

# Lec 12: Digital Signatures and Certificates

CSED415: Computer Security  
Spring 2026

Seulbae Kim

**POSTECH**  
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Administrivia

- Lab 03 is due April 6
- Midterm exam:
  - Monday, April 13 / 15:30 – 16:45 / in the regular classroom
  - 2025 midterm exam questions and solutions have been uploaded!

## 8Week [13 April - 19 April]



Spring 2025 Midterm + Solutions [307.3KB PDF document](#)

This year's midterm exam will follow the same format as the Spring 2025 exam. Please refer to the attached file for details.

# Cryptography roadmap

Goal \ Scheme	Symmetric Key	Asymmetric Key
<b>Confidentiality</b>	<ul style="list-style-type: none"> <li>✓ One Time Pad (OTP)</li> <li>✓ Block ciphers (DES, AES)</li> <li>✓ Stream ciphers</li> </ul>	<ul style="list-style-type: none"> <li>✓ DH secure key exchange</li> <li>✓ ElGamal encryption</li> <li>✓ RSA encryption</li> </ul>
<b>Integrity &amp; Authentication</b>	<ul style="list-style-type: none"> <li>✓ Message Authentication Code (MAC)</li> </ul>	<ul style="list-style-type: none"> <li>• Digital signature</li> </ul>
<b>CIA at the same time</b>	<ul style="list-style-type: none"> <li>✓ Authenticated encryption</li> </ul>	

## Additional Tool

- ✓ Hash functions

# Digital Signatures

# Missing integrity and authenticity

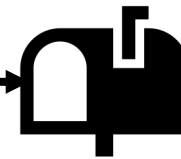
- Asymmetric encryption (like symmetric encryption) only provides confidentiality, not integrity
    - Message Authentication Code (MAC) solves the integrity problem in the symmetric-key setting
    - However, MAC has the same key sharing problem
      - Secret key needs to be shared between Alice and Bob
- Question: Can we use asymmetric cryptography to provide integrity and authenticity of messages?

# Authenticity in real world

- Anonymous document



*Party tonight at  
my place.  
Come for  
unlimited *

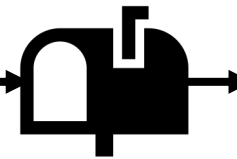


# Authenticity in real world

- Anonymous document



*Party tonight at  
my place.  
Come for  
unlimited* 

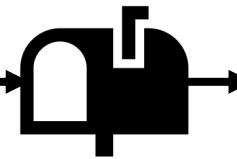


# Authenticity in real world

- **Signed** (stamped) document



*Party tonight at  
my place.  
Come for  
unlimited*   



# Digital signatures

- Key idea:
  - In asymmetric cryptography, each user has a secret key  $k_s$  and a public key  $k_p$
  - Only the owner of the secret key  $k_s$  can **sign** a message with the secret key
  - **Everyone else** can **verify** the signature using the corresponding public key  $k_p$

# Digital signatures

- Method:
  - Given: A key pair  $\langle k_p, k_s \rangle$ 
    - $k_p$ : public key
    - $k_s$ : secret key (a.k.a. signing key or private key)
  - $S(k_s, m)$ : Sign  $m$  using secret key  $k_s$  to generate signature  $\sigma$
  - $V(k_p, m, \sigma)$ : Verify signature  $\sigma$  of message  $m$  using public key  $k_p$

# Difference in key usage

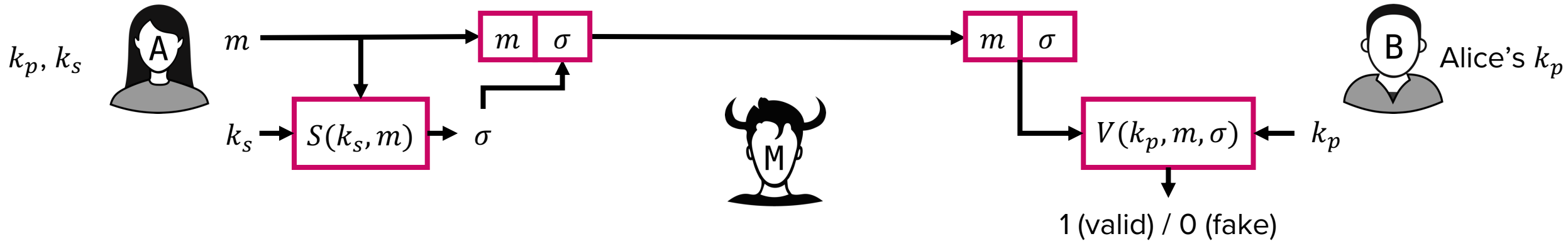
- Note: Digital signatures use the asymmetric key pair in the **opposite** order of asymmetric encryption schemes
  - Asymmetric encryption:
    - Alice (sender) encrypts using Bob's (receiver's) public key  $k_p$
    - Bob (receiver) decrypts using his (receiver's) secret key  $k_s$
  - Digital signature:
    - Alice (sender) signs using her (sender's) secret key  $k_s$
    - Bob or anyone (receiver) verifies using Alice's (sender's) public key  $k_p$

