public-key cryptography

slides
bit.ly/cs161-disc

feedback
bit.ly/extended-feedback

~~hack~~ mitigation of the day

~~hack~~ mitigation of the day

- Google introduces client-side encryption to Gmail and Calendar

~~hack~~ mitigation of the day

- Google introduces client-side encryption to Gmail and Calendar
  - data encrypted on your computer before sent to Google (via HTTPS)

~~hack~~ mitigation of the day

- Google introduces client-side encryption to Gmail and Calendar
  - data encrypted on your computer before sent to Google (via HTTPS)
- “...customers have sole control over their keys

~~hack~~ mitigation of the day

- Google introduces client-side encryption to Gmail and Calendar
  - data encrypted on your computer before sent to Google (via HTTPS)
- “…customers have sole control over their keys

SERVICE	DATA THAT'S CLIENT-SIDE ENCRYPTED	DATA THAT'S *NOT* CLIENT-SIDE ENCRYPTED
Google Drive	• Files created with Google Docs Editors (documents, spreadsheets, presentations) • Uploaded files, like PDFs and Microsoft Office files	• File title • File metadata, such as owner, creator, and last-modified time • Drive labels (also called Drive metadata) • Linked content that's outside of Docs or Drive (for example, a YouTube video linked from a Google document) • User preferences, such as Docs header styles
Gmail	• Email body, including inline images • Attached files **Note:** Attaching client-side encrypted Drive files isn't yet supported	• Email header, including subject, timestamps, and recipients lists
Google Calendar	• Event description • Attached Drive files (if CSE for Drive is turned on) • Meet audio and video streams (if CSE for Meet is turned on)	Any content other than the event description, attachments, and Meet data, such as: • Event title • Event starting and ending times • Attendees list • Booked rooms • Join by phone numbers • Link for Meet
Google Meet	• Audio streams • Video streams (including screen sharing)	• Any data other than audio and video streams

general questions, concerns, etc.

public-key cryptography

public-key cryptography

- every individual has two keys

public-key cryptography

- every individual has two keys
  - public key: known to everybody

public-key cryptography

- every individual has two keys
  - public key: known to everybody
  - private key
    - corresponds to public key

public-key cryptography

- every individual has two keys
  - public key: known to everybody
  - private key
    - corresponds to public key
- no longer need to assume alice and bob share key

public-key cryptography

- every individual has two keys
  - public key: known to everybody
  - private key
    - corresponds to public key
- no longer need to assume alice and bob share key
- much slower than symmetric-key cryptography
  - relies on number theory calculations

el gamal

el gamal

- diffie-hellman, but with simultaneous encrypt

el gamal

- diffie-hellman, but with simultaneous encrypt
- Bob: private key b and public key $B = g^b \bmod p$

el gamal

- diffie-hellman, but with simultaneous encrypt
- Bob: private key b and public key $B = g^b \bmod p$
- Alice: random r and computes $R = g^r \bmod p$

el gamal

- diffie-hellman, but with simultaneous encrypt
- Bob: private key b and public key $B = g^b \bmod p$
- Alice: random r and computes $R = g^r \bmod p$
- Alice sends $C1 = R$, $C2 = M \times B^r \bmod p$

el gamal

- diffie-hellman, but with simultaneous encrypt
- Bob: private key b and public key $B = g^b \bmod p$
- Alice: random r and computes $R = g^r \bmod p$
- Alice sends $C1 = R$, $C2 = M \times B^r \bmod p$
- Bob: $C2 \times C1^{-b} = M \times B^r \times R^{-b} = M \times g^{br} \times g^{-br} = M \bmod p$

RSA - KeyGen()

RSA - KeyGen()

- random large primes p and q

RSA - KeyGen()

- random large primes p and q
- Compute $N = pq$

RSA - KeyGen()

- random large primes p and q
- Compute $N = pq$
- Choose e

RSA - KeyGen()

- random large primes p and q
- Compute $N = pq$
- Choose e
  - Requirement: e is relatively prime to $(p - 1)(q - 1)$

RSA - KeyGen()

- random large primes p and q
- Compute $N = pq$
- Choose e
  - Requirement: e is relatively prime to $(p - 1)(q - 1)$
  - Requirement: $2 < e < (p - 1)(q - 1)$

RSA - KeyGen()

- random large primes p and q
- Compute $N = pq$
- Choose e
  - Requirement: e is relatively prime to $(p - 1)(q - 1)$
  - Requirement: $2 < e < (p - 1)(q - 1)$
- Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
  - Extended Euclid's algorithm

RSA - KeyGen()

- random large primes p and q
- Compute $N = pq$
- Choose e
  - Requirement: e is relatively prime to $(p - 1)(q - 1)$
  - Requirement: $2 < e < (p - 1)(q - 1)$
- Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
  - Extended Euclid's algorithm
- Public key: N and e

RSA - KeyGen()

- random large primes p and q
- Compute $N = pq$
- Choose e
  - Requirement: e is relatively prime to $(p - 1)(q - 1)$
  - Requirement: $2 < e < (p - 1)(q - 1)$
- Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
  - Extended Euclid's algorithm
- Public key: N and e
- Private key: d

