid MitM attacks



Alice

(pk_A, sk_A)

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

$X = g^x$ →

Bob

(pk_B, sk_B)

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

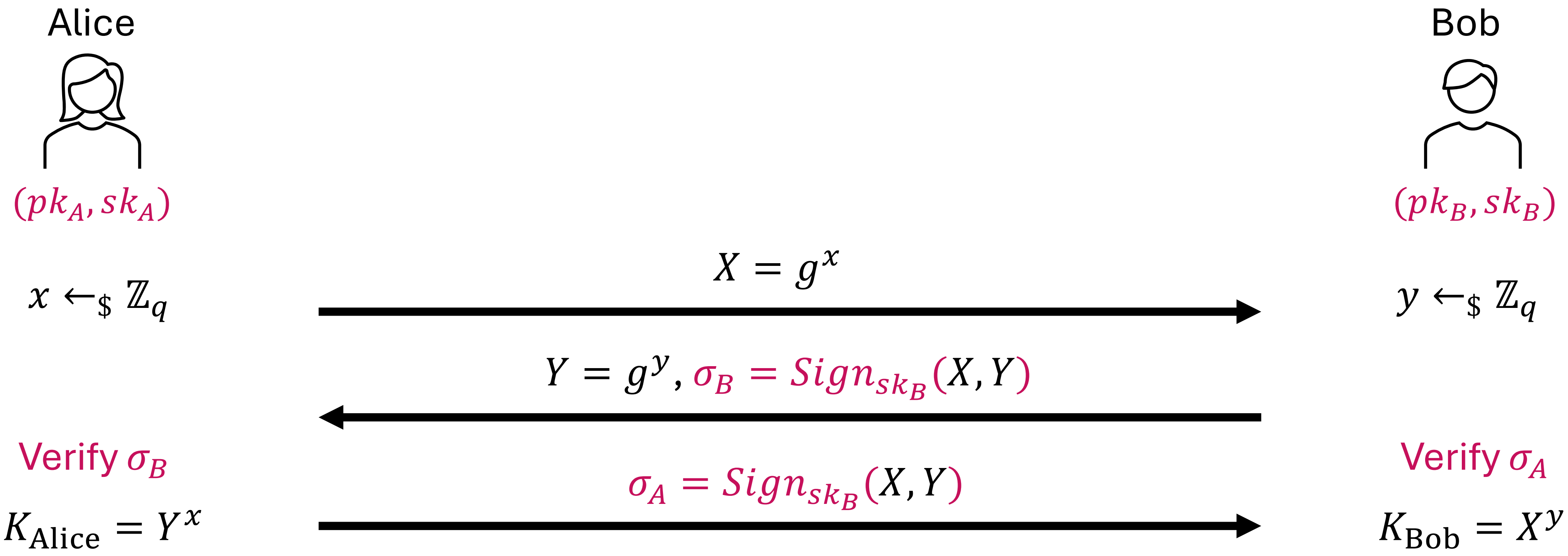
← $Y = g^y$

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

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

Signed Diffie-Hellman Protocol (Simplified)

- Use signature to avoid MitM attacks



Alice

(pk_A, sk_A)

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

Bob

(pk_B, sk_B)

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

$X = g^x$ →

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

Verify σ_B

$\sigma_A = Sign_{sk_B}(X, Y)$ →

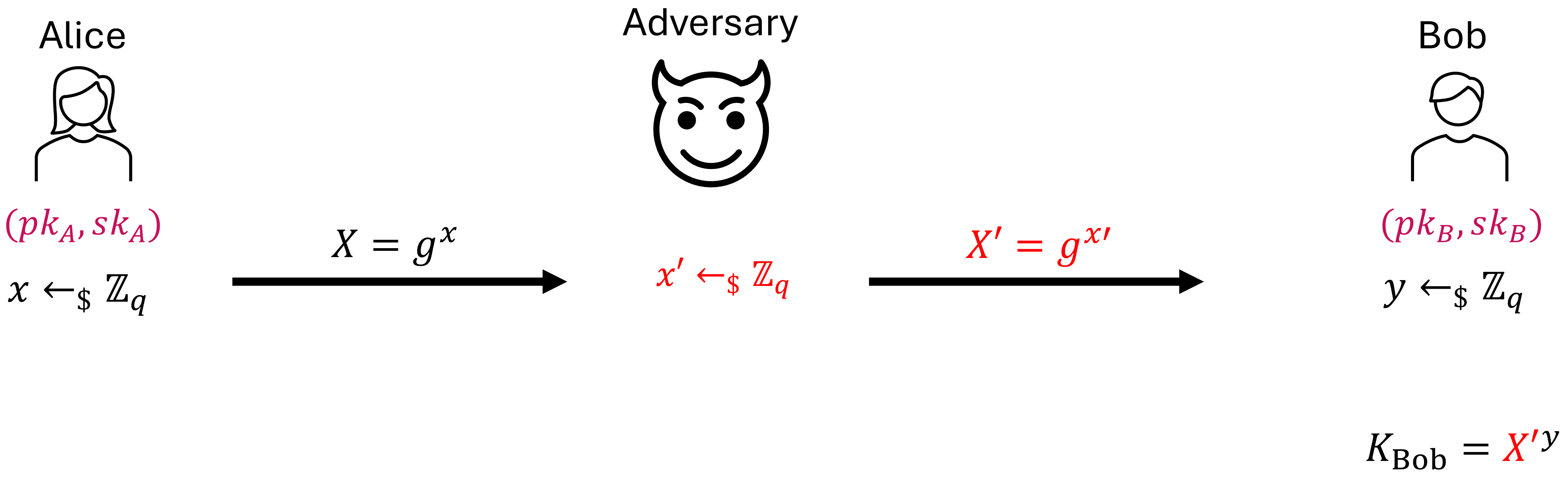
Verify σ_A

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

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

Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



Alice

Adversary

Bob

(pk_A, sk_A)

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

$X = g^x$

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

$X' = g^{x'}$

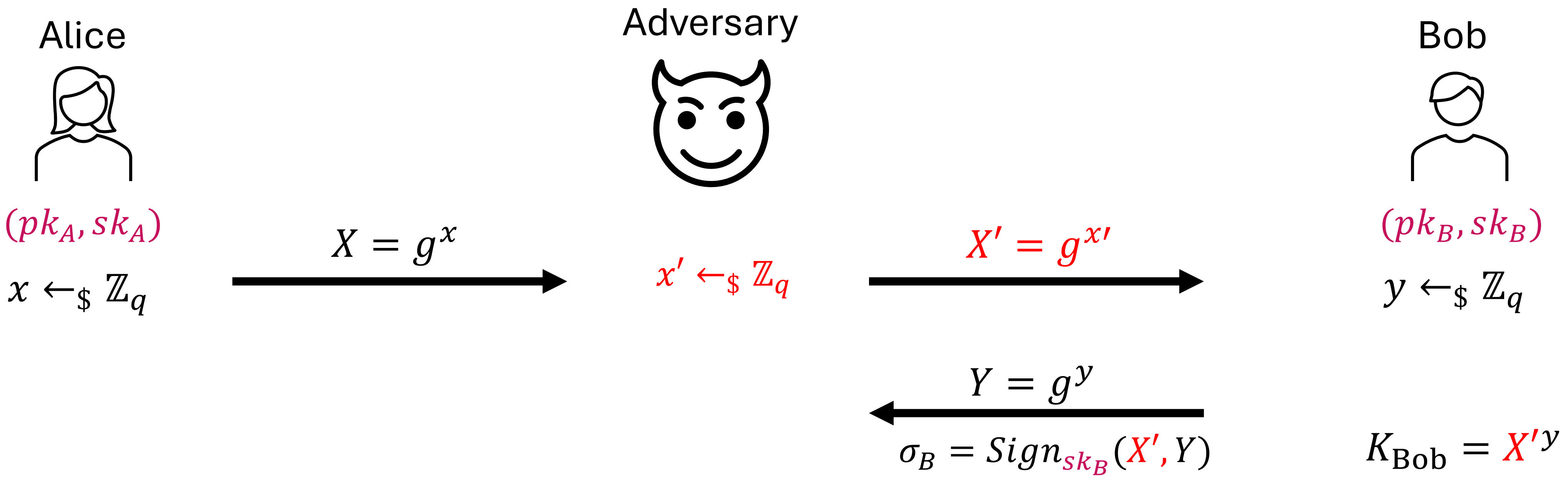
(pk_B, sk_B)

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

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

Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



Alice

Adversary

Bob

(pk_A, sk_A)

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

$X = g^x$

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

$X' = g^{x'}$

(pk_B, sk_B)

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

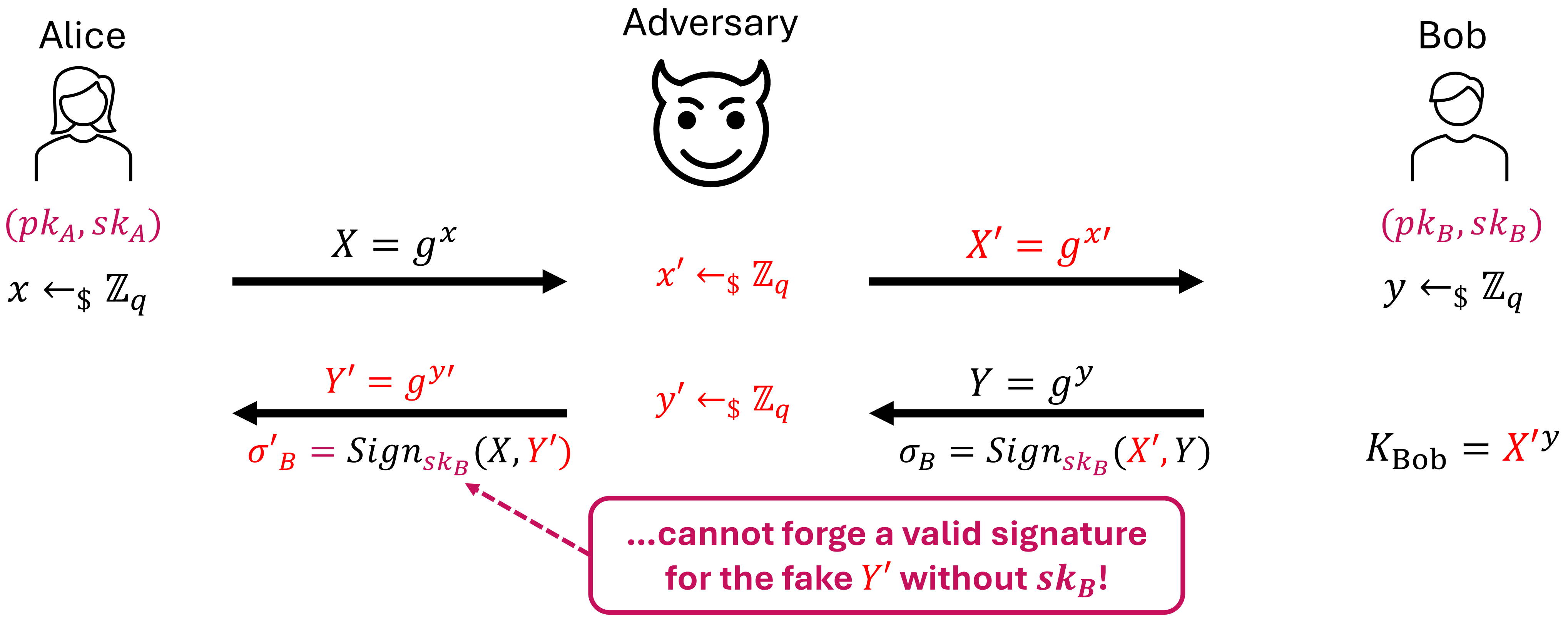
$Y = g^y$

$\sigma_B = Sign_{sk_B}(X', Y)$

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

Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



Alice

Adversary

Bob

(pk_A, sk_A)

