# public key distribution

## certificates, passwords

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

feedback
**bit.ly/extended-feedback**

# hack of the day

# hack of the day

- Trusted Platform Module 2.0, billions of PCs/IoT

# hack of the day

- Trusted Platform Module 2.0, billions of PCs/IoT
    - “crypto-processor” to store hardware-secured cryptographic functions

# hack of the day

- [Trusted Platform Module 2.0, billions of PCs/IoT]
    - “crypto-processor” to store hardware-secured cryptographic functions
    - out of bounds read/write vulnerabilities are “the result of a lack of necessary length checks, resulting in buffer overflows that could pave the way for local information disclosure or escalation of privileges.”

# general questions, concerns, etc.

PRNGs → rollback resistance

↳ if internal state compromised, can't get past states / outputs

↓

IND-CPA

↳ CBC

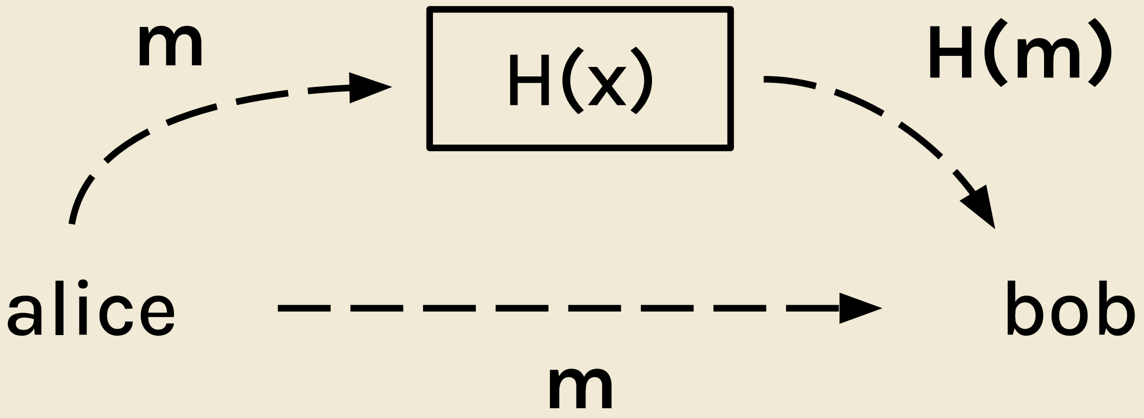
↳ CTR

IVs / nonce

# reminder: cryptography

- why?
  - secure communication
- goals:
  - confidentiality: adversary cannot <u>read</u> messages
  - integrity: adversary cannot <u>change</u> messages
  - authenticity: message is from the claimed author

