Cryptography, TLS, and Certificates

CMSC 23200, Spring 2026, Lecture 7

David Cash and Grant Ho

University of Chicago

Logistics

- Assignment 3 is due Thursday at 11:59pm
- Looking ahead: Midterm is Tuesday May 5, at lecture time
  - Location TBD

Outline of Lecture 7

1. Digital Signatures
2. Putting Everything Together: Secure Channels
3. Certificates
4. Attacks on Certificates, and Countermeasures

Outline of Lecture 7

1. **Digital Signatures**
2. Putting Everything Together: Secure Channels
3. Certificates
4. Attacks on Certificates, and Countermeasures

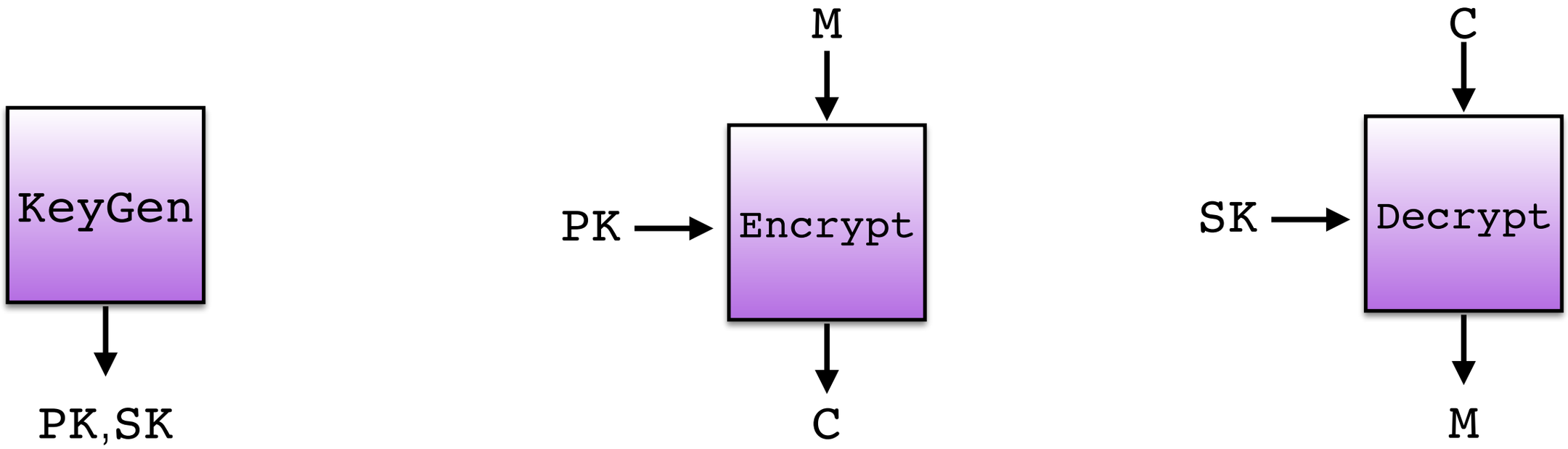
Recall: Public-Key Cryptography

Basic question: If two people do not have pre-shared a key, is there any way they can send private messages in the presence of an attacker?

Pre-shared key? / Security Goal	Confidentiality	Authenticity/Integrity
Yes ("Symmetric")	Symmetric Encryption (Stream Ciphers and Block Ciphers)	Message Authentication Codes (MACs)
No ("Asymmetric")	Public-Key Encryption (RSA) and Key Exchange (Diffie-Hellman)	Digital Signatures

Recall: Public-Key Encryption

A public-key encryption scheme consists of three algorithms **KeyGen**, **Encrypt,** and **Decrypt**



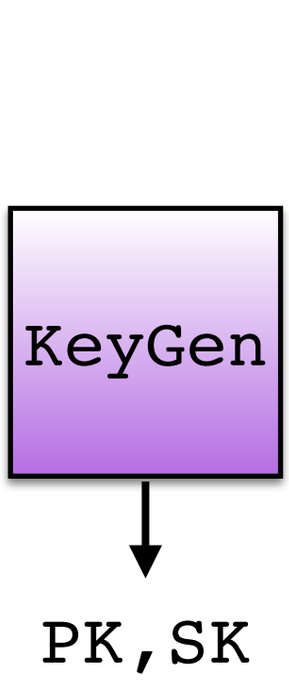
KeyGen: Outputs two keys. **PK** published openly, and **SK** kept secret.

Encrypt: Uses **PK** and **M** to produce a ciphertext **C**.

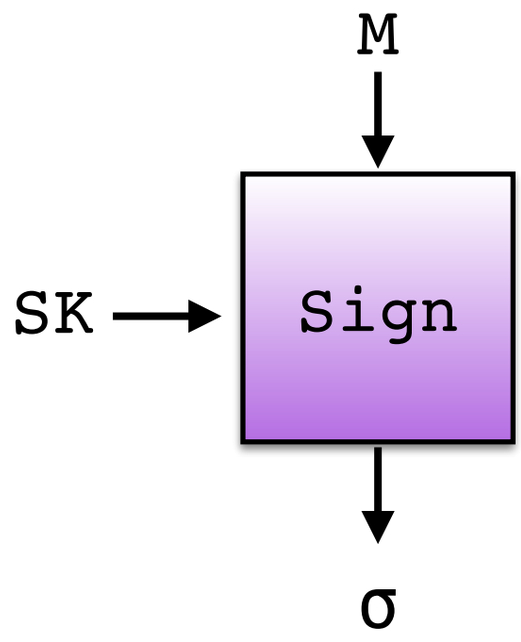
Decrypt: Uses **SK** and **C** to recover **M**.

Digital Signatures Schemes

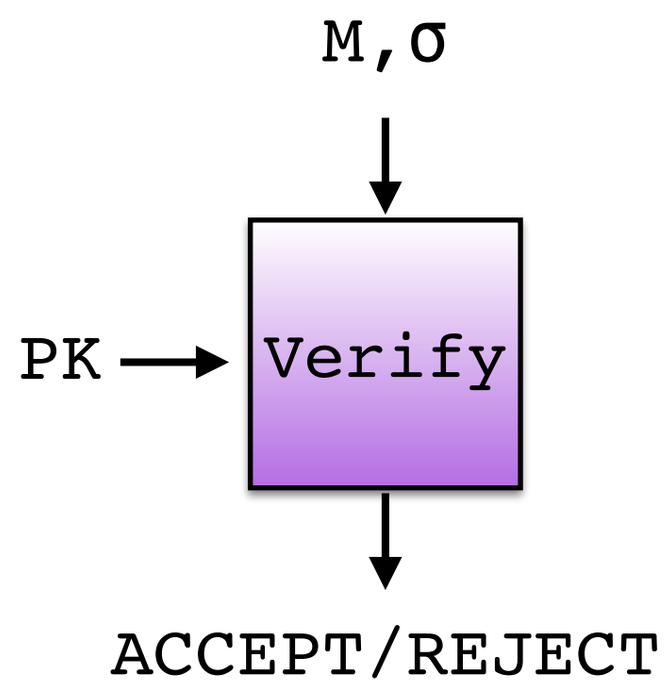
A digital signature scheme consists of three algorithms **`KeyGen`**, **`Sign`**, and **`Verify`**



`KeyGen`: Outputs two keys. **`PK`** published openly, and **`SK`** kept secret.

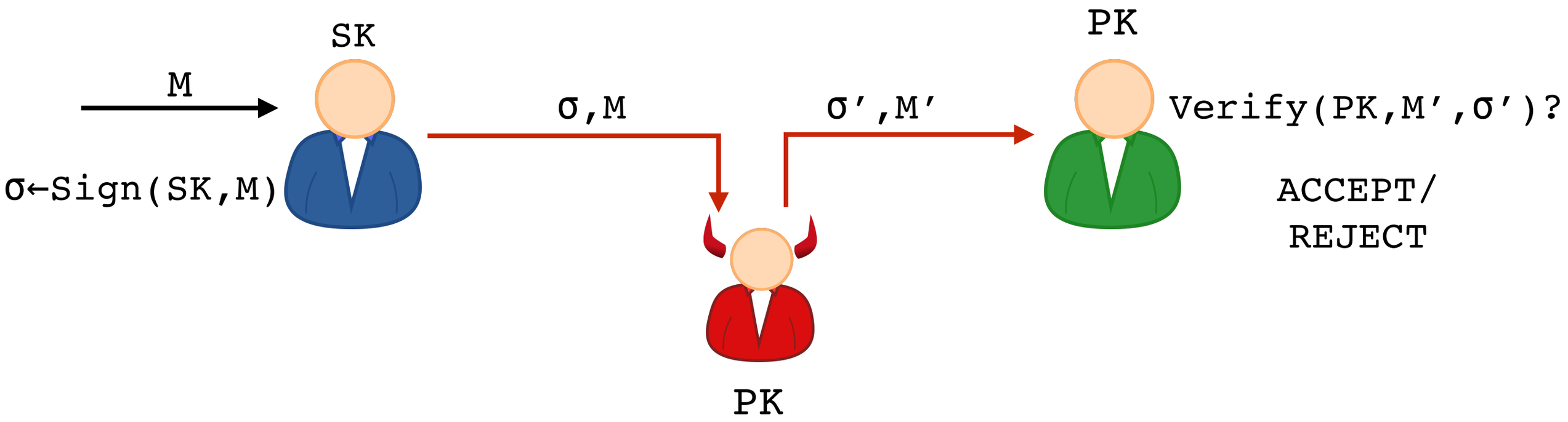


`Sign`: Uses **`SK`** to produce a "signature" **σ** on **`M`**.



`Verify`: Uses **`PK`** to check if signature **σ** is valid for **`M`**.

Digital Signature Security Goal: Unforgeability



Scheme satisfies **unforgeability** if it is unfeasible for Adversary (who knows `PK`) to fool Bob into accepting `M'` not previously sent by Alice.

Plain RSA Signatures

RSA-KeyGen(n)

1. Choose random n-bit primes p and q
2. N = pq; phi(N) = (p-1)(q-1)
3. Set e to default value
4. Find d such that ed == 1 (mod phi(N))
5. Output
   PK = (N,e)
   SK = (N,d)

RSA-Sign(SK,M)

1. Parse SK as (N,d)
2. Compute $\sigma = M^d \pmod N$
3. Output σ

RSA-Verify(PK,M,σ)

1. Parse PK as (N,e)
2. Compute $V = \sigma^e \pmod N$
3. Accept if V==M

- Note similarity to encryption
- **Intuition**: Computing "**e**-th roots modulo **N**" is hard if you can't factor **N**
- **Warning**: Not secure as stated; needs further protections

Warning: Broken

"Plain" RSA Weaknesses

Assume $e=3$.

RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute $\sigma = M^d \pmod N$
3. Output σ

RSA-Verify(PK,M,σ)

1. Parse PK as (N,e)
2. Compute $V = \sigma^e \pmod N$
3. Accept if V==M

To forge a signature on message M: Find number σ such that $\sigma^3 = M \bmod N$

$M=1$ weakness: If $M=1$ then it is easy to forge. Take $\sigma=1$:

$$\sigma^3 = 1^3 = 1 = M \bmod N$$ ✓

Cube-M weakness: If M is a *perfect cube* then it is easy to forge.
Just take $\sigma = M^{1/3}$:, i.e. the usual cube root of M:

Example: To forge on $M=8$, which is a perfect cube, set $\sigma=2$.

$$\sigma^3 = 2^3 = 8 = M \bmod N$$ ✓

(Intuition: If cubing does not "wrap modulo N", then it is easy to un-do.)

Further "Plain" RSA Weaknesses

Assume $e=3$.

RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute $\sigma = M^d \pmod N$
3. Output σ

RSA-Verify(PK,M,σ)

1. Parse PK as (N,e)
2. Compute $V = \sigma^e \pmod N$
3. Accept if V==M

To forge a signature on message M: Find number σ such that $\sigma^3 = M \bmod N$

***Malleability* weakness:** If σ is a valid signature for M, then it is easy to forge a signature on $M' = 8M \bmod N$.

Given (M,σ), compute $\sigma' = (2*\sigma \bmod N)$

Then $\mathtt{Verify}((N,3),M',\sigma')$ checks:

$$(\sigma')^3 = (2*\sigma \bmod N)^3 = (2^3*\sigma^3 \bmod N) = (2^3*M \bmod N) = 8M \bmod N$$

$\sigma^3 = M \bmod N$ b/c σ is valid sig. on M

Further "Plain" RSA Weaknesses

Assume `e=3`.

RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute $\sigma = M^d \pmod N$
3. Output σ

RSA-Verify(PK,M,σ)

1. Parse PK as (N,e)
2. Compute $V = \sigma^e \pmod N$
3. Accept if V==M

To forge a signature on message `M`: Find number σ such that $\sigma^3 = M \bmod N$

***Backwards signing* weakness:** Generate *some* valid signature by picking σ' first, and then defining $M' = (\sigma'^3 \bmod N)$

Then `Verify((N,3),M',σ')` checks:

$$(\sigma')^3 = (M' \bmod N)$$ ✓

Further "Plain" RSA Weaknesses

Assume `e=3`.

RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. Compute $\sigma = M^d \pmod N$
3. Output σ

RSA-Verify(PK,M,σ)

1. Parse PK as (N,e)
2. Compute $V = \sigma^e \pmod N$
3. Accept if V==M

To forge a signature on message `M`: Find number σ such that $\sigma^3 = M \bmod N$

Summary:

- Plain RSA Signatures allow several types of forgeries
- It was sometimes argued that these forgeries aren't important: If `M` is english text, then `M'` is unlikely to be meaningful for these attacks
- But often they are damaging anyway

RSA Signatures with Encoding

- Select a function `Encode` that "randomizes" messages
- Sign/Verify `R=Encode(M)` instead of `M`

RSA-Sign(SK,M)

1. Parse PK as (N,d)
2. R = Encode(M)
3. Compute $\sigma = R^d \pmod N$
4. Output σ

RSA-Verify(PK,M,σ)

1. Parse PK as (N,e)
2. R = Encode(M)
3. Compute $V = \sigma^e \pmod N$
4. Accept if V==R

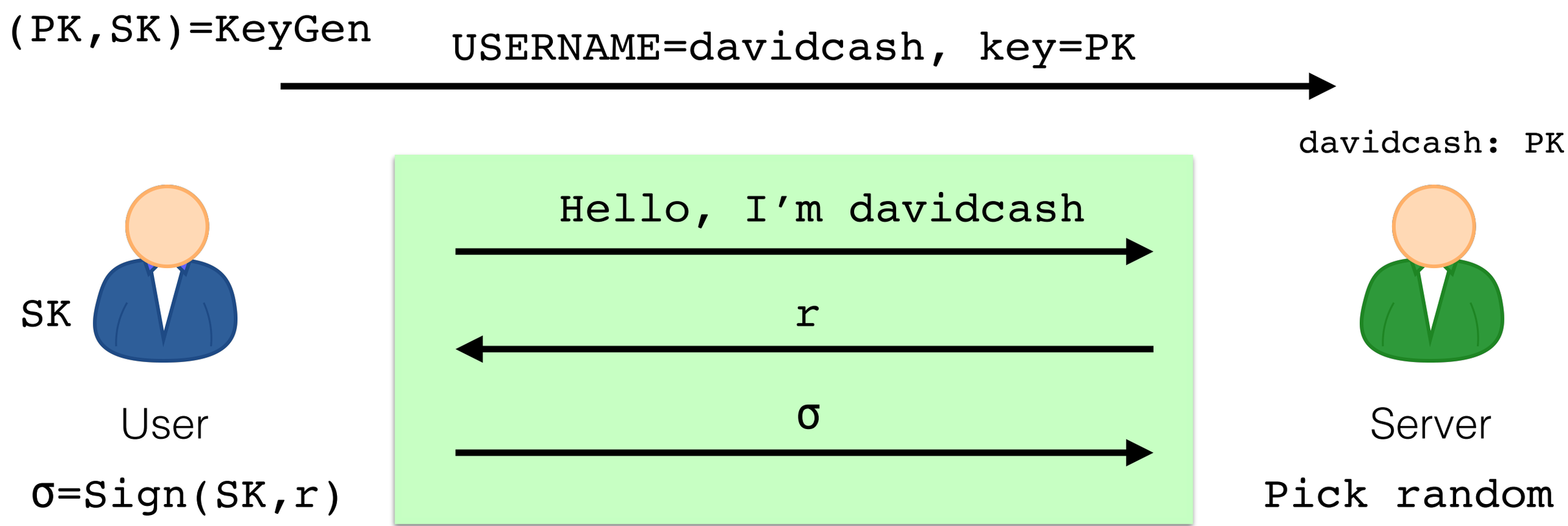
- For example, signing `M=1` now requires signing `Encode(1)` instead of `1`

Encoding must be chosen with extreme care.


Broken


Authentication with Digital Signatures (SSH, TLS, ...)

- User wants to use a single key to log into several servers

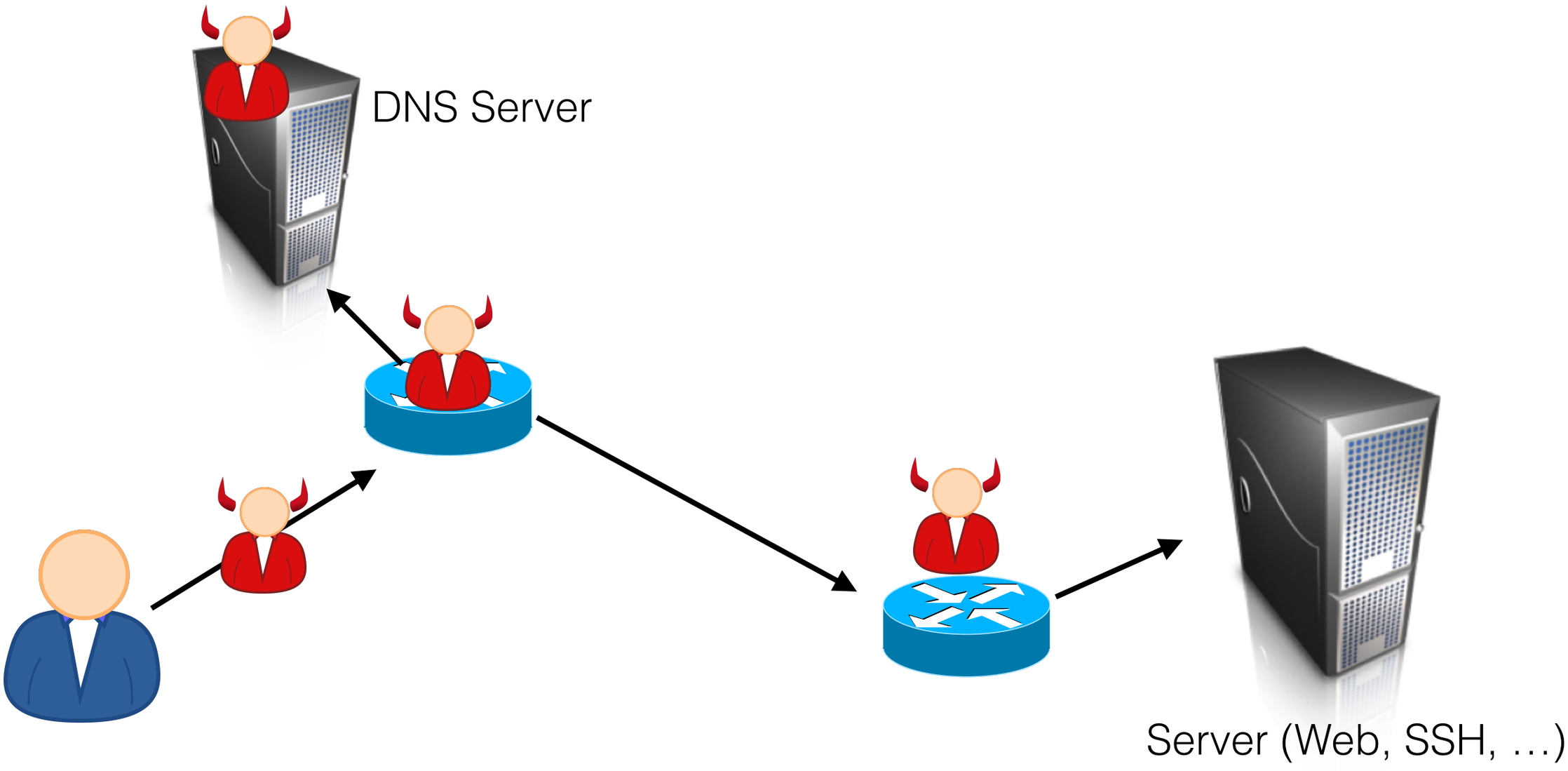


- Why a random challenge?
- In practice, challenge string also session ID and other info

Outline of Lecture 7

1. Digital Signatures
2. **Putting Everything Together: Secure Channels**
3. Certificates
4. Attacks on Certificates, and Countermeasures

Threat Model for Secure Channels on the Internet

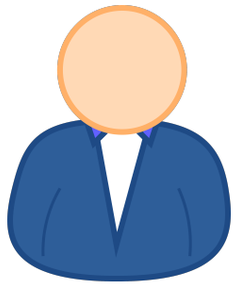


- Malicious/eavesdropping infrastructure
  - Examples: router, person at coffeeshop, ISP…
- Malicious DNS Server (more on these later in networking part)

- **Not** in threat model: Compromised endpoint

Key Exchange

- **Goal**: Agree on key `K` for symmetric encryption that only endpoints know



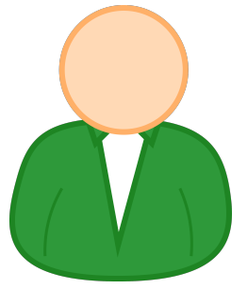
1. `Pick random K`
2. `C = Enc(PK,K)`
3. `Output K`

Key Exchange ("Handshake")

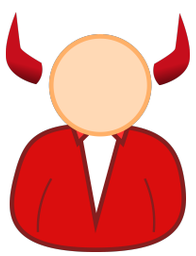
`PK`

`C`

`(PK,SK)=KeyGen`

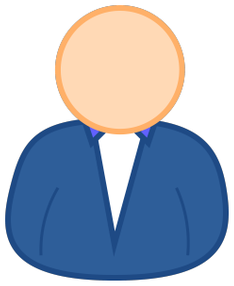


1. `K=Dec(SK,C)`
2. `Output K`



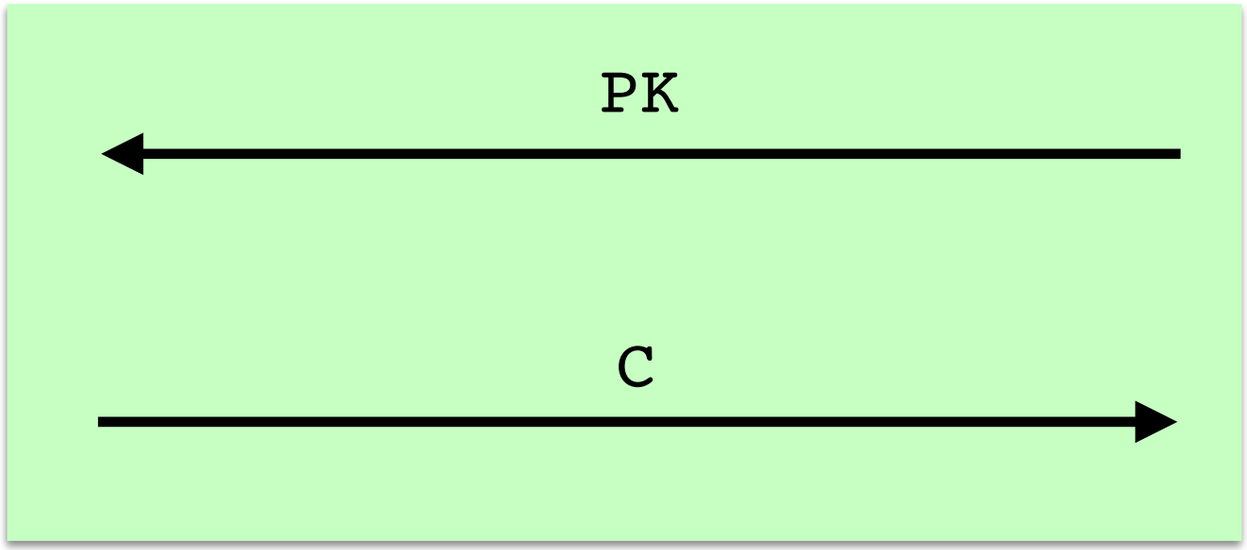
`K?`

Key Exchange and Secure Channels

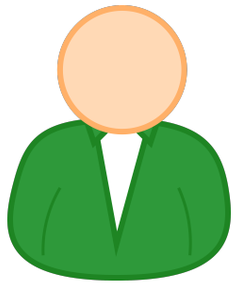


1. Pick random K
2. C = Enc(PK,K)
3. Output K

Key Exchange (“Handshake”)

