رمزنگاری، امنیت اطلاعات و حریم خصوصی

ارائه: دکتر سیدعلی لاجوردی

بخش نهم

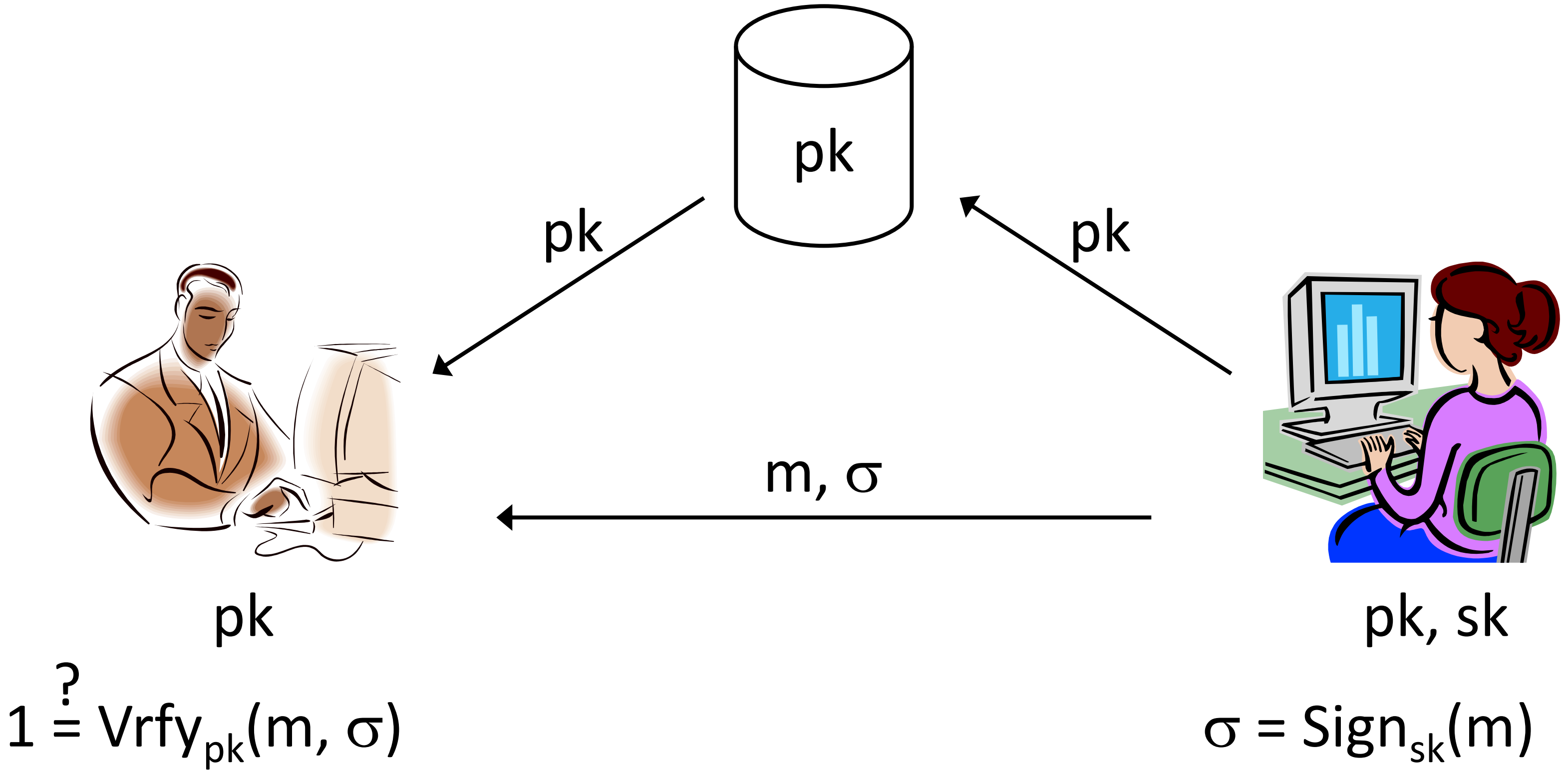
Digital signatures

Digital signatures

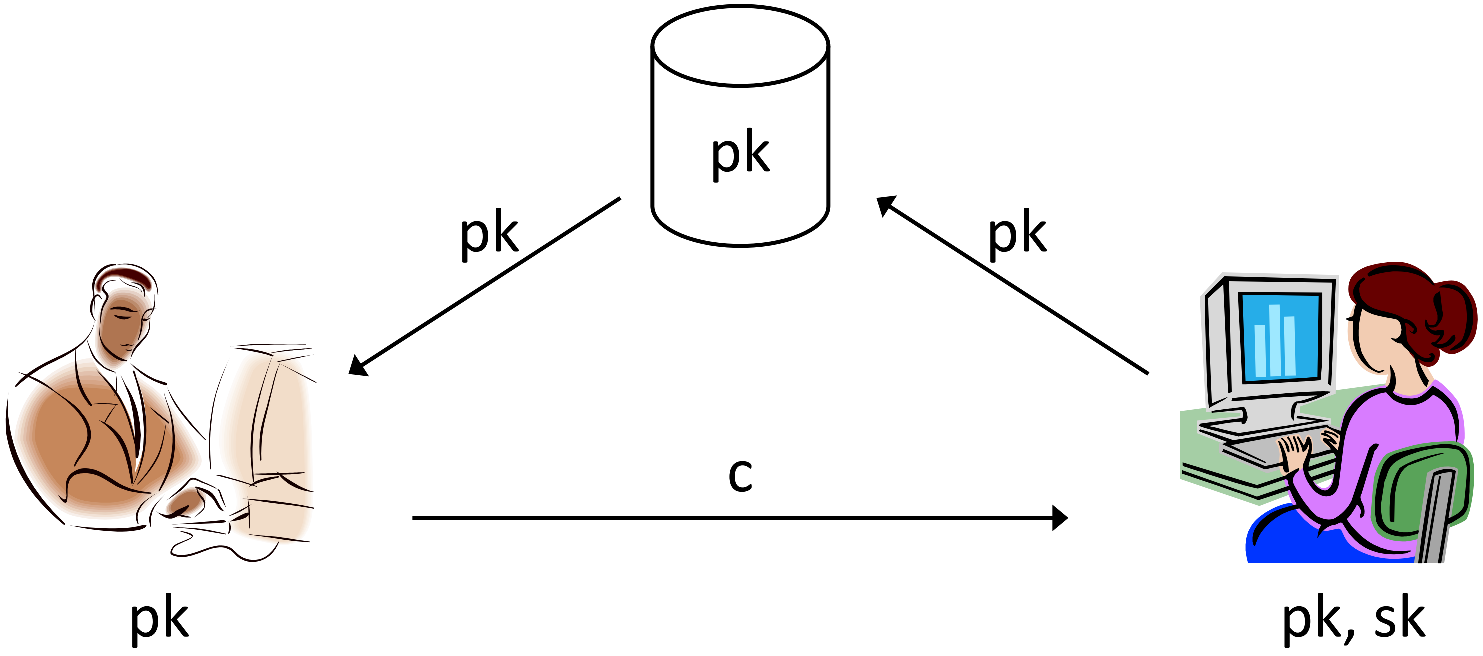
- Provide *integrity* in the public-key setting

