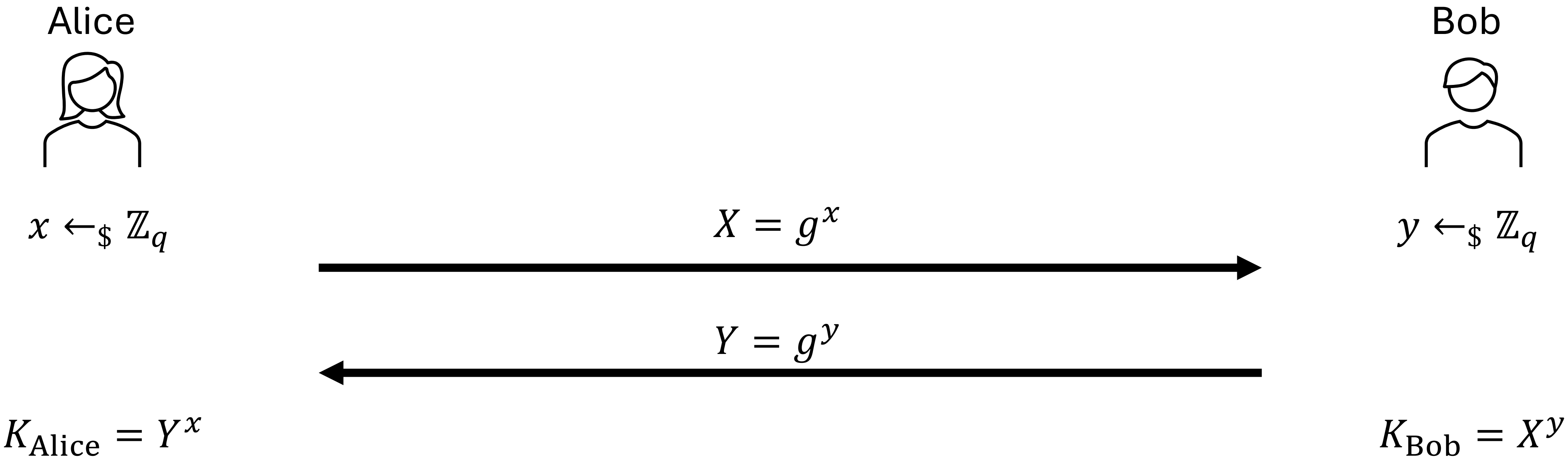
# Cryptography Engineering

- Lecture 2 (Oct 30, 2024)

- Today's notes:
  - Man-in-the-Middle attacks
  - DSA signature, and nonce reuse
  - Certificate
- Today's coding tasks (and homework):
  - Man-in-the-Middle attacks on DHKE
  - Nonce reuse attacks on DSA
  - Transporting pk using certificates (signatures)

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange



Alice

Bob

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

$X = g^x$

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

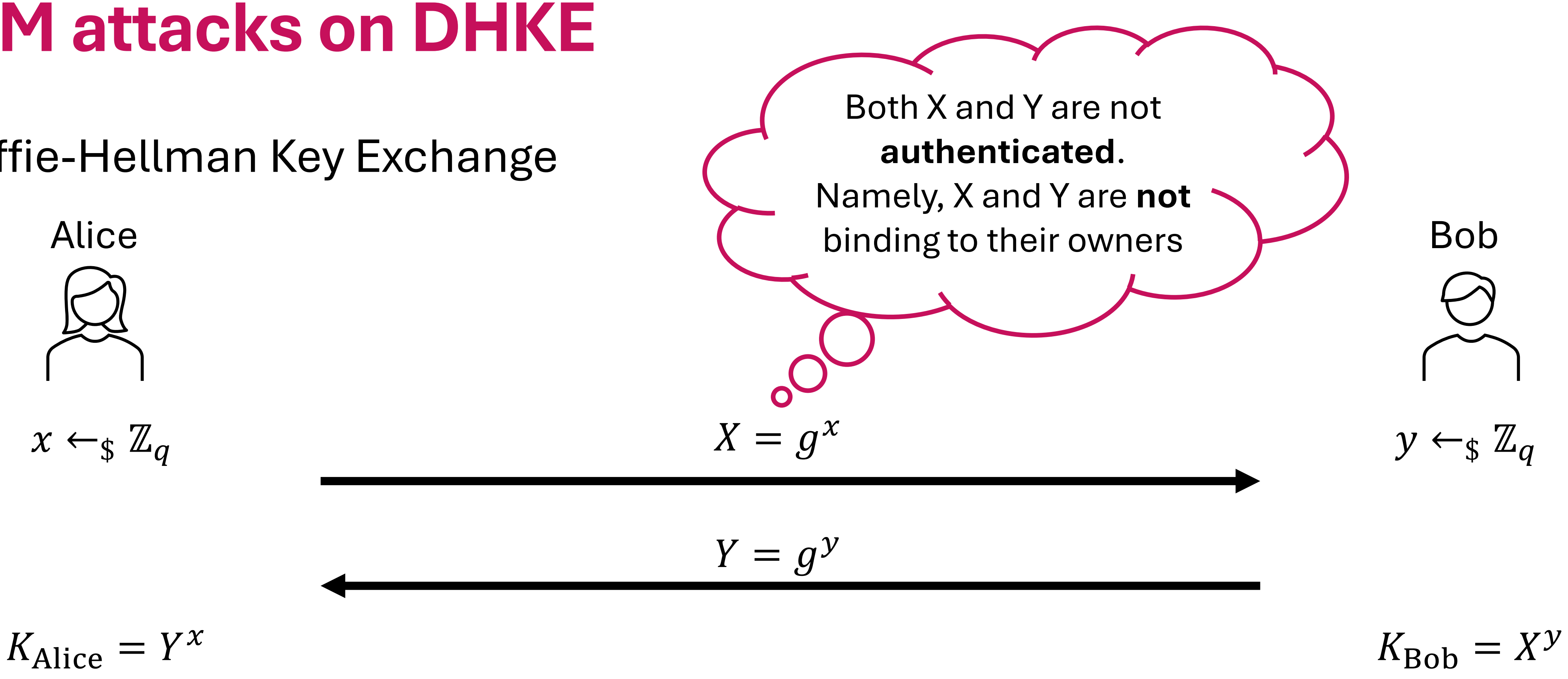
$Y = g^y$

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

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

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange



Both X and Y are not **authenticated**. Namely, X and Y are **not** binding to their owners

Alice

Bob

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

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

$X = g^x$

$Y = g^y$

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

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

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange

Alice

Adversary

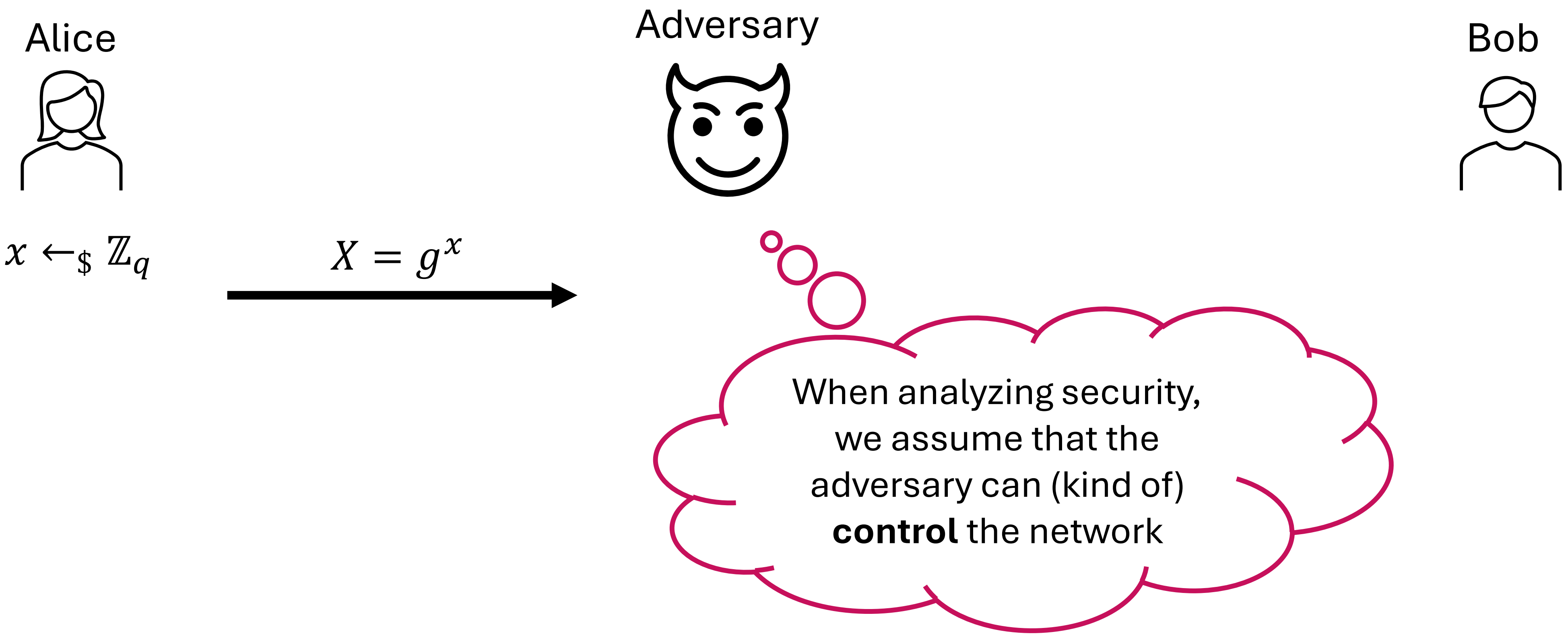
Bob

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

$X = g^x$

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange



Alice

Adversary

Bob

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

$X = g^x$

When analyzing security, we assume that the adversary can (kind of) **control** the network

