Lecture 7: Public-key Infrastructure

6.1600 - MIT

Fall 2022

Corrigan-Gibbs, Kaashoek,
Zeldovich

Plan

* Recap: Digital Signatures
* Signatures in practice
* Public-key infrastructure (PKI)
  - API / Goal
  - Common strategies
  - Common pitfalls

[Set up laptop.]

Logistics

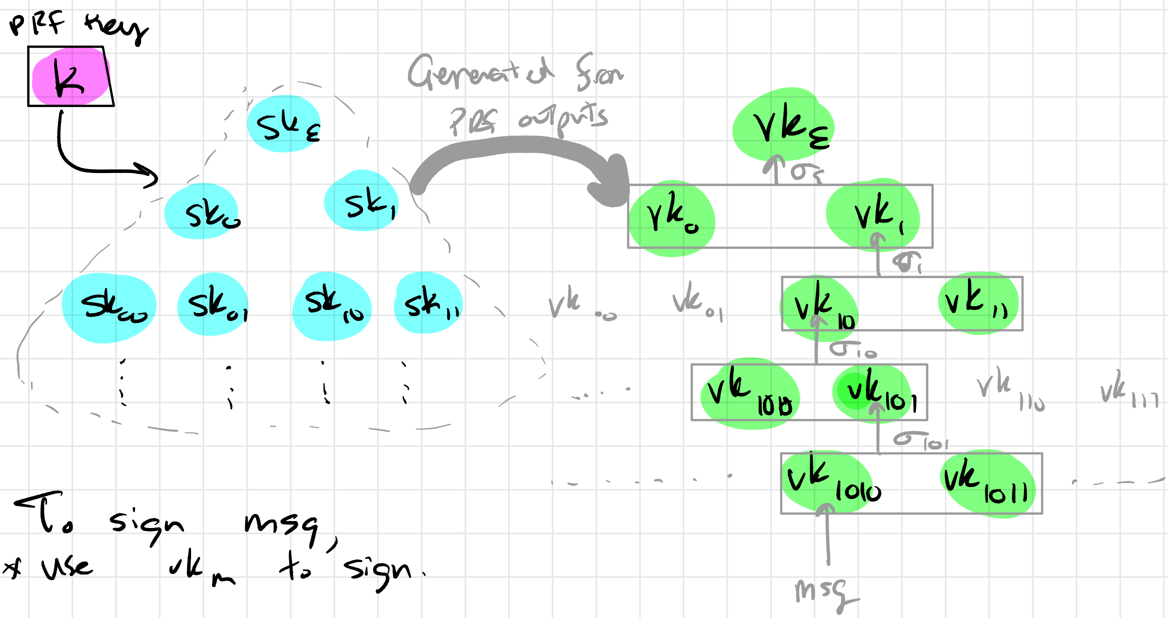
* Lab 1 theory & code due tomorrow 10pm ET
* Lab 2 out on 9/30

Recap: Digital Signatures

Key idea: Message integrity w/o shared secret
↳ unlike MAC or password-based auth
↳ really revolutionary — no shared secret!

(Gen, Sign, Verify)

Hash-based signature (unbounded msg len, many time sec)



To sign msg,
* use vk_m to sign.
* Return all vks on path to root with siblings
* Use sk_i to sign $(vk_{i10} \| vk_{i11})$
* Return all signatures.

[See lecture notes for a more formal description.]

Signatures in practice (briefly)

- One of the most widely used crypto tools
  * HTTPS
  * Software updates
  * Encrypted messaging
  * SSH
  * VPN
  * Essentially any protocol that sends msgs over the Internet

- Two widely used protocols ... both use "hash & sign"
  - ↳ RSA (classic, ... going away)
  - ↳ EC-DSA + friends (extremely popular)

  (both based on hard problems in number theory)

Choice of sig schemes

PQ Standard

	Pk size	Sig size	sign/s	ver/s
SPHINCS+-128 (~2010s)	32B (sk: 64B)	8000B	5	750
RSA-2048 (~1970s)	256B (sk: ")	256B	2,000	50,000
ECDSA 256 (Schnorr, Ed25519) (~1990s)	32 B (sk: ")	64B	42,000	14,000