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

$X = g^x$

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

$X' = g^{x'}$

(pk_B, sk_B)

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

$Y' = g^{y'}$

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

$Y = g^y$

$\sigma'_B = Sign_{sk_B}(X, Y')$

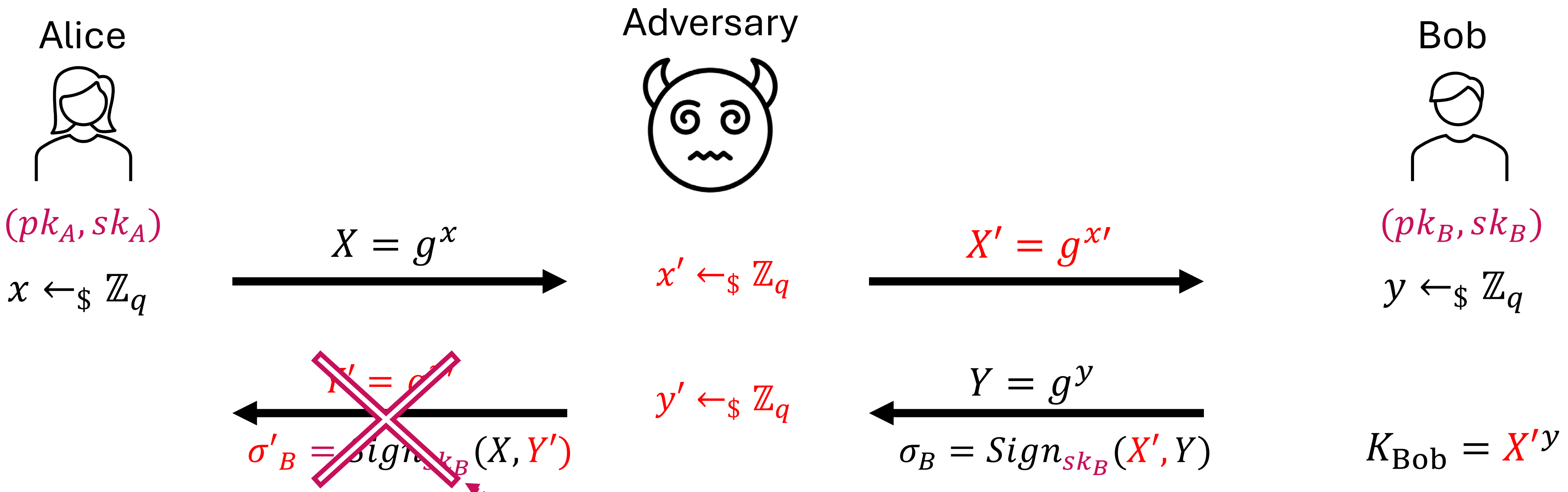
$\sigma_B = Sign_{sk_B}(X', Y)$

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

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

Signed Diffie-Hellman Protocol

- Can we launch a MitM attack on SigDH?



Alice

(pk_A, sk_A)

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

$X = g^x$

Adversary

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

$X' = g^{x'}$

Bob

(pk_B, sk_B)

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

$\cancel{Y' = g^{y'}}$

$\cancel{\sigma'_B = Sign_{sk_B}}(X, Y')$

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

$Y = g^y$

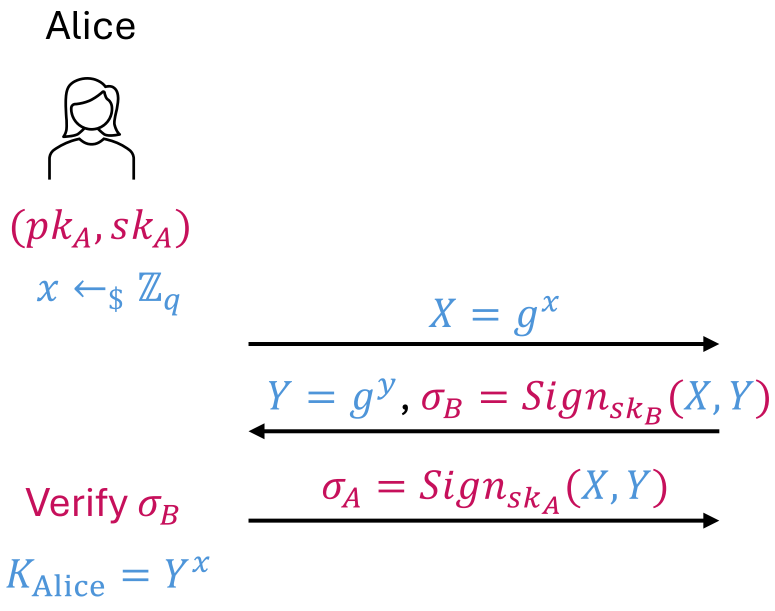
$\sigma_B = Sign_{sk_B}(X', Y)$

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

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

- **No, by unforgeability...**

Signed Diffie-Hellman Protocol



Alice

(pk_A, sk_A)

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

Bob

(pk_B, sk_B)

$$\xrightarrow{X = g^x}$$

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

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

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

Verify σ_B

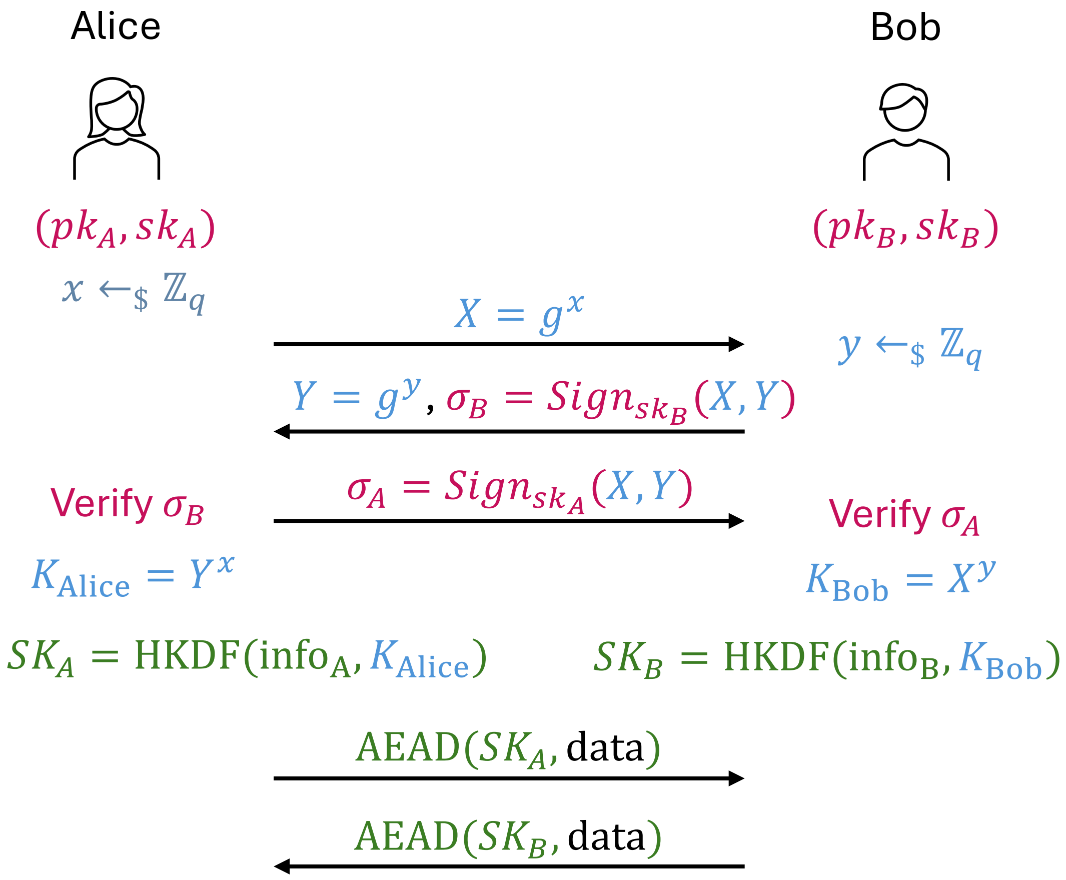
$K_{\text{Alice}} = Y^x$

Verify σ_A

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

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

Signed Diffie-Hellman Protocol



Alice

(pk_A, sk_A)

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

Bob

(pk_B, sk_B)

$X = g^x$

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

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

Verify σ_B

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

Verify σ_A

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

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

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

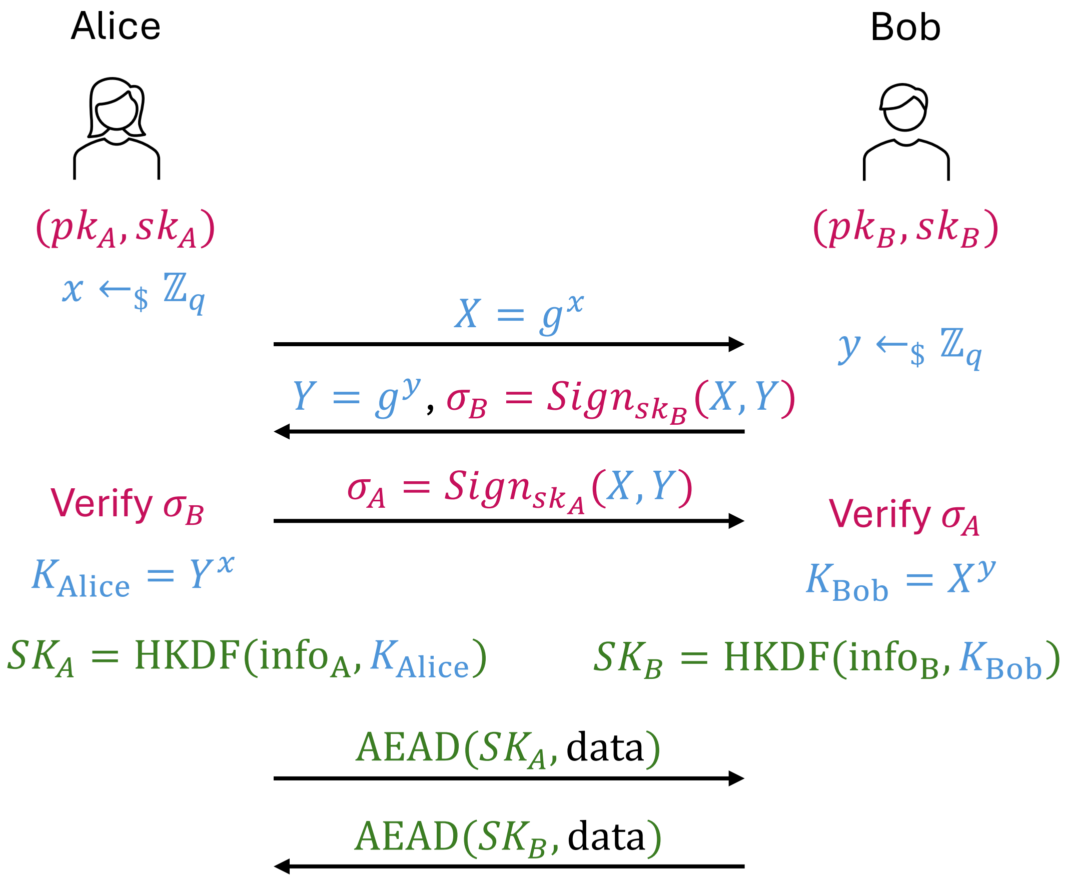
$SK_B = \text{HKDF}(\text{info}_\text{B}, K_{\text{Bob}})$

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

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

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

Signed Diffie-Hellman Protocol



Alice

Bob

(pk_A, sk_A)