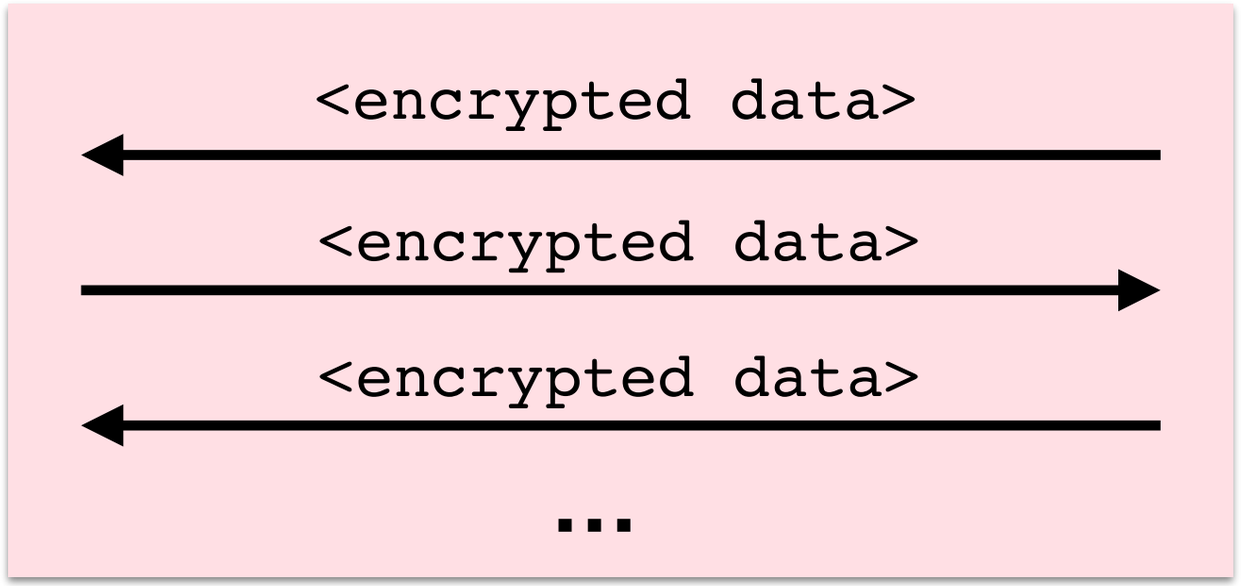


(PK,SK)=KeyGen



1. K=Dec(SK,C)
2. Output K

Symmetric Encryption (“Record Protocol”)



Diffie-Hellman: More efficient key exchange

- They modulus `N` for RSA is relatively large
  - Slows down encryption/decryption, uses bandwidth

- Now: A totally different, faster approach based on different math
  - Invented in 1970s, but new ideas have made it the standard choice
  - Strictly speaking, not public-key encryption, but can adapted into it if needed

The Setting: Discrete Logarithm Problem

Discrete Logarithm Problem:
Input: Prime p, integers g, X.
Output: Integer r such that $g^r = X \bmod p$.

- Different from factoring: Only one prime.
- Contrast with logarithms with real numbers, which are easy to compute. *Discrete* logarithms appear to be hard to compute
- Largest solved instances: 795-bit prime p (Dec 2019)

Diffie-Hellman Key Exchange

<u>Parameters: (fixed in standards, used by everyone)</u>:
Prime p (1024 or 2048 bit usually)
Number g (usually 2)

$$(p, g)$$

Diffie-Hellman Key Exchange

Parameters: (fixed in standards, used by everyone):

Prime p (1024 or 2048 bit usually)

Number g (usually 2)

$$(p, g)$$

Network Working Group
Request for Comments: 5114
Category: Informational

M. Lepinski
S. Kent
BBN Technologies
January 2008

Additional Diffie-Hellman Groups for Use with IETF Standards

Status of This Memo

This memo provides information for
not specify an Internet standard o
memo is unlimited.

Abstract

This document describes eight Diff
in conjunction with IETF protocols
communications. The groups allow
with a variety of security protoco
(SSH), Transport Layer Security (T
(IKE).

3. 2048-bit MODP Group

This group is assigned id 14.