← Short msg

SPHINCS+: similar to what we saw w/ siglen opts

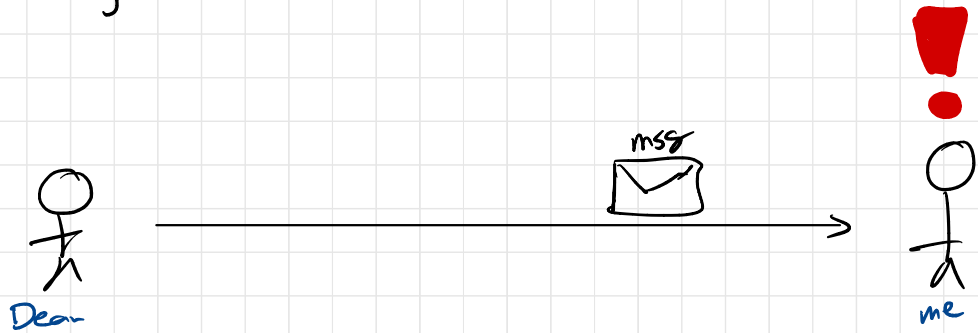
RSA, ECDSA: widely used

SHA256 Hash 64 bytes ≈ 10,000,000/s

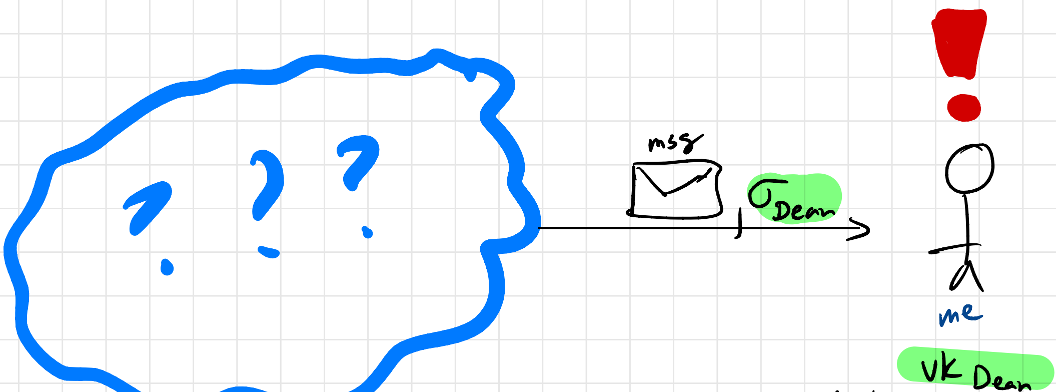
- 99% of time, use ECDSA (or modern variant)
- In rare cases, want to choose a diff scheme.
  - * Post-quantum security (RSA and ECDSA aren't! Hash-based sigs seem to be. Also lattice-based.)
  - * Extra features: aggregation, blind signing, etc...

Public-key infrastructure (PKI)

Last year



The right image...



$$Ver(vk_{Dean}, msg, \sigma_{Dean}) \stackrel{?}{=} 1$$

How do I know it was dean who sent me this email?

Now that we have signatures, answer is clear! (add vk_{Dean})

But where do we get vk_{Dean}?

Option: Use public key as name.

Dean's "name" is the vk.

Instead of calling him "Dan",

call him 0x2EEC9DB33.....0668

32 bytes

- Can imagine that at birth, we're each given an (sk, vk) pair. Everyone calls us by vk.

This sort-of works! Used in Bitcoin & friends, also Tor hidden services, ...

Problem: Cumbersome. Hard to remember 32B names.

↳ PKI

Problem: What happens if you lose your secret key? Or if it gets stolen? Or you realize you generated it incorrectly?

↳ Revocation

Crypto bugs :(

PKI is all about mapping...

human-intelligible names to **public keys.**

- email addr
- domain name
- legal entity
- phone #
- kerberos ID

Can think of PKI as having the API (grossly simplified)

$$\text{IsKeyFor}(vk, \langle name \rangle) \to \{0, 1\}$$

* Many many ways to implement a PKI.
  ... we will see some.
* But all serve this same purpose.
* No "perfect" solution here — lots of trade-offs.

We will look at a few common schemes...
* key as name, TOFU, cert based

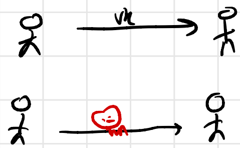
Trust on first use (TOFU)

→ Accept only first key you see for a name.