- Analogous to message authentication codes, but some key differences…

Digital signatures



Public-key encryption



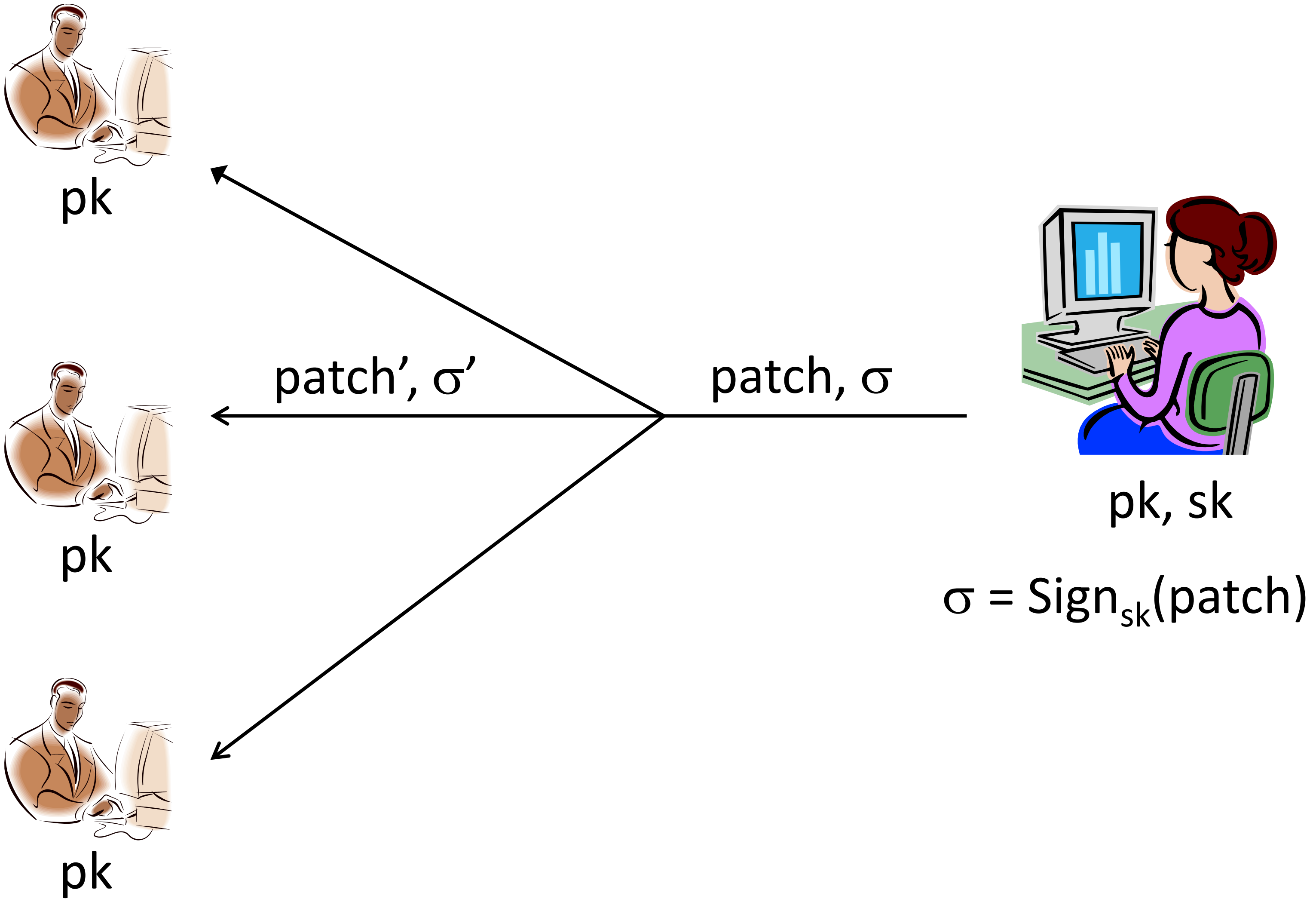