This prime is: 2^2048 - 2^1984 - 1 + 2^64 * { [2^1918 pi] + 124476 }

Its hexadecimal value is:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
15728E5A 8AACAA68 FFFFFFFF FFFFFFFF
```

The generator is: 2.

Diffie-Hellman Key Exchange

Parameters: (fixed in standards, used by everyone):
Prime p (1024 or 2048 bit usually)
Number g (usually 2)

$$(p, g)$$

Pick $r_A \in \{1,\ldots,p-1\}$
$X_A \leftarrow g^{r_A} \bmod p$

Pick $r_B \in \{1,\ldots,p-1\}$
$X_B \leftarrow g^{r_B} \bmod p$

X_A

X_B

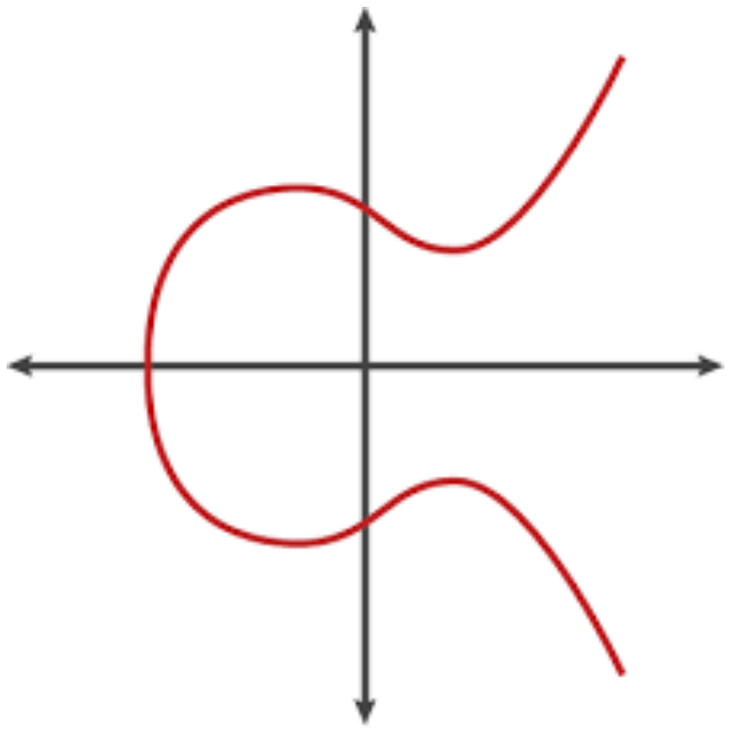
$K \leftarrow X_B^{r_A} \bmod p$

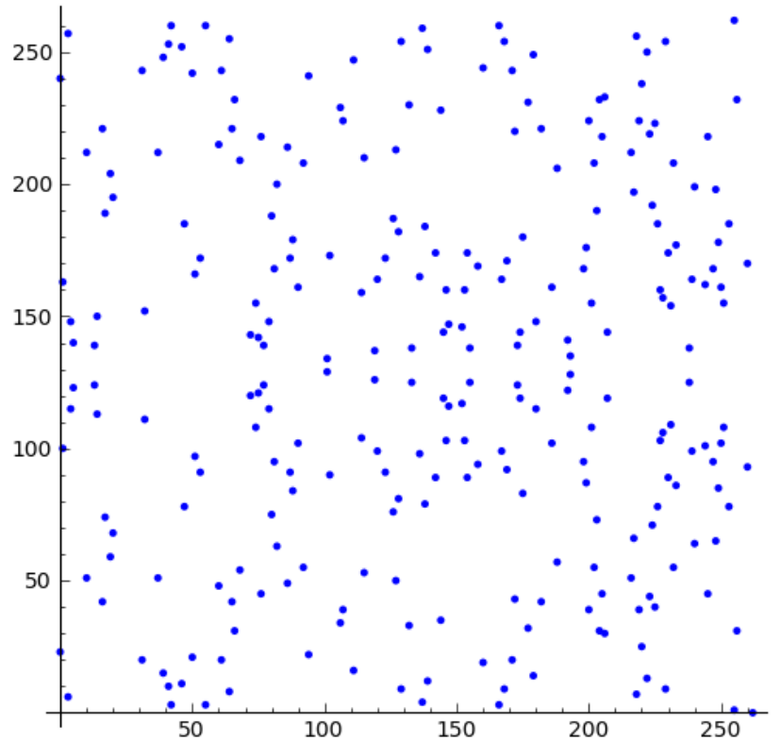
$K \leftarrow X_A^{r_B} \bmod p$

Correctness: $X_B^{r_A} = (g^{r_B})^{r_A} = g^{r_A r_B} = (g^{r_A})^{r_B} = X_A^{r_B} \bmod p$

Modern Key Exchange: *Elliptic Curve* Diffie-Hellman

- Totally different math from RSA
- Advantage: Bandwidth and computation (due to higher security)
  - 512 bit vs 2048-bit values.





- Used by default in TLS

Public-Key Encryption/Key Exchange Wrap-Up

- RSA-OAEP and Diffie-Hellman (either mod a prime or in an elliptic curve) are unbroken and run fine in TLS/SSH/etc.
- Elliptic-Curve Diffie-Hellman is preferred choice going forward.

Quantum computers will break:

- RSA (any padding)
- Diffie-Hellman (including Elliptic Curve)

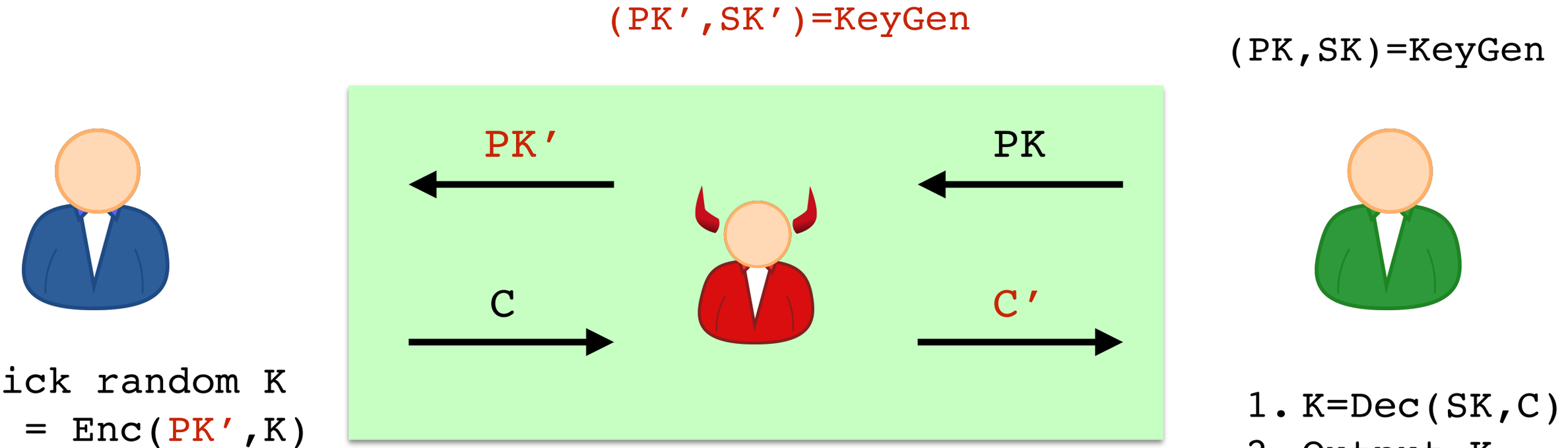
Shor's algorithm, 1994

Peter Shor

- Quantum computers are far from large enough, currently
- "Post-quantum" crypto = crypto not known to be broken by quantum computers (i.e. not RSA or DH)
- On-going research on post-quantum cryptography from hard problems on lattices, with first deployments in recent years

Active “Man-in-the-Middle” Attacks on Key Exchange

- Adversary can transparently learn result of key exchange, and read subsequent session



- **Plan**: Prevent substitution of public key by binding keys to servers

Outline of Lecture 7

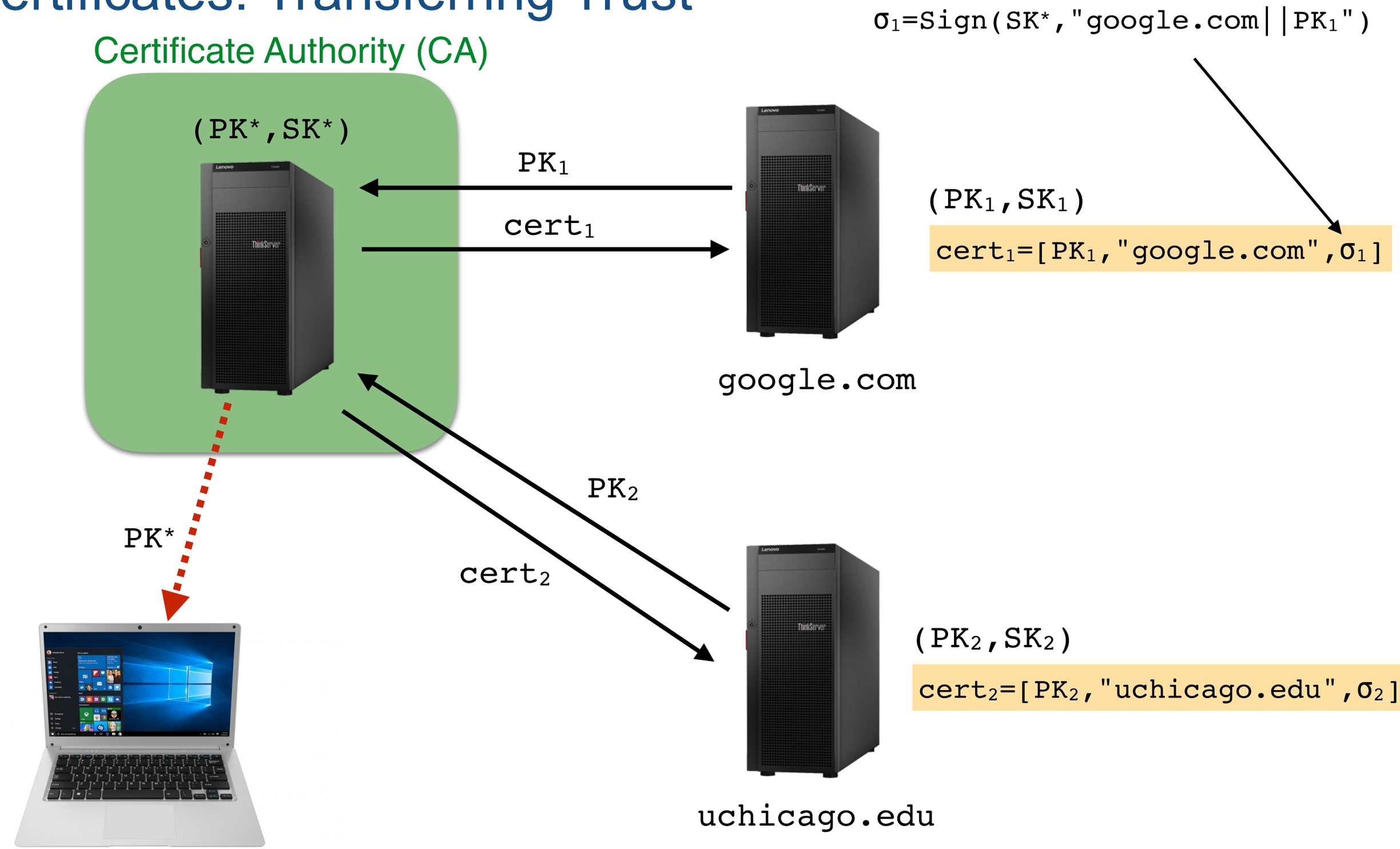
1. Digital Signatures
2. Putting Everything Together: Secure Channels
3. **Certificates**
4. Attacks on Certificates, and Countermeasures

Certificates (“Certs”)

- Assume that:
  - Everyone trusts some single organization (say, **`GrantCorp`**)
  - Everyone knows **`GrantCorp`**’s public key is **`PK`***

- We would call **`GrantCorp`** a *Certificate Authority (CA)*

Certificates: Transferring Trust