# MAC vs Digital signature

- When using MAC (symmetric cryptography):
  - The sender must share a secret (key  $k$ ) with the verifier
  - Consequently, the verifier could potentially impersonate the sender using the shared key!
    - e.g., by forging MAC tags using the shared key
- When using digital signatures (asymmetric cryptography):
  - The verifier utilizes the sender's public key
    - Verification does not require any shared secret
  - Consequently, the verifier cannot impersonate the sender!
    - Only the holder of the secret key (the sender) can generate valid signatures

# Security of DS

- DS scheme



- Metric to evaluate security (Same as MAC's)
  - **Unforgeability:** No polynomial time adversary should be able to produce forgery (i.e.,  $m$  and sig  $\sigma$ , where  $m$  was never queried to  $S$ ) with non-negligible probability, even after seeing multiple legitimate  $\langle m, \sigma \rangle$  pairs

# Naïve DS: Sign $m$

- Let's utilize the Vanilla RSA encryption for building  $S$  and  $V$ 
  - Recall RSA:
    - Select two large primes  $p$  and  $q$ .  $N = pq$
    - Compute totient  $T = (p - 1)(q - 1)$
    - Select  $k_p$ , which is coprime to  $T$  //  $k_p = e$  (notation we used in *Lecture 10*)
    - Compute  $k_s = k_p^{-1} \bmod T$  //  $k_s = d$  (notation we used in *Lecture 10*)
    - Ciphertext  $c = E(k_p, m) = m^{k_p} \bmod N$
    - Decrypted  $m = D(k_s, c) = c^{k_s} \bmod N$

# Naïve DS: Sign $m$

- Let's utilize the Vanilla RSA encryption for building  $S$  and  $V$ 
  - Given: Message  $m$ , secret key  $k_s$ , public  $k_p$ , public parameter  $N$
  - Alice signs  $m$ . Signature  $\sigma = S(k_s, m) = m^{k_s} \bmod N$
  - Alice sends  $m$  with  $\sigma$  to Bob
  - Bob verifies the signature  $\sigma$  via  $V(k_p, \sigma, m)$ :
    - $m' = \sigma^{k_p} \bmod N$  // message retrieved by decrypting  $\sigma$
    - If  $m == m'$  then return 1, else return 0

Security (unforgeability)?

# Naïve DS: Sign $m$

- Unforgeability of naïve DS
  - Q) Can an attacker forge a valid pair  $\langle m, \sigma \rangle$  without knowing  $k_s$ ?

A) Yes! Any attacker can forge  $\langle m = 1, \sigma = 1 \rangle$

Verification:

(1)  $m' = \sigma^{k_p} \bmod N = 1^{k_p} \bmod N = 1.$

(2)  $m == m'$  holds. Return true

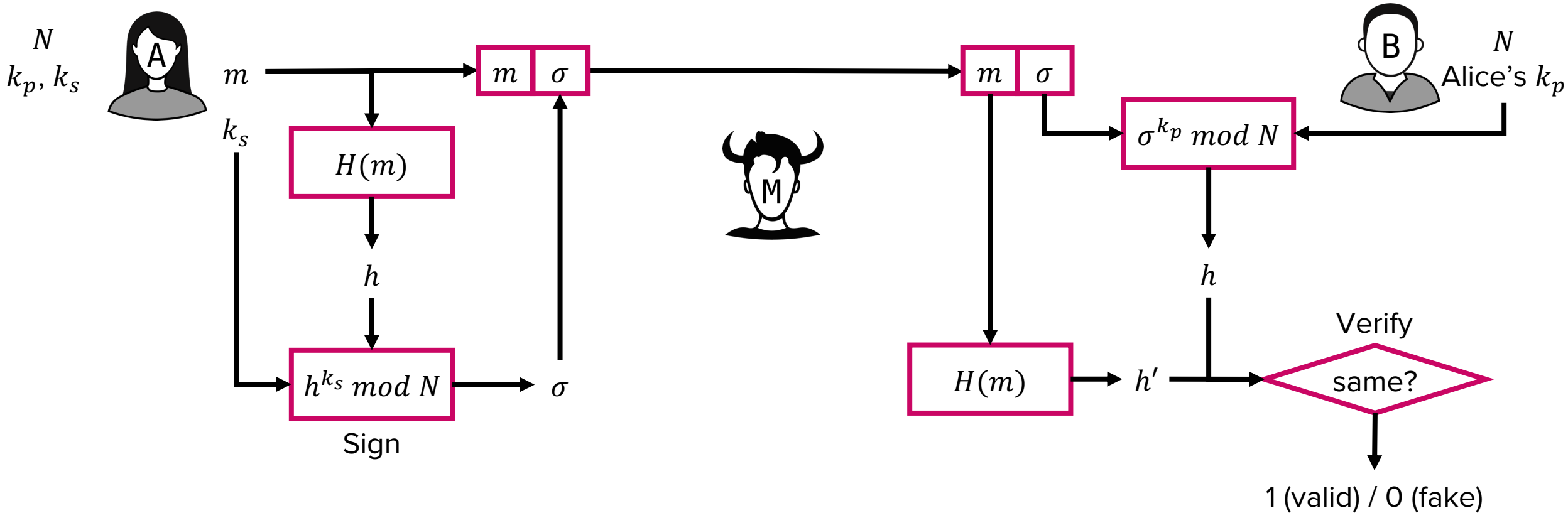
→ Unforgeability is broken



# Secure DS: Sign $H(m)$

- A better scheme: Hash the message and then sign it
    - Given: Message  $m$ , secret key  $k_s$ , public  $k_p$ , public parameter  $N$
    - $h \leftarrow H(m)$
    - Sign  $S(k_s, h)$ :  $\sigma \leftarrow h^{k_s} \bmod N$
    - Send  $m$  and  $\sigma$
    - Verify  $V(k_p, \sigma, m)$ :
      - $h \leftarrow H(m)$  // compute the hash of the received message  $m$
      - $h' \leftarrow \sigma^{k_p} \bmod N$  // hash retrieved by decrypting  $\sigma$
      - if  $h = h'$  then return 1, else return 0
- The previous forgery using  $(m = 1, \sigma = 1)$  no longer works

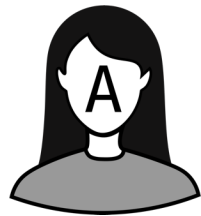
# Summary: Digital signature using hash and RSA



We can now provide integrity using an asymmetric scheme!

# Digital signature in practice

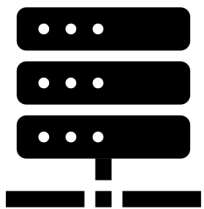
- Passwordless authentication of SSH (secure shell)



**Alice**

## Initial setup for account “Alice”

1. Alice logs in using her account password
2. Alice registers her public key in `/home/Alice/.ssh/authorized_keys`
3. She can then disable password login in the SSH configuration file



**Server**

## Passwordless login

1. Alice attempts to log in
2. She signs her identity with her secret key and sends it to the server
3. The server verifies the signature using the stored public key of Alice
4. If valid, Alice is granted access (no password required)

**Only Alice can securely log in as long as her secret key is not leaked**

# Rethinking the “authentication” problem

- Pizza prank scenario

- Mallory creates an order message:
- Mallory **signs** the order with **his own secret key**
- Mallory sends the signed order to the Pizza Store
- The store responds, “Hey **Bob**, please send us your public key”
- Mallory sends **his public key**
- The store verifies the signature using Mallory’s public key, incorrectly assuming the key is Bob’s public key
- The store delivers four pepperoni pizzas to Bob
- Bob is vegan

```
Dear Pizza Store,  
I'd like to order four pepperoni pizzas.  
Thank you,  
- Bob
```

# Rethinking the “authentication” problem

- Pizza prank scenario

- Mallory creates an order message:
- Mallory **signs** the order with **his own secret key**

```
Dear Pizza Store,  
I'd like to order four pepperoni pizzas.  
Thank you,  
- Bob
```

**Key question: Are public keys alone enough for strong authentication?**

The store verifies the signature using Mallory's public key, incorrectly assuming the key is Bob's public key

- The store delivers four pepperoni pizzas to Bob
- Bob is vegan

# Cryptography roadmap

Goal \ Scheme	Symmetric Key	Asymmetric Key
<b>Confidentiality</b>	<ul style="list-style-type: none"> <li>✓ One Time Pad (OTP)</li> <li>✓ Block ciphers (DES, AES)</li> <li>✓ Stream ciphers</li> </ul>	<ul style="list-style-type: none"> <li>✓ DH secure key exchange</li> <li>✓ ElGamal encryption</li> <li>✓ RSA encryption</li> </ul>
<b>Integrity &amp; Authentication</b>	<ul style="list-style-type: none"> <li>✓ Message Authentication Code (MAC)</li> </ul>	<ul style="list-style-type: none"> <li>• Digital signature</li> </ul>
<b>CIA at the same time</b>	<ul style="list-style-type: none"> <li>✓ Authenticated encryption</li> </ul>	

Authentication ←

Really? —

## Additional Tool

- ✓ Hash functions

# Certification Authorities

# Problem: Distributing public keys



Hey Bob, I want to talk to you.  
Give me your public key.

Generate public-secret key pair  
 $\langle k_p^B, k_s^B \rangle$

Generate public-secret key pair  
 $\langle k_p^M, k_s^M \rangle$

Send  $k_p^B$

Store  $k_p^B$

Send  $k_p^M$

Receive  $k_p^M$   
(Thinks it's Bob's public key)

Encrypt  $m$  using  $k_p^M$   
Send  $c = m^{k_p^M} \bmod N$

Decrypt  $c$  using  $k_s^M$   
Alter  $m$  to  $m'$

Encrypt  $m'$  using  $k_p^B$

Decrypt  $c'$  using  $k_s^B$   
Sees  $m' =$  "Hate you, Bob" again..