$c \leftarrow \mathrm{Enc}_{pk}(m)$

$m = \mathrm{Dec}_{sk}(c)$

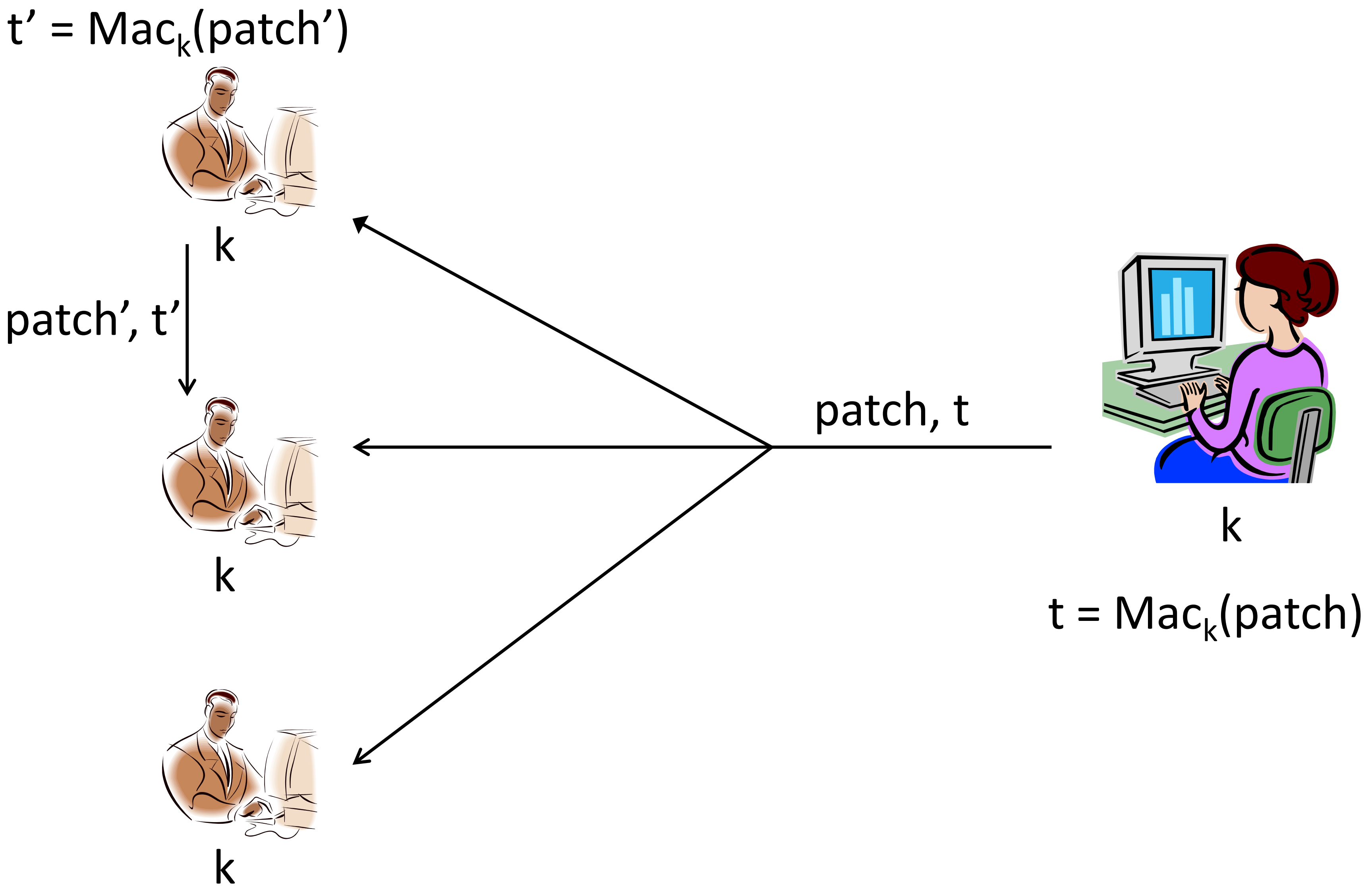
Security (informal)