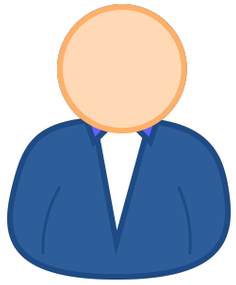


- **PK*** installed on laptop by manufacturer or user.
- Trusted CA “issues certs”, i.e. signs public keys of other orgs.

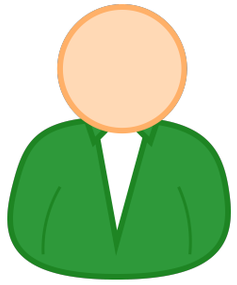
Securing Key Exchange against Active Attackers

Authenticated Key Exchange

CA key: PK*



(PK,SK)=KeyGen



cert=[PK,"Bob",σ]

C

Check that endpoints agreed on same key

1. K=Dec(SK,C)
2. Output K

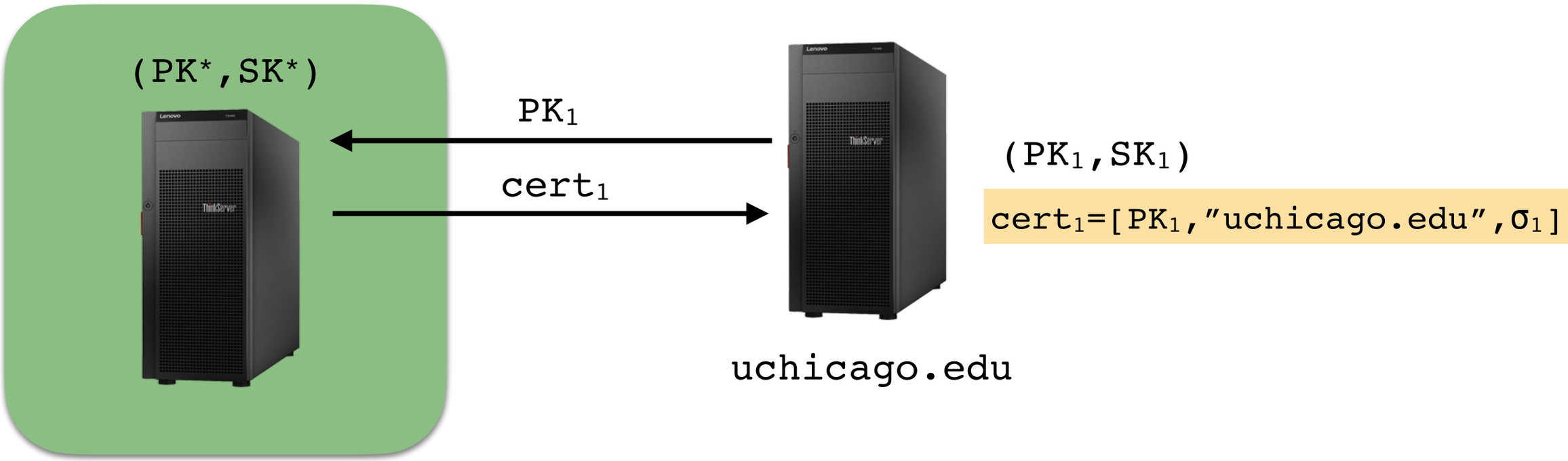
1. Check:
   a. Cert is for Bob
   b. Verify(PK*,cert,σ) ← Use CA's key here
2. Pick random K
3. C = Enc(PK,K) ← Use Bob's key from cert here
4. Output K

- Many subtle details omitted

Outline of Lecture 7

1. Digital Signatures
2. Putting Everything Together: Secure Channels
3. Certificates
4. **Attacks on Certificates, and Countermeasures**

Issuing Certificates: Validation

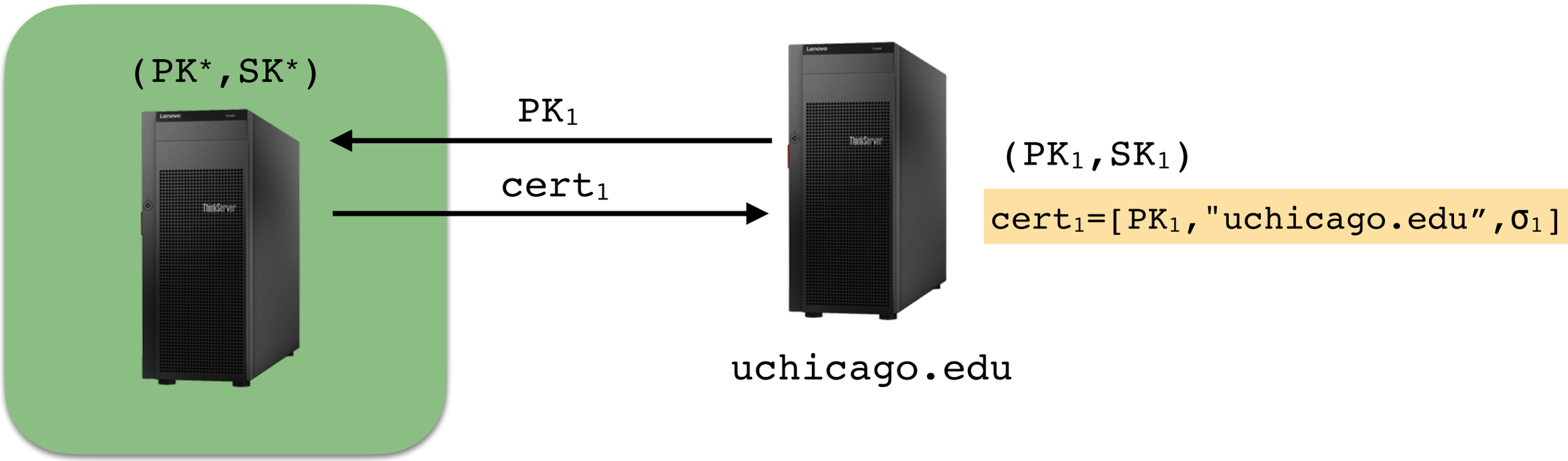


- CA must check that key really does "belong to `google.com`"

Domain Validation (DV): Check that party with that key can control domain.

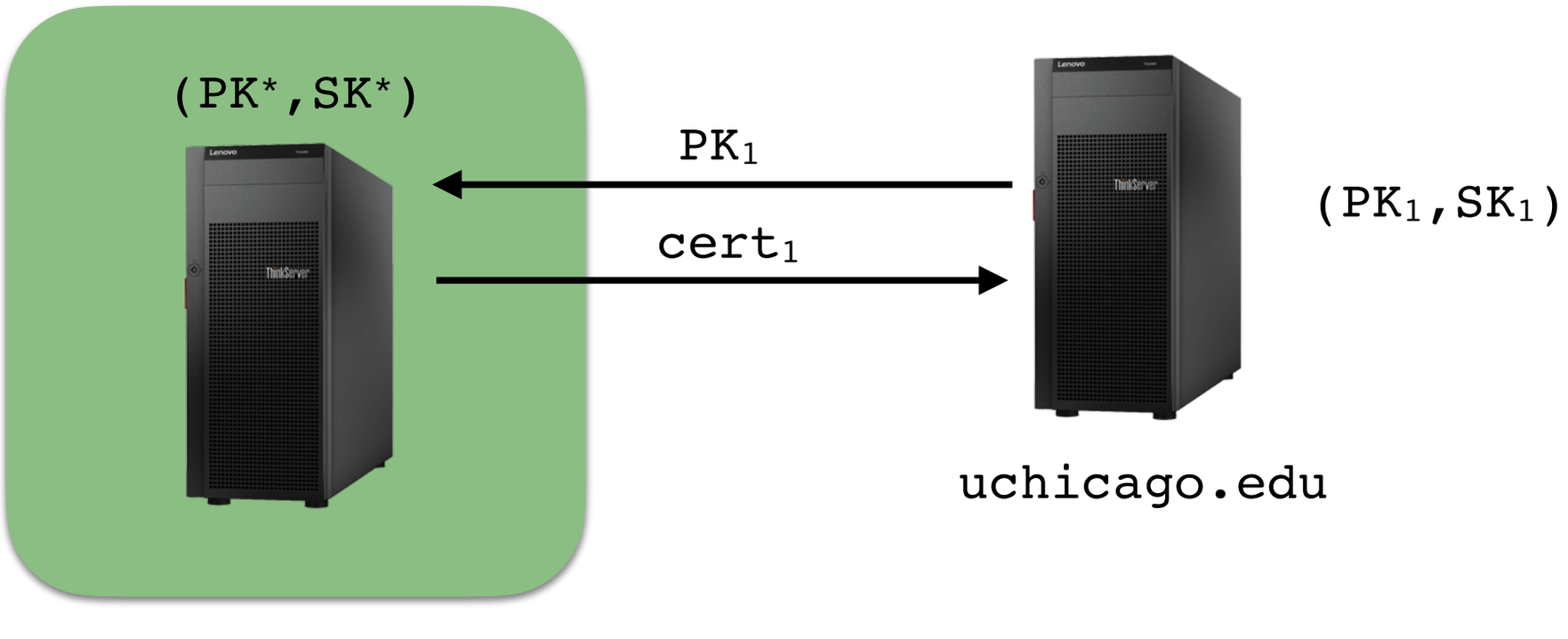
Org. Validation (OV) and **Extended Validation (EV)**: Also check company name, location etc via public records.

ACME Protocol by Let's Encrypt



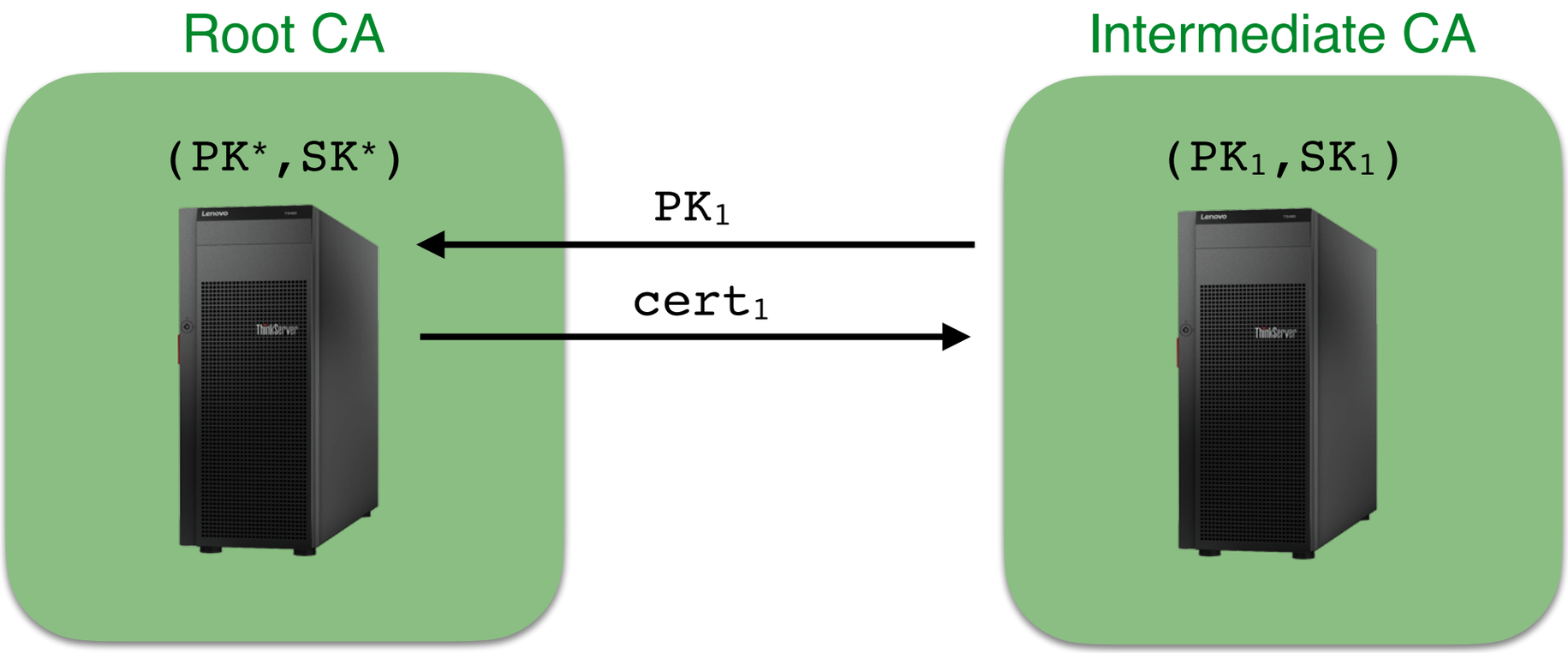
1. Requestor submits public key and request to CA
2. CA gives a challenge to requestor
3. Requestor places challenge on server or DNS records
4. CA checks challenge and then issues cert

Scaling Certificates to the Internet



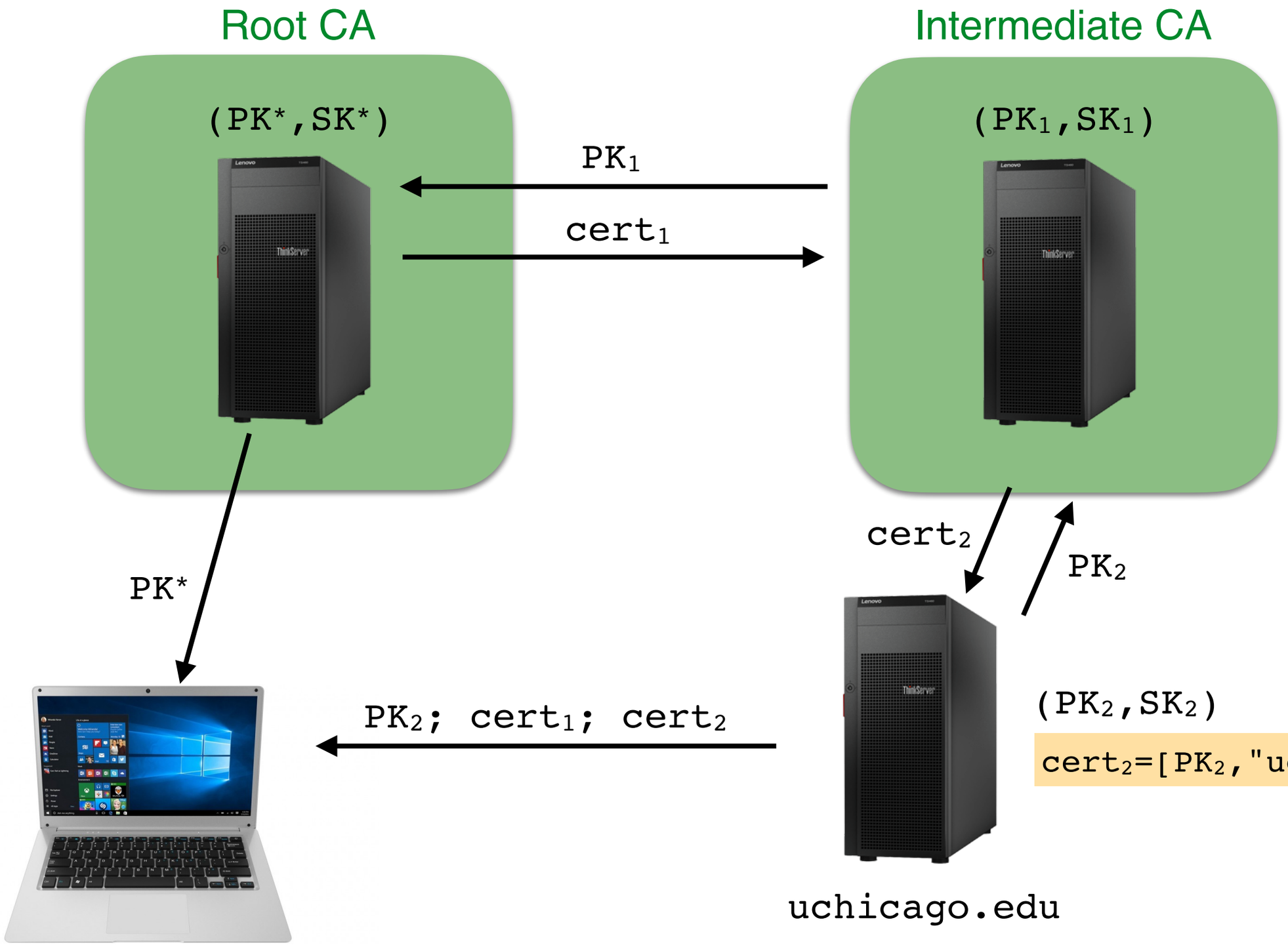
- What are downsides of having exactly one CA with exactly one key?
  - Not everyone trusts one org (e.g. across different countries)
  - Having only one key is very risky

Intermediate CAs



- “Root” CAs will sign the keys of “Intermediate” CAs
- Users who trust the Root will also trust the Intermediate CAs
- 100s of Intermediate CAs on the Internet
- Also used for defense-in-depth within one CA organization

Intermediate CAs and Cert Chains



PK* bound to root ⇒ **PK$_1$** bound to CA ⇒ **PK$_2$** bound to uchicago.edu

USERTrust RSA Certification Authority
↳ InCommon RSA Server CA
↳ *.uchicago.edu

Public Key Info

Algorithm RSA Encryption (1.2.840.113549.1.1.1)

Parameters None