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange

Alice

Adversary

Bob

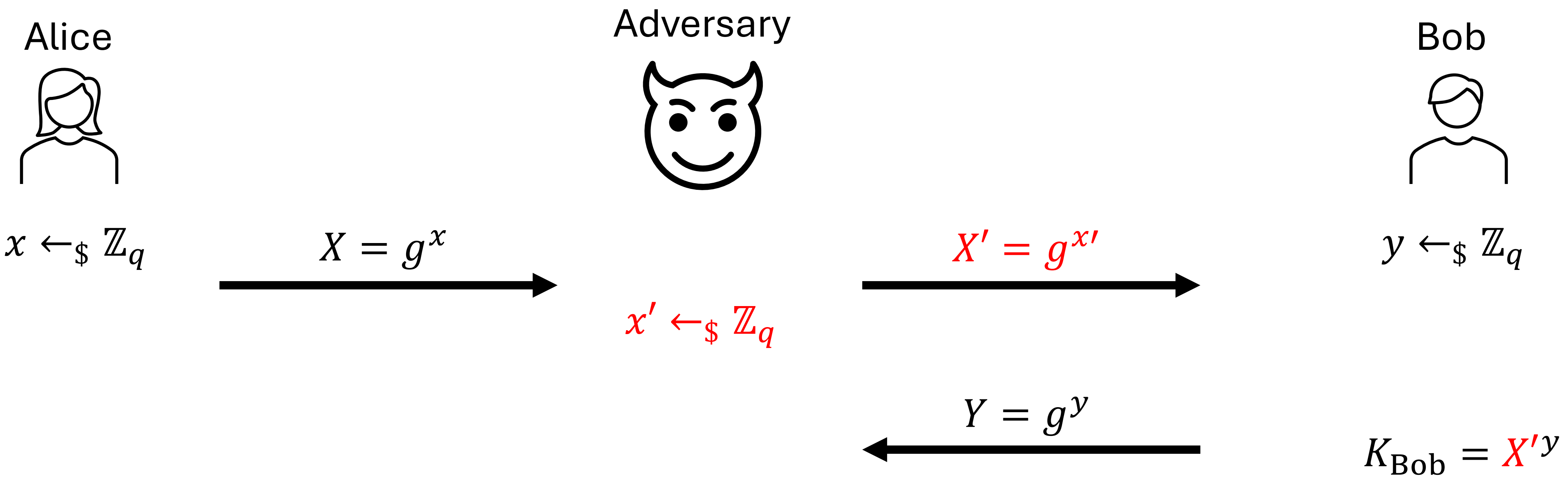
$x \leftarrow_{\$} \mathbb{Z}_q$

$X = g^x$

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

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange



Alice

Adversary

Bob

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

$X = g^x$

$X' = g^{x'}$

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

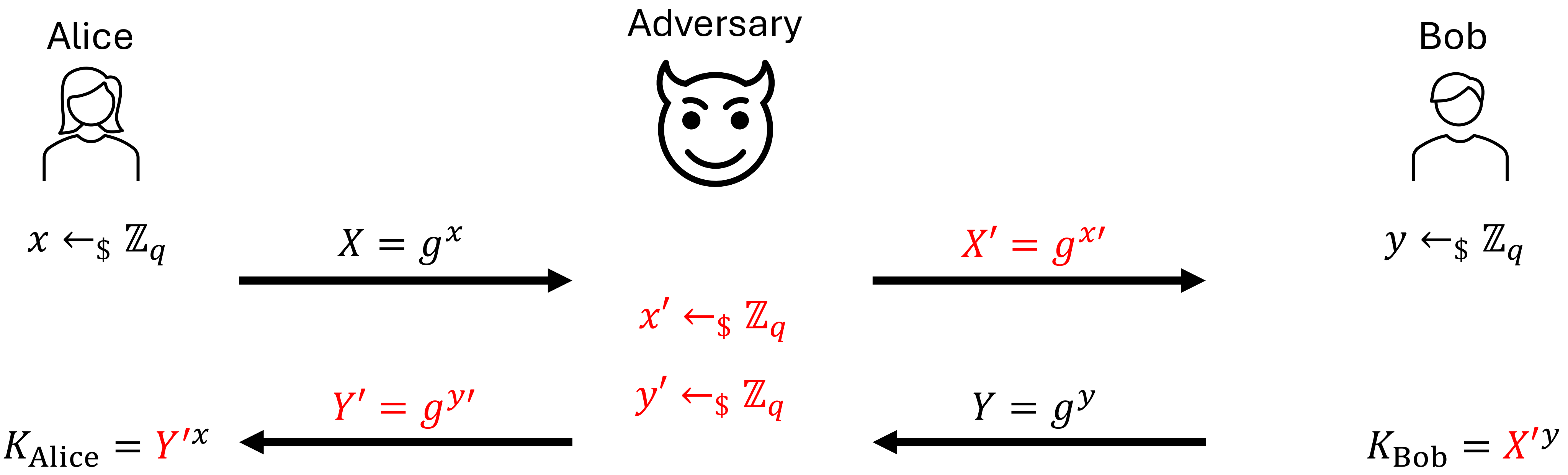
$x' \leftarrow_{\$} \mathbb{Z}_q$

$Y = g^y$

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

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange



Alice

Adversary

Bob

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

$X = g^x$

$X' = g^{x'}$

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

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

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

$Y' = g^{y'}$

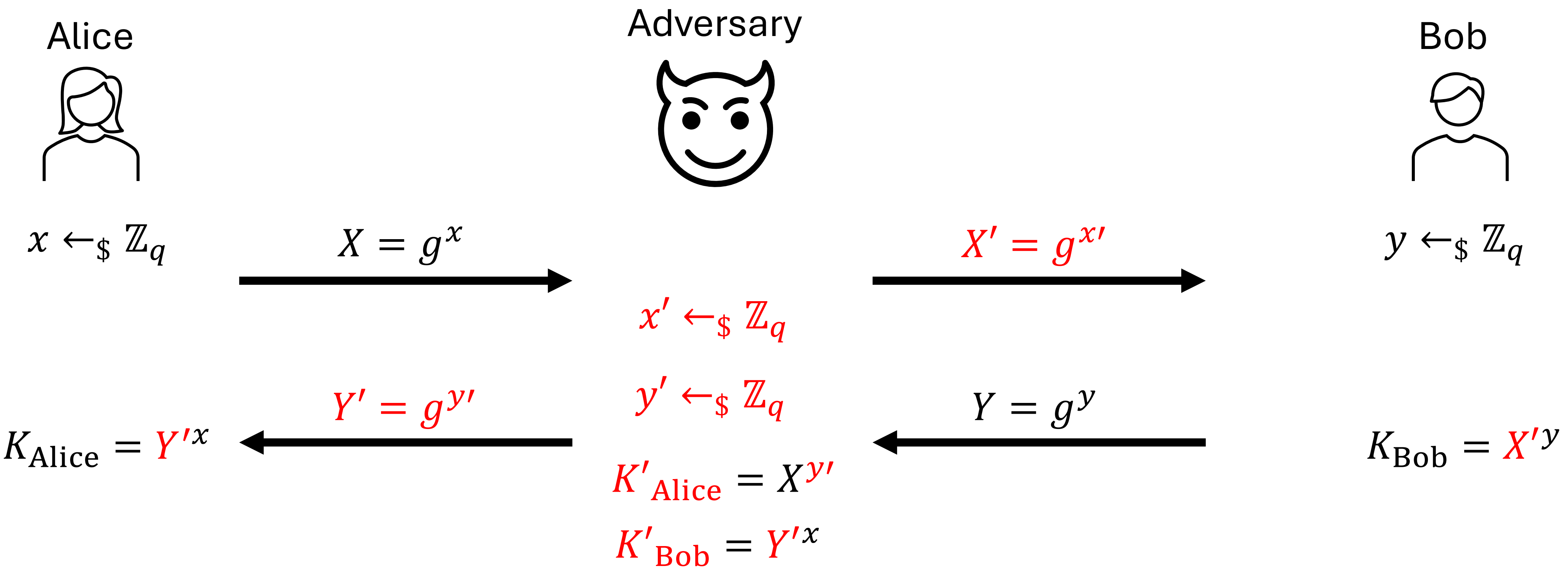
$Y = g^y$

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

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

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange



Alice

Adversary

Bob

$x \leftarrow_{\$} \mathbb{Z}_q$ $\xrightarrow{X = g^x}$ $\xrightarrow{X' = g^{x'}}$ $y \leftarrow_{\$} \mathbb{Z}_q$

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

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

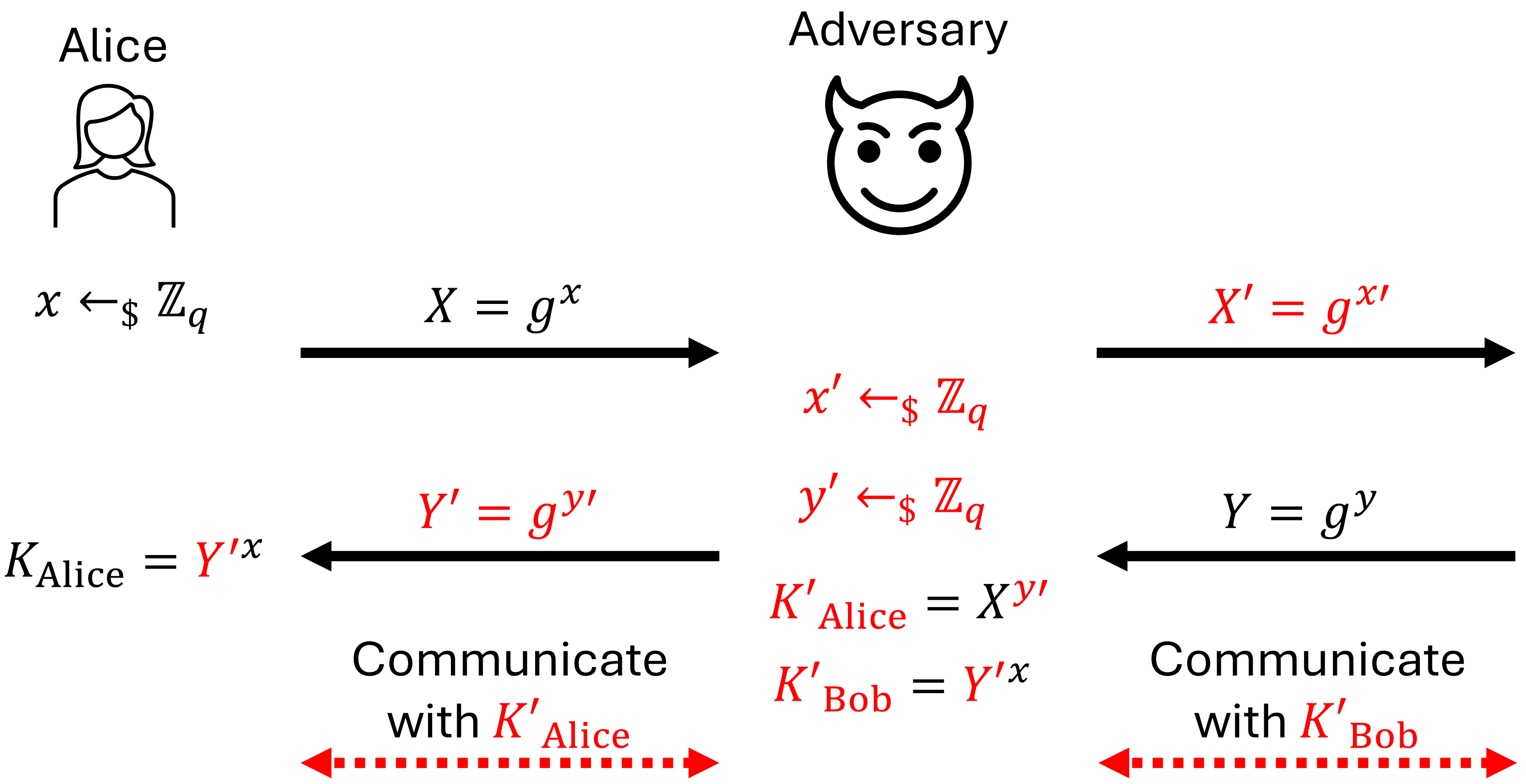
$K_{\text{Alice}} = Y'^{x}$ $\xleftarrow{Y' = g^{y'}}$ $\xleftarrow{Y = g^{y}}$ $K_{\text{Bob}} = X'^{y}$