- Even after observing signatures on multiple messages, an attacker should be unable to *forge* a valid signature on a *new* message

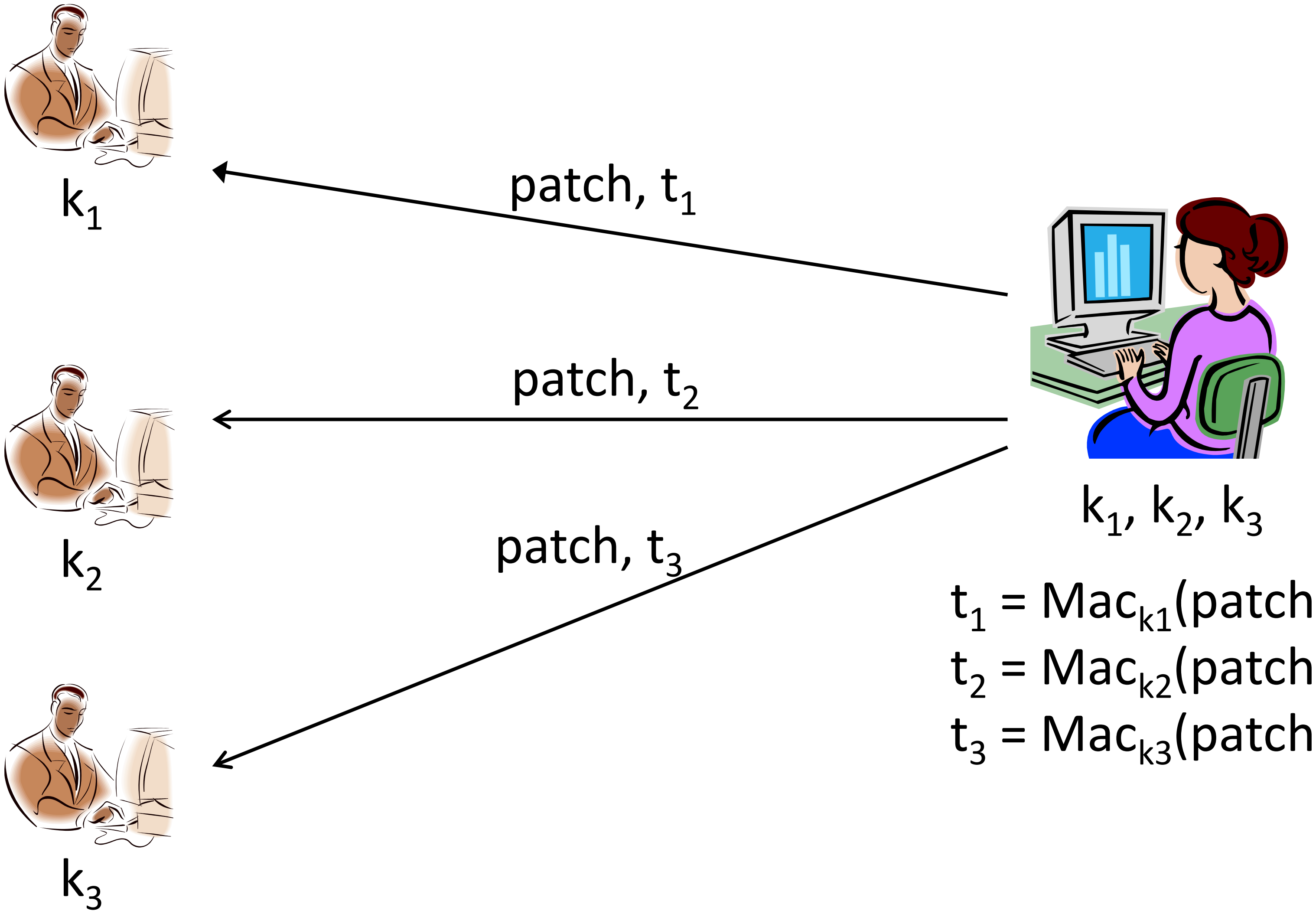
Prototypical application



Comparison to MACs?



Comparison to MACs?



$t_1 = \mathrm{Mac}_{k1}(\text{patch})$
$t_2 = \mathrm{Mac}_{k2}(\text{patch})$
$t_3 = \mathrm{Mac}_{k3}(\text{patch})$

Comparison to MACs?