Client keeps a cache = {} ← dictionary/hash table

```
IsKeyFor(vk, name):
    if name not in cache:
        cache[name] = vk
        return true
    else:
        return vk == cache[name]
```

Used in SSH, Signal, WhatsApp
(Could use this in my email example. Protection if have already gotten email from Dean)



Pros:

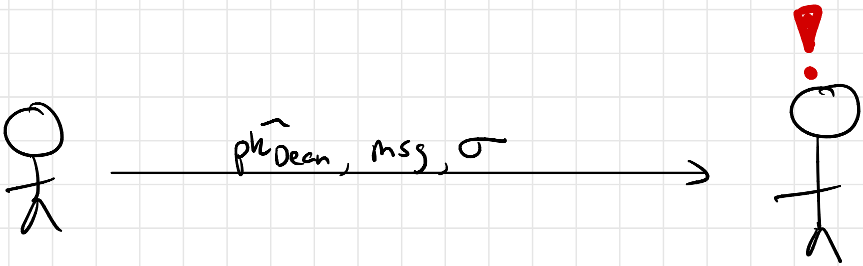
- Simple
- Easy to understand
- Surprisingly effective - protects you against an attacker that hijacks 2nd connection.

Cons:

- No protection on first communication
- What happens when key changes?
  ↳ SSH: Warn ... then what?

Trust on first use (TOFU)

→ Accept only first key you see for a name.



$\{(\text{dean@mit.edu}, pk_{Dean})\}$

Check $\hat{pk}_{Dean} = pk_{Dean}$

Verify σ on msg ...

Certificate-Based System

← Used for HTTPS...

→ Let certification authorities (CAs) manage name → key mapping

Client keeps a list of known CAs' verif keys.

$CAs = \{vk_{Verisign}, vk_{Google}, \ldots\}$

List of CAs is packaged with browser/OS.

⇒ Client accepts (vk, name) pair iff known CA signed it.

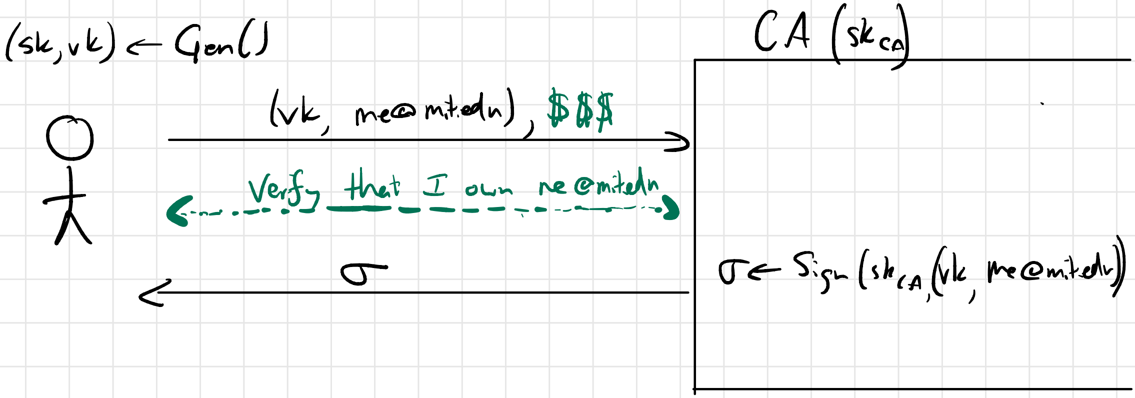
↳ CAs "attest" to name → vk mappings.

```
IsKeyFor((vk, σ), name):
    For each vk_CA in CAs:
        if Verify(vk_CA, (vk, name), σ)
            return true
    return false
```

When a client generates a new keypair, it must get a CA to sign its vk

[Pub-key certs introduced in 1978 by Loren Kohnfelder in B.S. thesis.]

Certificate Issuance



Common extension: Accept a (vk, name) pair if it's signed by someone whose key was signed by a known CA

Lots of extra metadata in cert: Expiration date, ...

Used on web (HTTPS/TLS), code signing, S/MIME, ...
...also at MIT

Pros:
- Client only needs a few vks — Scales well!
- Client can choose which CAs to trust
- No online interaction w/ CA

Cons:
- Weakest link security — attacker who compromises one CA can impersonate anyone!
- Validation is typically pretty weak... TOFU almost
- Stolen keys?

Demo

- Show cert & chain of trust for mit.edu

- Dump CRL data

```
openssl crl -inform DER -text -noout -in <CRL>
```

- Q: Why intermediate CAs?

There are many variants on certificate-style systems – key directory, web of trust, ...

"Key" idea: To prove (vk, name) binding, I can give you signature on (vk, name) from someone you trust.

Problems with CA-based PKI

1. Any malicious/compromised CA can issue certs for any domain.
   - → Your browser trusts many sketchy CAs (gov'ts, random businesses, etc.)
   - → "AAA cert services" can issue cert for mit.edu... you'll never know

2011: - Diginotar signing key stolen
- Attackers used it to issue cert for google.com
- Used to decrypt Gmail traffic in Iran

- Browsers pull Diginotar from list of known CAs
- Dutch gov't websites break

"Certificate transparency" is one partial answer...