$K'_{\text{Alice}} = X^{y'}$

$K'_{\text{Bob}} = Y'^{x}$

# MitM attacks on DHKE

- Diffie-Hellman Key Exchange



Alice

Adversary

Bob

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

$X = g^x$

$X' = g^{x'}$

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

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

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

$Y' = g^{y'}$

$Y = g^y$

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

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

$K'_{\text{Alice}} = X^{y'}$

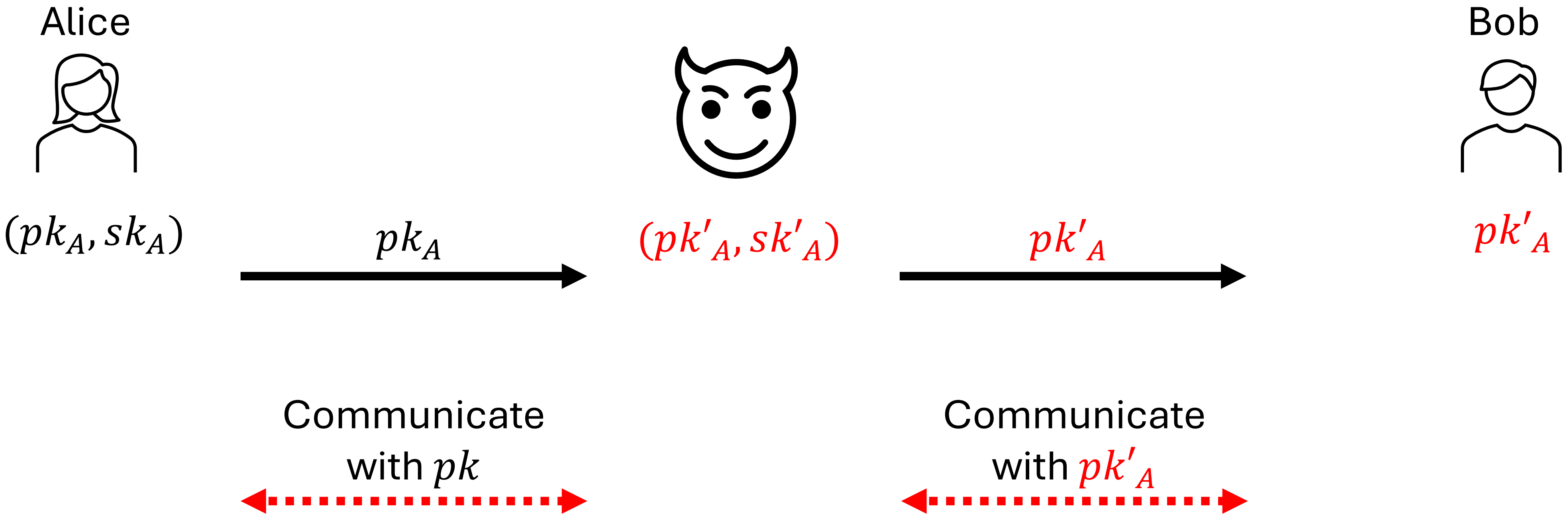
$K'_{\text{Bob}} = Y'^x$

Communicate with $K'_{\text{Alice}}$

Communicate with $K'_{\text{Bob}}$

# MitM attacks (in General)

- Transporting **(malicious)** public keys



Alice

Bob

$(pk_A, sk_A)$

$pk_A$

$(pk'_A, sk'_A)$

$pk'_A$

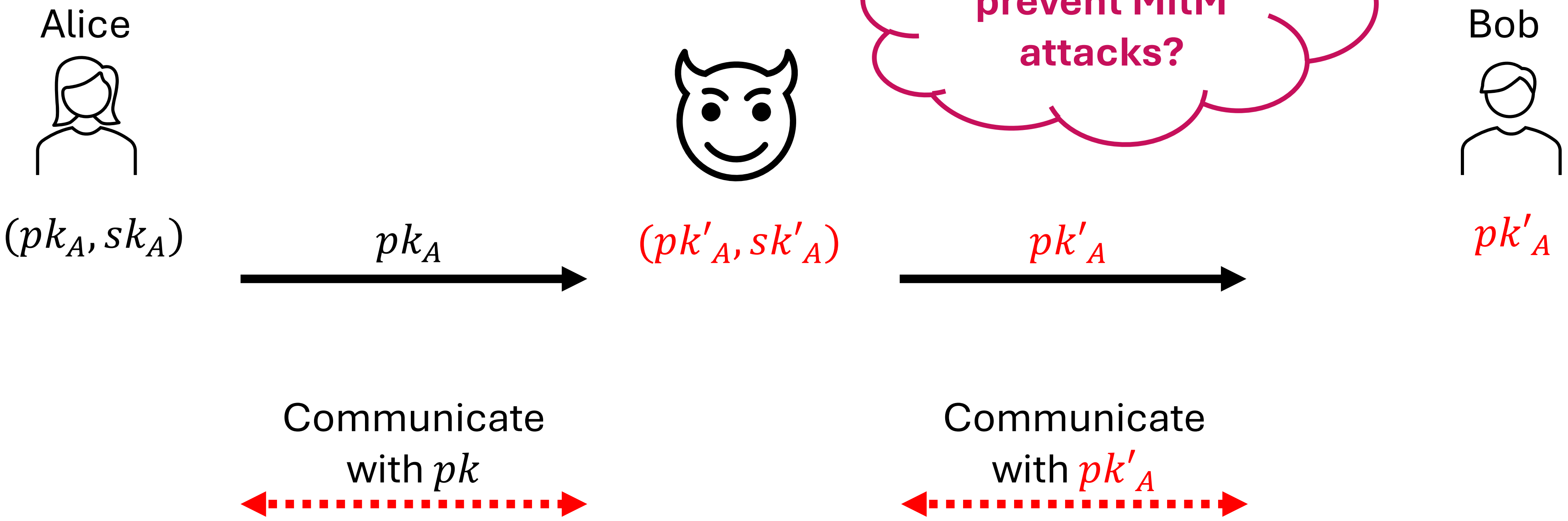
$pk'_A$

Communicate with $pk$

Communicate with $pk'_A$

# MitM attacks (in General)

- Transporting **(malicious)** public keys



**How can we prevent MitM attacks?**

Alice

$(pk_A, sk_A)$

$pk_A$

$(pk'_A, sk'_A)$

$pk'_A$

Bob

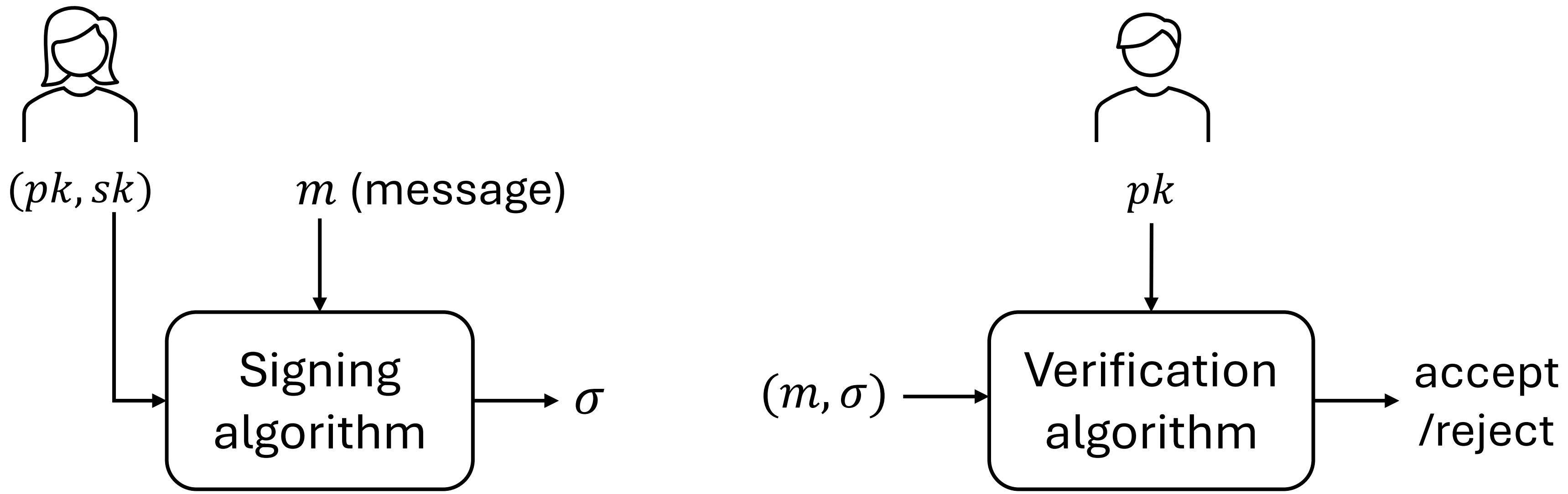
$pk'_A$

Communicate with $pk$

Communicate with $pk'_A$

# Digital Signature

- Signature Schemes



$(pk, sk)$

$m$ (message)