- *Public verifiability*
    - “Anyone” can verify a signature
    - (Only a holder of the key can verify a MAC tag)

$\Rightarrow$ *Transferability*

- Can forward a signature to someone else…

$\Rightarrow$ *Non-repudiation*

Non-repudiation

- Signer cannot deny issuing a signature
  - Crucial for legal applications
  - Judge can verify signature using public copy of pk

- MACs cannot provide this functionality!
  - Without access to the key, no way to verify a tag
  - Even if receiver gives key to judge, how can the judge verify that the key is correct?
    - Even if key is correct, receiver could have generated the tag also!

Signature schemes

- A *signature scheme* is defined by three PPT algorithms (Gen, Sign, Vrfy):
  - Gen: takes as input 1^n; outputs pk, sk
  - Sign: takes as input a private key sk and a message $m \in \{0,1\}^*$; outputs signature σ

$$\sigma \leftarrow \mathrm{Sign}_{sk}(m)$$

  - Vrfy: takes public key pk, message m, and signature σ as input; outputs 1 or 0

For all m and all pk, sk output by Gen,
$\mathrm{Vrfy}_{pk}(m, \mathrm{Sign}_{sk}(m)) = 1$

Security?

- Exactly analogous to security for MACs
- Threat model
  - “Adaptive chosen-message attack”
  - Assume the attacker can induce the sender to sign *messages of the attacker’s choice*
- Security goal
  - “Existential unforgeability”
  - Attacker should be unable to forge valid signature on *any* message not signed by the sender
- Attacker gets the public key…

Formal definition

- Fix A, Π
- Define randomized experiment $\text{Forge}_{A,\Pi}(n)$:
  1. pk, sk $\leftarrow$ Gen(1^n)
  2. A given pk, and interacts with oracle $\text{Sign}_{sk}(\cdot)$; let M be the set of messages sent to this oracle
  3. A outputs (m, σ)
  4. A *succeeds*, and the experiment evaluates to 1, if $\text{Vrfy}_{pk}(m, \sigma)=1$ and $m \notin M$

Security for signature schemes

- Π is *secure* if for all PPT attackers A, there is a negligible function ε such that

$$\Pr[\mathrm{Forge}_{A,\Pi}(n) = 1] \leq \varepsilon(n)$$

Replay attacks

- Replay attacks need to be addressed just as in the symmetric-key setting

Hash-and-sign paradigm

- Given
  - A signature scheme Π = (Gen, Sign, Vrfy) for “short” messages of length n
  - Hash function H: $\{0,1\}^* \rightarrow \{0,1\}^n$
- Construct a signature scheme Π’=(Gen, Sign’, Vrfy’) for arbitrary-length messages:
  - $\text{Sign'}_{sk}(m) = \text{Sign}_{sk}(H(m))$
  - $\text{Vrfy'}_{pk}(m, \sigma) = \text{Vrfy}_{pk}(H(m), \sigma)$

Hash-and-sign paradigm

- Theorem: If Π is secure and H is collision-resistant, then Π' is secure
- Proof: Same as for MACs

- Can be viewed as a counterpart of hybrid encryption
  - The *functionality* of digital signatures at the asymptotic cost of a *symmetric-key* solution

Signature schemes

- We will discuss how to construct signature schemes for “short” messages
    - Using hash-and-sign, this implies signatures for arbitrary length messages

Signature schemes in practice

- RSA-based signatures
  - Can be proven secure (based on RSA assumption, in random-oracle model)
- Dlog-based signatures
  - Shorter signatures, faster signing than RSA-based signatures
  - (EC)DSA
    - Widely used, no proof of security
  - Schnorr
    - Can be proven secure (based on dlog assumption, in random-oracle model)

RSA-based signatures

Recall…

- Choose random, equal-length primes p, q
- Compute modulus N=pq
- Choose e, d such that $e \cdot d = 1 \bmod \phi(N)$

- The e^{th} root of m modulo N is $[m^d \bmod N]$

  $$(m^d)^e = m^{de} = m^{[ed \bmod \phi(N)]} = m \bmod N$$

- *RSA assumption*: given N, e only, hard to compute the e^{th} root of a uniform $m \in \mathbb{Z}^*_N$

“Plain” RSA signatures

$$\xleftarrow{\quad N,\ e \quad}$$

$$\xleftarrow{\quad m,\ \sigma \quad}$$

$m \stackrel{?}{=} [\sigma^e \bmod N]$

$(N, e, d) \leftarrow \mathrm{RSAGen}(1^n)$

$pk = (N, e)$

$sk = d$

$\sigma = [m^d \bmod N]$

Security?

- Intuition
  - Signature of m is the e^{th} root of m – supposedly hard to compute given only the public key!