RSA - encrypt

Alice

has access to Bob's
public keys N, e

Bob

has private key d,
public keys N, e

RSA - encrypt

message

M

Alice

has access to Bob's
public keys N, e

Bob

has private key d,
public keys N, e

RSA - encrypt

message

M

encrypted message

$C = M^e \bmod N$

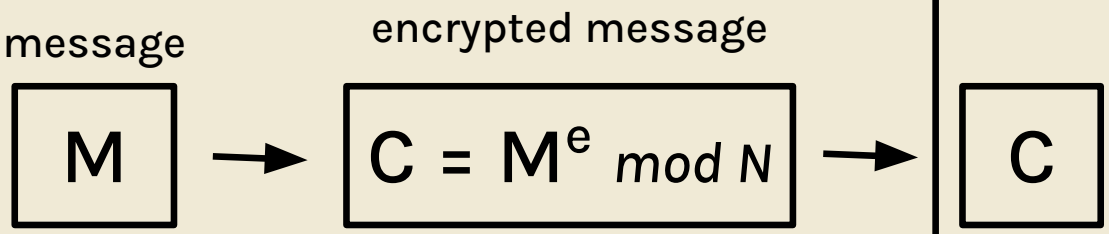
Alice

has access to Bob's
public keys N, e

Bob

has private key d,
public keys N, e

RSA - encrypt



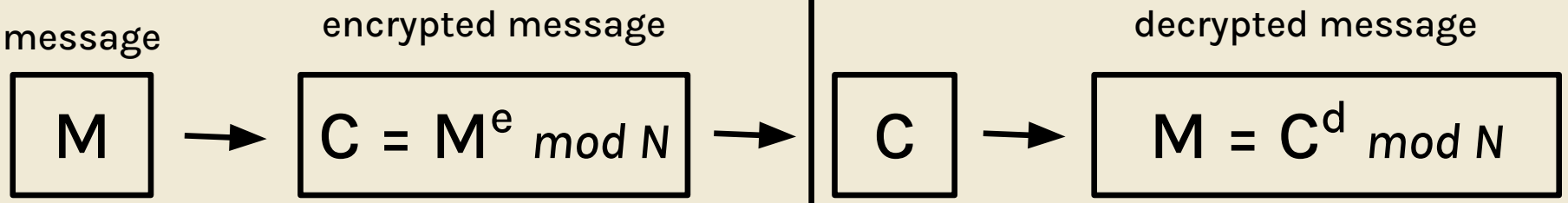
Alice

has access to Bob's
public keys N, e

Bob

has private key d,
public keys N, e

RSA - encrypt



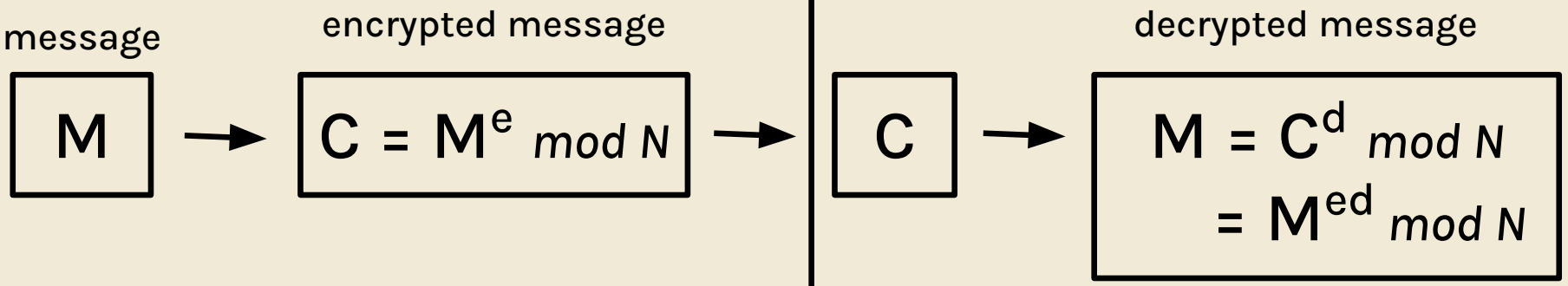
Alice

has access to Bob's
public keys N, e

Bob

has private key d,
public keys N, e

RSA - encrypt



Alice

has access to Bob's
public keys N, e

Bob

has private key d,
public keys N, e

public key’s downside

public key’s downside

- relies on accurate public keys given

public key’s downside

- relies on accurate public keys given
  - is public key cryptography secure against man-in-the-middle attacks?

public key's downside

- relies on accurate public keys given
  - is public key cryptography secure against man-in-the-middle attacks?
  - no! what if mallory gives alice her public key instead of bob's?

distributing public keys

distributing public keys

- bob can just sign his public key to ensure it’s his!

distributing public keys

- bob can just sign his public key to ensure it’s his!
  - problems with this?