Signing algorithm

$\sigma$

$pk$

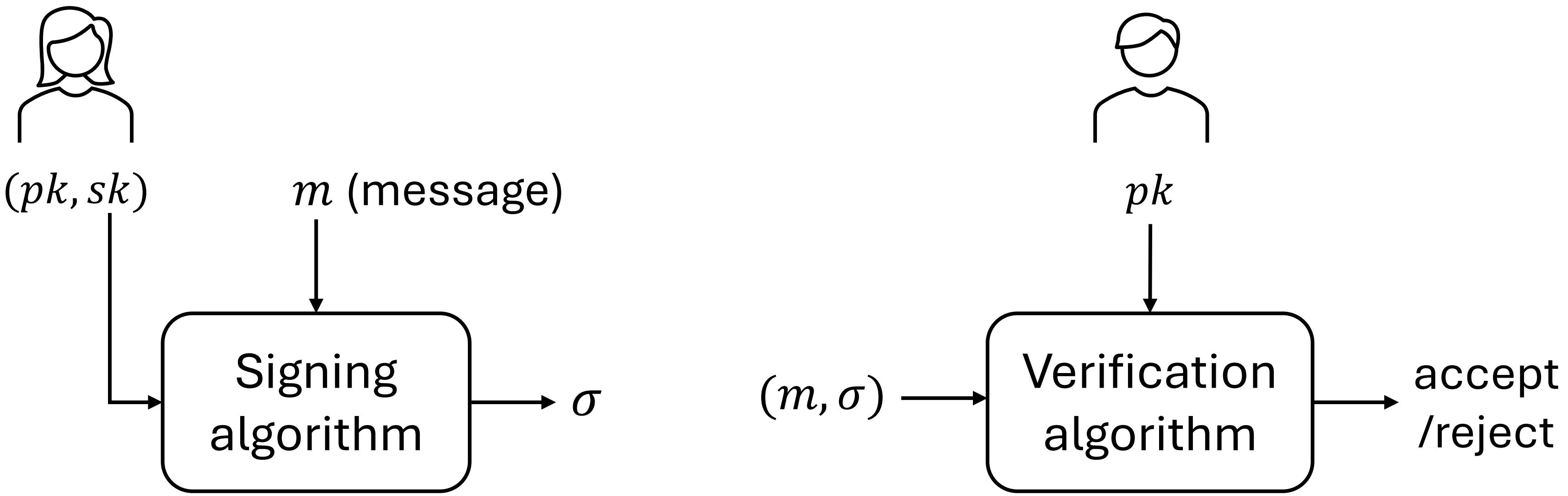
$(m, \sigma)$

Verification algorithm

accept /reject

# Digital Signature

- Signature Schemes



- Security: **Unforgeability**
  - **Unable to forge** a valid signature on any message without $sk$

# 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:

$$E\colon\ y^2 = x^3 + ax + b$$

- In Elliptic-Curve Cryptography (ECC), we use ECs over finite fields.
  - Example: SECP256R1 (used in our example Python code)

# Case Study: ECDSA

- ECDSA (Elliptic Curve Digital Signature Algorithm): DSA based on Elliptic Curve

- Public parameter (publicly known): $(\mathbb{CURVE}, \mathbb{G}, g, p)$
  - $\mathbb{CURVE}$: Tell the users what *the elliptic curve* and *equations* are being used.
  - $(\mathbb{G}, g, p)$: A subgroup $\mathbb{G}$ over $\mathbb{CURVE}$ with a **large prime** order $p$. The base point (generator) $g$ generates $\mathbb{G}$.