2. Revocation is difficult...

Revocation

- After a CA has issued a cert, it may want to revoke it → make sure clients reject it in the future.

Why?
* site owner has their secret key stolen (Heartbleed) - 2011
* site owner realizes they generated key using bad randomness (Debian bug) - 2008
* MIT student graduates, account inactivated
* Crypto standards change (SHA1, RSA1024, ...)

Approach: Expiration

* Cert has expiration date, clients will reject cert after that date
* If expiration date is not far away, this handles many routine revocation cases

e.g. MIT certs expire June 30 every year.

e.g. Let's Encrypt uses 90-day expiration

Approach: Software vendor (e.g. Mozilla) ships update to client w/ full list of revoked certs.

- window of vulnerability... as long as update latency
- b/w storage cost after wave of revocations

"CRLSet" "CRLite"

Approaches: fallen out of favor

- Certificate revocation list (CRL)
  - ↳ ask CA for list of all revoked unexpired certs
    - expensive after a wave of revocations
    - what happens if can't reach CA server?

OCSP

↳ Ask CA each time you use cert

- browsing history leaks to CA
- CA on critical path of page load

"Stapling" & short-lived cert

Bottom line:

- PKI is about names ⇒ public keys
- Key idea: **Certificates** — signed attestation of name ↦ vk binding
- Key challenge: **Revocation** — stolen key, invalid binding

The End

Recap: Many-time signatures from one-time sigs

(unbounded-length msgs)

Claim: Given
* a PRF w/ keyspace $\mathcal{K}$
* a one-time sig scheme $(\text{Gen}_0, \text{Sign}_0, \text{Ver}_0)$

can construct a 2^t-time secure sig scheme for all $t \geq 0$ where running time of all algs grows as poly(t).

Pf idea: By induction on t

Base case ($t=0$): This is one-time scheme. Done.

Induction: Assume for $t-1$.

$$\text{Gen}_t(): \begin{cases} k \xleftarrow{R} \mathcal{K} \quad // \text{PRF key} \\ (sk_\varepsilon, vk_\varepsilon) \leftarrow \text{Gen}_0^k() \\ \text{output } (k, vk_\varepsilon) \end{cases}$$

Use randomness as $PRF(k, \varepsilon)$

$$\text{Sign}_t(k, m) \begin{cases} (sk_\varepsilon, vk_\varepsilon) \leftarrow \text{Gen}_0^k() \\ (sk_0, vk_0) \leftarrow \text{Gen}_{t-1}^k() \\ (sk_1, vk_1) \leftarrow \text{Gen}_{t-1}^k() \\ \sigma_\varepsilon \leftarrow \text{Sign}_0(sk_\varepsilon, vk_0 \| vk_1) \\ \sigma_m \leftarrow \text{Sign}_{t-1}(sk_{m[0]}, m[1:]) \\ \text{output } \sigma = (vk_0, vk_1, \sigma_\varepsilon, \sigma_m) \end{cases}$$

} Grows linearly with t!

$$\text{Ver}_t(vk_\varepsilon, m, \sigma) \begin{cases} (vk_0, vk_1, \sigma_\varepsilon, \sigma_m) \leftarrow \sigma \\ \text{Ver}_0(vk_\varepsilon, vk_0 \| vk_1, \sigma_\varepsilon) \; \&\& \\ \text{Ver}_{t-1}(vk_{m[0]}, m[1:], \sigma_m) \end{cases}$$

How to detect "rogue" CA?

- Have client software look for certain misbehavior
  e.g. Chrome has list of Google pks hardcoded
  If CA issues a rogue Google cert, Chrome will (I believe) notify Google
  - ↳ Doesn't really solve the problem. Only works for friends of Google
  - ↳ If client knew what the right cert was, wouldn't need PKI.

Certificate Transparency (Some browsers, sort of)

- Require CAs to publish all certs they sign in a public log ... many logs run by many different orgs
- mit.edu can inspect logs regularly to make sure that no CA has issued rogue certs for its domains
- In theory, when browser gets a cert from a web server, it can "audit" the cert by checking that it appears in the log.
- Lots of messy implementation details
  - ↳ prevent logs from cheating
  - ↳ ensure that everyone sees same log
  - ↳ ensure that client can audit recently issued certs
  - ↳ privacy issues w/ auditing

  ⋮