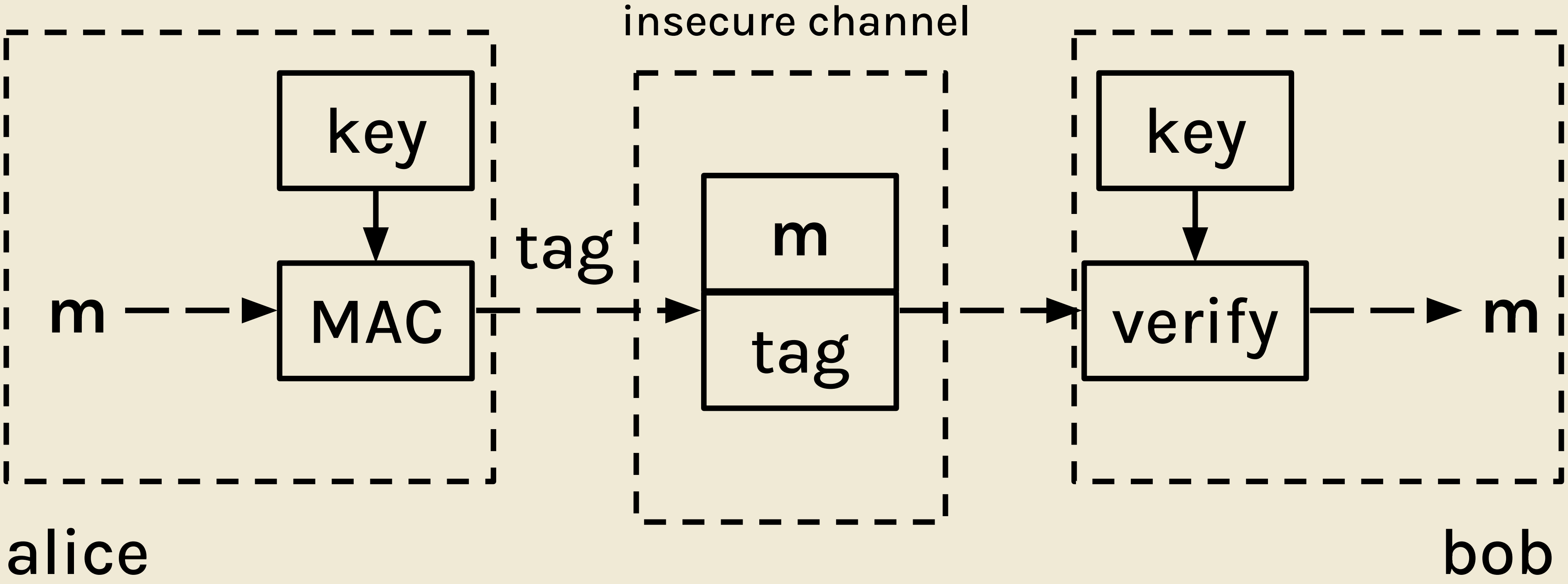
# recap: hashes and integrity




if the message is tampered with (and H(m) isn't), Bob can compute H(m) himself and make sure it matches the sent hash

# recap: message authentication codes

m: message



# recap: authenticated encryption (AE)

- MAC-then-encrypt: $\mathrm{Enc}(K1, M \,||\, \mathrm{MAC}(K2, M))$
- encrypt-then-MAC: $\mathrm{MAC}(K2, \mathrm{Enc}(K1, M))$
- which is better?
  - both are technically IND-CPA and EU-CPA
  - but MAC-then-encrypt requires decryption before tag verification
  - **always use encrypt-then-MAC**
    - more robust to mistakes

# diffie-hellman key exchange (lecture)

Public: $g, p$

Alice

Eve

Bob

Secret key

Public key

| Alice | Eve | Bob |
|---|---|---|
| Generate $a$ | | Generate $b$ |
| Calculate $g^a$ mod $p$ | $g^a$    $g^b$ | Calculate $g^b$ mod $p$ |
| Receive $g^b$ mod $p$ | | Receive $g^a$ mod $p$ |
| Calculate $(g^b)^a$ mod $p$ | | Calculate $(g^a)^b$ mod $p$ |
| $a, g^a, g^b \Rightarrow g^{ab}$ | $g^a, g^b \not\Rightarrow g^{ab}$ | $b, g^a, g^b \Rightarrow g^{ab}$ |

**Shared symmetric key is $g^{ab}$**

# RECAP
# public key crypto

# 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 $C_1 = R$, $C_2 = M \times B^r \bmod p$
- Bob: $C_2 \times C_1^{-b} = M \times B^r \times R^{-b} = M \times g^{br} \times g^{-br} = M \bmod p$

# RSA - encrypt

message: $\boxed{M}$ $\rightarrow$ encrypted message: $\boxed{C = M^e \mod N}$ $\rightarrow$ $\boxed{C}$ $\rightarrow$ decrypted message: $\boxed{\begin{aligned} M &= C^d \mod N \\ &= M^{ed} \mod N \end{aligned}}$

**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

“the key ‘iamabhi200%’
is definitely abhi’s
public key”