Attack 1

- Can sign *specific* messages
  - E.g., easy to compute the e^{th} root of m = 1, or the cube root of m = 8

Attack 2

- Can generate signatures on “random” messages
  - Choose arbitrary σ; set $m = [\sigma^e \bmod N]$

Attack 3

- Can combine two signatures to obtain a third
  - Say σ_1, σ_2 are valid signatures on m_1, m_2 with respect to public key N, e
  - Then $\sigma' = [\sigma_1 \cdot \sigma_2 \bmod N]$ is a valid signature on the message $m' = [m_1 \cdot m_2 \bmod N]$
    - $(\sigma_1 \cdot \sigma_2)^e = \sigma_1^e \cdot \sigma_2^e = m_1 \cdot m_2 \bmod N$

RSA-FDH

- Main idea: apply "cryptographic transformation" to messages before signing

- Public key: (N, e)   private key: d
- $\mathrm{Sign}_{sk}(m) = H(m)^d \bmod N$
  - H must map onto all of $\mathbb{Z}^*_N$
- $\mathrm{Vrfy}_{pk}(m, \sigma)$: output 1 iff

  $$\sigma^e = H(m) \bmod N$$
- (This also handles long messages without additional hashing)

Intuition for security?

- Look at the three previous attacks…
  - Not easy to compute the e^{th} root of H(1), …
  - Choose σ…, but how do you find an m such that $H(m) = \sigma^e \bmod N$?
    - Computing inverses of H should be hard
  - $H(m_1) \cdot H(m_2) = \sigma_1^e \cdot \sigma_2^e = (\sigma_1 \cdot \sigma_2)^e \neq H(m_1 \cdot m_2)$

Security of RSA-FDH

- If the RSA assumption holds, and H is modeled as a random oracle (mapping onto $\mathbb{Z}^*_N$), then RSA-FDH is secure

- In practice, H is instantiated with a (modified) cryptographic hash function
  - Must ensure that the range of H is large enough!

RSA-FDH in practice

- The RSA PKCS #1 v2.1 standard includes a signature scheme inspired by RSA-FDH
  - Essentially a randomized variant of RSA-FDH

dlog-based signatures

Digital signature standard (DSS)

- US government standard for digital signatures
  - DSA, based on discrete-logarithm problem in subgroup of $\mathbb{Z}^*_p$
  - ECDSA, based on elliptic-curve groups

- No security proof, even in RO model

- Compared to RSA-based signatures
  - Shorter signatures and public keys (especially for EDCSA)
  - Can have faster signing
  - Slower verification