- Key Generation:
  - $sk = d \leftarrow_{\$} \mathbb{Z}_p^*$ // (= {1, 2, …, p-1} , here "*" means that we exclude zero)
  - $pk = d \circ g$ // "∘" 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) $e' = \mathrm{Hash}(m)$                                 // “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) $r = x \mod p$                                          // Now r is an integer module p. Given x, y is determined.
6) Assert $[x \mod p \neq 0]$                              // Make sure we do not get a “trivial point”
7) $s = k^{-1} \cdot (e + r \cdot d) \mod p$               // Signing
8) return $(r, s)$
```

# Case Study: ECDSA

- Verification algorithm ($pk$: public key, $m$: message, $(r, s)$: signature):
  1) $e' = \mathrm{Hash}(m)$ // "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) $u_1 = e \cdot s^{-1} \mod p$
  4) $u_2 = r \cdot s^{-1} \mod p$
  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) $e' = \mathrm{Hash}(m)$ // "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) $u_1 = e \cdot s^{-1} \mod p$
  4) $u_2 = r \cdot s^{-1} \mod p$
  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: $m_1, (r, s_1)$ with nonce $k$
$m_2, (r, s_2)$ with nonce $k$

(Same value $r$ if $k$ is the same)

By DSA construction:

$$s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \mod p$$
$$s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \mod p$$

# 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: $m_1, (r, s_1)$ with nonce $k$
$m_2, (r, s_2)$ with nonce $k$

(Same value $r$ if $k$ is the same)

By DSA construction:

$$s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \mod p$$
$$s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \mod p$$

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: $m_1, (r, s_1)$ with nonce $k$
$m_2, (r, s_2)$ with nonce $k$

(Same value $r$ if $k$ is the same)

By DSA construction:

$$s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \mod p$$
$$s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \mod p$$

By Linear Algebra:

$$\begin{bmatrix} s_1 & r \\ s_2 & r \end{bmatrix} \begin{bmatrix} k \\ d \end{bmatrix} = \begin{bmatrix} H(m_1) \\ H(m_2) \end{bmatrix} \mod p$$

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: $m_1, (r, s_1)$ with nonce $k$
$m_2, (r, s_2)$ with nonce $k$

(Same value $r$ if $k$ is the same)

By DSA construction:

$$s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \mod p$$
$$s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \mod p$$

By Linear Algebra:

$$\begin{bmatrix} k \\ d \end{bmatrix} = \begin{bmatrix} (s_1 - s_2)^{-1} \cdot (H(m_1) - H(m_2)) \\ r^{-1} \cdot (s_1 \cdot k - H(m_1)) \end{bmatrix} \mod p$$

# 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: $m_1, (r, s_1)$ with nonce $k$; $m_2, (r, s_2)$ with nonce $k$ (Same value $r$ if $k$ is the same)

By DSA construction:

$$s_1 = k^{-1} \cdot (H(m_1) + r \cdot d) \mod p$$
$$s_2 = k^{-1} \cdot (H(m_2) + r \cdot d) \mod p$$

By Linear Algebra:

$$\begin{bmatrix} k \\ d \end{bmatrix} = \begin{bmatrix} (s_1 - s_2)^{-1} \cdot (H(m_1) - H(m_2)) \\ r^{-1} \cdot (s_1 \cdot k - H(m_1)) \end{bmatrix} \mod p$$

- 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.

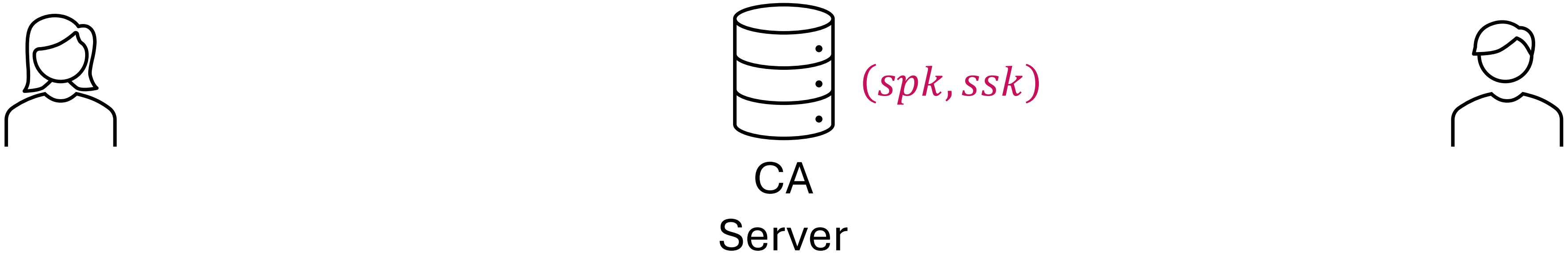
# Digital Signature

- Other standard properties of Digital Signature:
    - Authentication // Verify the identity...
    - Publicly verifiable // Everyone with pk can verify the signature...
    - Non-repudiation // A party cannot deny having sent or signed a message...
    - ...

- One of the most important application: **Digital Certificate**

# Digital Certificate

- Certificate: A signature generated by a trusted party (In short)
  - Verifies an ID and binds it to a public key
  - Securely distribute public keys
  - Issued by **CA** (Certificate Authority)

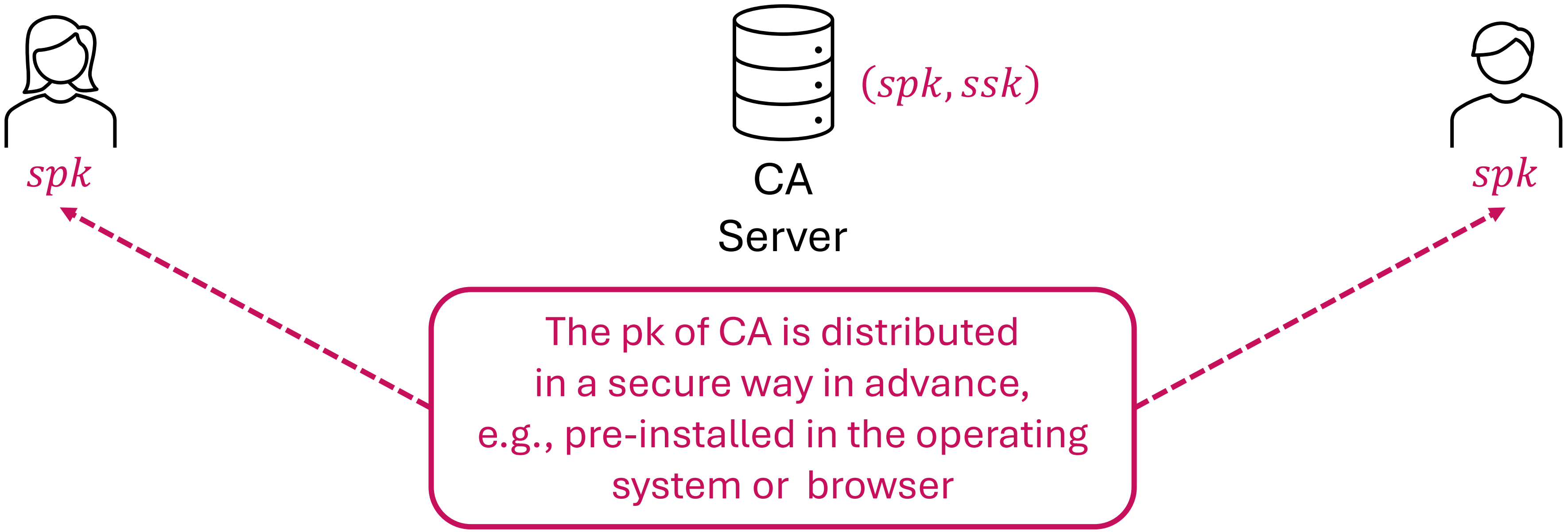


$(spk, ssk)$

CA
Server

# Digital Certificate

- Certificate: A signature generated by a trusted party (In short)
    - Verifies an ID and binds it to a public key
    - Securely distribute public keys
    - Issued by **CA** (Certificate Authority)



$(spk, ssk)$

CA
Server

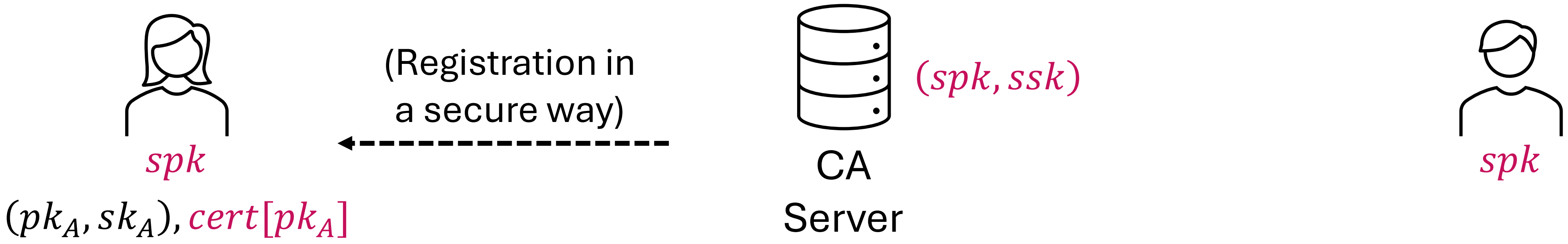
$spk$

$spk$

The pk of CA is distributed
in a secure way in advance,
e.g., pre-installed in the operating
system or browser

# Digital Certificate

- Certificate: A signature generated by a trusted party (In short)
  - Verifies an ID and binds it to a public key
  - Securely distribute public keys
  - Issued by **CA** (Certificate Authority)



(Registration in a secure way)

$spk$

$(pk_A, sk_A), cert[pk_A]$

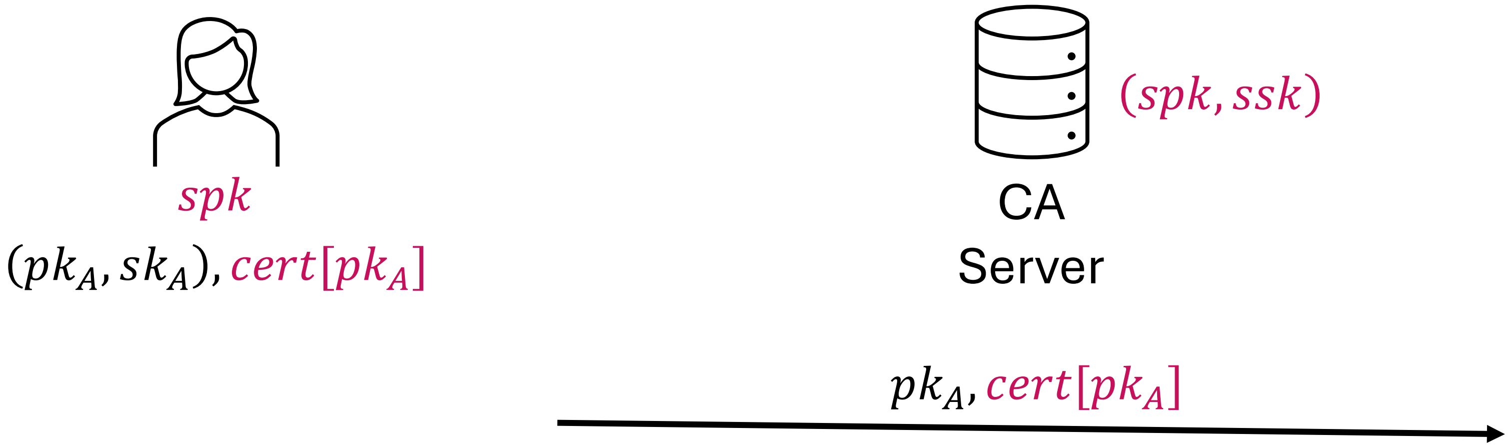
$(spk, ssk)$

CA Server

$spk$

# Digital Certificate

- Certificate: A signature generated by a trusted party (In short)
  - Verifies an ID and binds it to a public key
  - Securely distribute public keys
  - Issued by **CA** (Certificate Authority)



$spk$

$(pk_A, sk_A), cert[pk_A]$

$(spk, ssk)$

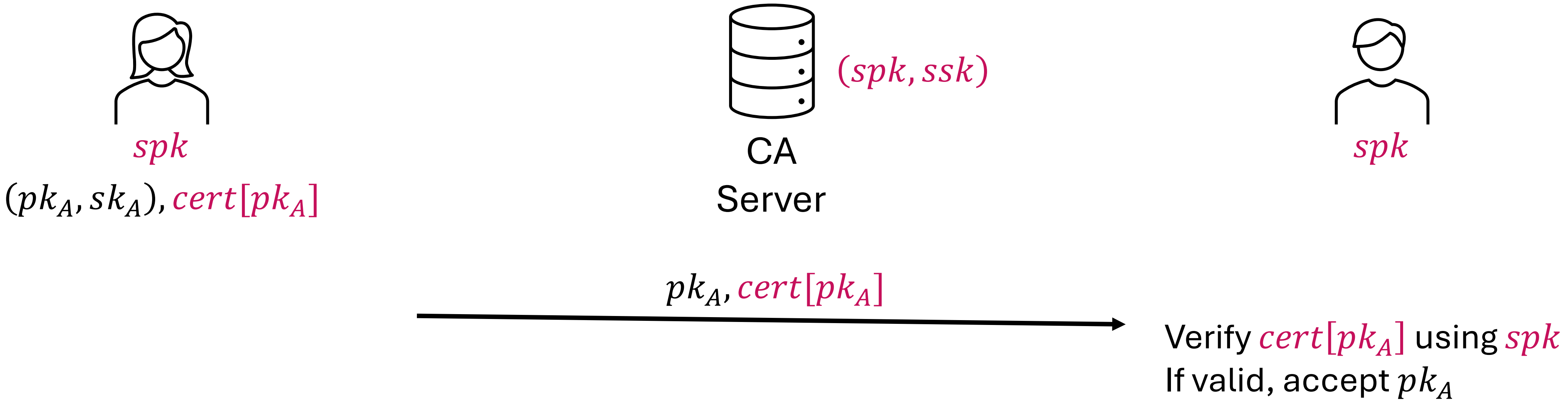
CA
Server

$spk$

$pk_A, cert[pk_A]$

# Digital Certificate

- Certificate: A signature generated by a trusted party (In short)
  - Verifies an ID and binds it to a public key
  - Securely distribute public keys
  - Issued by **CA** (Certificate Authority)



$spk$

$(pk_A, sk_A), cert[pk_A]$

$(spk, ssk)$

CA
Server

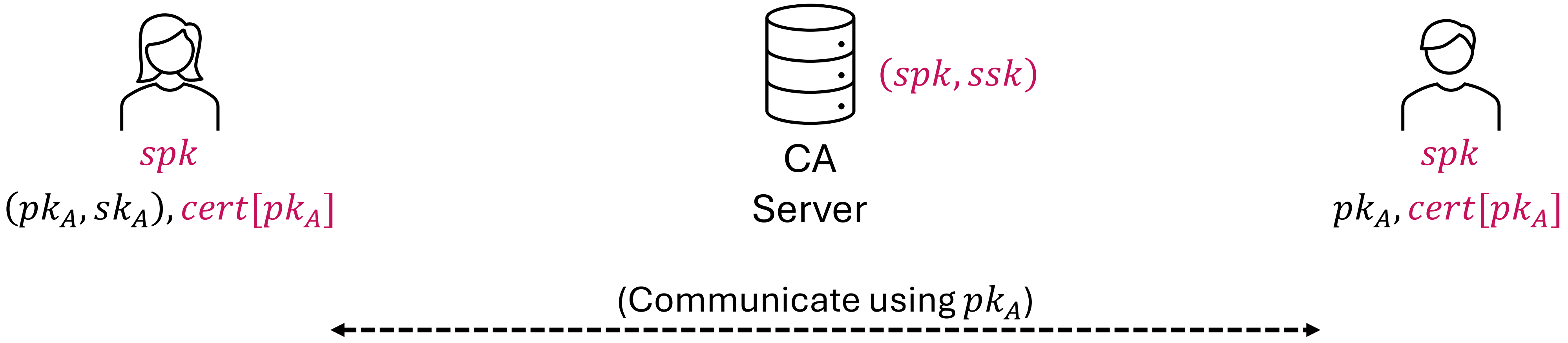
$spk$

$pk_A, cert[pk_A]$

Verify $cert[pk_A]$ using $spk$
If valid, accept $pk_A$

# Digital Certificate

- Certificate: A signature generated by a trusted party (In short)
  - Verifies an ID and binds it to a public key
  - Securely distribute public keys
  - Issued by **CA** (Certificate Authority)



$(spk, ssk)$

CA
Server

$spk$

$(pk_A, sk_A), cert[pk_A]$

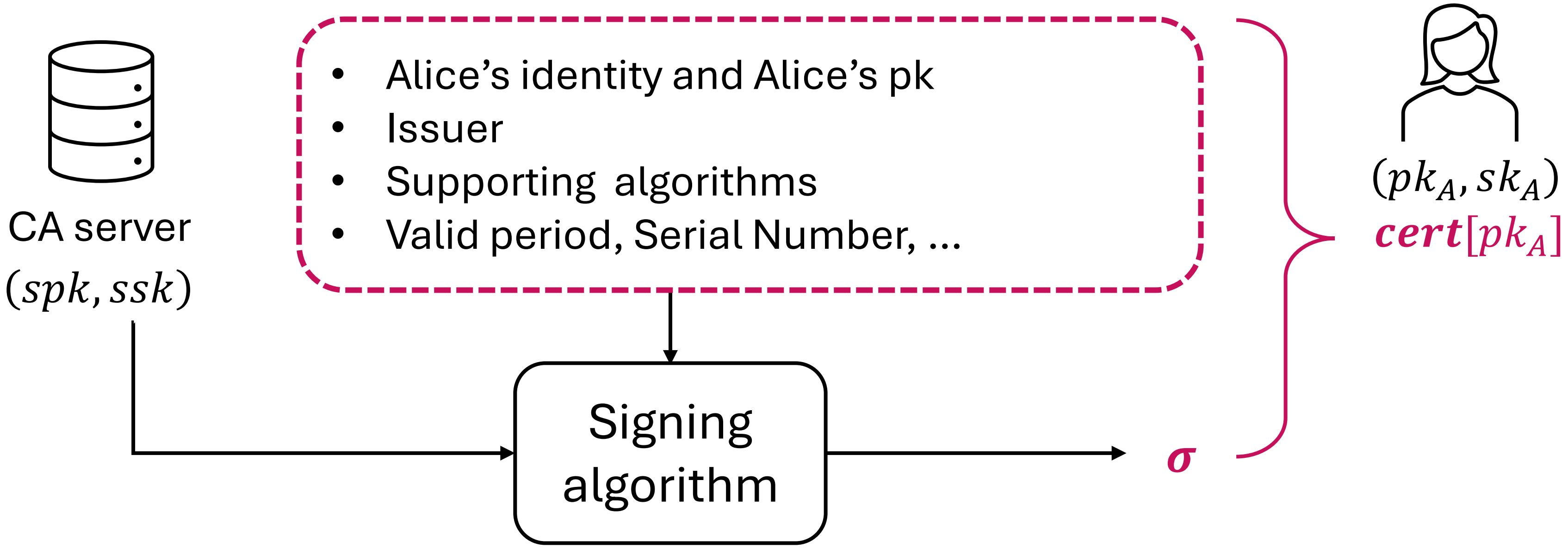
$spk$

$pk_A, cert[pk_A]$

(Communicate using $pk_A$)

# Digital Certificate

- What information does a certificate include?
  - X.509 standard: defines the format of public key certificates.



CA server

$(spk, ssk)$

- Alice's identity and Alice's pk