Sincerely,
peyrin

# certificates

- signed endorsement of someone’s public key

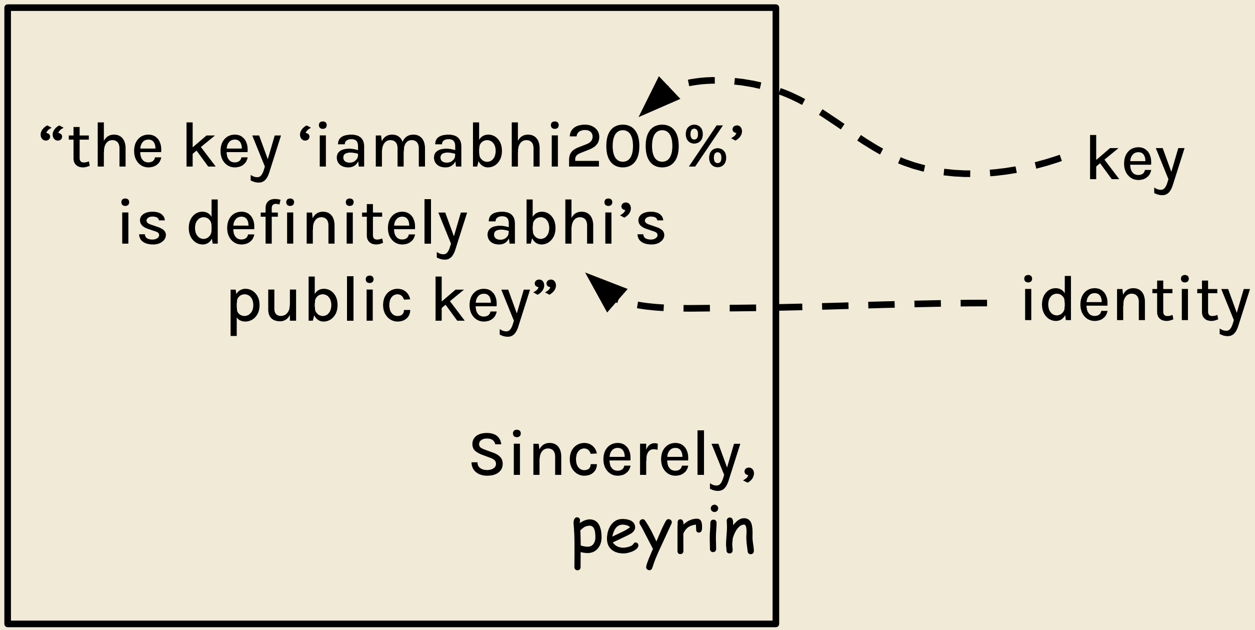
“the key ‘iamabhi200%’
is definitely abhi’s
public key”

Sincerely,
peyrin

key

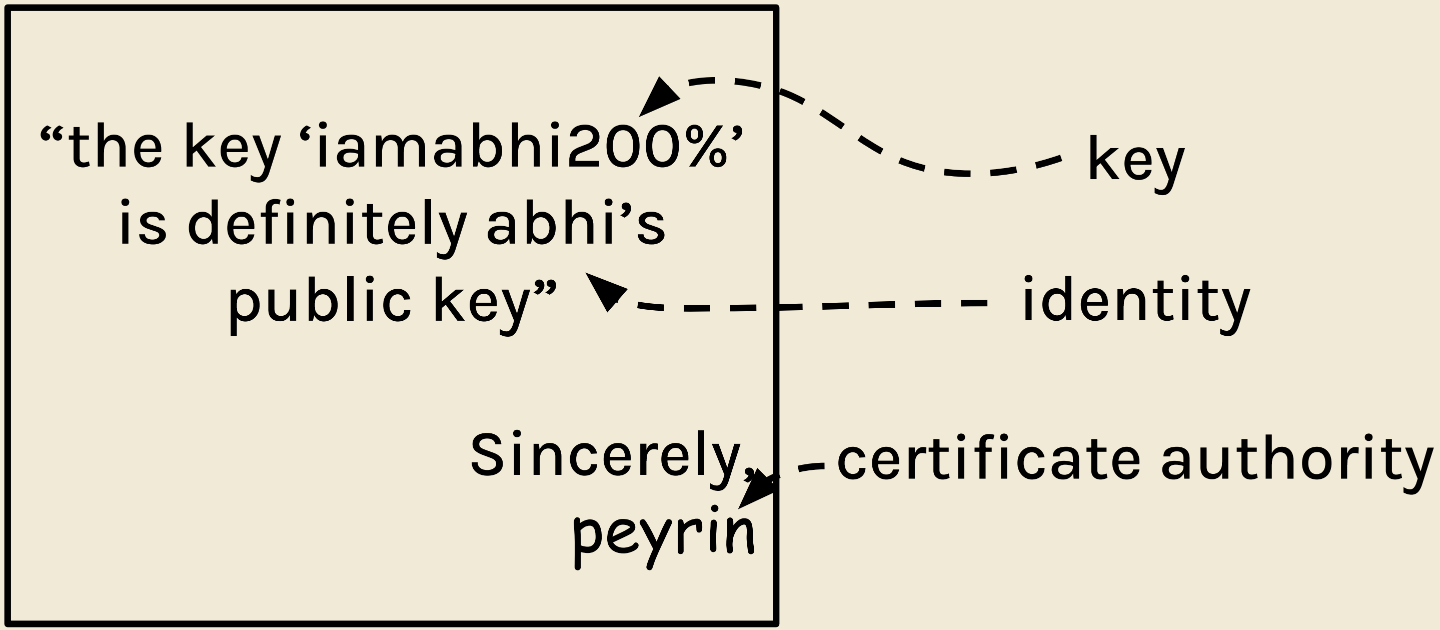
# certificates

- 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*: $\{\text{“Message”}\}_{PK}$