(pk_B, sk_B)

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

$X = g^x$ $\longrightarrow$

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

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

Verify σ_B

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

Verify σ_A

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

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

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

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

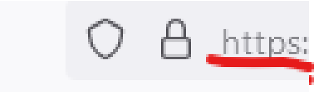
$\text{AEAD}(SK_A, \text{data})$ $\longrightarrow$

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

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

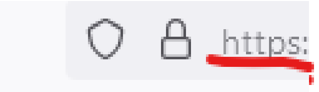
- Important Application: **TLS handshake protocol...**
  - Note: Cryptographic algorithms “in textbooks” often contrasts with their real-world implementation

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, …

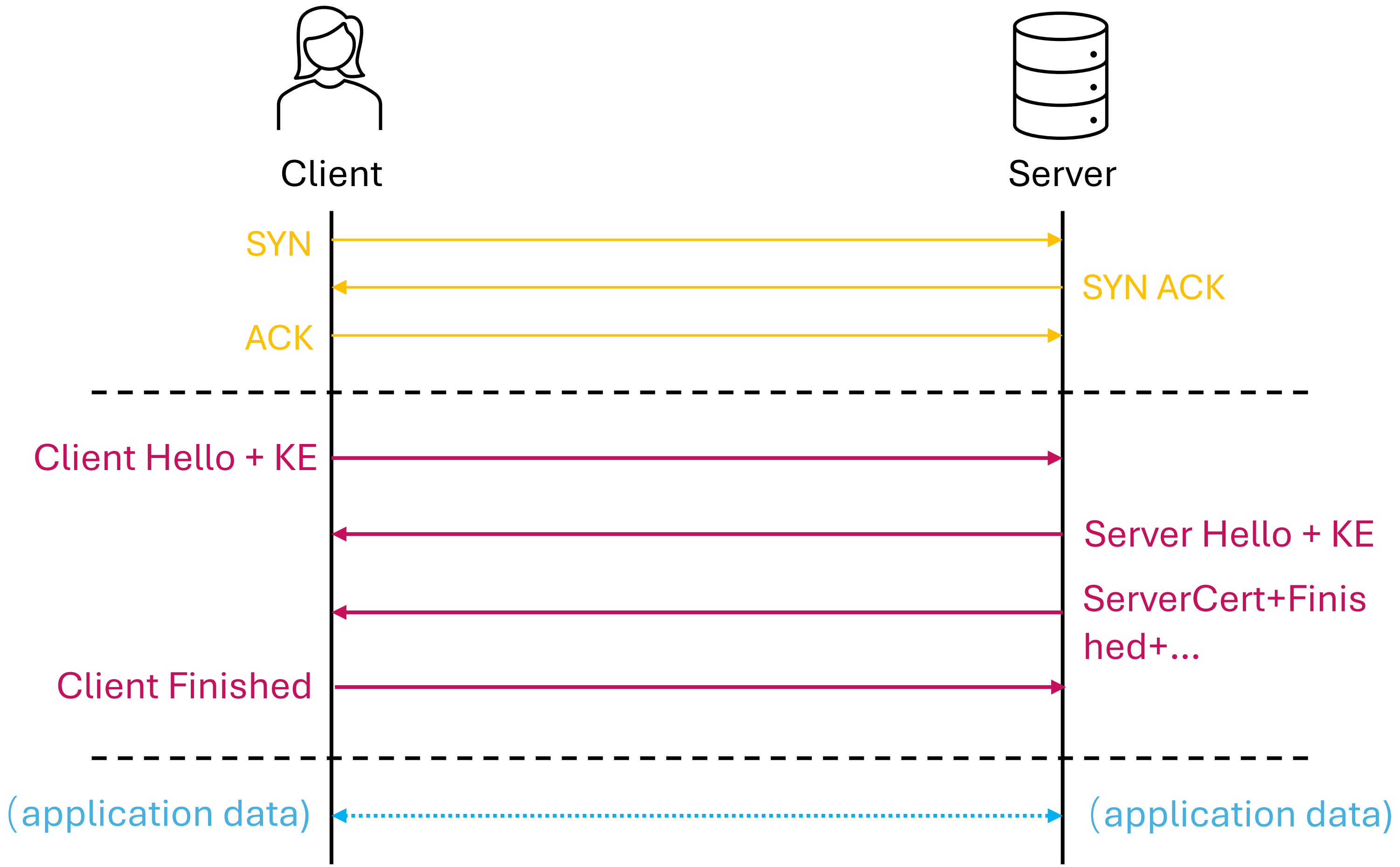

https:

  - 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):



Client

Server

1. Session initialization (TCP/IP)

SYN

SYN ACK

ACK

2. TLS handshake

Client Hello + KE

Server Hello + KE

ServerCert+Finished+...

Client Finished

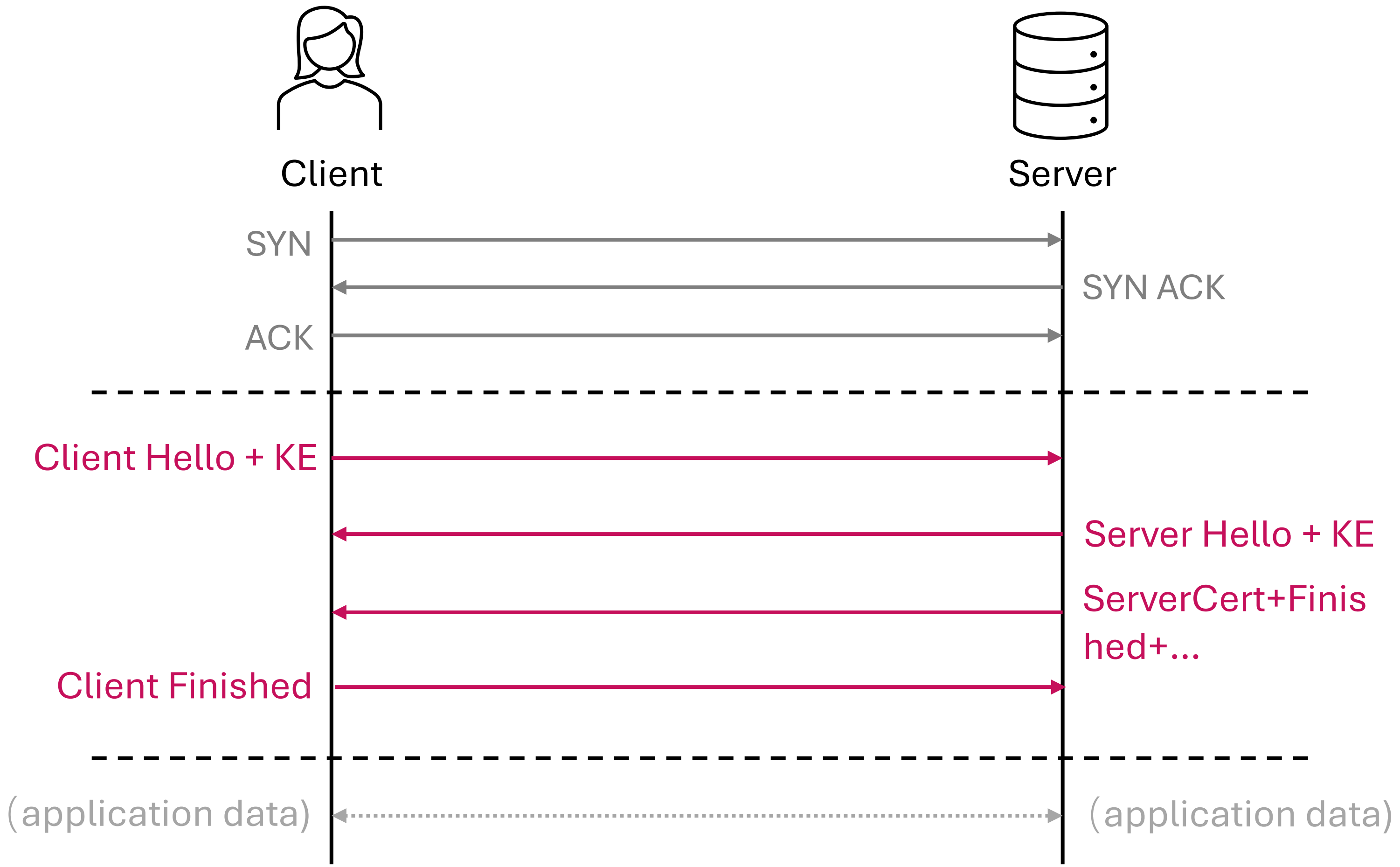
3. Encrypted Data Transfer

(application data)

(application data)

TLS Handshake Protocol

- TLS process (simplified):



Client

Server

1. Session initialization (TCP/IP)

SYN

SYN ACK

ACK

2. **TLS handshake**

Client Hello + KE

Server Hello + KE

ServerCert+Finished+...

Client Finished

3. Encrypted Data Transfer

(application data)

(application data)