distributing public keys

- bob can just sign his public key to ensure it’s his!
  - problems with this?
  - must have bob’s PK to verify signature

distributing public keys

- bob can just sign his public key to ensure it’s his!
  - problems with this?
  - must have bob’s PK to verify signature
- can’t gain trust without trusting anything

distributing public keys

- bob can just sign his public key to ensure it's his!
  - problems with this?
  - must have bob's PK to verify signature
- can't gain trust without trusting anything
- **trust anchor:** a root of trust—implicit trust

trust-on-first-use

trust-on-first-use

- trust the user’s PK on first communication

trust-on-first-use

- trust the user’s PK on first communication
- warn user if PK changes

trust-on-first-use

- trust the user's PK on first communication
- warn user if PK changes
- idea: attacks are infrequent, probably won't happen the first time

trust-on-first-use

- trust the user's PK on first communication
- warn user if PK changes
- idea: attacks are infrequent, probably won't happen the first time
- used by SSH

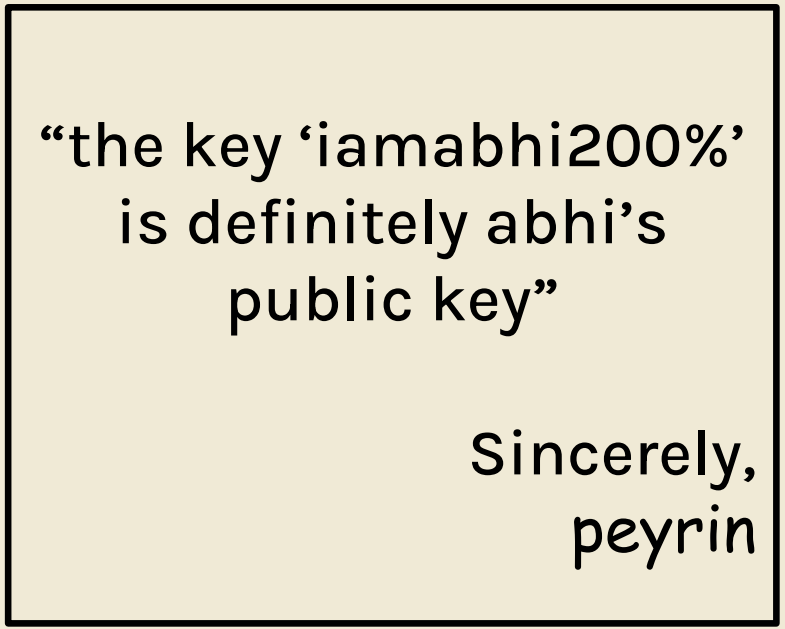
certificates

certificates

- signed endorsement of someone’s public key

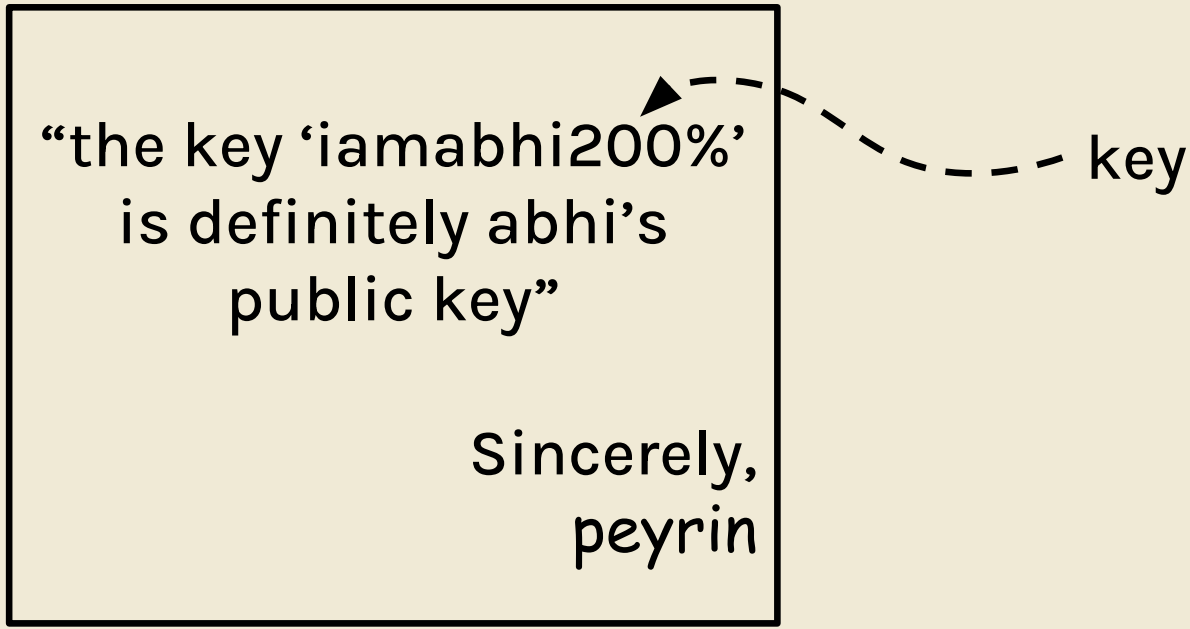
certificates

- signed endorsement of someone's public key



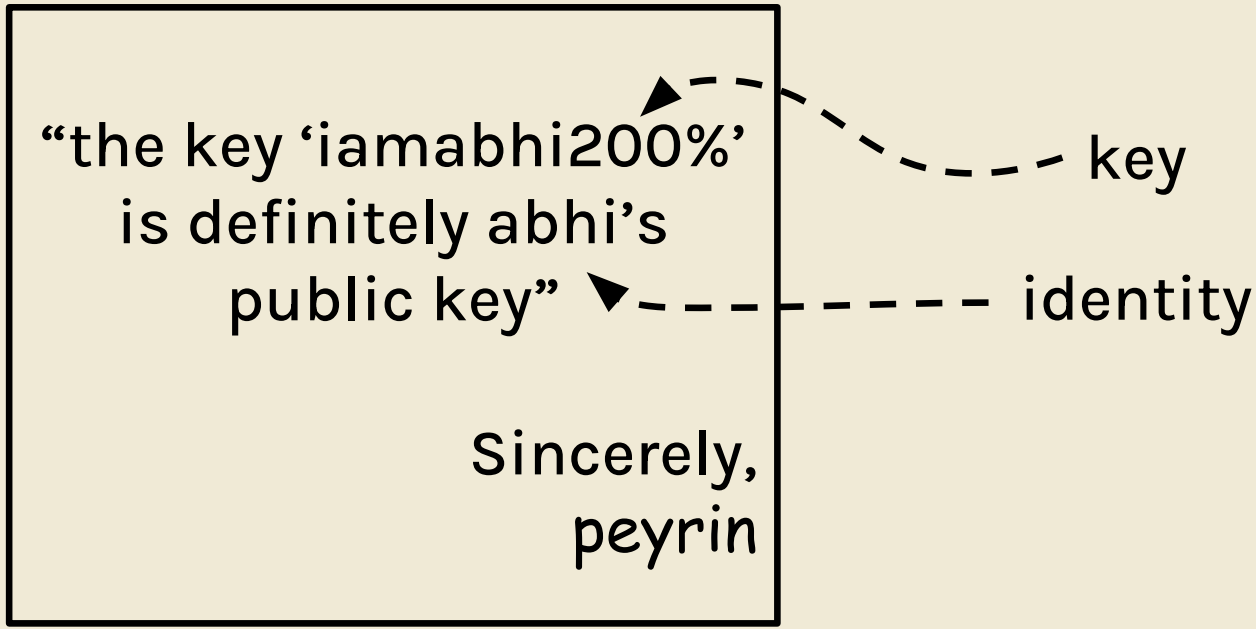
certificates

- signed endorsement of someone's public key



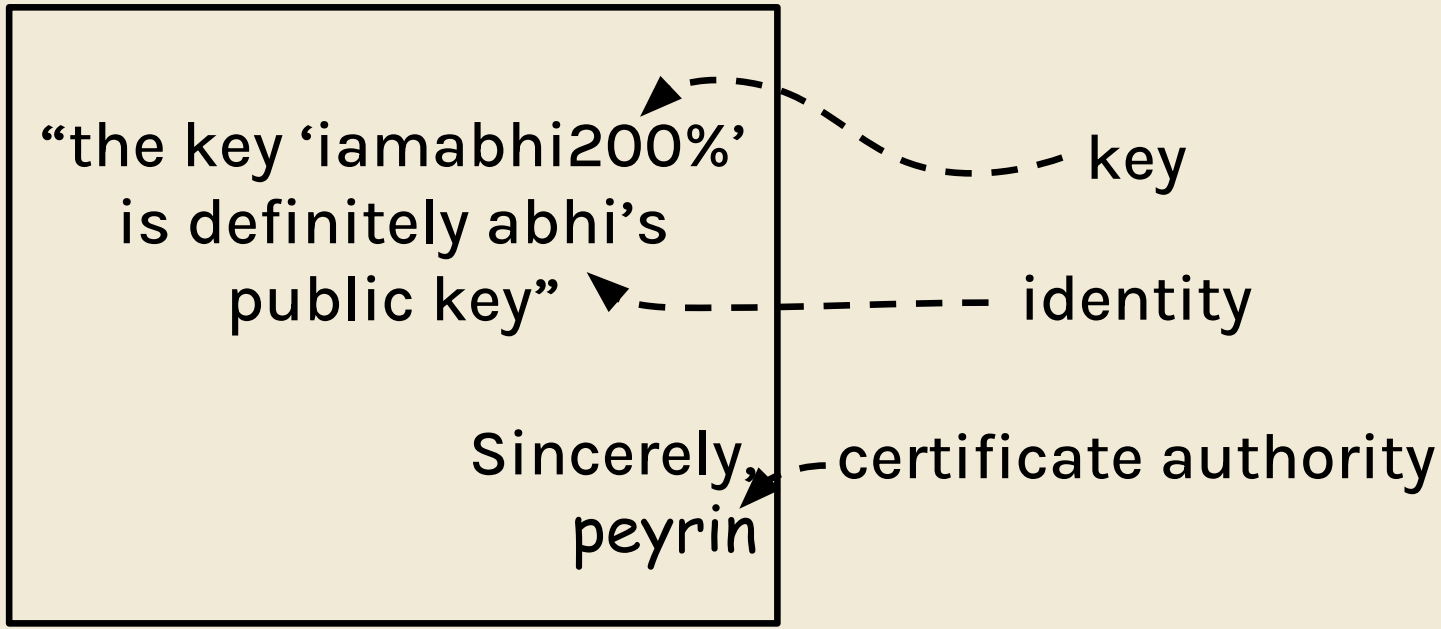
certificates

- signed endorsement of someone's public key



certificates

- signed endorsement of someone's public key



certificates

- signed endorsement of someone's public key
- encryption under a public key *PK*: {"Message"}$_{PK}$