- Issuer
- Supporting algorithms
- Valid period, Serial Number, …

Signing algorithm

$\boldsymbol{\sigma}$

$(pk_A, sk_A)$

$\boldsymbol{cert}[pk_A]$

- Export a certificate and run the example code 'ReadCert.py'…

# Digital Certificate

- Root Certificate and Certificate Chains
  - Hierarchical sequence of certificates
  - Trace the authenticity of a certificate back to a trusted **Root CA**
  - Only **root certificates** need to be pre-installed…

# Digital Certificate

- Root Certificate and Certificate Chains
  - Hierarchical sequence of certificates
  - Trace the authenticity of a certificate back to a trusted **Root CA**
  - Only **root certificates** need to be pre-installed...

Root CA

$(rpk, rsk)$

$cert[rpk]$

Intermediate CA 1

$(pk_1, sk_1)$

Intermediate CA 2

$(pk_2, sk_2)$

End server

$(pk_3, sk_3)$

# Digital Certificate

- Root Certificate and Certificate Chains
  - Hierarchical sequence of certificates
  - Trace the authenticity of a certificate back to a trusted **Root CA**
  - Only **root certificates** need to be pre-installed…

Root CA

$(rpk, rsk)$

$cert[rpk]$

- Pre-installed pk
- Self-signed cert

Intermediate CA 1

$(pk_1, sk_1)$

Intermediate CA 2

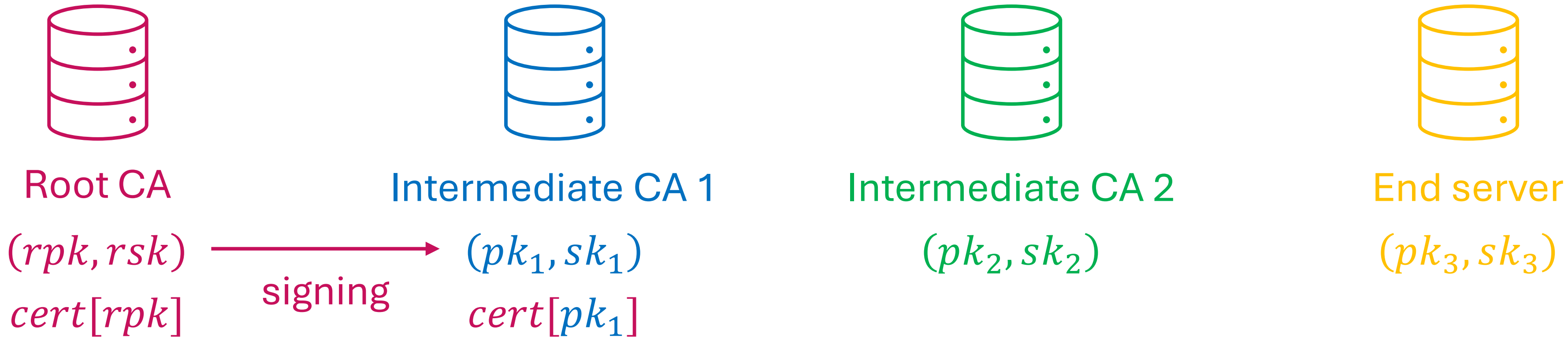
$(pk_2, sk_2)$

End server

$(pk_3, sk_3)$

# Digital Certificate

- Root Certificate and Certificate Chains
  - Hierarchical sequence of certificates
  - Trace the authenticity of a certificate back to a trusted **Root CA**
  - Only **root certificates** need to be pre-installed...



Root CA

$(rpk, rsk)$

$cert[rpk]$

signing

Intermediate CA 1

$(pk_1, sk_1)$

$cert[pk_1]$

Intermediate CA 2

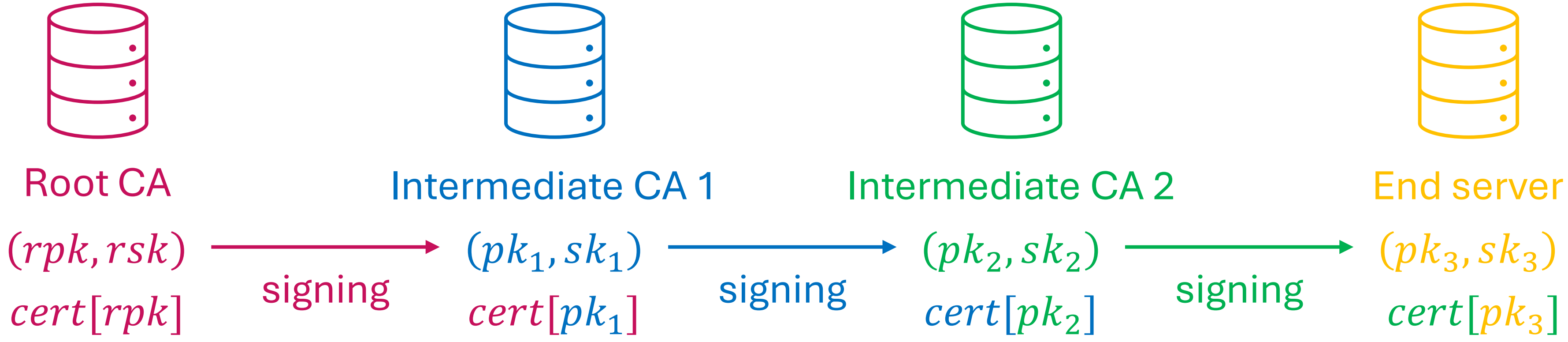
$(pk_2, sk_2)$

End server

$(pk_3, sk_3)$

# Digital Certificate

- Root Certificate and Certificate Chains
  - Hierarchical sequence of certificates
  - Trace the authenticity of a certificate back to a trusted **Root CA**
  - Only **root certificates** need to be pre-installed…



Root CA

$(rpk, rsk)$

$cert[rpk]$

signing

Intermediate CA 1

$(pk_1, sk_1)$

$cert[pk_1]$

signing

Intermediate CA 2

$(pk_2, sk_2)$

$cert[pk_2]$

signing

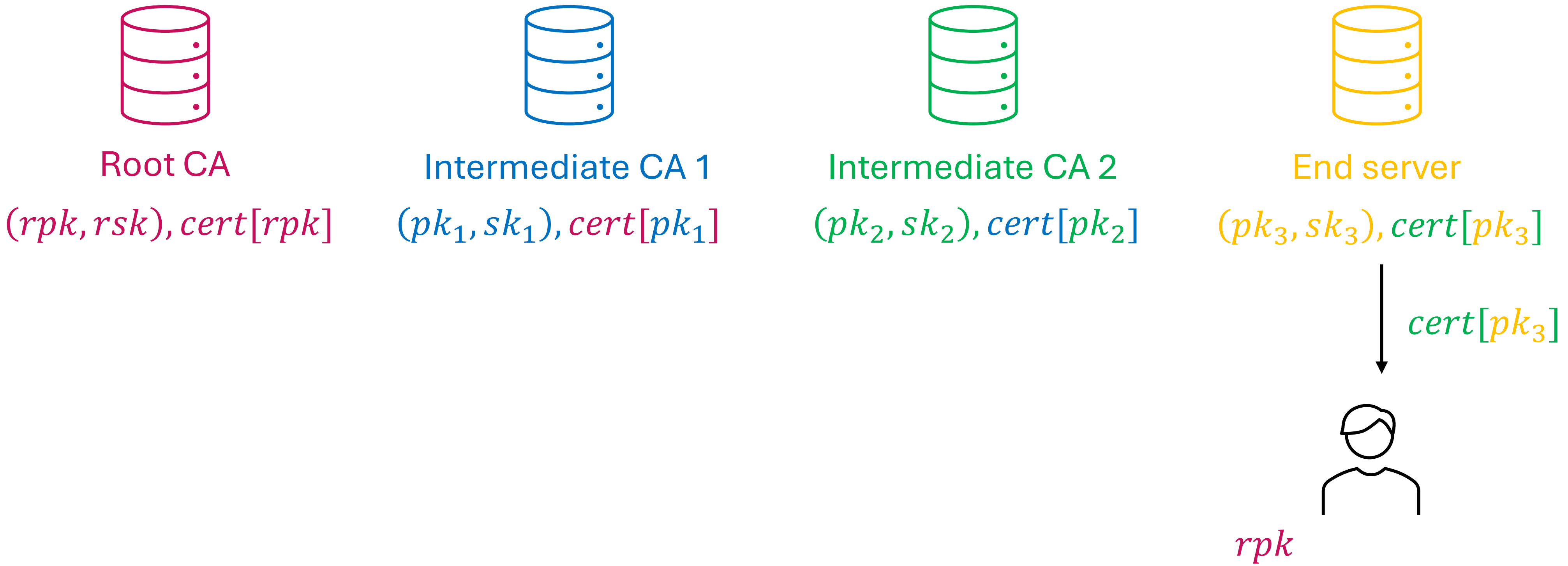
End server

$(pk_3, sk_3)$

$cert[pk_3]$

# Digital Certificate

- Root Certificate and Certificate Chains



Root CA

$(rpk, rsk), cert[rpk]$

Intermediate CA 1

$(pk_1, sk_1), cert[pk_1]$

Intermediate CA 2

$(pk_2, sk_2), cert[pk_2]$

End server

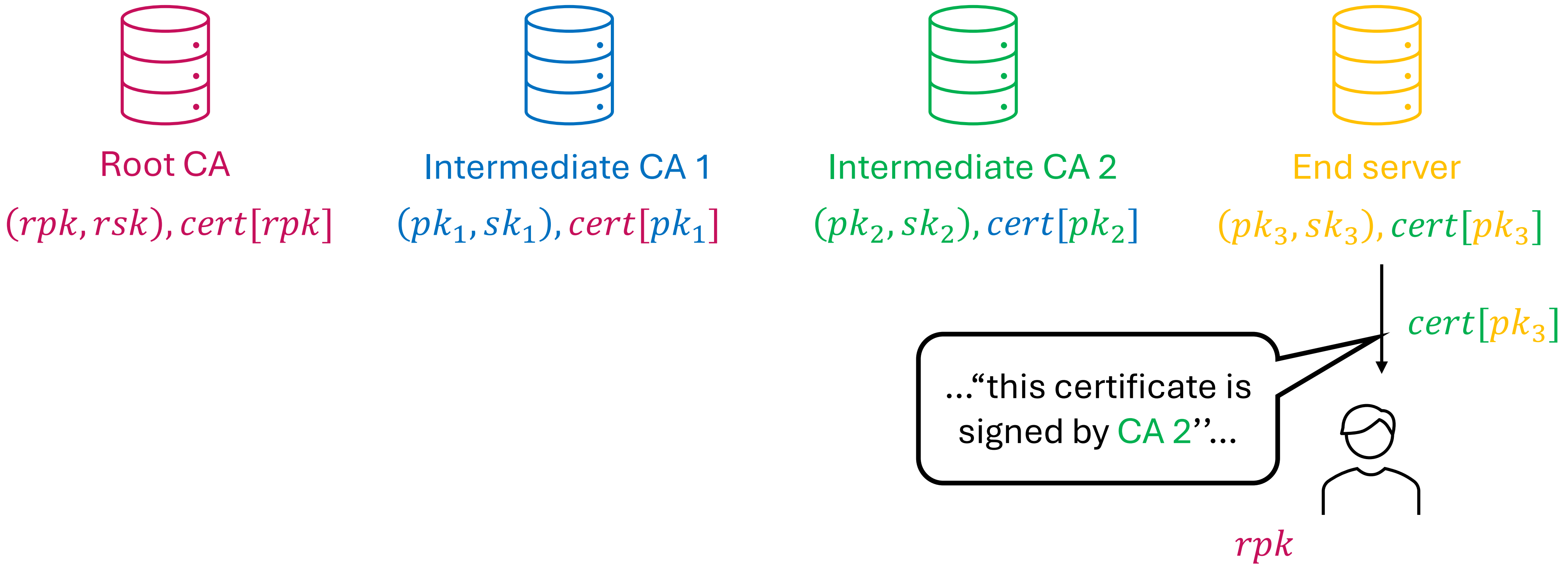
$(pk_3, sk_3), cert[pk_3]$

$cert[pk_3]$

$rpk$

# Digital Certificate

- Root Certificate and Certificate Chains



Root CA

$(rpk, rsk), cert[rpk]$

Intermediate CA 1

$(pk_1, sk_1), cert[pk_1]$

Intermediate CA 2

$(pk_2, sk_2), cert[pk_2]$

End server

$(pk_3, sk_3), cert[pk_3]$

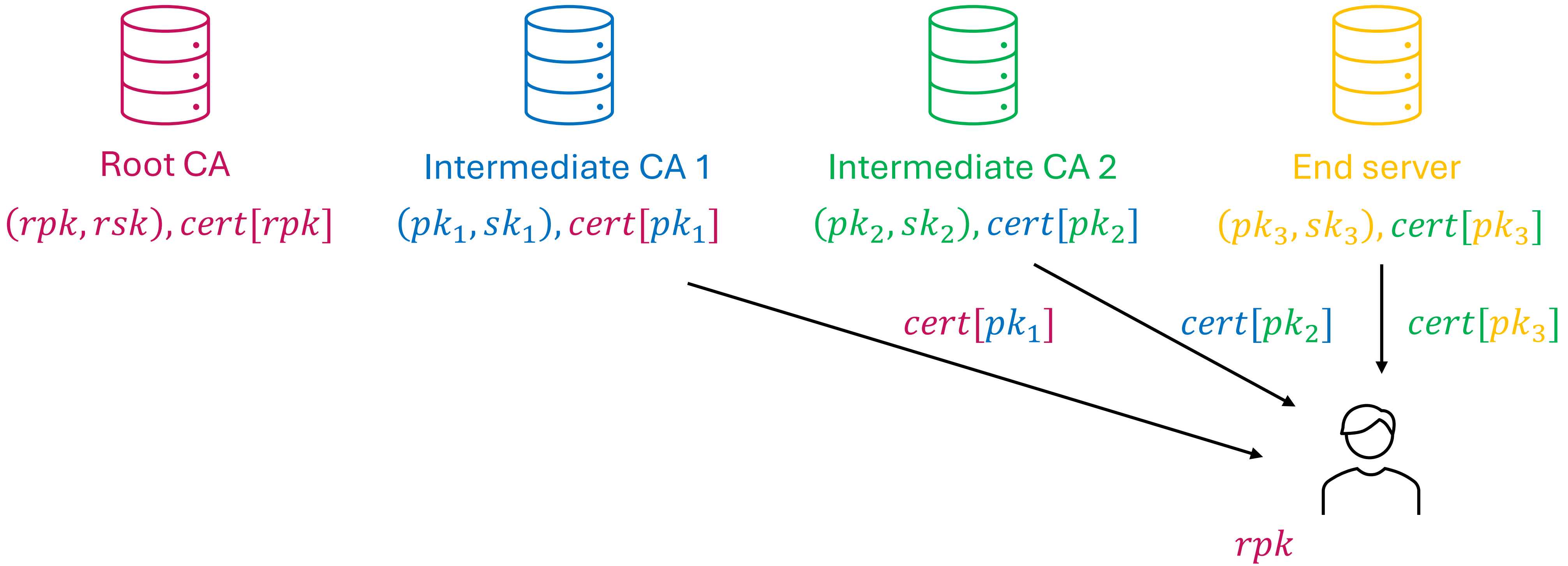
$cert[pk_3]$

…"this certificate is signed by CA 2''…

$rpk$

# Digital Certificate

- Root Certificate and Certificate Chains



Root CA

$(rpk, rsk), cert[rpk]$

Intermediate CA 1

$(pk_1, sk_1), cert[pk_1]$