TLS Handshake Protocol

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

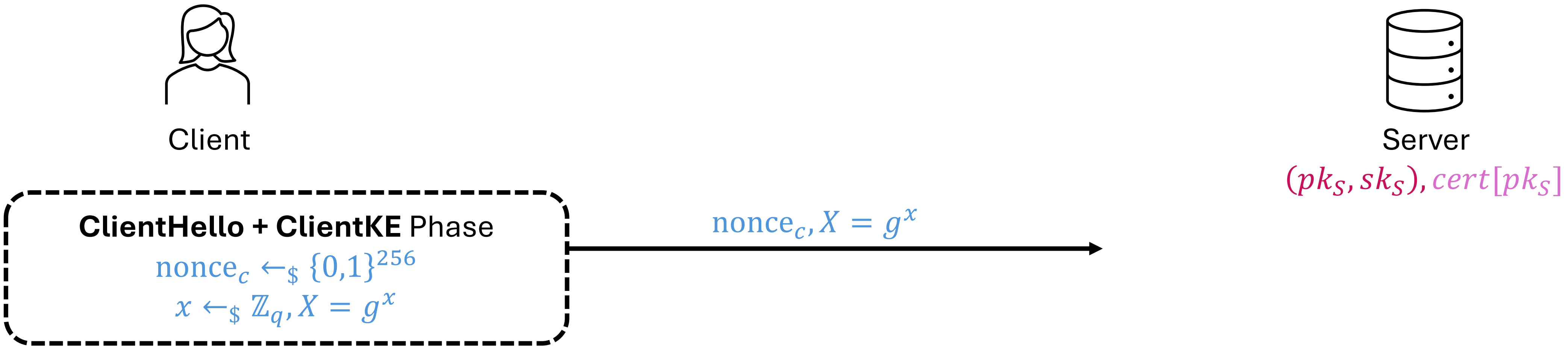
Client

Server

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

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

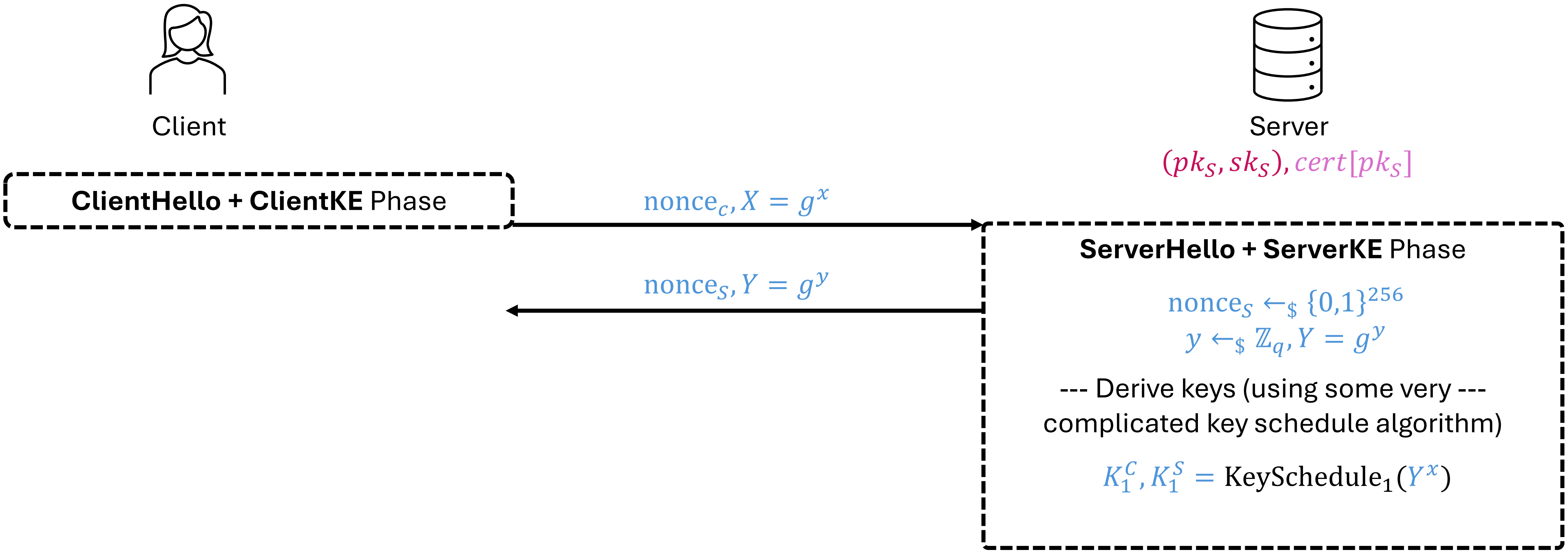
$\text{nonce}_c \leftarrow_\$ \{0,1\}^{256}$

$x \leftarrow_\$ \mathbb{Z}_q, X = g^x$

$\text{nonce}_c, X = g^x$

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\text{nonce}_c, X = g^x$

$\text{nonce}_S, Y = g^y$

ServerHello + ServerKE Phase

$\text{nonce}_S \leftarrow_\$ \{0,1\}^{256}$

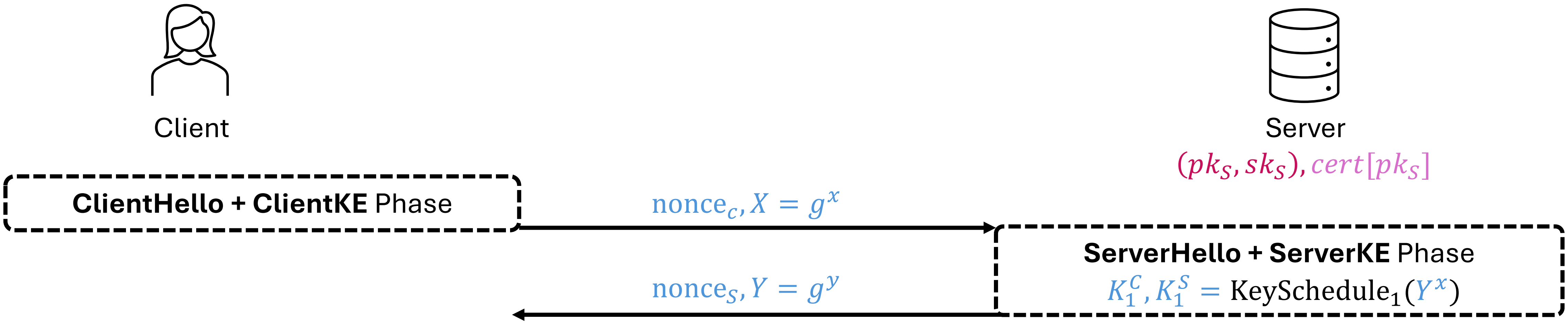
$y \leftarrow_\$ \mathbb{Z}_q, Y = g^y$

--- Derive keys (using some very ---
complicated key schedule algorithm)

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

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\mathrm{nonce}_c, X = g^x$

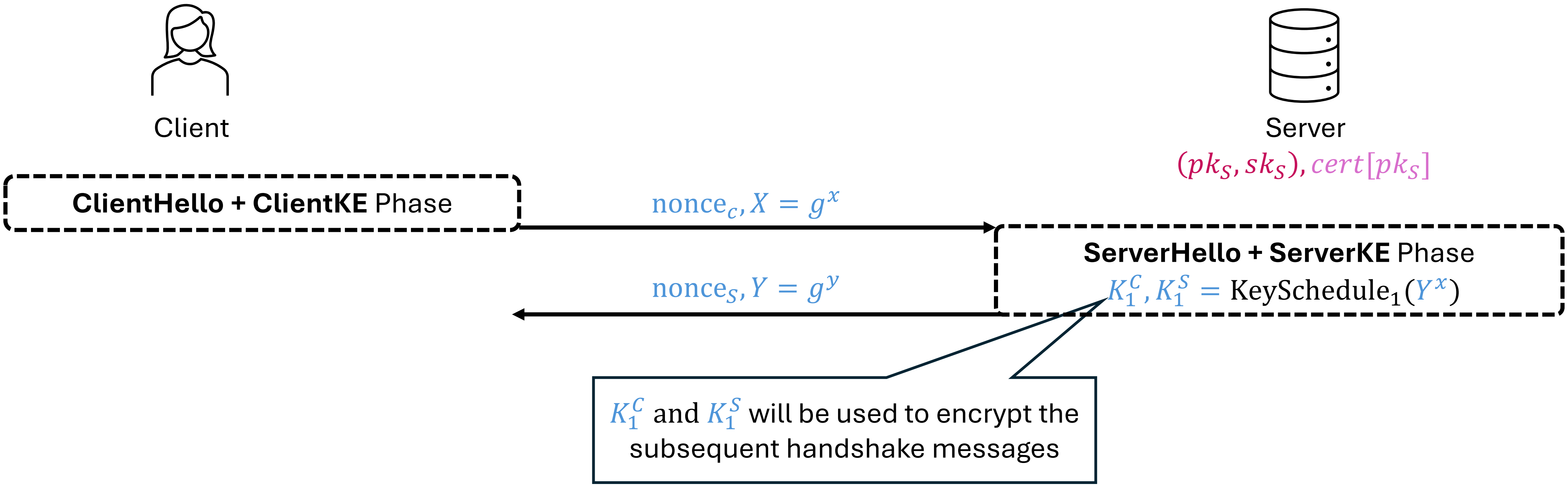
ServerHello + ServerKE Phase

$K_1^C, K_1^S = \mathrm{KeySchedule}_1(Y^x)$

$\mathrm{nonce}_S, Y = g^y$

TLS Handshake Protocol

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



Client

Server

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

ClientHello + **ClientKE** Phase

$\text{nonce}_c, X = g^x$

ServerHello + **ServerKE** Phase

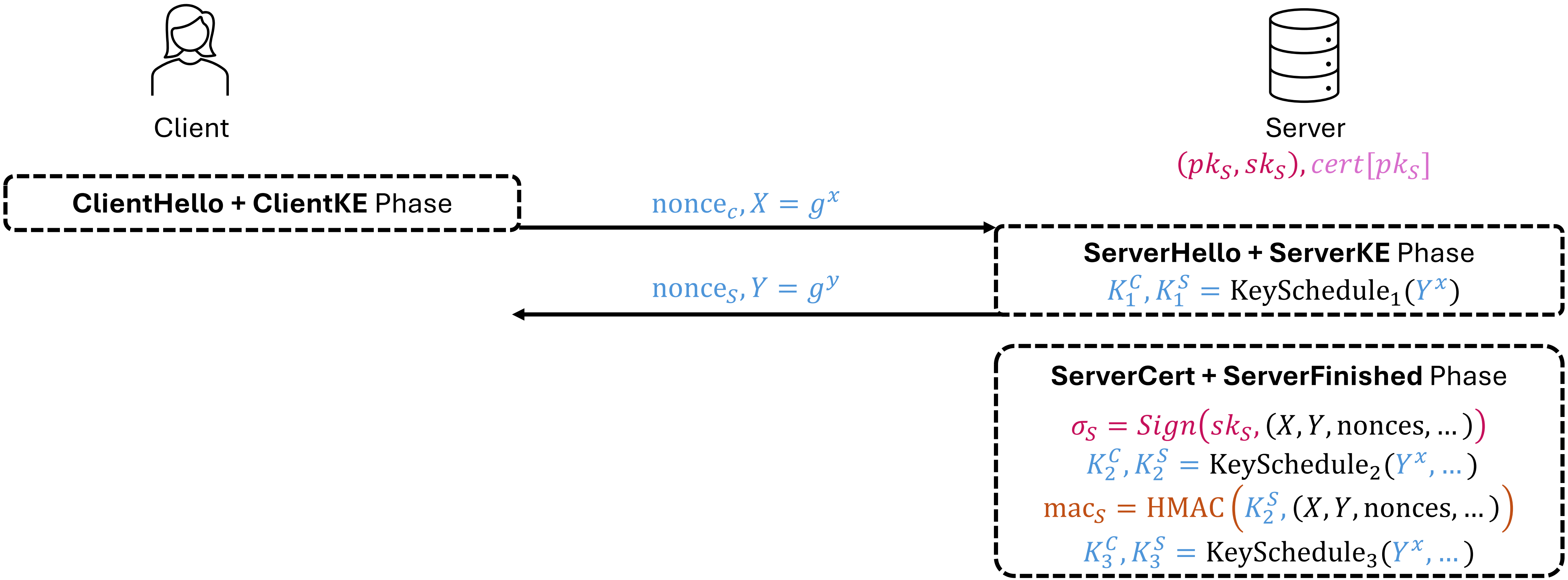
$K_1^C, K_1^S = \text{KeySchedule}_1(Y^x)$

$\text{nonce}_S, Y = g^y$

K_1^C and K_1^S will be used to encrypt the subsequent handshake messages

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\text{nonce}_c, X = g^x$

ServerHello + ServerKE Phase

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

$\text{nonce}_S, Y = g^y$

ServerCert + ServerFinished Phase

$\sigma_S = Sign\left(sk_S, (X, Y, \text{nonces}, \ldots)\right)$

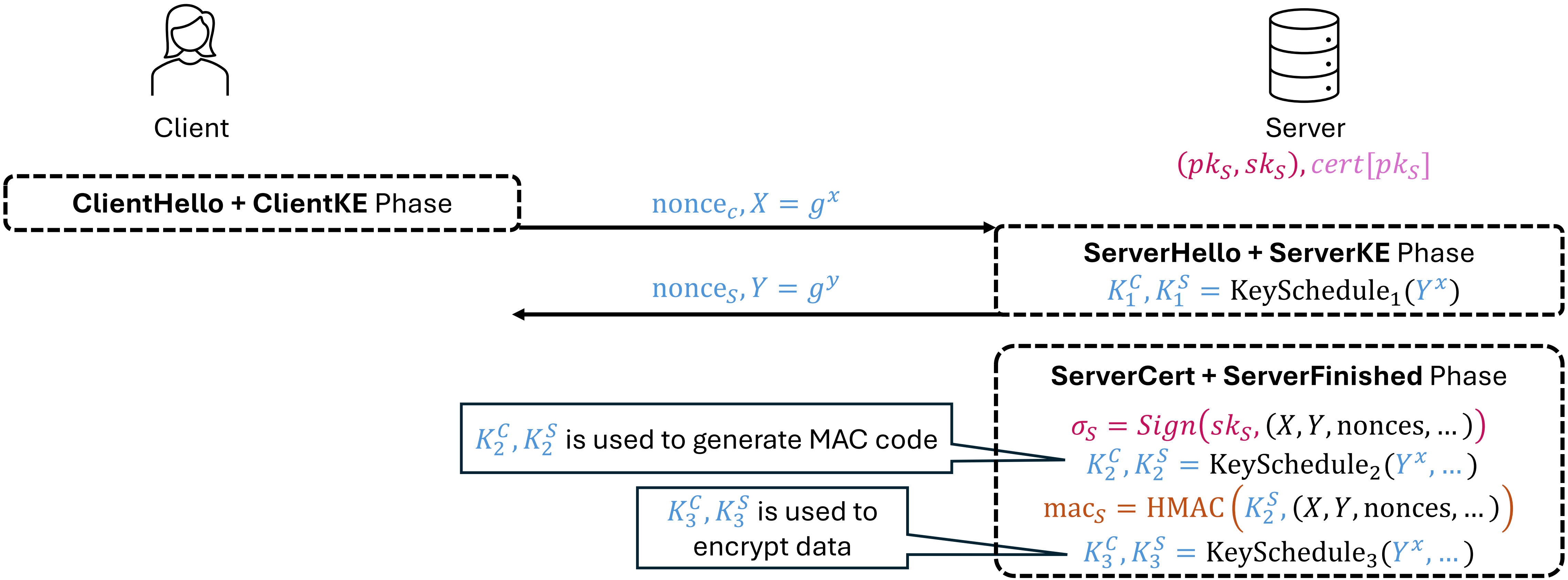
$K_2^C, K_2^S = \text{KeySchedule}_2(Y^x, \ldots)$