certificates

- signed endorsement of someone's public key
- encryption under a public key *PK*: {"Message"}$_{PK}$
- signing with a private key *SK*: {"Message"}$_{SK^{-1}}$

certificates

- signed endorsement of someone's public key
- encryption under a public key *PK*: {"Message"}$_{PK}$
- signing with a private key *SK*: {"Message"}$_{SK^{-1}}$
  - signed message = message & signature on message (not just a signature)

certificates

- signed endorsement of someone's public key
- encryption under a public key *PK*: {"Message"}$_{PK}$
- signing with a private key *SK*: {"Message"}$_{SK^{-1}}$
  - signed message = message & signature on message (not just a signature)
- if we trust EvanBot and he sends {"Bob's public key is *PK*$_B$"}$_{SK_E^{-1}}$, we trust this certificate

models of certificate trust

models of certificate trust

- **trusted directory**

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - we ask the TD for bob's PK

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - we ask the TD for bob's PK
    - TD says {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - we ask the TD for bob's PK
    - TD says {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
  - trust assumptions

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - we ask the TD for bob's PK
    - TD says {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
  - trust assumptions
    - we receive the right PK_{TD}

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - we ask the TD for bob's PK
    - TD says {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
  - trust assumptions
    - we receive the right PK_{TD}
    - the TD won't sign keys unless the owner is verified

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - problems:

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - problems:
    - scalability: one directory for whole world

models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
  - problems:
    - scalability: one directory for whole world
    - single point of failure: if directory is compromised, can't trust anyone

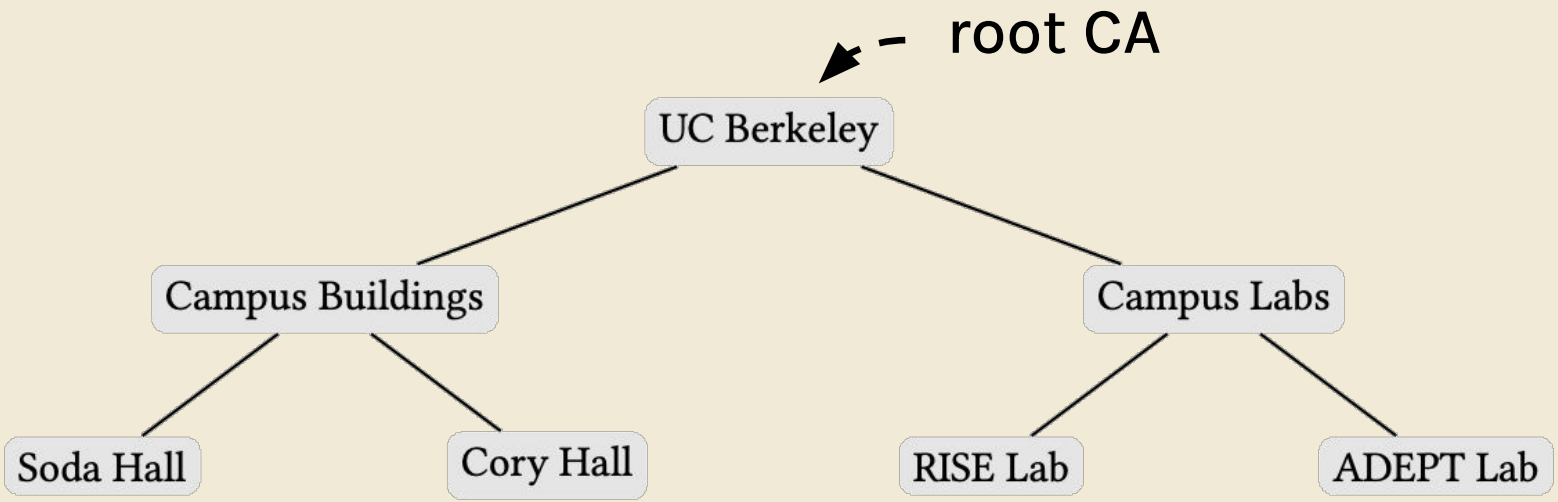
models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
- **certificate authority**