Intermediate CA 2

$(pk_2, sk_2), cert[pk_2]$

End server

$(pk_3, sk_3), cert[pk_3]$

$cert[pk_1]$

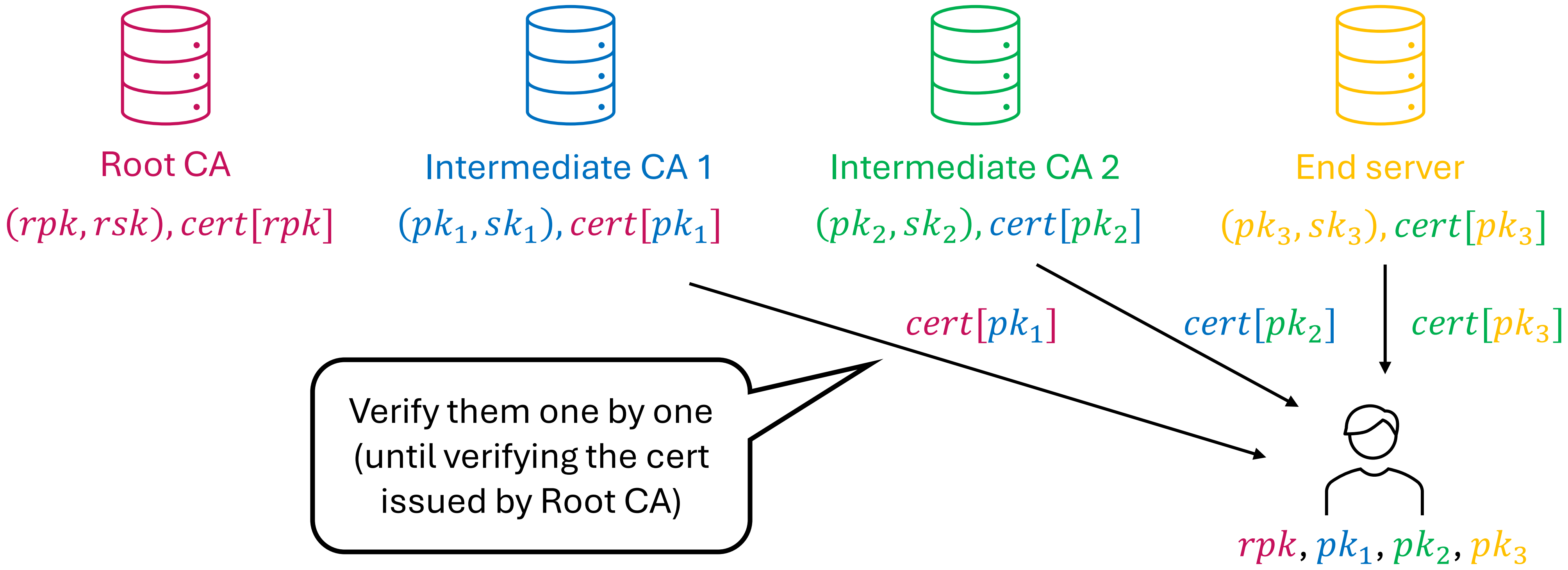
$cert[pk_2]$

$cert[pk_3]$

$rpk$

# Digital Certificate

- Root Certificate and Certificate Chains



Root CA

$(rpk, rsk), cert[rpk]$

Intermediate CA 1

$(pk_1, sk_1), cert[pk_1]$

Intermediate CA 2

$(pk_2, sk_2), cert[pk_2]$

End server

$(pk_3, sk_3), cert[pk_3]$

$cert[pk_1]$

$cert[pk_2]$

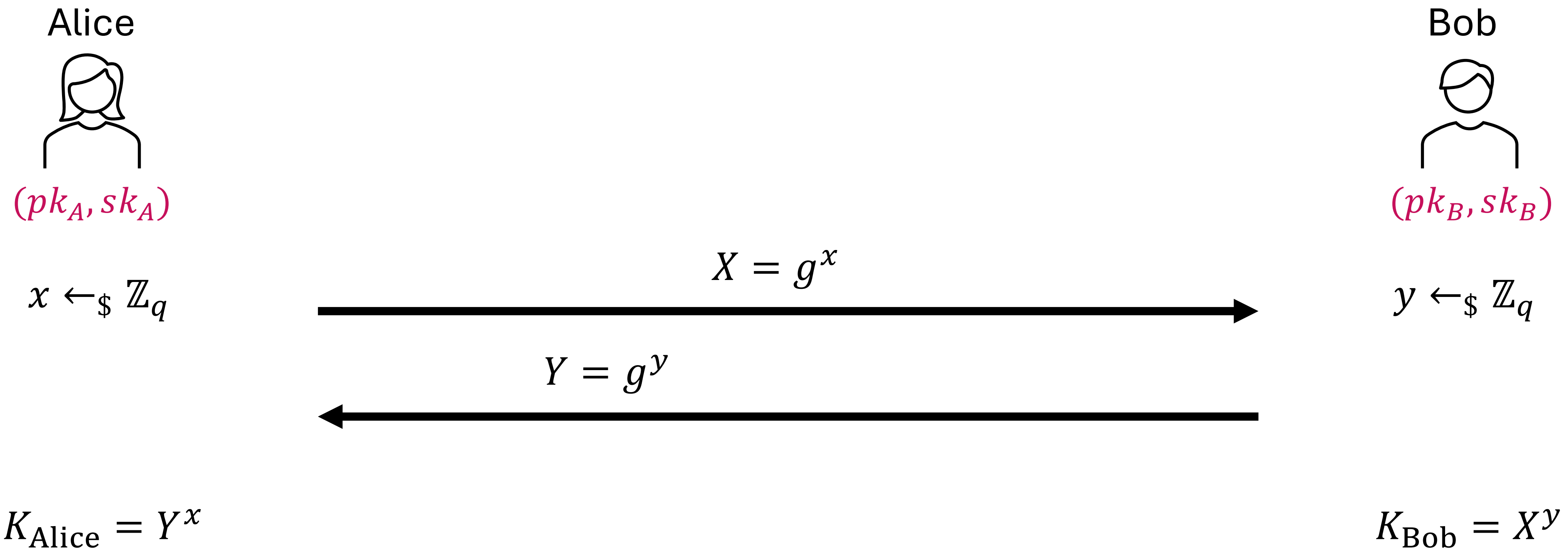
$cert[pk_3]$

Verify them one by one (until verifying the cert issued by Root CA)

$rpk, pk_1, pk_2, pk_3$

# Signed DH Key Exchange (Next Lecture)

- Use signature to avoid MitM attacks on DHKE:



Alice

$(pk_A, sk_A)$

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

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

Bob

$(pk_B, sk_B)$

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

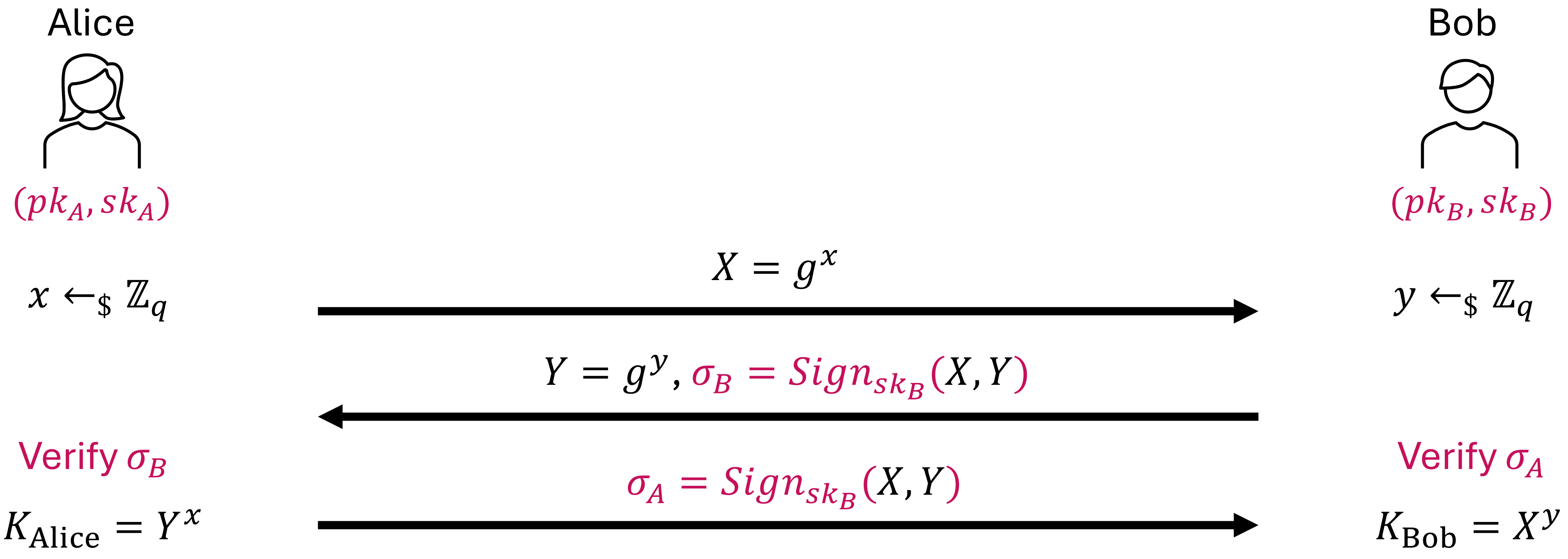
$K_{\text{Bob}} = X^y$

$X = g^x$

$Y = g^y$

# Signed DH Key Exchange (Next Lecture)

- Use signature to avoid MitM attacks on DHKE:



# Coding Tasks

1. Export a certificate from a website, and then use the example code *ReadCert.py* to read the certificate.
2. Find and export a pre-installed certificate on your laptop or PC (via browser), and use the example code to read the certificate.

# Homework

- Implement a man-in-the-middle attack (in one program) on DHKE.
- Use the example code ‘ECDSA.py’ to demonstrate the nonce-reuse attack on ECDSA (i.e., recover the secret key given two valid signatures using the same randomness)
- **Bonus**: Implement a man-in-the-middle attack on DHKE using sockets.
- **Bonus**: Use a trusted server and signatures to securely exchange public keys (using sockets): See next slide.

# Homework

1. Alice and Bob each have the **server's public key pre-installed**, which they will use to verify the server's digital signatures.
2. To initiate the key exchange, **Alice first requests the server to generate a digital signature** for her public key.
3. After receiving the signed public key from the server, **Alice sends her public key and the server's signature to Bob.**
4. **Bob, upon receiving (pk_alice, signature of pk_alice),** verifies the signature with the server's public key. If the signature is valid, Bob accepts pk_alice. **Next, Bob requests a signature for his own public key from the server**, following a similar process as Alice.
5. **Finally, Bob sends (pk_bob, signature of pk_bob) to Alice.** Alice verifies the signature using the server's public key and, if valid, accepts pk_bob.

# Further Reading

- DigiCert (one of the largest and most widely trusted CAs): https://www.digicert.com/
- Elliptic Curves: https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/
- P-256 (secp256r1) curve: https://neuromancer.sk/std/nist/P-256
- The X.509 standard: https://en.wikipedia.org/wiki/X.509
- Public Key Infrastructure (PKI): https://en.wikipedia.org/wiki/Public_key_infrastructure