$\text{mac}_S = \text{HMAC}\left(K_2^S, (X, Y, \text{nonces}, \ldots)\right)$

$K_3^C, K_3^S = \text{KeySchedule}_3(Y^x, \ldots)$

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\text{nonce}_c, X = g^x$

ServerHello + ServerKE Phase

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

$\text{nonce}_S, Y = g^y$

ServerCert + ServerFinished Phase

$\sigma_S = Sign\big(sk_S, (X, Y, \text{nonces}, \ldots)\big)$

$K_2^C, K_2^S = \text{KeySchedule}_2(Y^x, \ldots)$

$\text{mac}_S = \text{HMAC}\big(K_2^S, (X, Y, \text{nonces}, \ldots)\big)$

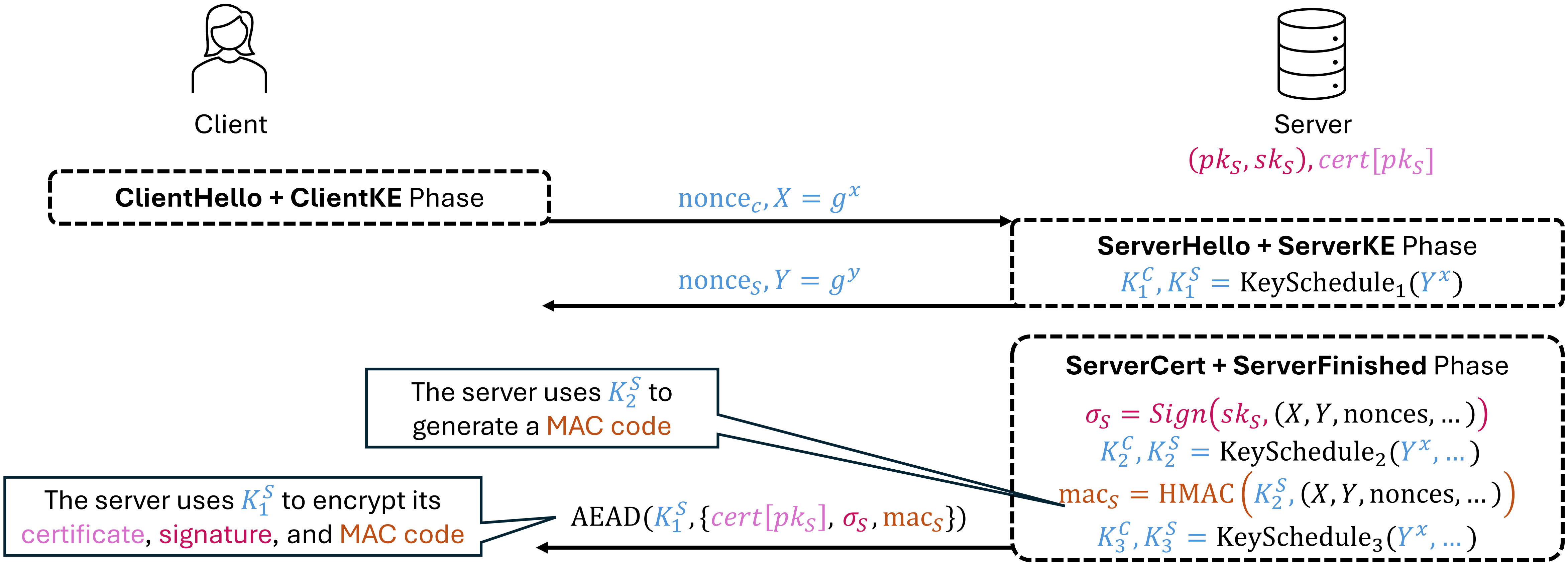
$K_3^C, K_3^S = \text{KeySchedule}_3(Y^x, \ldots)$

K_2^C, K_2^S is used to generate MAC code

K_3^C, K_3^S is used to encrypt data

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\text{nonce}_c, X = g^x$

ServerHello + ServerKE Phase

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

$\text{nonce}_S, Y = g^y$

ServerCert + ServerFinished Phase

$\sigma_S = Sign\left(sk_S, (X, Y, \text{nonces}, \ldots)\right)$

$K_2^C, K_2^S = \text{KeySchedule}_2(Y^x, \ldots)$

$\text{mac}_S = \text{HMAC}\left(K_2^S, (X, Y, \text{nonces}, \ldots)\right)$

$K_3^C, K_3^S = \text{KeySchedule}_3(Y^x, \ldots)$

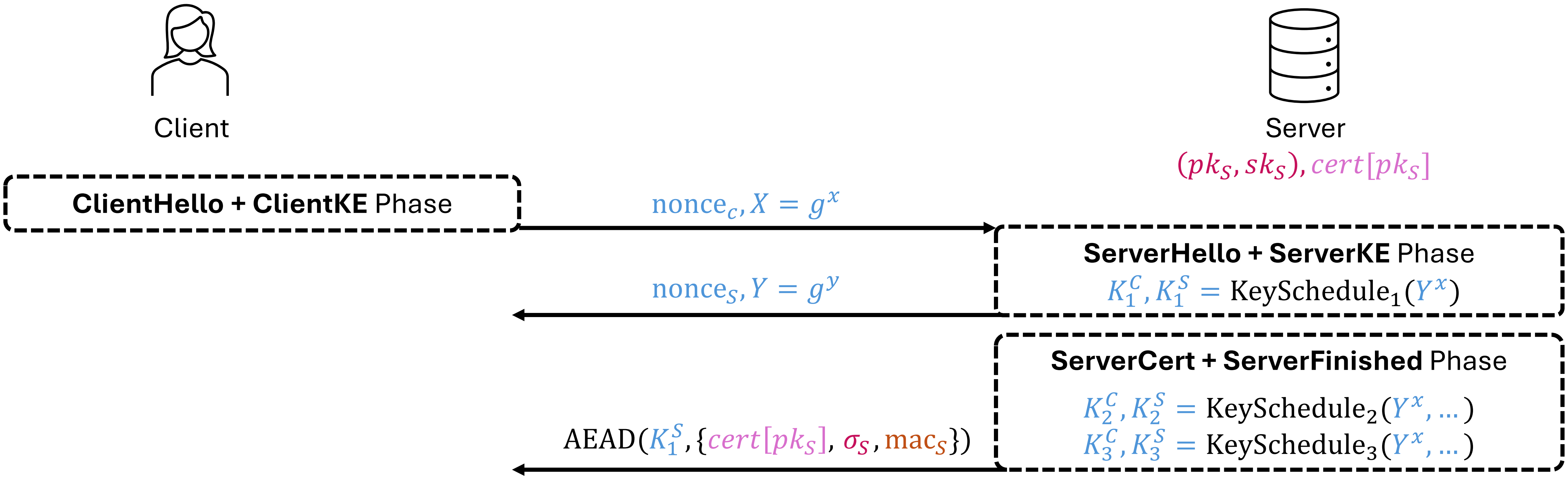
The server uses K_2^S to generate a MAC code

The server uses K_1^S to encrypt its certificate, signature, and MAC code

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

TLS Handshake Protocol

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



Client

Server

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

ClientHello + **ClientKE** Phase

$\text{nonce}_c, X = g^x$

ServerHello + **ServerKE** Phase

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

$\text{nonce}_S, Y = g^y$

ServerCert + **ServerFinished** Phase

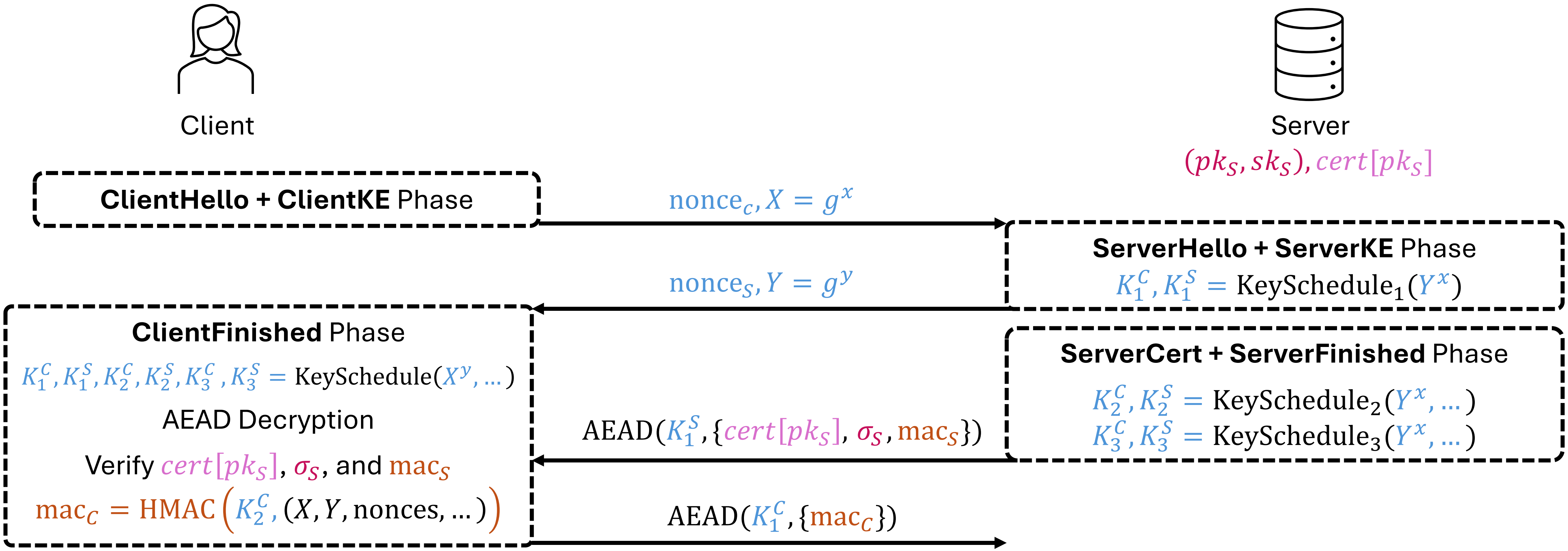
$K_2^C, K_2^S = \text{KeySchedule}_2(Y^x, \ldots)$