Public Key 256 bytes : CA E9 01 25 77 E9 74 B8 CB F7 99 DA D6 87 79 35 D7 31 CA D7 83 11 83 32 FA FA 43 CC C8 85 7B 76 EF 79 BB 4B 8B E0 35 87 EE A4 34 17 DC 5A 0D 5A 04 D3 F1 BA E7 98 9F 49 FC D5 B9 2C FB C8 DD 36 47 4D 07 FE 41 11 75 B0 42 F7 6D 40 4C BF F5 B6 C7 FE 05 0D DE 3B 7C E9 9F 6A 1C 1C 89 2E AA E8 F5 E3 5B 04 55 16 B0 48 92 C7 F9 37 11 89 F8 C5 85 C1 24 96 71 6F 78 B6 6B 35 39 92 8C EF 17 91 D1 97 D7 EF 93 6E 95 F1 EE C6 0D 5A EA 39 C6 4E 33 E2 CA F2 9A 41 F4 A2 41 9C E8 EA 46 FB EF 71 C0 A6 D3 C6 A5 94 81 4B 12 5E 80 63 87 7C 2F A6 8A A5 9A 31 9E 81 63 7F 0F 26 25 B6 6D 62 C2 AD B4 E7 68 FD C9 F8 86 2C 3F F8 E1 59 F3 3E 73 08 DF 6C 92 98 21 D2 AD EF 23 E7 33 A2 D4 5E 67 74 E3 AB 08 DF 15 31 9A 9D 3B 36 7D 6B 77 48 60 17 A4 10 F3 17 77 53 E0 21 D9 F9 A4 12 0F 39 DA D1

Exponent 65537

Key Size 2,048 bits

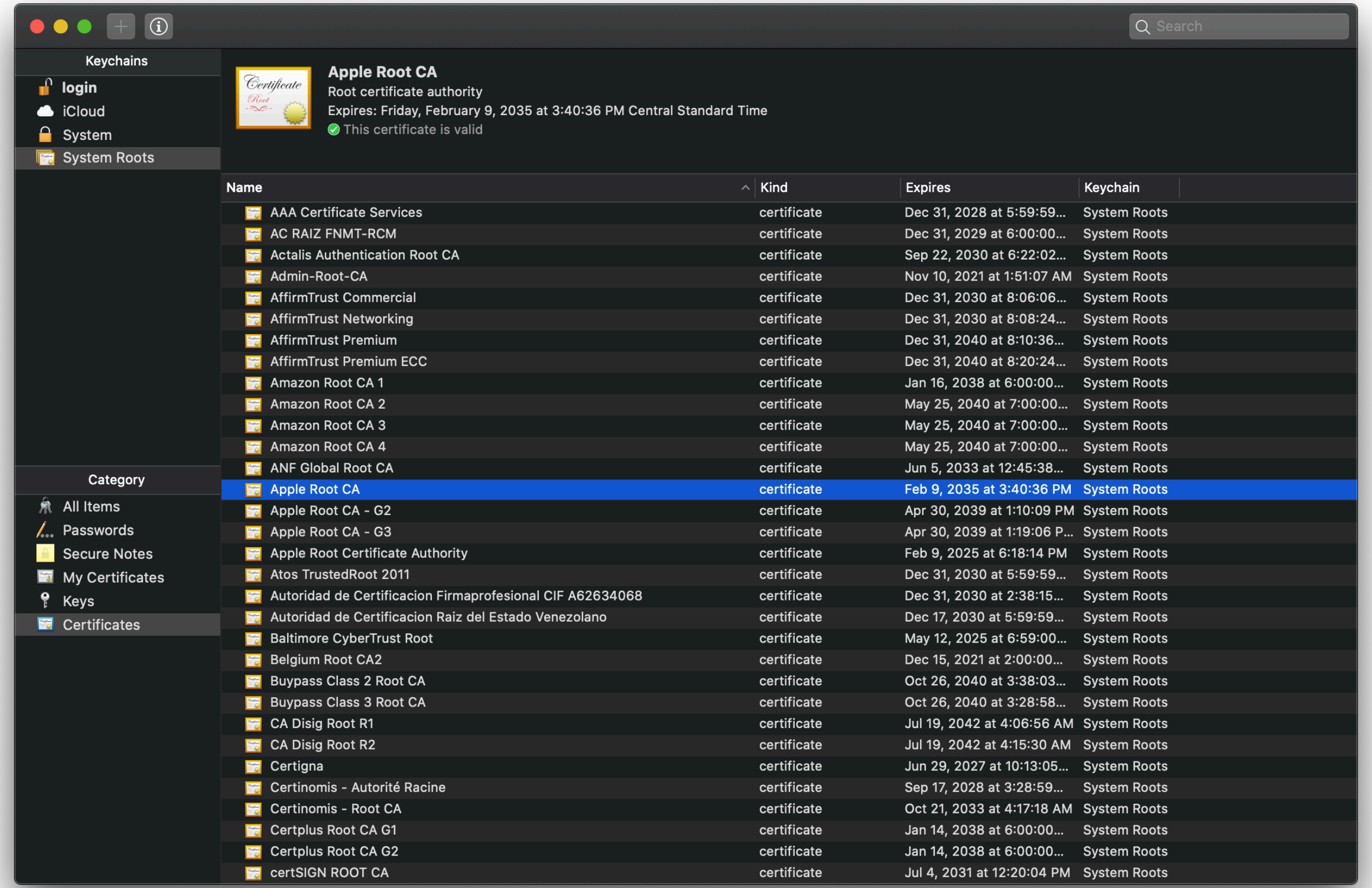
Key Usage Encrypt, Verify, Wrap, Derive

Signature 256 bytes : 11 F9 F9 6D C6 92 D1 B9 E7 13 E6 0D BA E6 19 65 BB 16 4B DE E1 C2 3A 62 55 D1 61 80 93 F0 2A B2 7D 9E 76 CE 10 4A D6 96 4E 5C 00 5D BD 8C 83 74 CF C1 14 91 2B 15 4B 2D 67 4A 84 A2 A4 54 7A B1 C9 8E F5 A7 93 8D 30 BF 0C 9B EF 98 36 D6 4B BD B6 11 63 C2 51 23 71 7B 8D 4C 9B B7 AD A9 FE A8 4E 48 B2 83 A1 36 75 97 2B 36 4A 72 C4 AA C6 B6 A8 4A C0 F4 37 BD 0E 85 B1 A8 FB EC B6 B5 BB A8 C2 C0 BB B7 47 D7 D4 DB 05 80 72 BA CB C7 79 81 63 CC 55 D7 68 9C 41 2B E7 D9 F0 C2 8F 11 15 7D C5 D5 34 27 5C 7C B5 D9 A8 3F 3C DF C5 1D AA 52 03 19 AE 5B FC FF 42 68 15 A3 01 CB F8 0E FE 9B A1 76 B8 43 1C 6B 9C 57 38 87 81 3B 4A 33 98 09 CF 25 F4 75 34 AE 1E 7B CD 0F EF A0 4C 5B 92 B7 F1 FD 66 1B 49 67 B0 65 5A 90 1D 1D 54 D2 CF FF FD 07 DC 7A 88 56 51 55 16 7F 83 D4 FC 19 F4 28

X.509 Certificates

- Certs include several pieces of information:
  - Cert's Serial number
  - Cert's Expiration Date
  - Common name of subject (ex: **`uchicago.edu`**)
  - Public key of subject
  - Extensions (possibly many)
  - CA info (public key, name, etc)
  - CA's signature on cert

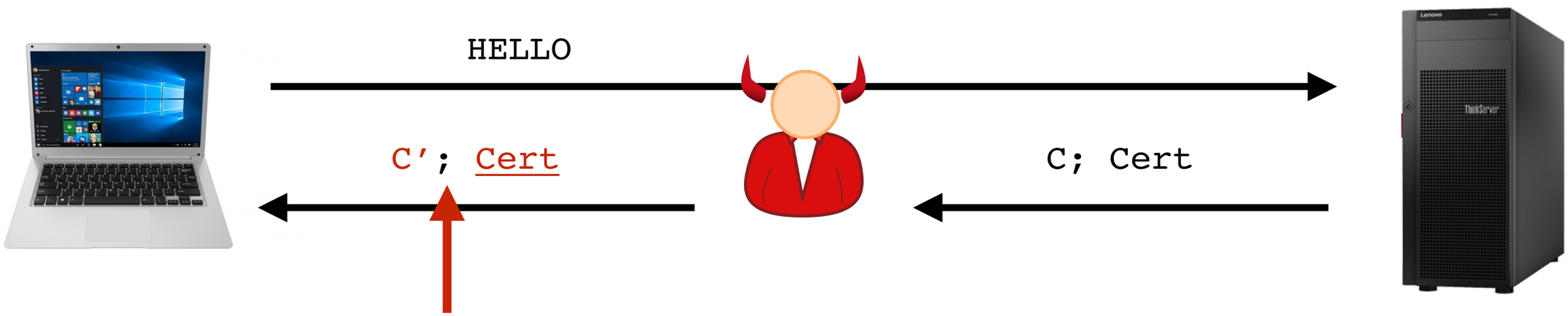
Root Certificates



Outline of Lecture 7

1. Digital Signatures
2. Putting Everything Together: Secure Channels
3. Certificates
4. **Attacks on Certificates, and Countermeasures**

What if you had “valid” cert for uchicago.edu?



- “Man-in-the-middle” can read/change all traffic undetected

TECHNOLOGY | August 31, 2011

Google warns of man-in-the-middle attacks on Iranian users