# Problem: Distributing public keys



Hey Bob, I want to talk to you.

Give me your public key.

Man-in-the-Middle (MitM) attack becomes possible  
by merely replacing the **public key!**

Alter  $m$  to  $m'$

Encrypt  $m'$  using  $k_p^B$

Decrypt  $c'$  using  $k_s^B$

Sees  $m' =$  "Hate you, Bob" again..

# Problem: Distributing public keys

- Countermeasure idea
  - Bob can sign his public key to prevent tampering?
- Dilemma:
  - For verification, we require his **public key**
  - Yet, the purpose was to verify Bob's **public key** in the first place
  - Creates a circular problem!
    - Alice cannot fully trust any public key

We need a “root of trust”!

# Establishing root of trust: Trust-on-first-use (TOFU)

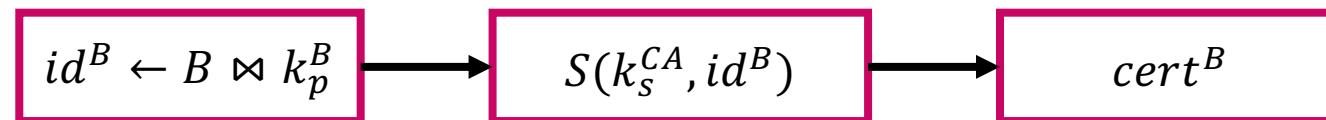
- TOFU policy: Trust the  $k_p$  used for the initial communication and warn the user if the key changes in the future
  - Rationale: Attacks are not frequent, so assume that the initial communication was not attacked
  - Used by SSH (Secure Shell)
    - Connect to a new server from my machine
    - Server's identification is saved on my machine (in `~/.ssh/known_hosts`)
    - If the server sends a different identification, we can suspect an MitM attack

```
ssh root@65.109.131.51
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@   WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!   @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ED25519 key sent by the remote host is
SHA256:qT2ctfBupHj1k0yC2WcJJpQ76ge58iyiBf9sPDQtkdE.
Please contact your system administrator.
Add correct host key in /Users/ask/.ssh/known_hosts to get rid of this messag
e.
Offending ECDSA key in /Users/ask/.ssh/known_hosts:12
Host key for 65.109.131.51 has changed and you have requested strict checking
.
Host key verification failed.
```

Practical, yet the assumption is too strong

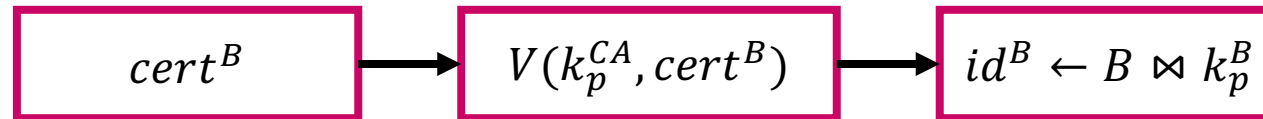
# Establishing root of trust: Certification Authority

- Certification Authority (CA) binds a public key to a specific entity (E)
  - Serves as a trusted third party (TTP)
- Procedure
  - Bob registers his public key to a CA. Bob provides a proof of identity
  - The CA creates an identity binding ( $\bowtie$ ) of Bob and his public key
  - The binding, digitally signed by the CA's secret key, serves as the certificate of Bob ( $cert^B$ )



# Certification Authority (CA)

- Now when Alice wants Bob's public key
  - Alice receives Bob's certificate ( $cert^B$ ) from the CA
  - Alice applies the CA's public key to verify Bob's identity



- If Alice trusts the CA (root of trust), Alice can trust that Bob's public key is truly Bob's

# Building a practical CA

- Naïve idea: Make a central, trusted directory (TD) from which you can fetch anyone's public key
  - The TD has a public/secret key pair:  $k_p^{TD}$  and  $k_s^{TD}$
  - The directory publishes  $k_p^{TD}$  to everyone
  - When someone requests Bob's public key, the directory sends a certificate for Bob's identity
    - $cert^B$ , which is  $B \bowtie k_p^B$  signed using  $k_s^{TD}$
  - If you trust the TD, you trust every public key

# Building a practical CA

- Naïve idea: Make a central, trusted directory (TD) from which you can fetch anyone's public key
- Problems
  - **Scalability:** One directory will not have enough computing power to serve all entities in the entire world
  - **Single point of failure:**
    - If the TD fails, every service depending on TD becomes unavailable
    - If the TD is compromised, you cannot trust anyone
    - If the TD is compromised, it is extremely difficult to recover