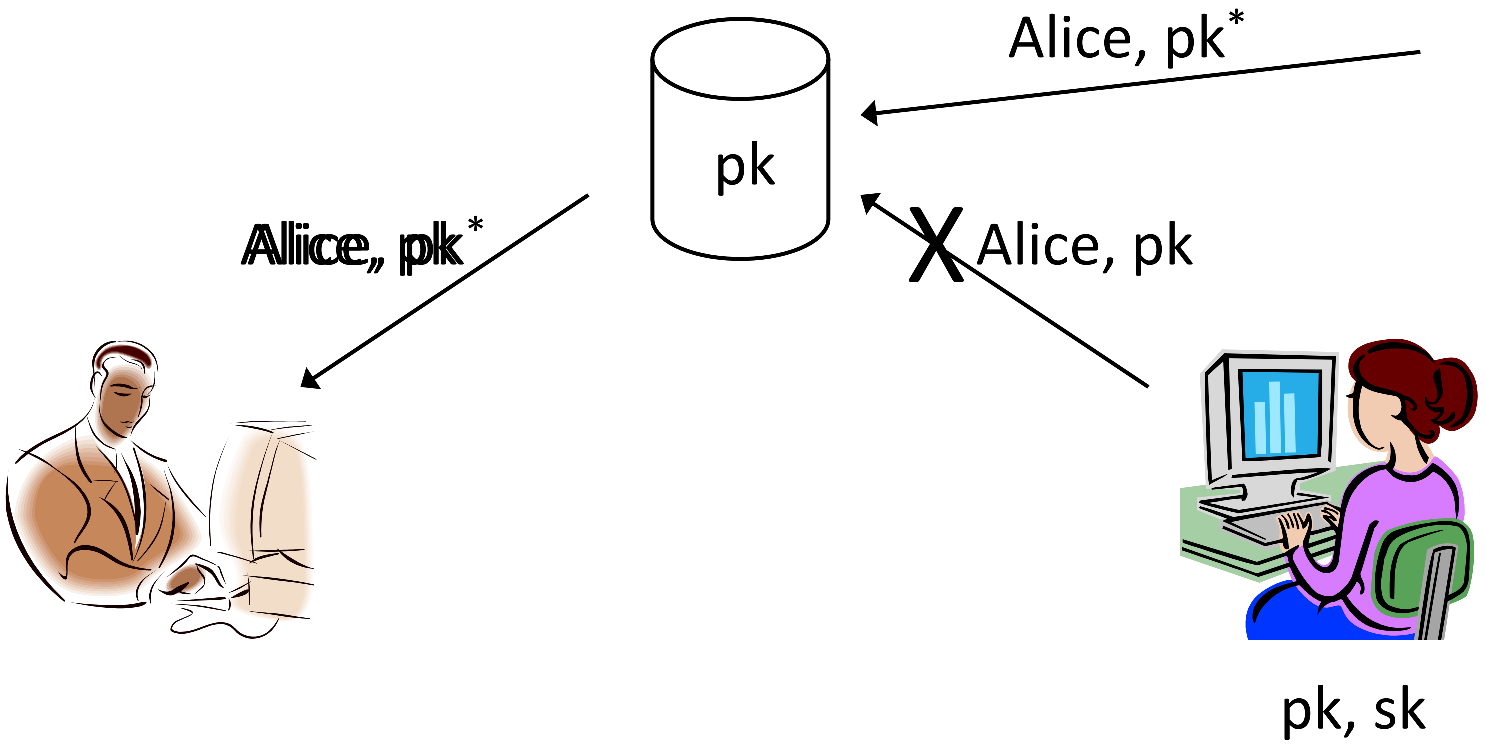
Signatures from identification schemes

- Two signature schemes that can be viewed as being derived from (public-key) *identification schemes*
  - Schnorr
  - DSA/ECDSA

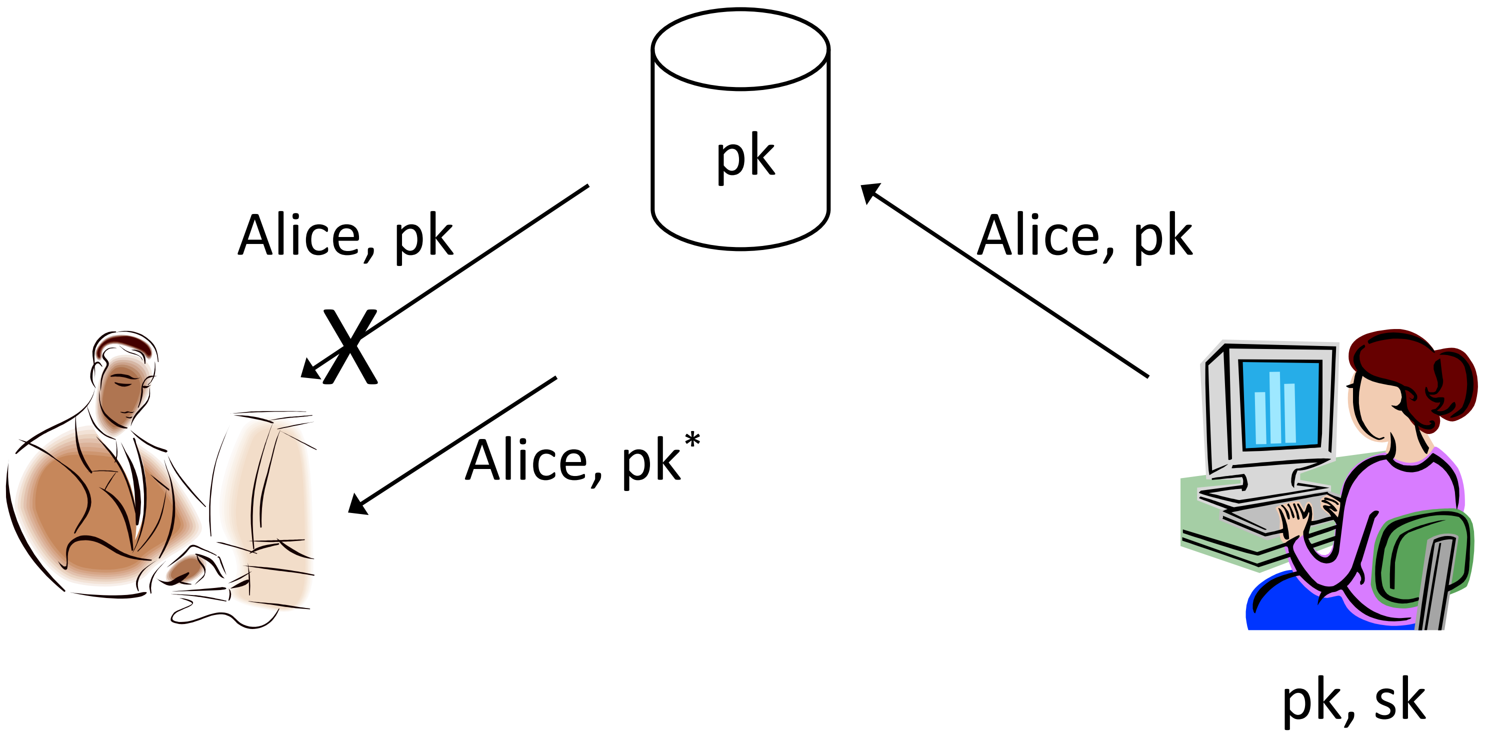
- Will return to this in later lecture

Public-key infrastructure (PKI)

Public-key distribution



Public-key distribution



Use signatures for secure key distribution!

- Assume a trusted party with a public key known to everyone
  - CA = certificate authority who acts as a “root of trust”
  - Public key pk_{CA}
  - Private key sk_{CA}

Use signatures for secure key distribution!