Intrusion into Dutch SSL provider led to cyber snooping

CA Security

- Can try to fool validation process
- Can also compromise CA organization to get keys or sign certs

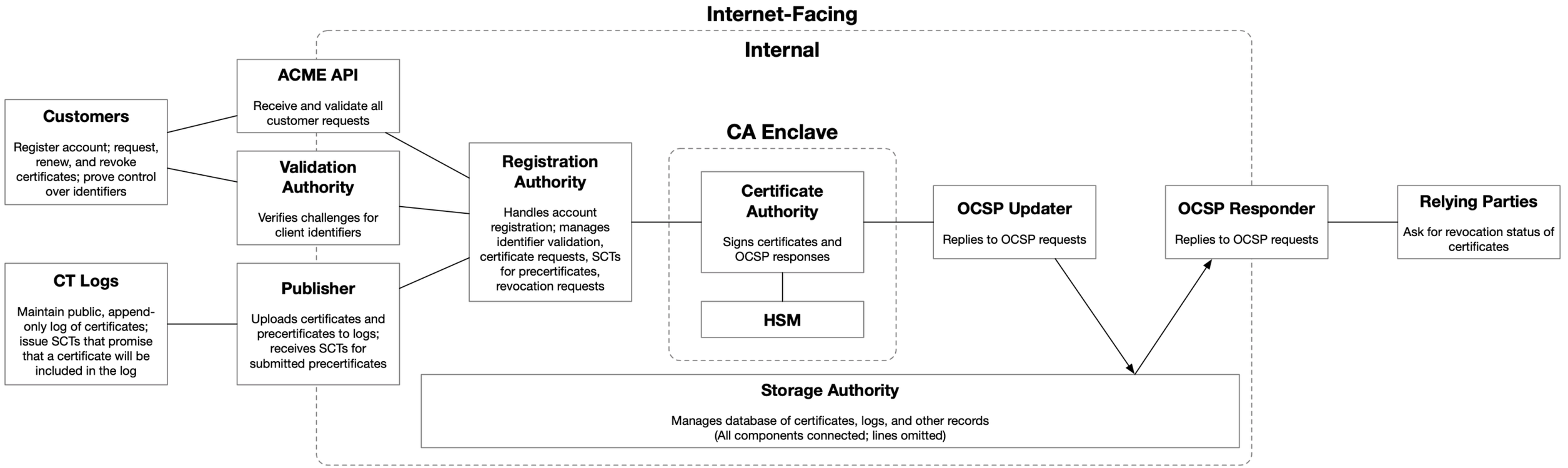


Figure 3: Boulder architecture. Let's Encrypt developed and operates a Go-based open-source CA software platform named Boulder, which is composed of single-purpose components that communicate over gRPC, as illustrated here. The certificate lifecycle unfolds roughly from left to right in the diagram.

"Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web" (CCS 2019)

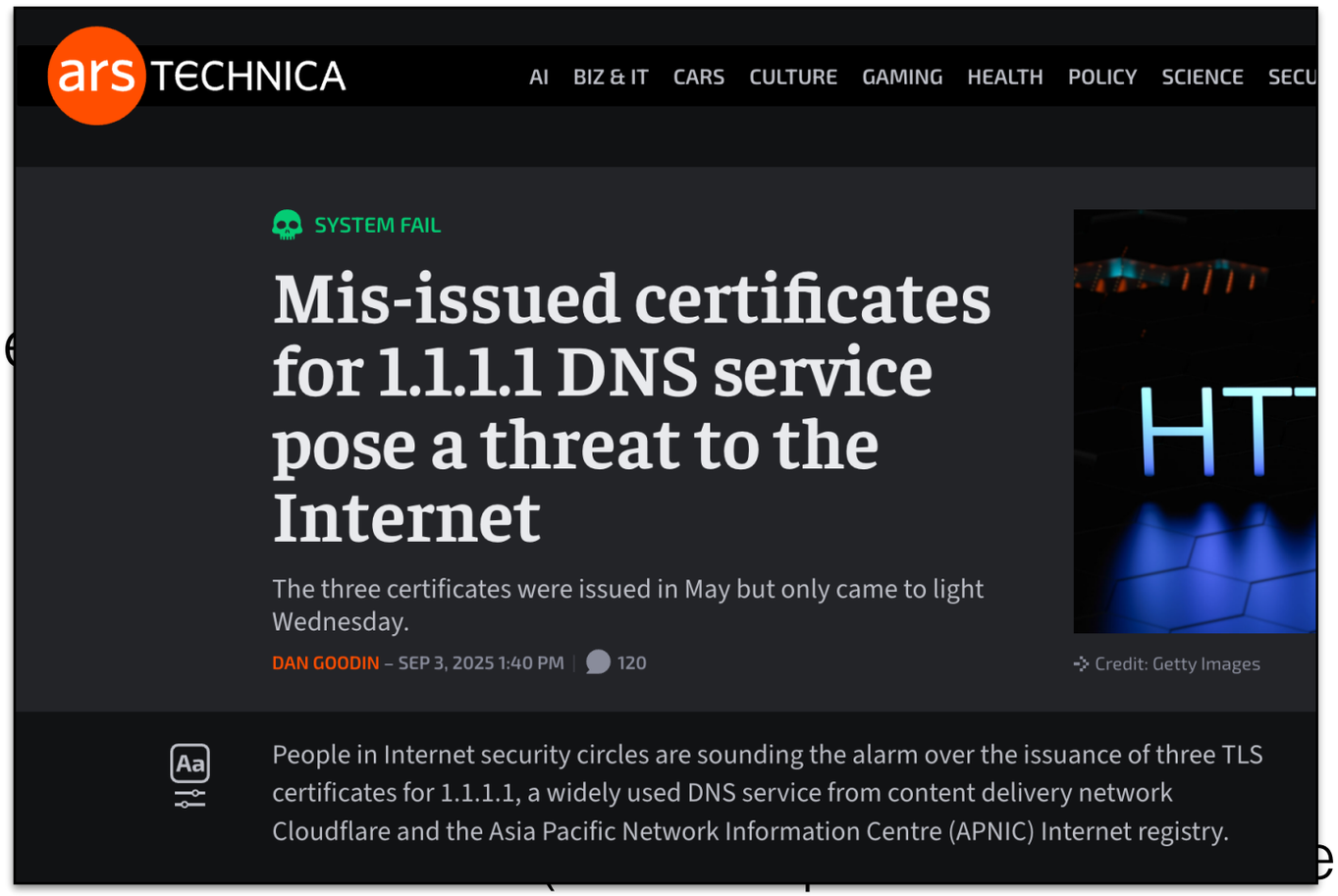
CA Security Incidents (A sampling…)

- 2011, Root CA Comodo: Login credentials stolen. Hacker issues certs for mail.google.com, login.live.com, www.google.com, login.yahoo.com…
- 2011, Root CA DigiNotar: Hacker issues rogue cert for *.google.com, others. Used to MitM by Iranian government.
- 2013, Root CA TurkTrust: Accidentally issues intermediate CA cert, used to issue gmail.com cert.
- …
- 2019, Root CA Comodo: Pushes email login credentials to public GitHub repo…

(Slide inspiration: Dan Boneh)

CA Security Incidents (A sampling…)

- 2011, Root CA Comodo: Login credentials stolen. Hacker issues certs for mail.google.com, login.live.com, www.google.com, login.yahoo.com…
- 2011, Root CA DigiNotar: Hacker issues rogue cert for *.google.com, others. Used to MitM by Iranian government.
- 2013, Root CA TurkTrust: Accidentally issues intermediate CA cert, used to issue gmail.com cert.
- …
- 2019, Root CA Comodo: Pushe
repo…



eh)

Countermeasure: Public-Key Pinning

- **Goal**: Limit damage of less trust-worthy CAs

- Server will tell client to only accept certs from certain CAs
  - E.g. Google tells browsers to only accept keys from their CAs
  - Can also pin in apps: Google apps only accept their own CAs
  - Helped discover some rogue certs from previous slide

- But… if site gets hacked… attacker can pin a malicious cert!
- Deprecated now.

Countermeasure: Revocation

- Explicitly list revoked certificates so they are no longer accepted

Mass Revocation: Millions of certificates revoked by Apple, Google & GoDaddy

The DarkMatter debate is already having industry-wide ramifications

Millions of SSL/TLS certificates – among other digital certificates – are being revoked right now as a result of an operational error that caused the generation of non-compliant serial numbers.