# Building a practical CA

- Practical idea #1: Hierarchical trust model
  - The roots of trust may **delegate** the identity bindings and signing power to other authorities
    - Alice's public key is  $k_p^A$  and I trust her to sign for POSTECH
    - Bob's public key is  $k_p^B$  and I trust him to sign for the CSE department
    - Charlie's public key is  $k_p^C$ . I don't let him sign for anyone else
  - Hierarchy
    - Root CA
    - Alice and Bob are intermediate CAs

Solves the scalability problem



# Building a practical CA

- Practical idea #2: Multiple trust anchors
  - There are more than 200 root CAs in the world
  - Most operating systems provide a built-in list of trusted root CAs
  - Most web browsers, too

Certificate name	Issued by	Type	Key size	Sig alg	Serial number	Expires	EV policy	Fingerprint (SHA-256)
AAA Certificate Services	AAA Certificate Services	RSA	2048 bits	SHA-1	01	23:59:59 Dec 31, 2028	Not EV	D7 A7 A0 FB 5D 7E 27 31 D7 71 E9 48 4E BC DE F7 1D 5F 0C 3E 0A 29 48 78 2B C8 3E E0 EA 69 9E F4
AC RAIZ FNMT-RCM	AC RAIZ FNMT-RCM	RSA	4096 bits	SHA-256	5D 93 8D 30 67 36 C8 06 1D 1A C7 54 84 69 07	00:00:00 Jan 1, 2030	Not EV	EB C5 57 0C 29 01 8C 4D 67 B1 AA 12 7B AF 12 F7 03 B4 61 1E BC 17 B7 DA B5 57 38 94 17 9B 93 FA
ACCVRAIZ1	ACCVRAIZ1	RSA	4096 bits	SHA-1	5E C3 B7 A6 43 7F A4 E0	09:37:37 Dec 31, 2030	Not EV	9A 6E C0 12 E1 A7 DA 9D BE 34 19 4D 47 8A D7 C0 DB 18 22 FB 07 1D F1 29 81 49 6E D1 04 38 41 13

Solves the single-point-of-failure problem

# Building a practical CA

- New problem: Revocation
  - What if a CA is compromised and issues forged certificates?
    - e.g., CA: “Bob’s public key is  $k_p^M$ ”
  - Everyone that trust the CA will trust forged public keys
    - e.g., if Mallory signs messages, people will think Bob did

We need to be able to revoke bad certificates!

# Building a practical CA – Revocation

- Approach #1: Expiration
  - A certificate expires after X days
    - e.g., Certificates issued by Let's Encrypt expire in 90 days
- Strength:
  - Can provide a hard upper bound on trust
    - Forged certificates will eventually expire
- Weakness: Renewal overhead
  - Frequent renewal: More security, less usability
  - Infrequent renewal: Less security, more usability

# Building a practical CA – Revocation

- Approach #2: Certificate Revocation List (CRL)
  - CA periodically releases a list of invalidated certificates (i.e., CRL)
  - Clients download updated CRLs before key validation
- Strength:
  - Invalid certificates are revoked as soon as a user downloads the CRL
- Weakness:
  - Size of the list grows linearly to the number of revoked certificates
    - Can be addressed through “delta CRL” (publishing incremental updates)
  - Revocation latency window exists
    - Revoked certificates may still be accepted until the next CRL update

# Current certificate standard: X.509

- Certificate contains
  - Issuer (name of CA)
  - Validity (Not Before, Not After)
  - Subject (entity, e.g., google.com)
  - Subject's public key
  - Certificate signature algorithm (e.g., SHA-256 with RSA Encryption)
  - Certificate signature value (signed with the issuer's secret key)

# Summary

---

- Certificate: A signed attestation of identity
- Trusted directory: One server holds all keys
- Certificate authorities: Provide delegated trust from a pool of multiple root CAs
  - Root CA can sign certificates for intermediate CAs
  - Certificates can be revoked (timed expiry or revocation list)

# Cryptography roadmap

Goal \ Scheme	Symmetric Key	Asymmetric Key
<b>Confidentiality</b>	<ul style="list-style-type: none"> <li>✓ One Time Pad (OTP)</li> <li>✓ Block ciphers (DES, AES)</li> <li>✓ Stream ciphers</li> </ul>	<ul style="list-style-type: none"> <li>✓ DH secure key exchange</li> <li>✓ ElGamal encryption</li> <li>✓ RSA encryption</li> </ul>
<b>Integrity &amp; Authentication</b>	<ul style="list-style-type: none"> <li>✓ Message Authentication Code (MAC)</li> </ul>	<ul style="list-style-type: none"> <li>✓ Digital signature + CA</li> </ul>
<b>CIA at the same time</b>	<ul style="list-style-type: none"> <li>✓ Authenticated encryption</li> </ul>	<ul style="list-style-type: none"> <li>• ???</li> </ul>