$K_3^C, K_3^S = \text{KeySchedule}_3(Y^x, \ldots)$

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

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\mathrm{nonce}_c, X = g^x$

ServerHello + ServerKE Phase

$K_1^C, K_1^S = \mathrm{KeySchedule}_1(Y^x)$

$\mathrm{nonce}_S, Y = g^y$

ClientFinished Phase

$K_1^C, K_1^S, K_2^C, K_2^S, K_3^C, K_3^S = \mathrm{KeySchedule}(X^y, \dots)$

AEAD Decryption

Verify $cert[pk_S]$, σ_S, and mac_S

$\mathrm{mac}_C = \mathrm{HMAC}\left(K_2^C, (X, Y, \mathrm{nonces}, \dots)\right)$

ServerCert + ServerFinished Phase

$K_2^C, K_2^S = \mathrm{KeySchedule}_2(Y^x, \dots)$

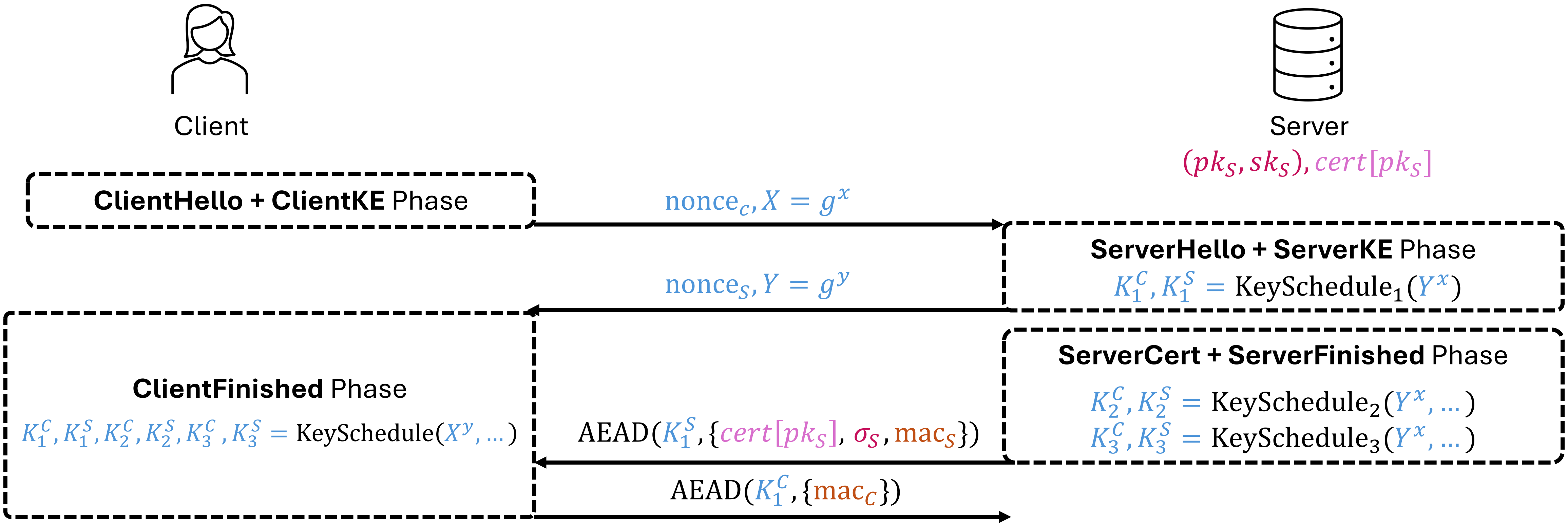
$K_3^C, K_3^S = \mathrm{KeySchedule}_3(Y^x, \dots)$

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

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

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\mathrm{nonce}_c, X = g^x$

ServerHello + ServerKE Phase

$K_1^C, K_1^S = \mathrm{KeySchedule}_1(Y^x)$

$\mathrm{nonce}_S, Y = g^y$

ServerCert + ServerFinished Phase

$K_2^C, K_2^S = \mathrm{KeySchedule}_2(Y^x, \ldots)$

$K_3^C, K_3^S = \mathrm{KeySchedule}_3(Y^x, \ldots)$

ClientFinished Phase

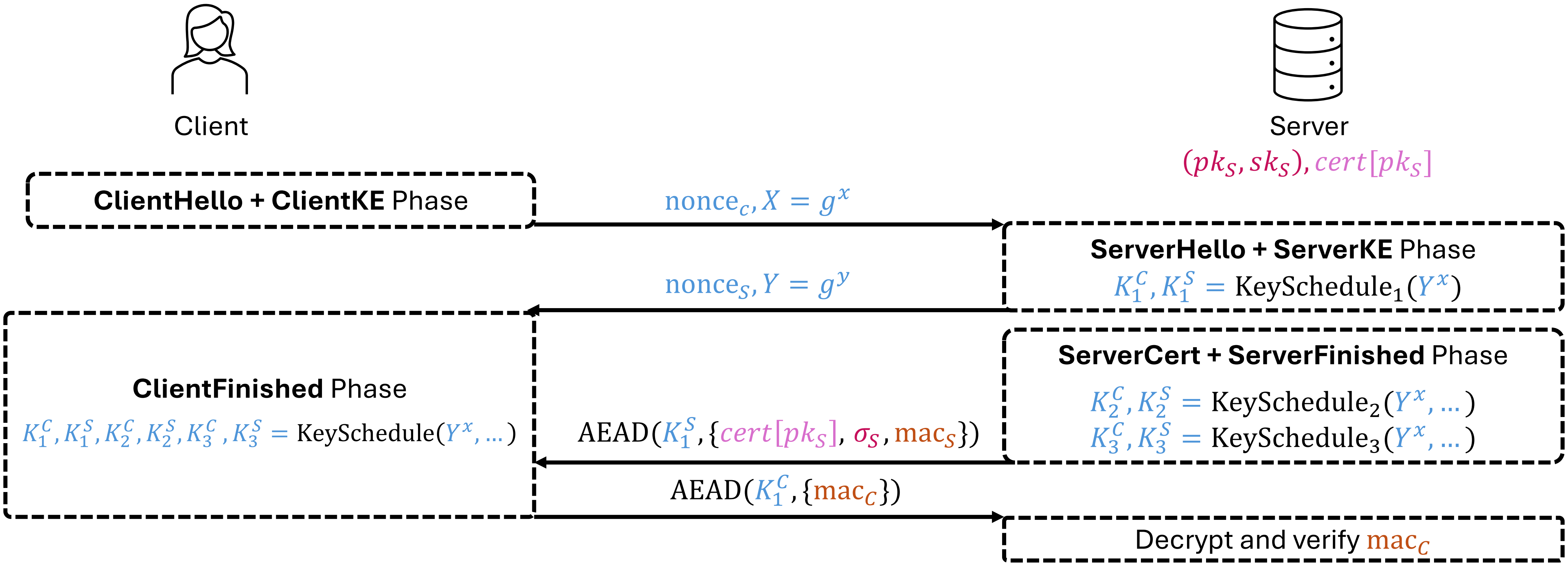
$K_1^C, K_1^S, K_2^C, K_2^S, K_3^C, K_3^S = \mathrm{KeySchedule}(X^y, \ldots)$

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

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

TLS Handshake Protocol

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



Client

Server

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

ClientHello + **ClientKE** Phase

$\mathrm{nonce}_c, X = g^x$

ServerHello + **ServerKE** Phase

$K_1^C, K_1^S = \mathrm{KeySchedule}_1(Y^x)$

$\mathrm{nonce}_S, Y = g^y$

ServerCert + **ServerFinished** Phase

$K_2^C, K_2^S = \mathrm{KeySchedule}_2(Y^x, \dots)$

$K_3^C, K_3^S = \mathrm{KeySchedule}_3(Y^x, \dots)$

ClientFinished Phase

$K_1^C, K_1^S, K_2^C, K_2^S, K_3^C, K_3^S = \mathrm{KeySchedule}(Y^x, \dots)$

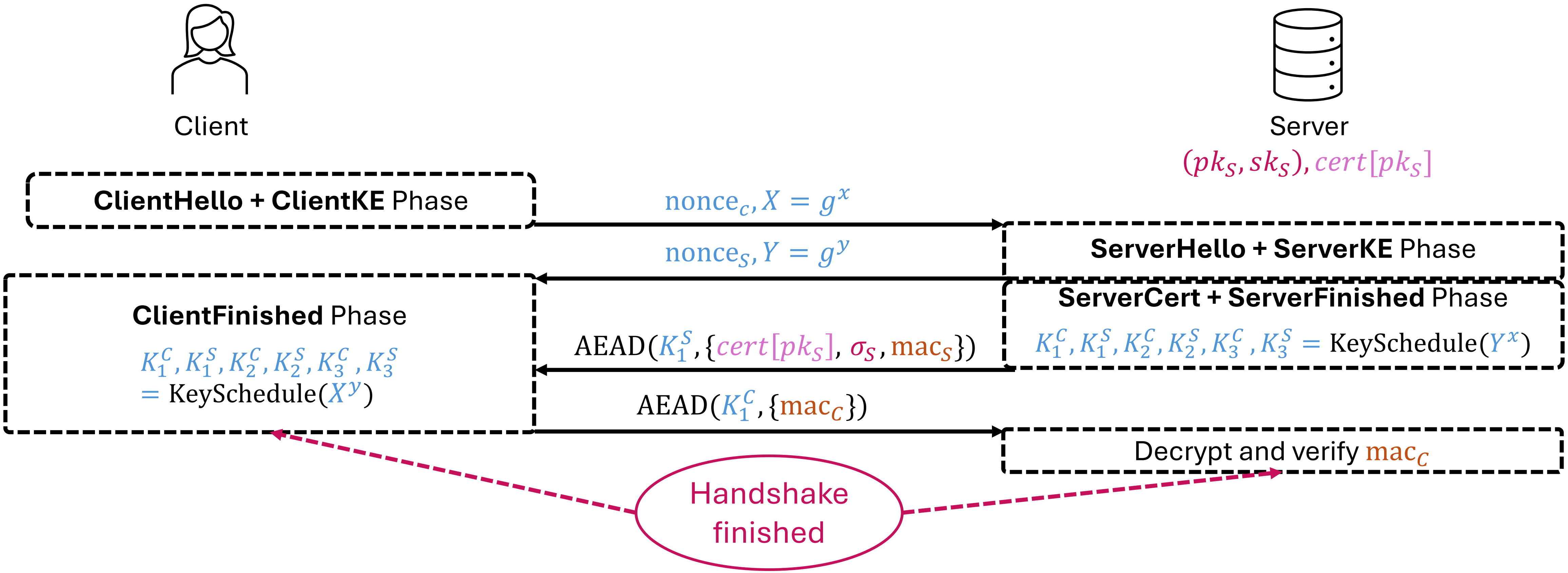
$\mathrm{AEAD}(K_1^S, \{cert[pk_S], \sigma_S, \mathrm{mac}_S\})$