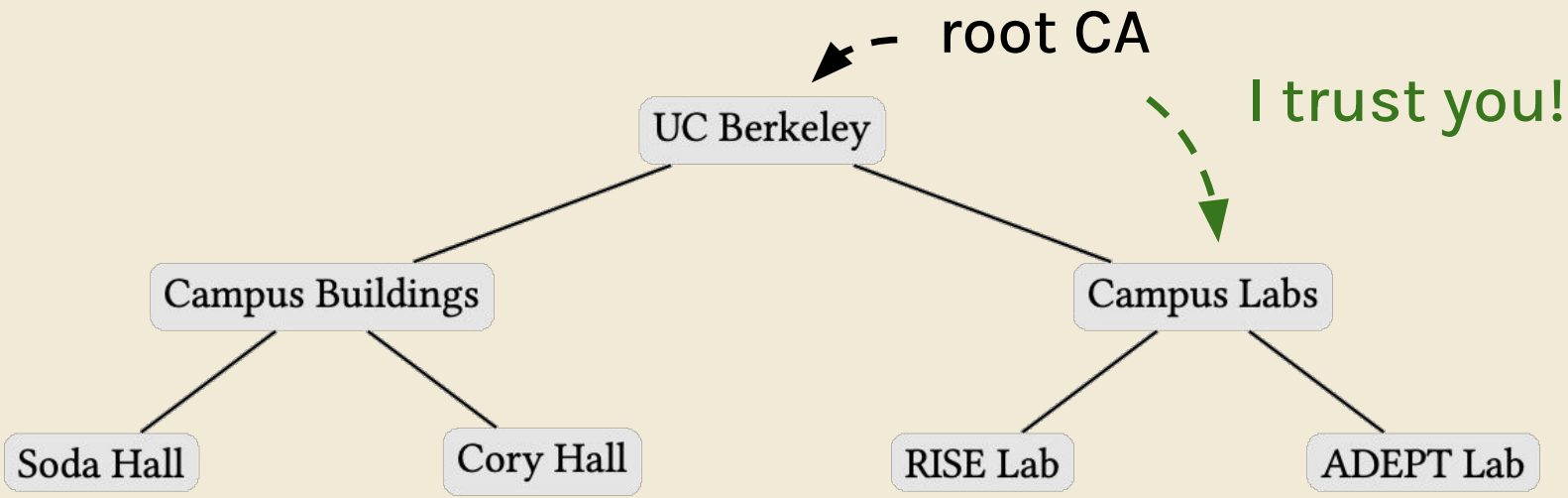
models of certificate trust

- **trusted directory**
  - central directory with PK_{TD}, SK_{TD}
- **certificate authority**
  - hierarchical trust: a root CA signs other CAs, and they can certificates as well