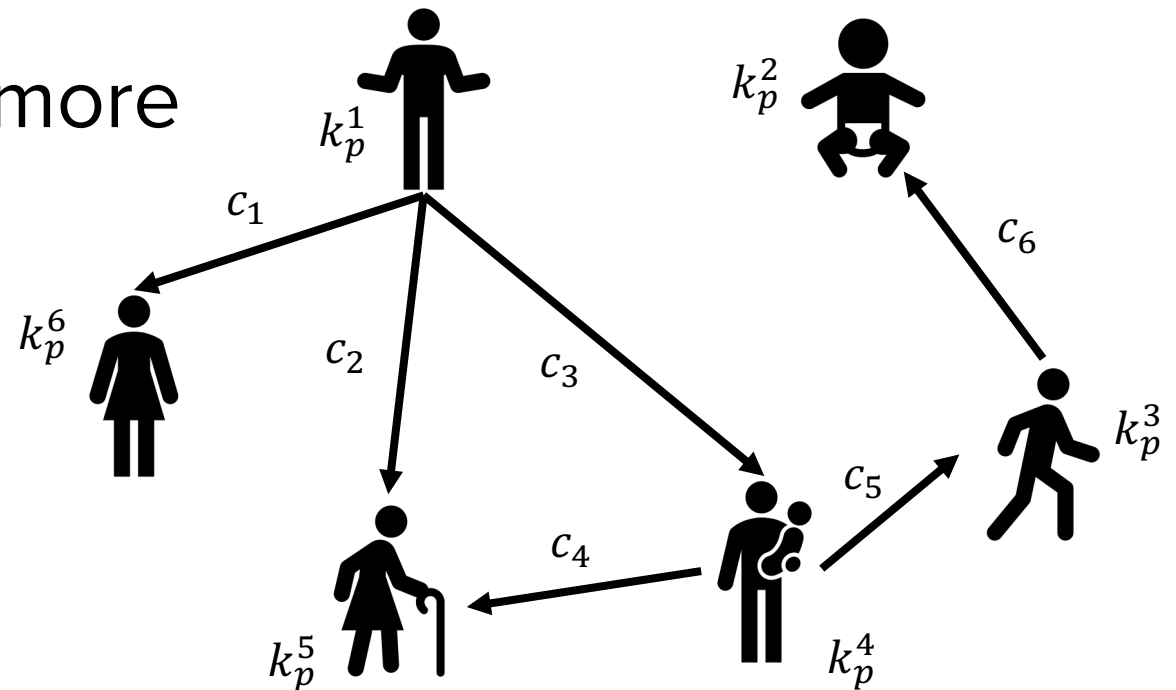
**Additional Tool**  
 ✓ Hash functions

# Multi-user Setting and Signcryption



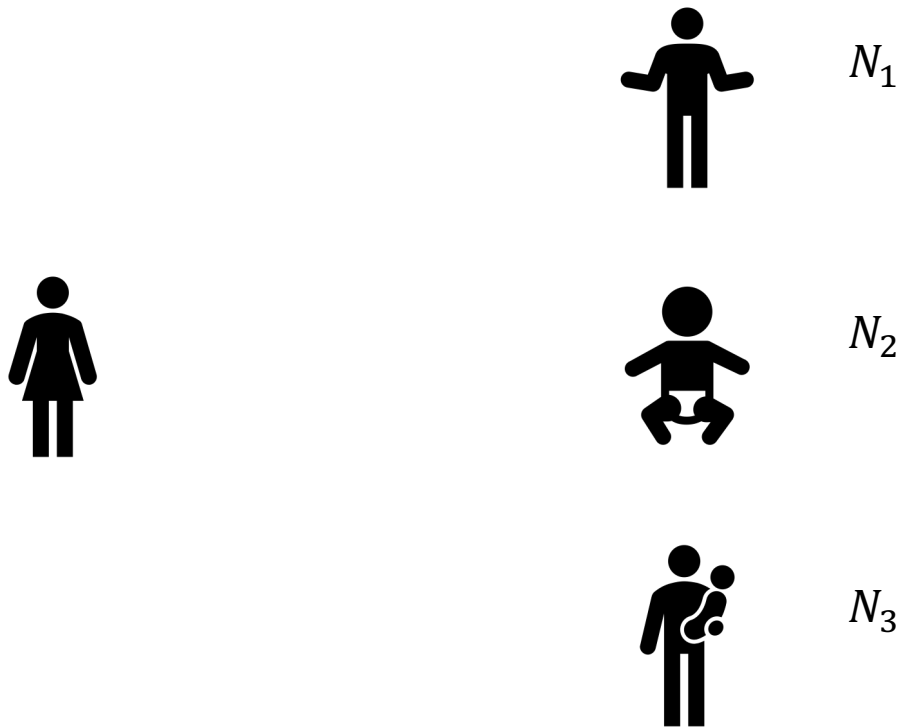
# Multi-user setting complications

- Security of asymmetric schemes considered a single user
  - “Can a sender have confidentiality?”
  - “Can a receiver verify a signature?”
- Real-world scenarios are much more complex



# Multi-user setting complications

- Known real-world attack: Hastad-type attack on RSA



Three people select different large numbers  $N_1, N_2, N_3$  for RSA key generation