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

Decrypt and verify mac_C

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\text{nonce}_c, X = g^x$

ServerHello + ServerKE Phase

$\text{nonce}_S, Y = g^y$

ClientFinished Phase

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

ServerCert + ServerFinished Phase

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

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

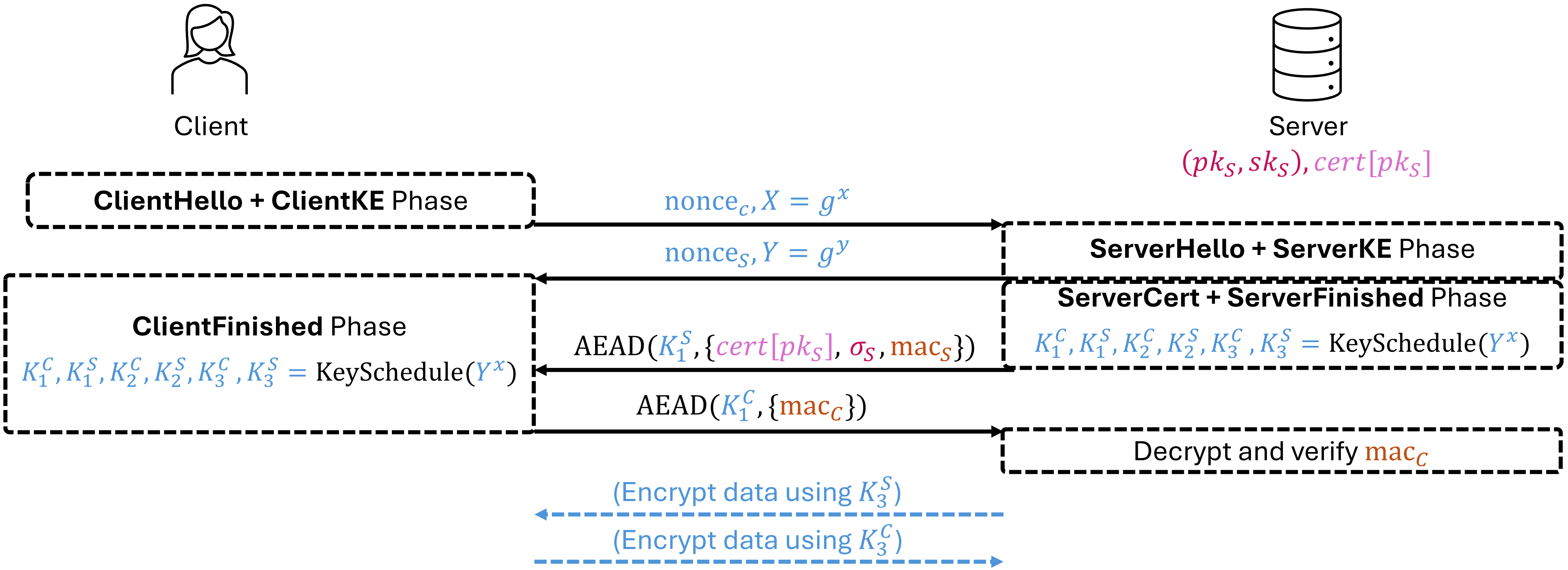
$\text{AEAD}(K_1^C, \{\text{mac}_C\})$

Decrypt and verify mac_C

Handshake finished

TLS Handshake Protocol

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



Client

Server

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

ClientHello + ClientKE Phase

$\text{nonce}_c, X = g^x$

ServerHello + ServerKE Phase

$\text{nonce}_S, Y = g^y$

ServerCert + ServerFinished Phase

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

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

ClientFinished Phase

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

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

Decrypt and verify mac_C

(Encrypt data using K_3^S)

(Encrypt data using K_3^C)

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

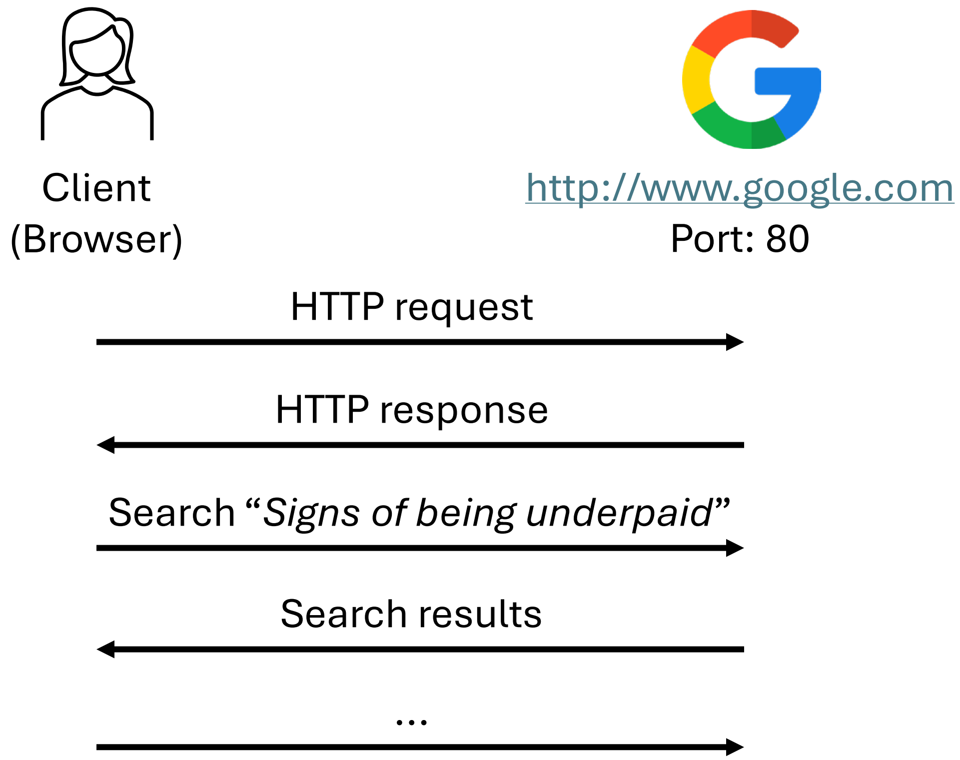
Just an example. You probably cannot access http://www.google.com because your browser or Google enforces HTTPs connections.

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

HTTP request →

← HTTP response

Search “*Signs of being underpaid*” →

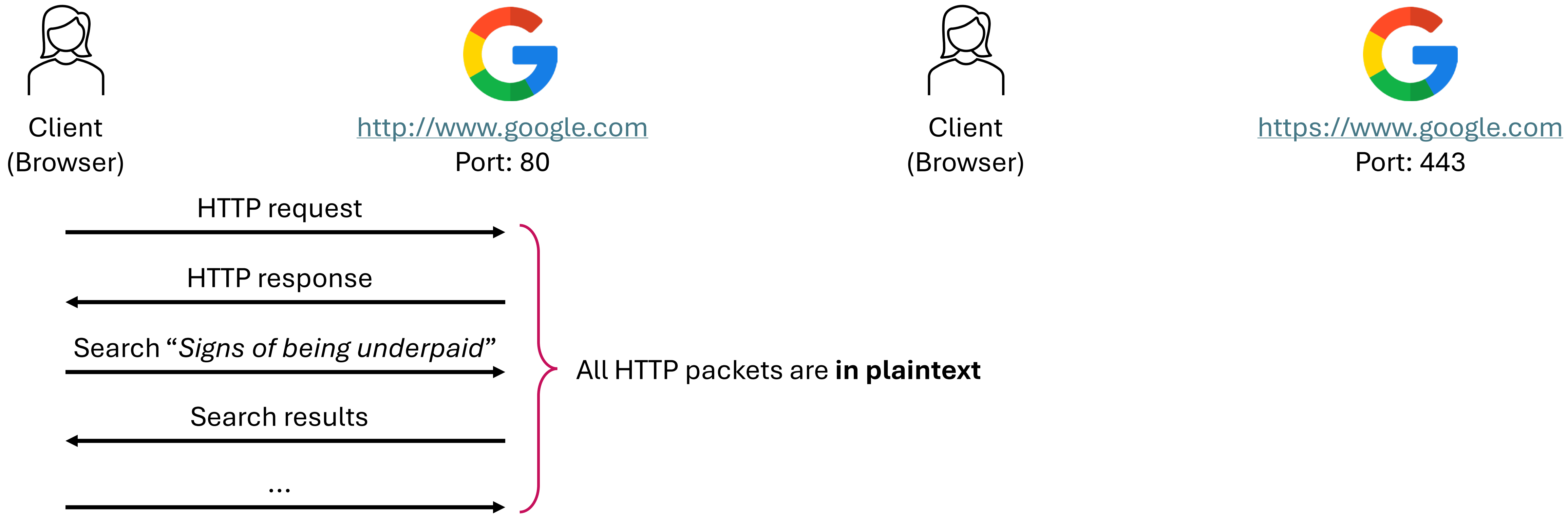
← Search results

… →

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

HTTP request →

← HTTP response

Search “*Signs of being underpaid*” →

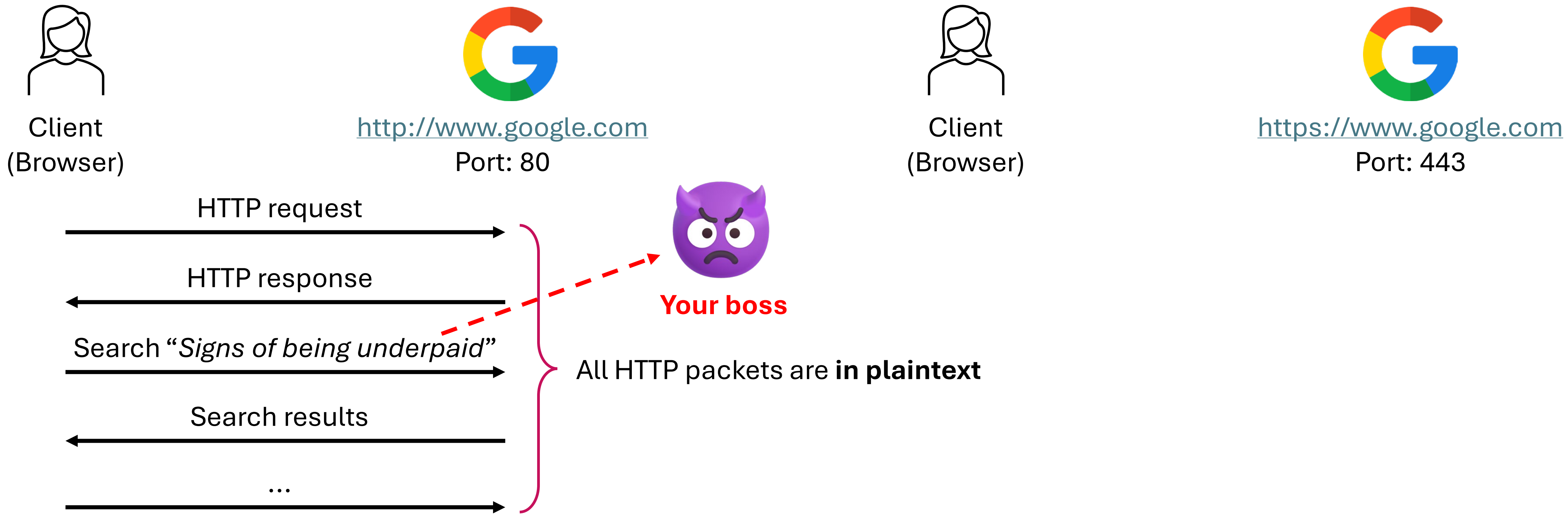
← Search results

… →

All HTTP packets are **in plaintext**

Case Study: HTTPs = HTTP over TLS

HTTP v.s. **HTTPs**



Client
(Browser)

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

HTTP request

HTTP response

Search “*Signs of being underpaid*”

Search results

...