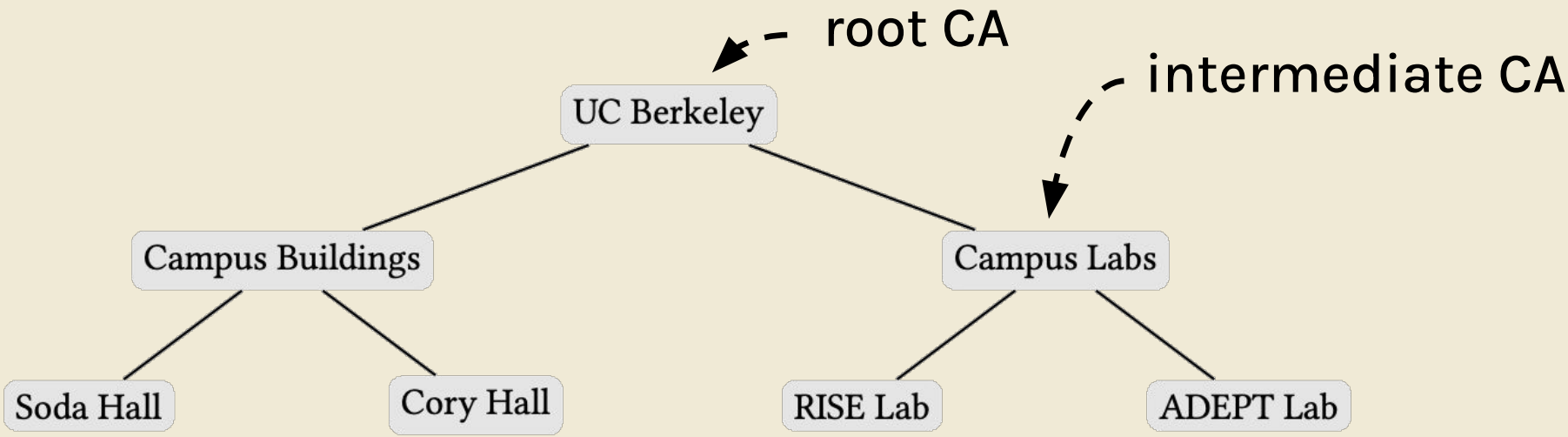
certificate authorities



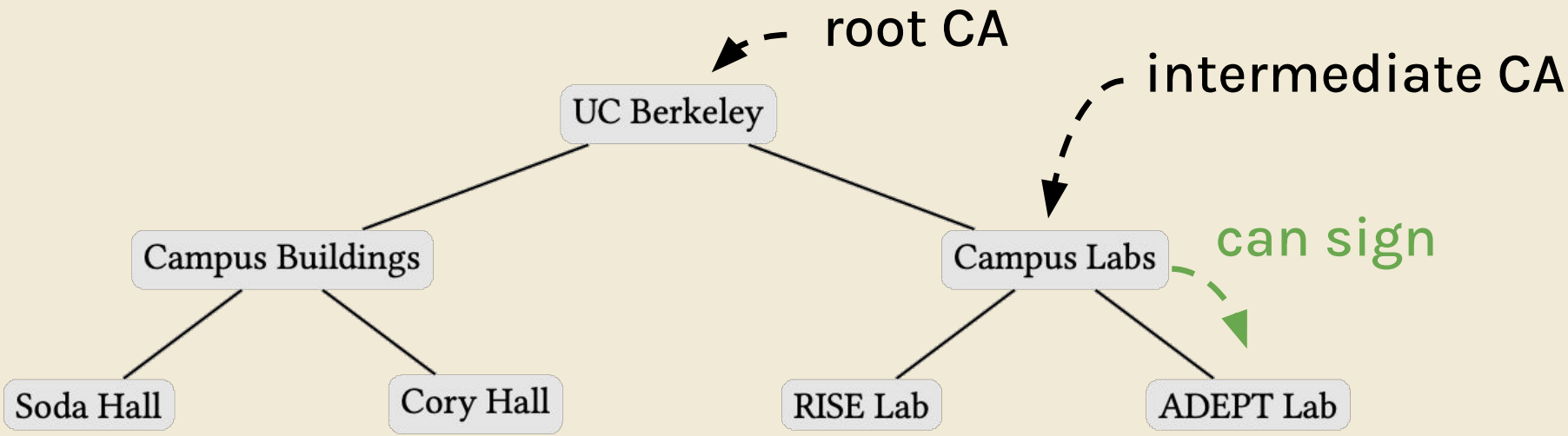
certificate authorities



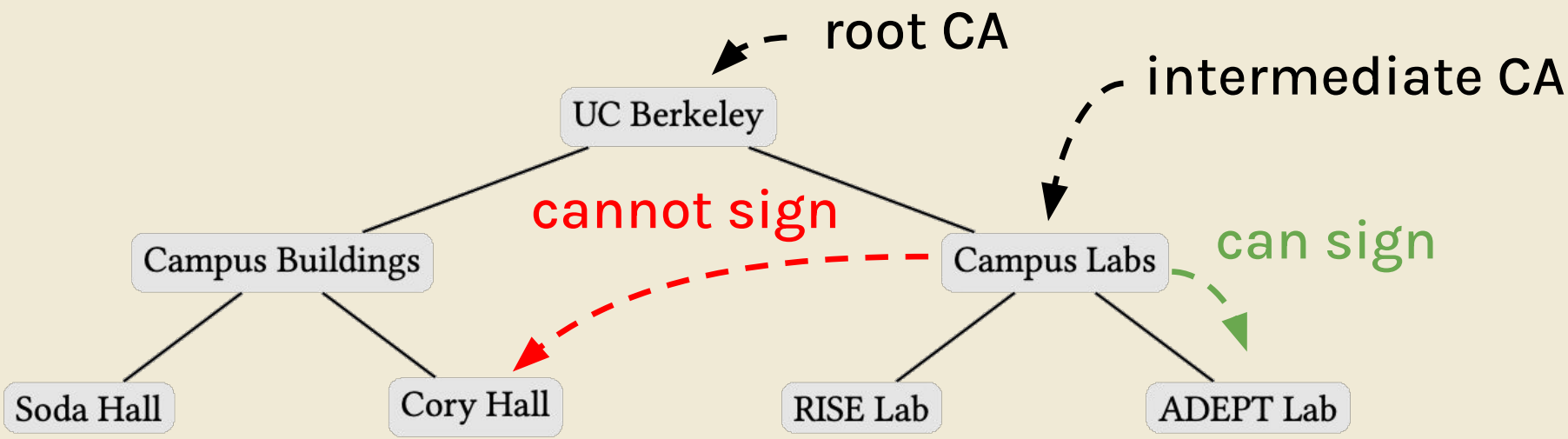
certificate authorities



certificate authorities



certificate authorities



certificate revocation

certificate revocation

- when CA issues bad certificate

certificate revocation

- when CA issues bad certificate

1) expiration dates: when certificate expires, re-request

certificate revocation

- when CA issues bad certificate

1) expiration dates: when certificate expires, re-request
    - tradeoff?

certificate revocation

- when CA issues bad certificate

1) expiration dates: when certificate expires, re-request
    - tradeoff?
2) certificate revocation list

certificate revocation

- when CA issues bad certificate

1) expiration dates: when certificate expires, re-request
   - tradeoff?
2) certificate revocation list
   - {“The certificate with serial number Oxdeadbeef is now revoked”}$_{SK_{CA}^{-1}}$

certificate revocation

- when CA issues bad certificate

1) expiration dates: when certificate expires, re-request
   - tradeoff?
2) certificate revocation list
   - {“The certificate with serial number Oxdeadbeef is now revoked”}$_{SK_{CA}^{-1}}$
   - tradeoff?