# Multi-user setting complications

- Known real-world attack: Hastad-type attack on RSA



$$\begin{matrix} N_1 \\ k_p^1 = 3 \end{matrix}$$



$$\begin{matrix} N_2 \\ k_p^2 = 3 \end{matrix}$$



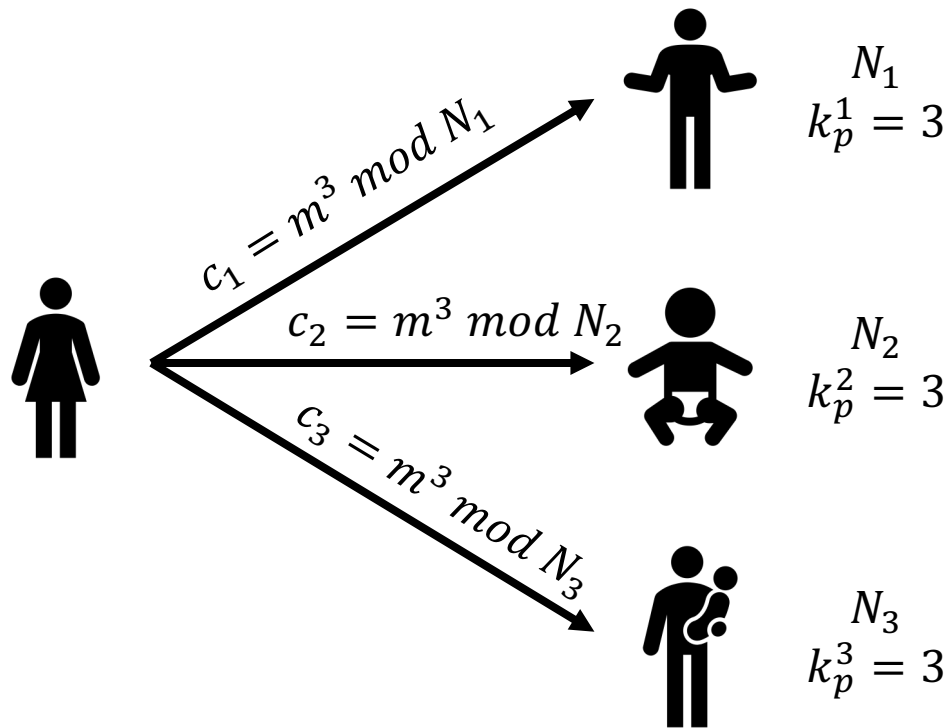
$$\begin{matrix} N_3 \\ k_p^3 = 3 \end{matrix}$$

Three people happen to select the same public key  $k_p^i$  that are coprime to the totient of each  $N$  (Recall, if  $N = pq$ , its totient  $T = (p - 1)(q - 1)$ )

e.g.,  $k_p^1 = k_p^2 = k_p^3 = 3$

# Multi-user setting complications

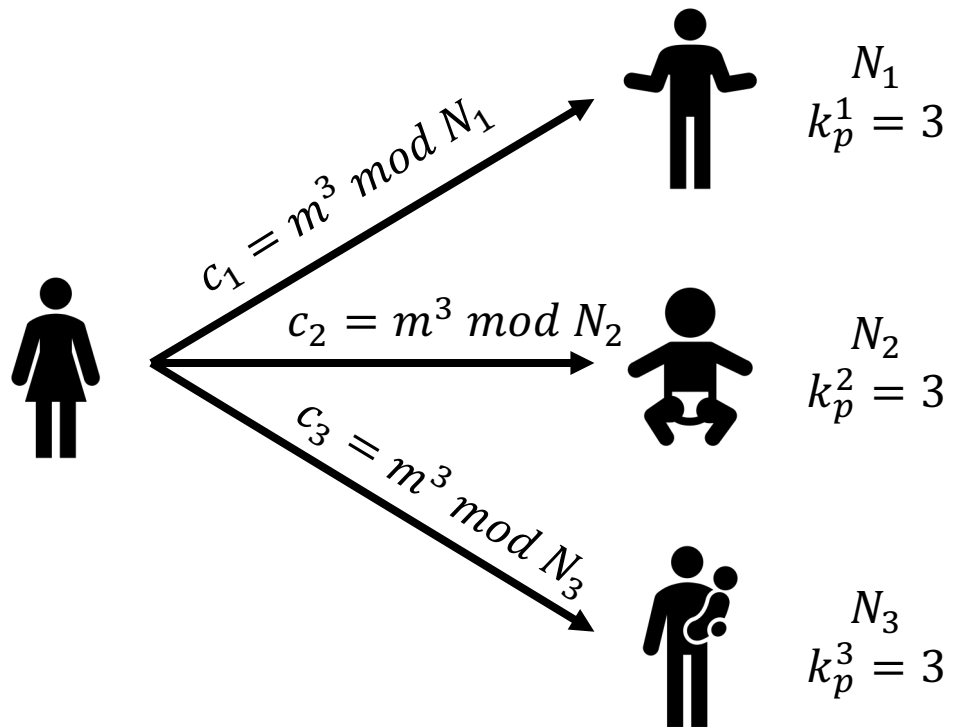
- Known real-world attack: Hastad-type attack on RSA