Your boss

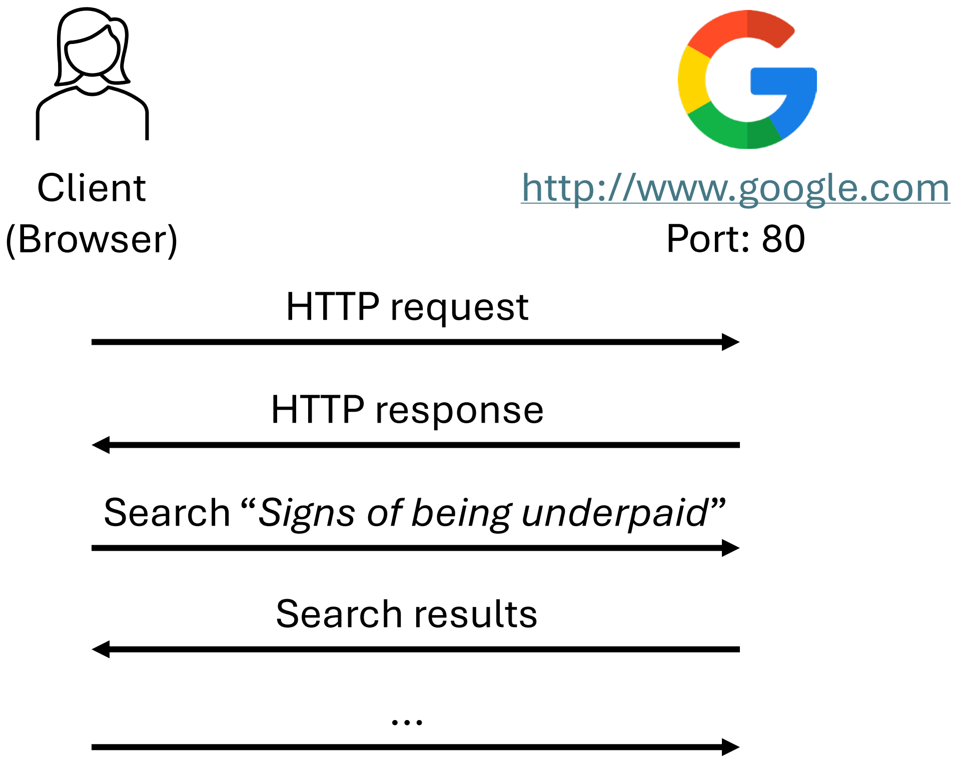
All HTTP packets are **in plaintext**

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

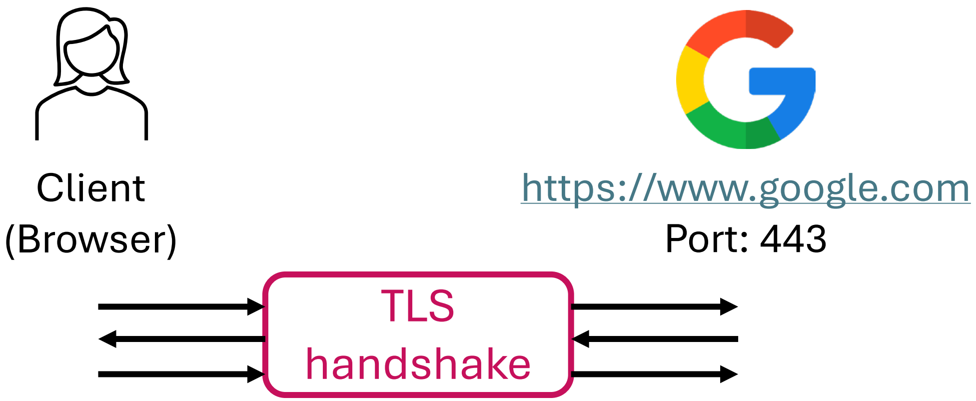
HTTP request

HTTP response

Search “*Signs of being underpaid*”

Search results

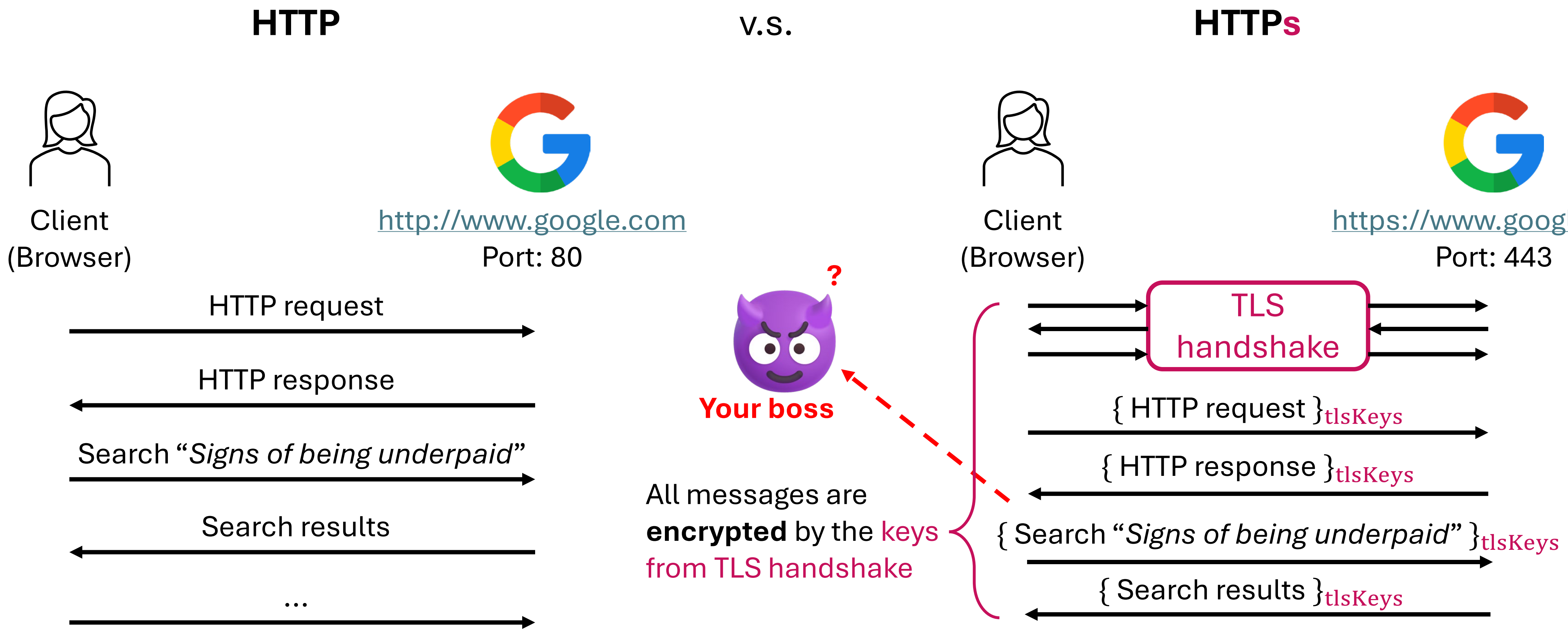
…


Client
(Browser)

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

TLS
handshake

Case Study: HTTPs = HTTP over TLS



HTTP v.s. **HTTPs**

Client (Browser)

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

HTTP request

HTTP response

Search “*Signs of being underpaid*”

Search results

…

?

Your boss

All messages are **encrypted** by the keys from TLS handshake

Client (Browser)

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

TLS handshake

$\{$ HTTP request $\}_{\text{tlsKeys}}$

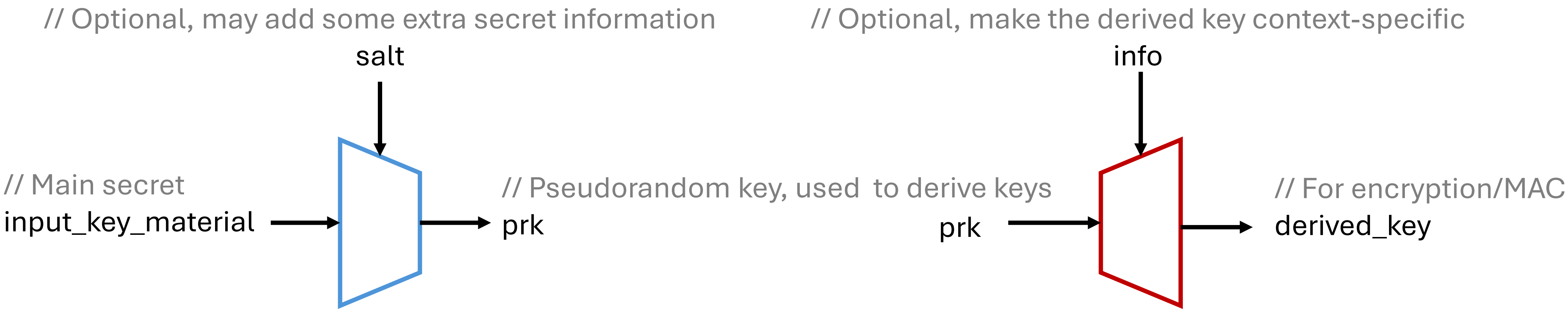
$\{$ HTTP response $\}_{\text{tlsKeys}}$

$\{$ Search “*Signs of being underpaid*” $\}_{\text{tlsKeys}}$

$\{$ Search results $\}_{\text{tlsKeys}}$

Coding Tasks

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



// Optional, may add some extra secret information

salt

// Main secret

input_key_material

// Pseudorandom key, used to derive keys

prk

// Optional, make the derived key context-specific

info

prk

// For encryption/MAC

derived_key

prk = HKDF.**Extract**(input_key_material, salt)

derived_key = HKDF.**Expand**(prk, info)

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:

```
$\mathbf{KeySchedule_1}(g^{xy})$:
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 $K_1^C$, $K_1^S$
```

```
$\mathbf{DeriveHS}(g^{xy})$:
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
```

```
$\mathbf{KeySchedule_2}(\mathrm{nonce}_C, X, \mathrm{nonce}_S, Y, g^{xy})$:
1. HS = DeriveHS($g^{xy}$)
2. ClientKC = SHA256($\mathrm{nonce}_C$ || $X$ || $\mathrm{nonce}_S$ || $Y$ || "ClientKC") // "||" is the concatenation operation
3. ServerKC = SHA256($\mathrm{nonce}_C$ || $X$ || $\mathrm{nonce}_S$ || $Y$ || "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

```
$\mathbf{KeySchedule_3}(\mathrm{nonce}_C, X, \mathrm{nonce}_S, Y, g^{xy}, \sigma, cert[pk_S], \mathrm{mac}_S)$:
1. HS = $\mathrm{DeriveHS}(g^{xy})$
2. dHS = HKDF.Expand(HS, SHA256(“DerivedHS”))
3. MS = HKDF.Extract(dHS, 0) // 0 = zeros (bytes) of length 32
2. ClientSKH = SHA256($\mathrm{nonce}_C$ || $X$ || $\mathrm{nonce}_S$ || $Y$ || $\sigma$ || $cert[pk_S]$ || $\mathrm{mac}_S$|| “ClientEncK”)
3. ServerSKH = SHA256($\mathrm{nonce}_C$ || $X$ || $\mathrm{nonce}_S$ || $Y$ || $\sigma$ || $cert[pk_S]$ || $\mathrm{mac}_S$|| “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: $\sigma = Sign\left(sk_S, \mathrm{SHA256}(\mathrm{nonce}_C \,||\, X \,||\, \mathrm{nonce}_S \,||\, Y \,||\, cert[pk_S])\right)$ // Use DSA with SHA256 and P256

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

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

- **How to verify HMAC:** To verify if mac is the *valid* HMAC code of M with respect to the key K,

  Just check: mac =? HMAC(K, M)

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

- **How to deal with the certificate:**

  You may generate a key pair (pk_S, sk_S) for server and a key pair (pk_{CA}, sk_{CA}) for the CA.

  1. "Hardcode" the key pair (pk_S, sk_S) for server into the program of the server.
  2. "Hardcode" the public key pk_{CA} of CA into the programs of the server and the client.
  3. Run a separate Python program to generate $\sigma_{CA} = Sign(pk_{CA}, pk_S)$ using ECDSA.
  4. "Hardcode" (pk_S, σ_{CA}) into the program of the server. And define $cert[pk_S] = (pk_S, \sigma_{CA})$
  5. When the server send $cert[pk_S]$ to the client, the client can use pk_{CA} to verify it by running:

  ECDSA.Verify(pk_{CA}, pk_S, σ_{CA})

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