hashing—a review

- H(M): M is an arbitrary length message
  - output: fixed length n-bit hash
  - $\{0, 1\}^* \rightarrow \{0, 1\}^n$
- “look” random
- fast
- **one-way:** hard to find x given a y such that H(x) = y
- **collision-resistant:** hard to find $x \neq x'$ s.t. H(x) = H(x’)

hashing passwords

hashing passwords

- given a password, how do I check it’s the right one

hashing passwords

- given a password, how do I check it’s the right one
- store the passwords in plaintext?

hashing passwords

- given a password, how do I check it’s the right one
- store the passwords in plaintext?
  - an attacker can find them

hashing passwords

- given a password, how do I check it's the right one
- store the passwords in plaintext?
  - an attacker can find them
- idea: *hash* the password and store it, then check if input's hash matches stored hash

hashing passwords

- given a password, how do I check it's the right one
- store the passwords in plaintext?
  - an attacker can find them
- idea: *hash* the password and store it, then check if input's hash matches stored hash
- problem: brute-forcing passwords

hashing passwords: attacks

hashing passwords: attacks

- attacker can precompute common passwords, e.g., H(“password”), H(“password123”), H(“peyrin”)

hashing passwords: attacks

- attacker can precompute common passwords, e.g., H(“password”), H(“password123”), H(“peyrin”)
  - dictionary attack: hash an entire dictionary of common passwords

hashing passwords: attacks

- attacker can precompute common passwords, e.g., H(“password”), H(“password123”), H(“peyrin”)
  - dictionary attack: hash an entire dictionary of common passwords
- rainbow tables: an algorithm to make brute-forcing easier

brute-forcing passwords

how hard is it to brute force a 10-bit password?

password = “1011010100”

hash = H(“1011010100”) = 158912

brute-forcing passwords

how hard is it to brute force a 10-bit password?

password = “1011010100”

hash = H(“1011010100”) = 158912

only guessing 1 or 0 for 10 bits, 2 options

for each of 10 bits, 2 * 2 * 2 …. * 2 = 2^{10}

guesses and then hashing each guess

mitigating brute force: salts

mitigating brute force: salts

- **salt:** a public, random bitstring for each user

mitigating brute force: salts

- **salt:** a public, random bitstring for each user
  - should be long and random

mitigating brute force: salts

- **salt:** a public, random bitstring for each user
  - should be long and random
  - public!!! like IVs or nonces

mitigating brute force: salts

- **salt:** a public, random bitstring for each user
  - should be long and random
  - public!!! like IVs or nonces
- store (user, salt, H(password || salt)) in database

mitigating brute force: salts

- **salt:** a public, random bitstring for each user
  - should be long and random
  - public!!! like IVs or nonces
- store (user, salt, H(password || salt)) in database
- if there are M possible passwords and N users

mitigating brute force: salts

- **salt:** a public, random bitstring for each user
  - should be long and random
  - public!!! like IVs or nonces
- store (user, salt, H(password || salt)) in database
- if there are M possible passwords and N users
  - brute force runtimes:

mitigating brute force: salts

- **salt:** a public, random bitstring for each user
  - should be long and random
  - public!!! like IVs or nonces
- store (user, salt, H(password || salt)) in database
- if there are M possible passwords and N users
  - brute force runtimes:
    - unsalted: O(M + N)

mitigating brute force: salts

- **salt:** a public, random bitstring for each user
  - should be long and random
  - public!!! like IVs or nonces
- store (user, salt, H(password || salt)) in database
- if there are M possible passwords and N users
  - brute force runtimes:
    - unsalted: O(M + N)
    - salted: O(MN)

mitigating brute force: slow hash

mitigating brute force: slow hash

- cryptographic hashes are fast

mitigating brute force: slow hash

- cryptographic hashes are fast
- password hashes are designed to be slow

mitigating brute force: slow hash

- cryptographic hashes are fast
- password hashes are designed to be slow
  - why?

mitigating brute force: slow hash

- cryptographic hashes are fast
- password hashes are designed to be slow
  - why?
  - guessing the wrong password should waste time

mitigating brute force: slow hash

- cryptographic hashes are fast
- password hashes are designed to be slow
  - why?
  - guessing the wrong password should waste time
  - users can’t tell the difference between 0.001 and 0.1 second hashes, attackers computing thousands of hashes can

online v.s. offline attacks

online v.s. offline attacks

- online attack

online v.s. offline attacks

- online attack
  - attacker guesses password via website input

online v.s. offline attacks

- online attack
  - attacker guesses password via website input
  - defense: rate limit (certain # of guesses/min)

online v.s. offline attacks

- online attack
  - attacker guesses password via website input
  - defense: rate limit (certain # of guesses/min)
- offline attack

online v.s. offline attacks

- online attack
  - attacker guesses password via website input
  - defense: rate limit (certain # of guesses/min)
- offline attack
  - computation done by attacker, can parallelize

online v.s. offline attacks

- online attack
  - attacker guesses password via website input
  - defense: rate limit (certain # of guesses/min)
- offline attack
  - computation done by attacker, can parallelize
  - defense: salted passwords, slow hashes, strong passwords

worksheet

(on 161 website)

feedback

bit.ly/extended-feedback

slides: **bit.ly/cs161-disc**