The sender wants to send  $m$  and RSA-encrypts it using  $N_i, k_p^i$  for each recipient

# Multi-user setting complications

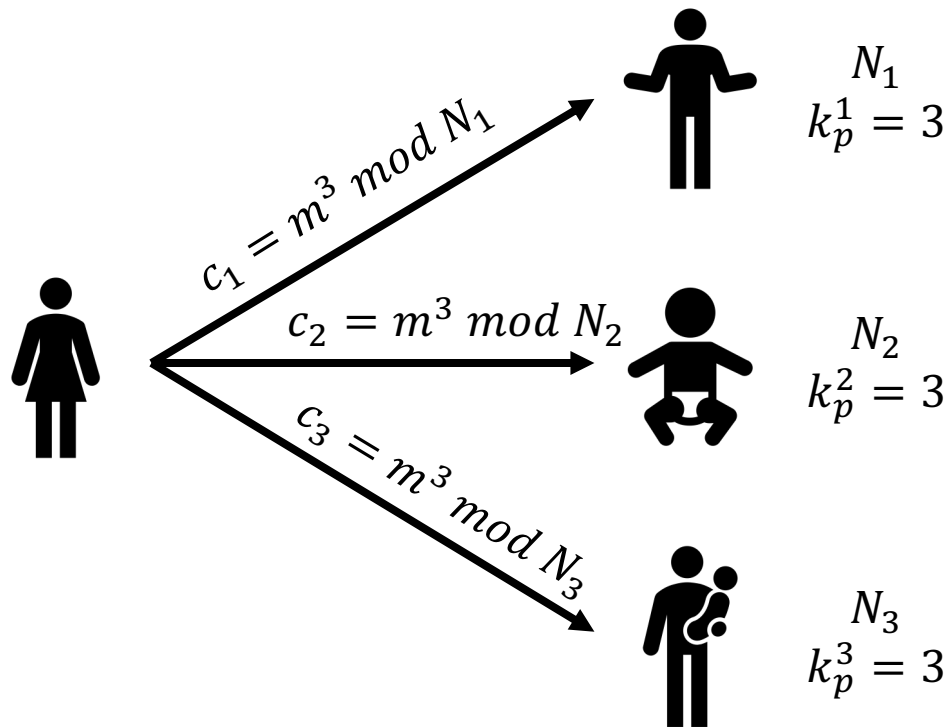
- Known real-world attack: Hastad-type attack on RSA



Only the three recipients, individually, should be able to decrypt  $m$  from  $c_i$  using their  $k_s^i$

# Multi-user setting complications

- Known real-world attack: Hastad-type attack on RSA



If  $N_1, N_2, N_3$  are relatively prime, then by the Chinese Remainder Theorem,

- $c_1 = m^3 \bmod N_1$
- $c_2 = m^3 \bmod N_2$
- $c_3 = m^3 \bmod N_3$

can be combined to find  $c$ :

$$c = m^3 \bmod N_1 N_2 N_3$$

Since  $m^3 < N_1 N_2 N_3$ , we get  $m = \sqrt[3]{c}$

$m$  can be completely recovered using public keys

# Signcryption

- Signcryption is a public key-based primitive that assures confidentiality, integrity, and authenticity at the same time
  - Not by separately utilizing encryption and digital signatures
  - Goal is to combine encryption and signing into a single operation
- e.g., sign-then-encrypt?
  - Signing involves an encryption (using a secret key)
  - Encrypting involves another encryption (using a public key)
  - Redundancy (== inefficiency)

# Signcryption

- Signcryption presents significant challenges:
  - Strong security should be provided:
    - Indistinguishability under chosen plaintext/ciphertext attacks
    - Unforgeability
  - Multi-user setting poses more challenges
    - e.g., Hastad-type attack
- As of now, no provably-secure algorithm has been developed



# Cryptography roadmap

Goal \ Scheme	Symmetric Key	Asymmetric Key
<b>Confidentiality</b>	<ul style="list-style-type: none"> <li>✓ One Time Pad (OTP)</li> <li>✓ Block ciphers (DES, AES)</li> <li>✓ Stream ciphers</li> </ul>	<ul style="list-style-type: none"> <li>✓ DH secure key exchange</li> <li>✓ ElGamal encryption</li> <li>✓ RSA encryption</li> </ul>
<b>Integrity &amp; Authentication</b>	<ul style="list-style-type: none"> <li>✓ Message Authentication Code (MAC)</li> </ul>	<ul style="list-style-type: none"> <li>✓ Digital signature + CA</li> </ul>
<b>CIA at the same time</b>	<ul style="list-style-type: none"> <li>✓ Authenticated encryption</li> </ul>	None

## Additional Tool

- ✓ Hash functions

# Coming up next

---

- Internet Security Protocols
  - Real-world applications that rely on cryptography

# Questions?