# certificates

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

# 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*: $\{\text{“Message”}\}_{PK}$
- signing with a private key *SK*: $\{\text{“Message”}\}_{SK^{-1}}$
    - signed message = message & signature on message (not just a signature)
- if we trust EvanBot and he sends $\{\text{“Bob’s public key is } PK_B\text{”}\}_{SK_E^{-1}}$, 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

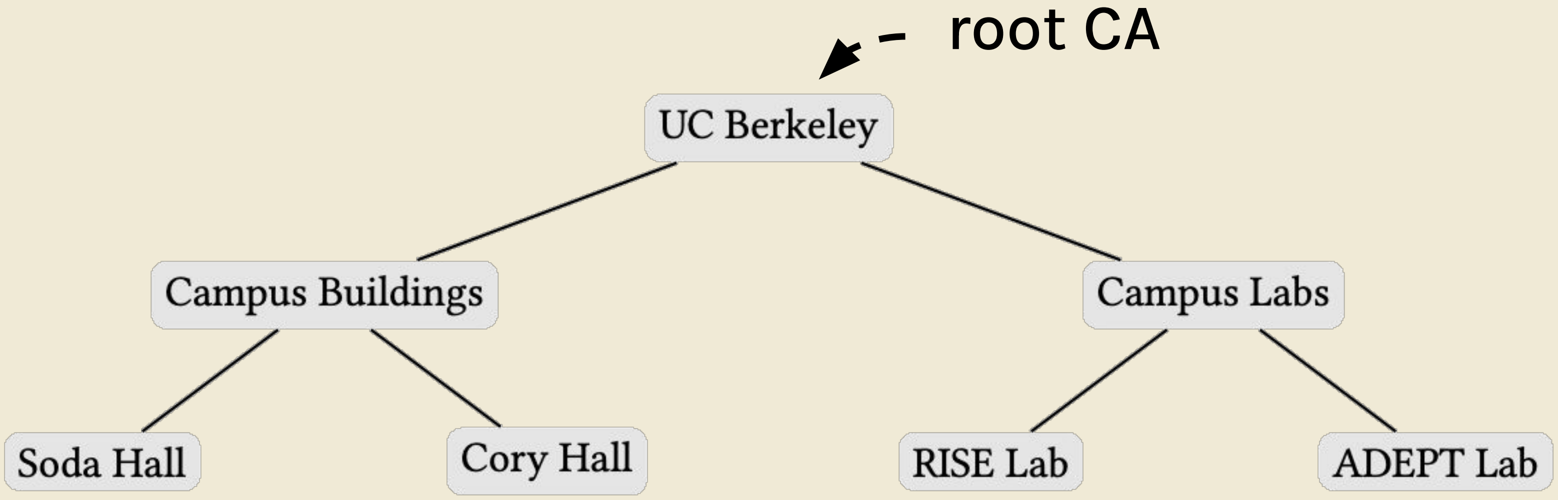
# models of certificate trust

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

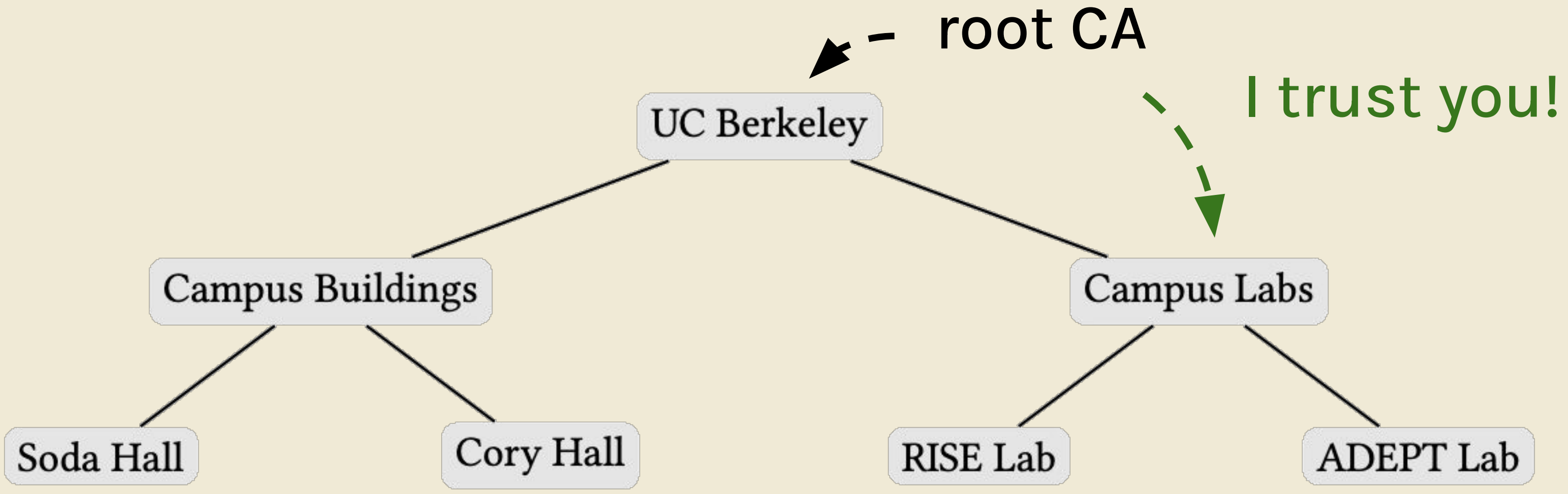
# models of certificate trust

- **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

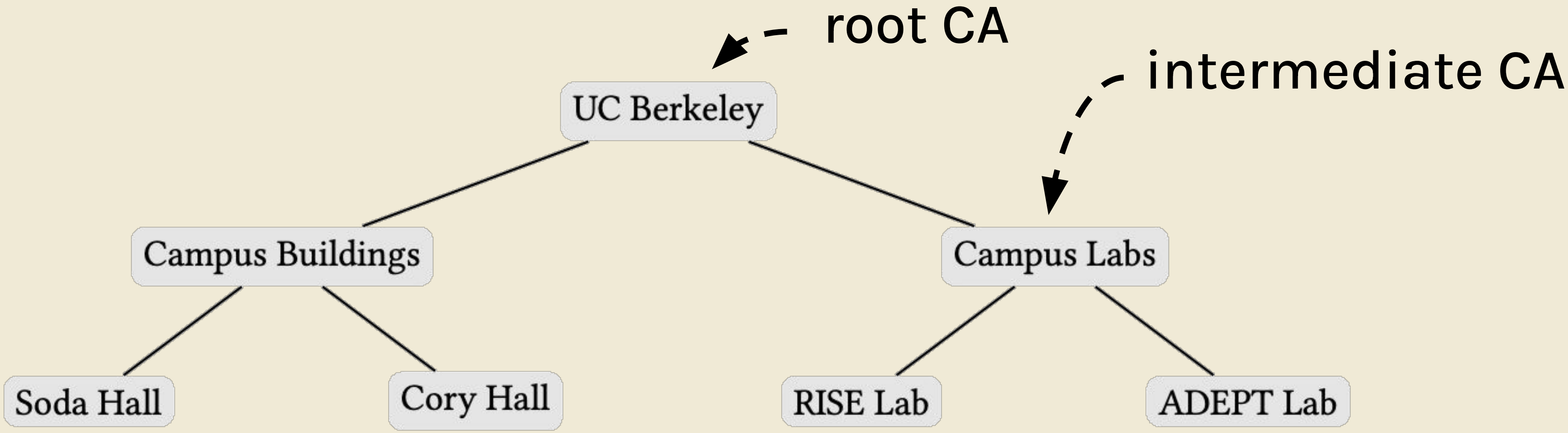
# certificate authorities



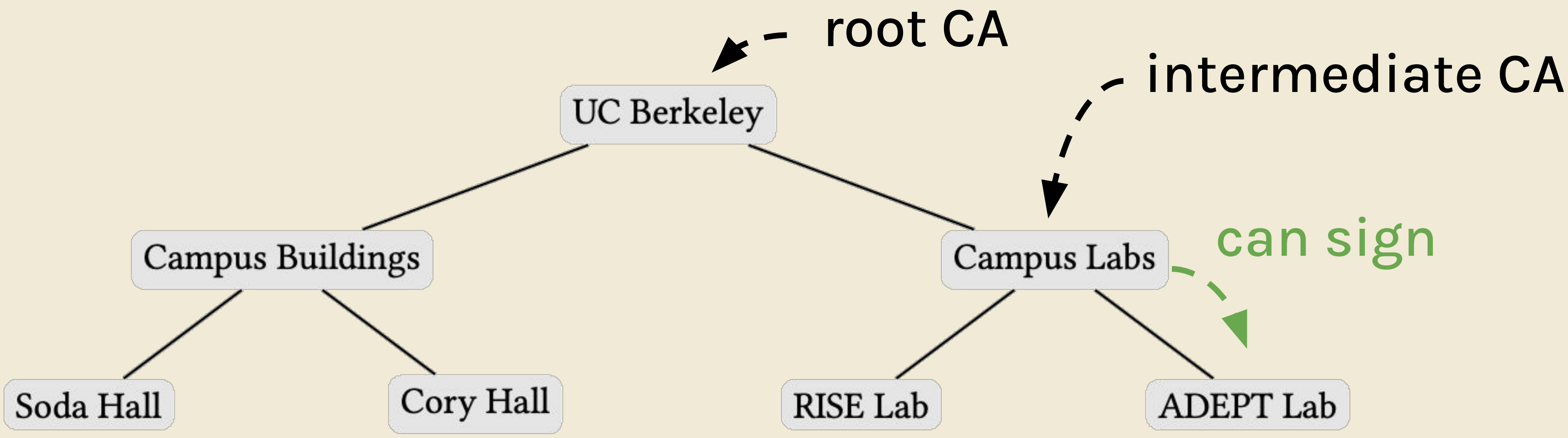
# certificate authorities



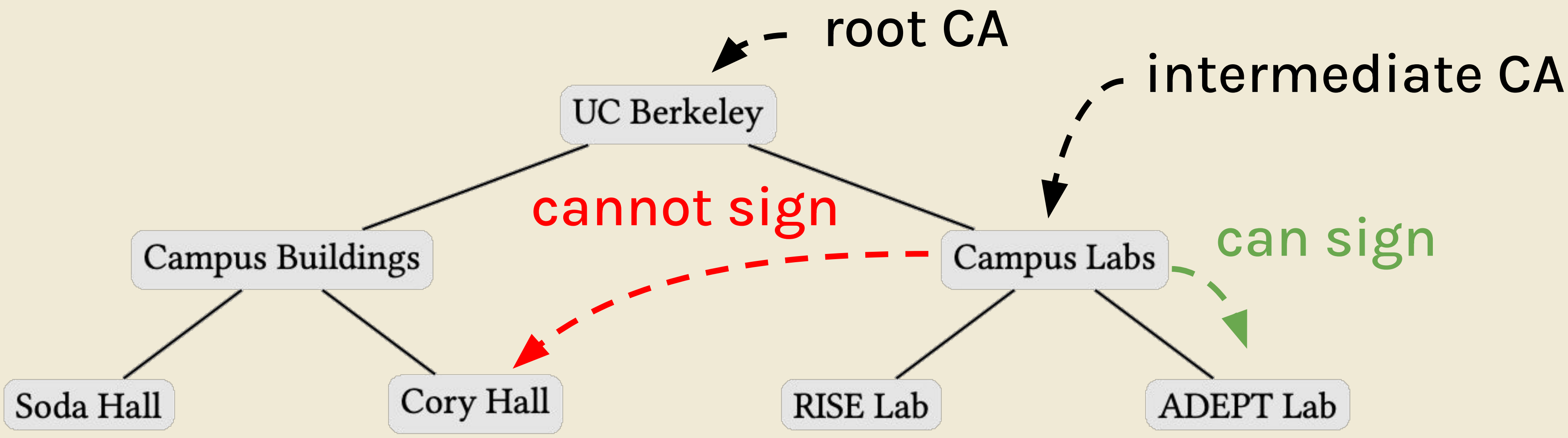
# certificate authorities



# 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 0xdeadbeef 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 0xdeadbeef is now revoked"}$_{SK_{CA}^{-1}}$
   - tradeoff?

# hashing—a review

- H(M): M is an arbitrary length message
  - output: <u>fixed length</u> 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 ≠ 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

H(password123 7$cafxY$

@

- 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**