March 3, 2020

0

Let’s Encrypt to Revoke 3 Million SSL Certificates on March 4

The world’s leading free SSL provider announces that millions of certificates are being revoked due to a bug they discovered days ago – giving subscribers potentially only hours to respond

Certification Revocation: Cert. Revocation Lists (CRLs)

CA's CRL Server

Revoked serial numbers:

09823342365

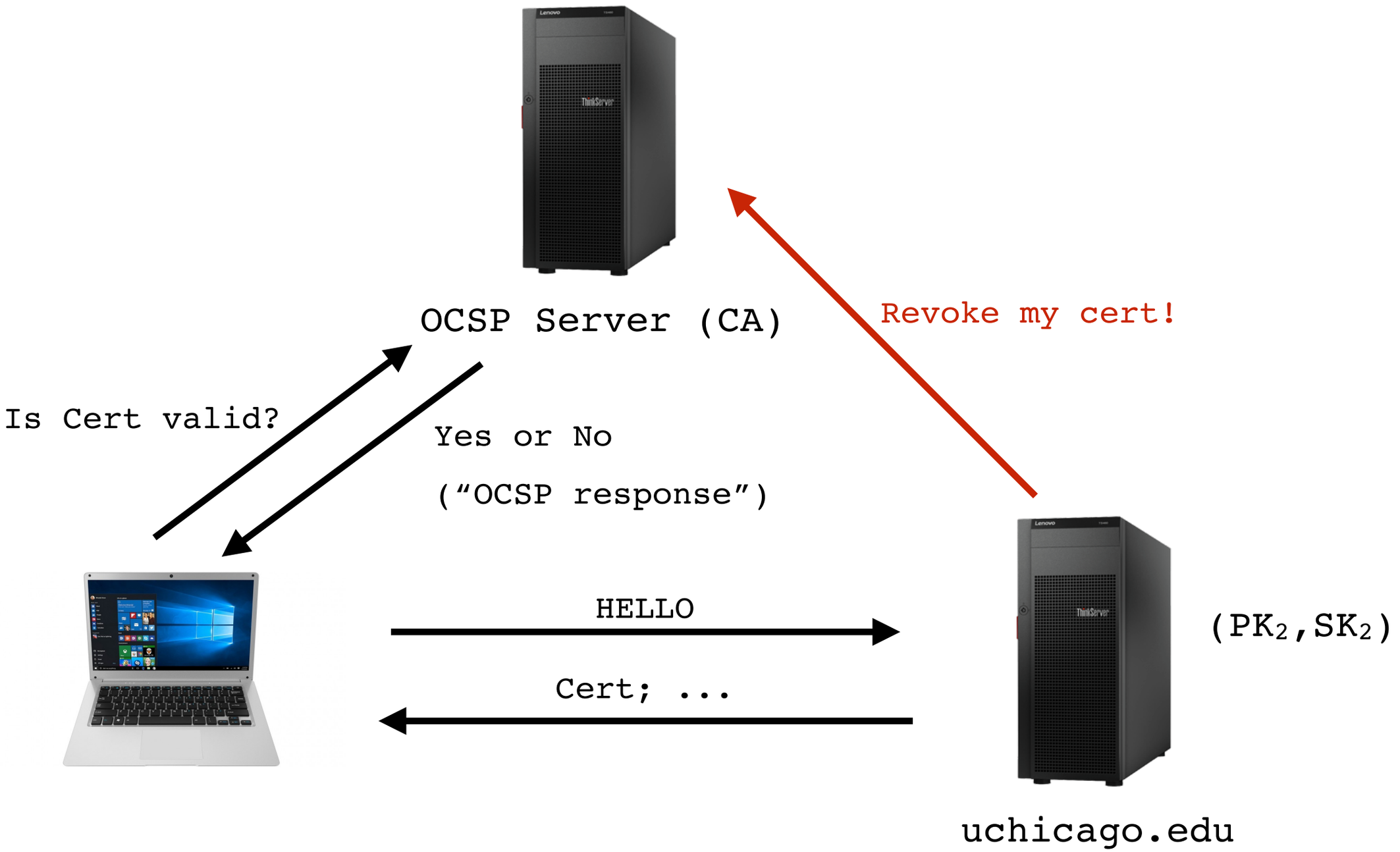
23423482349

98072344456

…

- Each CA provides list of revoked certs
- Problems:
  - CA provides list of revoked certs
  - List will get big, hard to keep current

Online Certificate Status Protocol (OCSP)



- Add another server to connect to, slowing connection
- What if OCSP server times out?
- Privacy problem?

OCSP Stapling

OCSP Server (CA)

Is Cert valid?

Yes or No

(“OCSP response”)

HELLO

Cert; OCSP response; ...

(PK_2, SK_2)

uchicago.edu

- TLS Extension that allows for OCSP response to be included with cert
- Client checks CA signature and time-stamp on response (~hours old).
- Certs can have “must staple” extension.

OCSP Stapling

OCSP Server (CA)

Is Cert valid?

Yes or No

(“OCSP response”)

HELLO

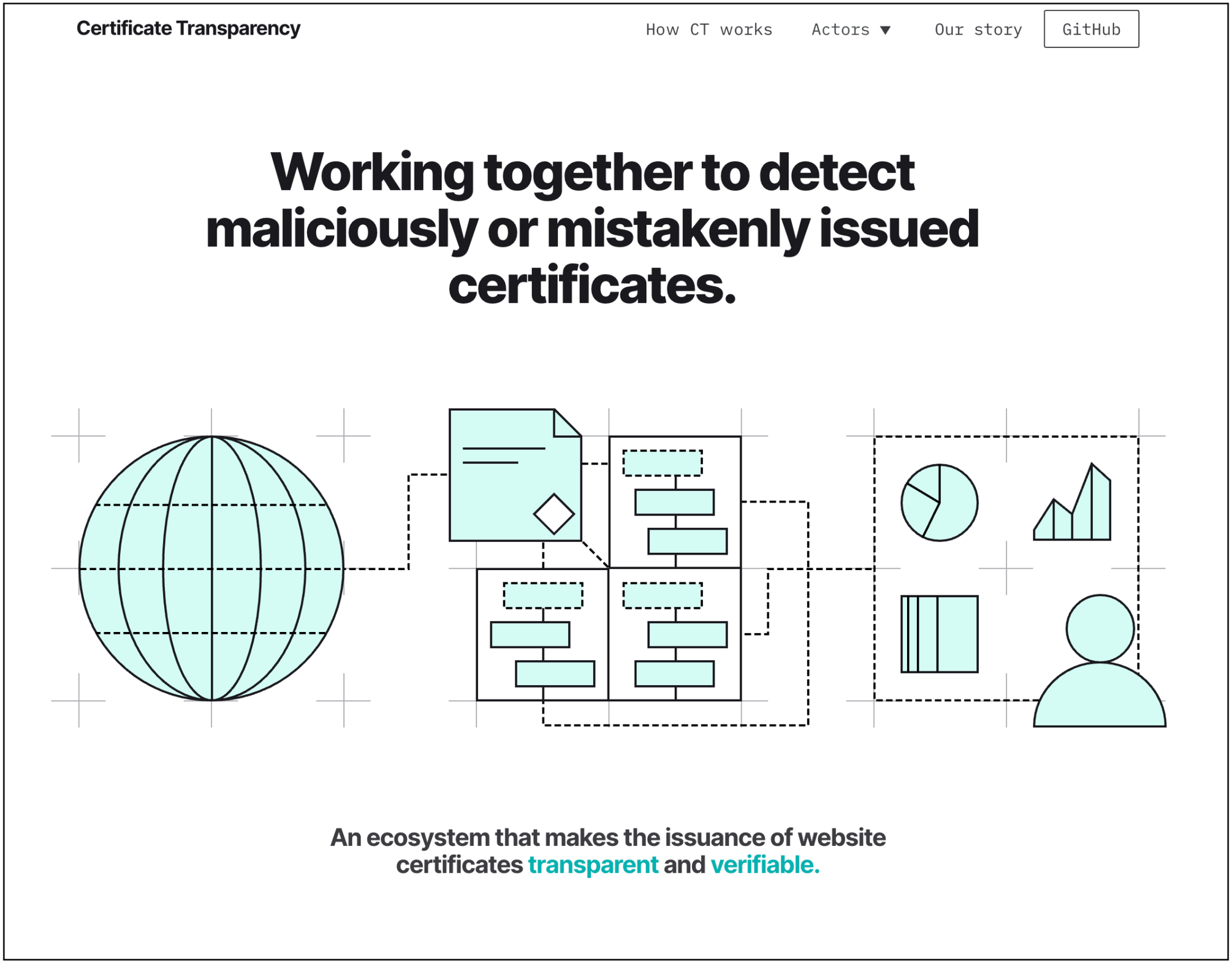
Cert; OCSP response; ...

(PK_2, SK_2)

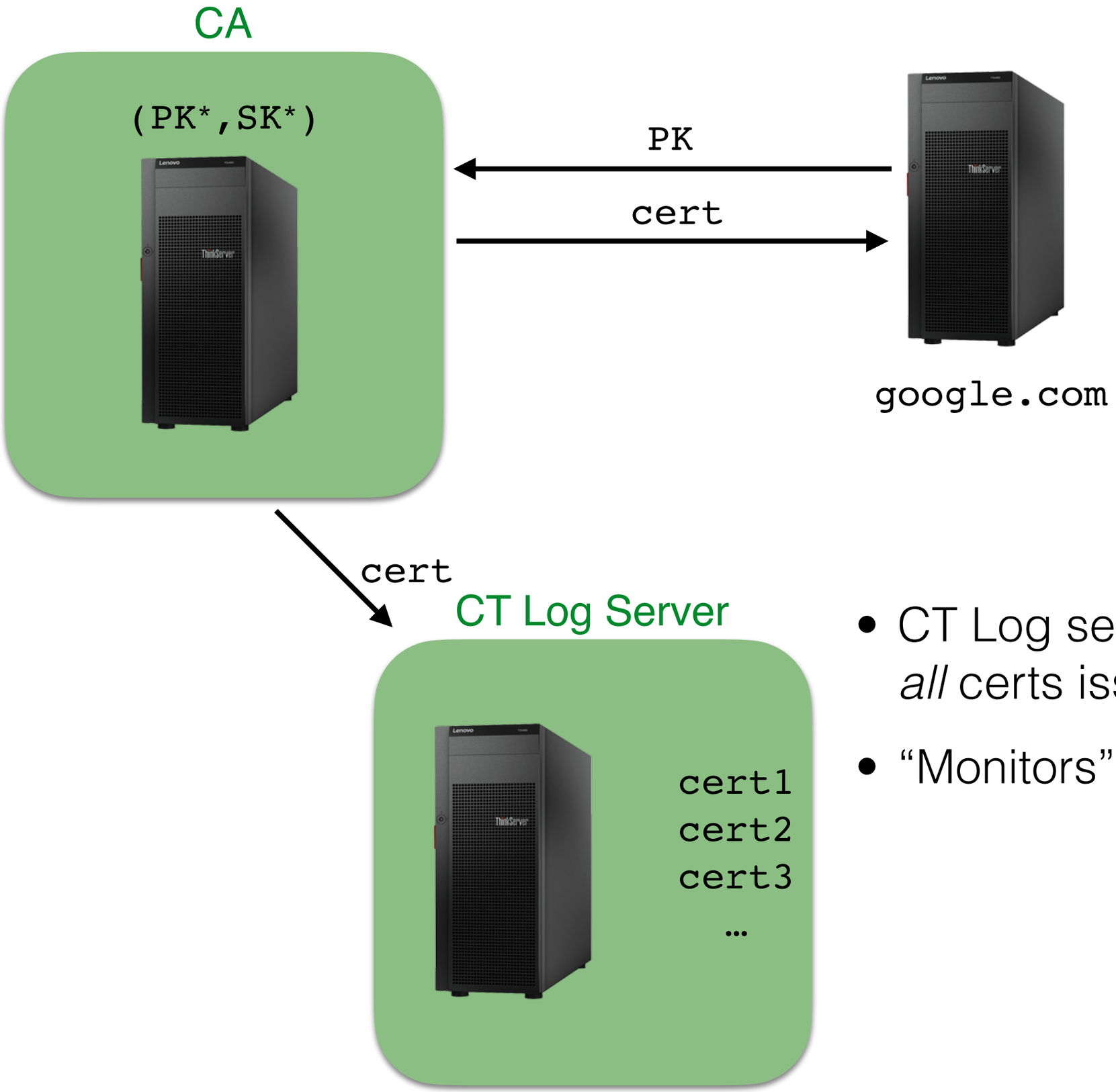
uchicago.edu

- Problems?
  - OCSP server goes down => uchicago.edu goes down (or ignore)

Certificate Transparency (CT)

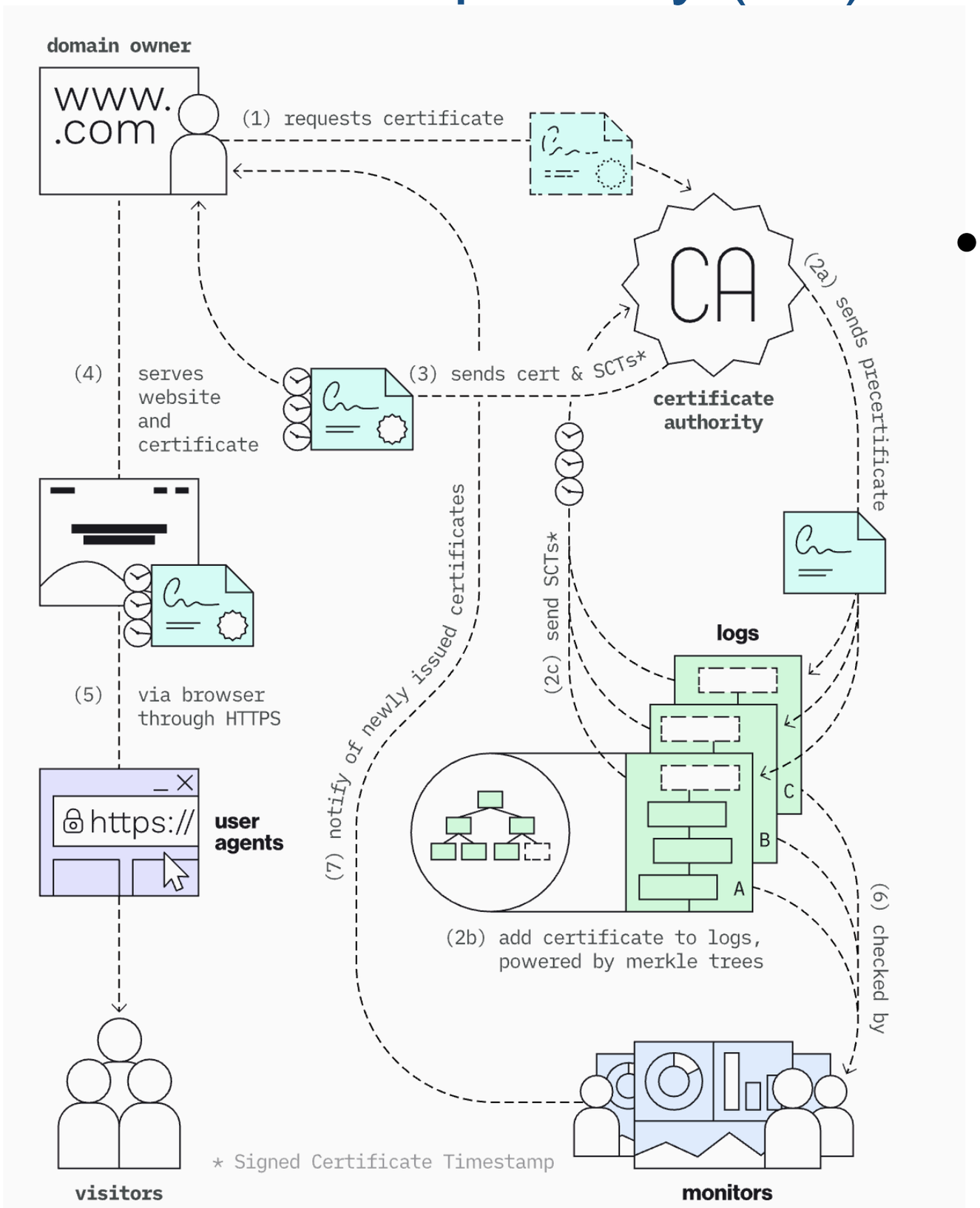


Certificate Transparency (CT)



- CT Log server maintains a list of *all* certs issued by CA(s).
- “Monitors” check for improper certs.

Certificate Transparency (CT)

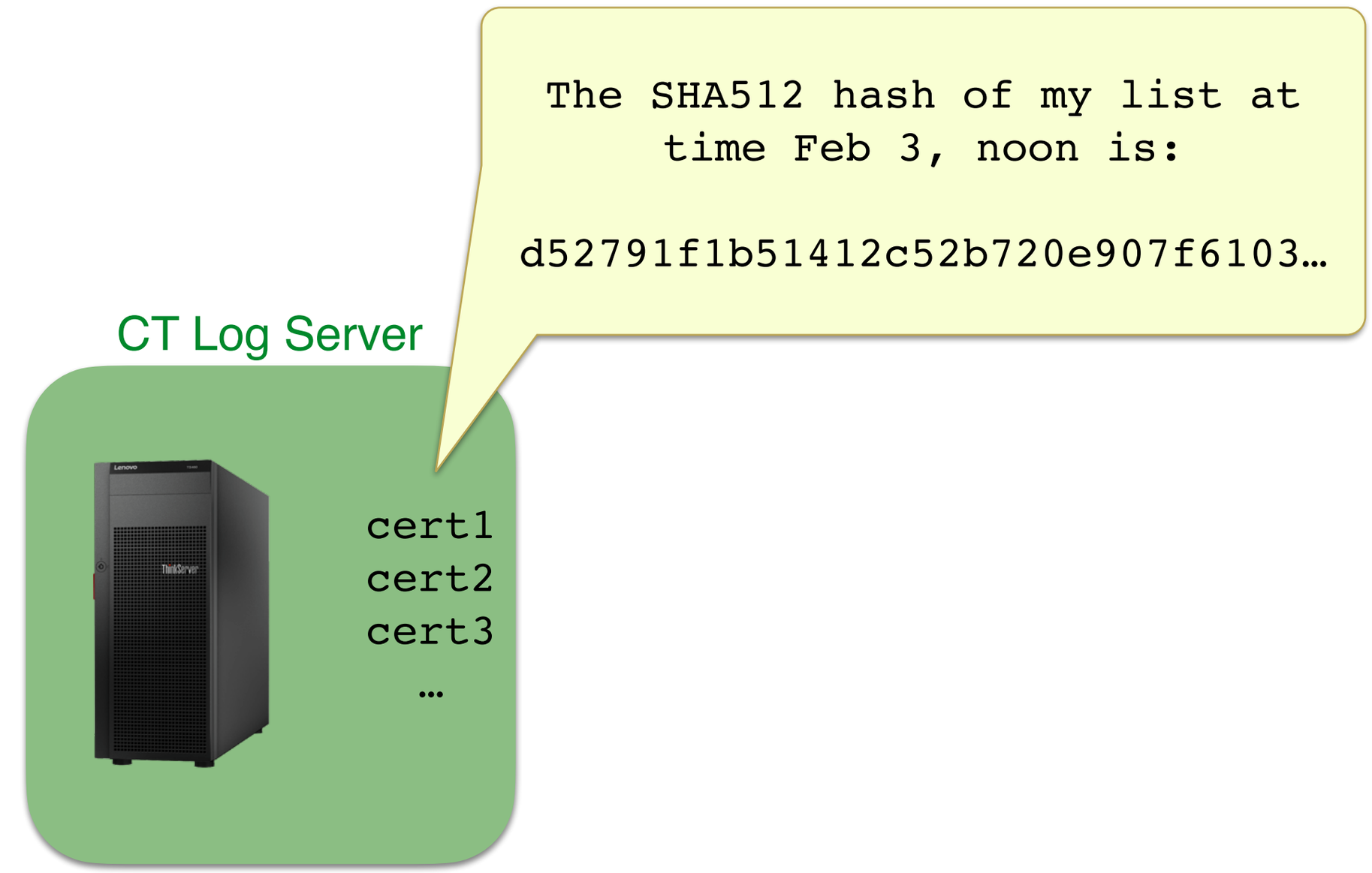


(https://certificate.transparency.dev/howctworks/)

- How do CT and OCSP compare?
  - What problems do they solve?

Challenges with CT

- List is huuuuge
- Trust the CT Log?
- Who checks?
- Privacy?



Software Bugs Can Undo All of This

[Moxie'2009]

- Step 1: Blase requests cert for domain

`google.com\0.blaseur.com`

- Step 2: Blase can validate this domain; He owns `blaseur.com`
- Step 3: Blase MitM's a victim, and presents his cert as a `google.com` cert
- Result: Browser runs

```
strcmp("google.com", "google.com\0.blaseur.com")
```

which returns 1, accepting cert.

The End