- Alice asks the CA to sign the *binding* (Alice, pk)

$$\text{cert}_{\text{CA}\rightarrow\text{Alice}} = \text{Sign}_{\text{sk}_{\text{CA}}}(\text{Alice, pk})$$

- (CA must verify Alice's identity out of band)

Use signatures for secure key distribution!

- Bob obtains Alice, pk, and the certificate $\text{cert}_{CA \to Alice}$ …
  - … check that $\text{Vrfy}_{pk_{CA}}((\text{Alice}, pk), \text{cert}_{CA \to Alice}) = 1$

- Bob is then assured that pk is Alice's public key
  - As long as the CA is trustworthy…
    - Honest, and properly verifies Alice's identity
  - …and the CA's private key has not been compromised

Chicken-and-egg problem?

- How does Bob get pk_{CA} in the first place?

- Several possibilities…

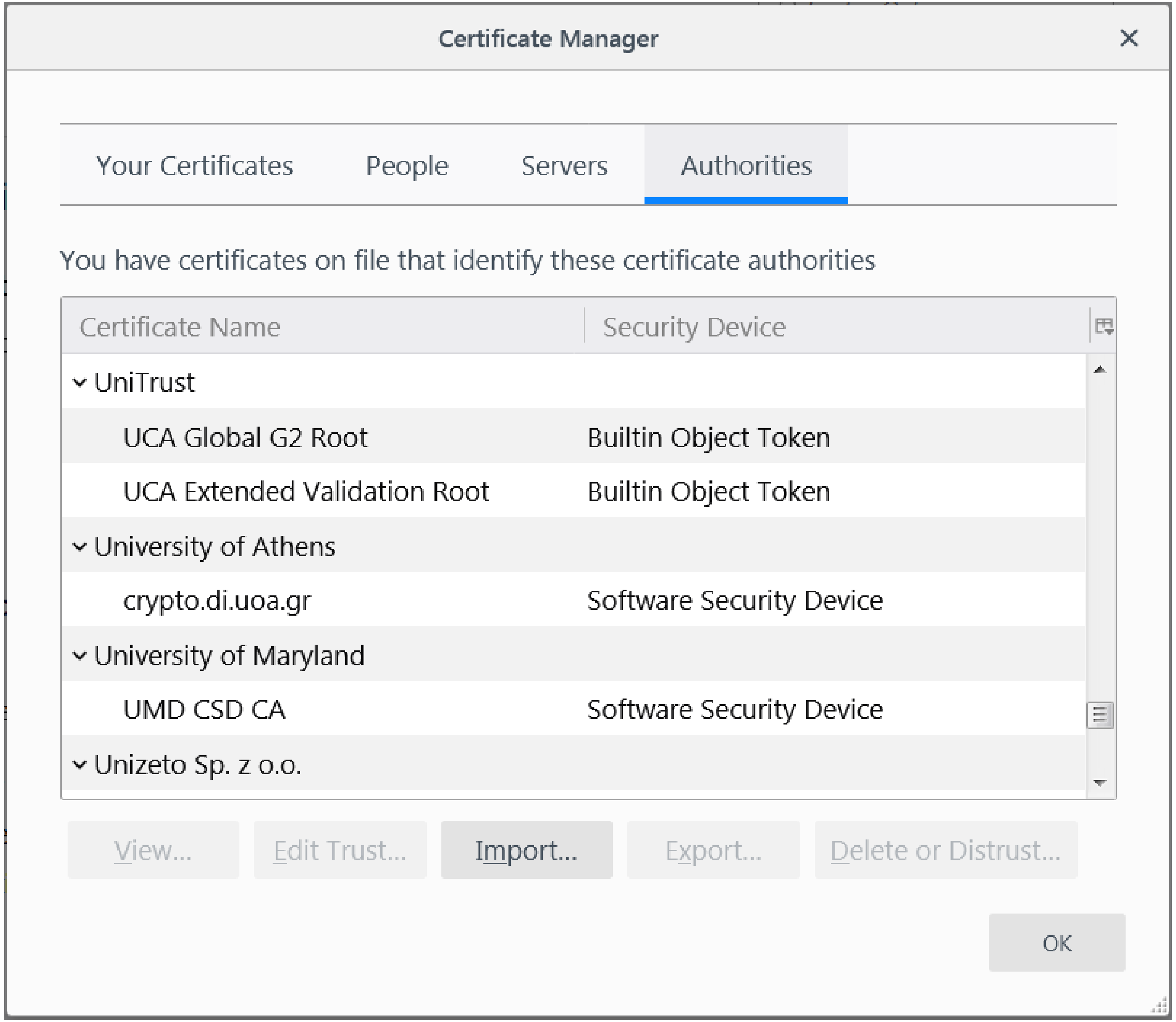
Certificate chains

- Can also have *chains* of certificates
- E.g., Bob holds pk_{CA}
- Alice has pk and $\mathrm{cert}_{CA' \rightarrow Alice}$
- Alice also sends $\mathrm{pk}_{CA'}$ and $\mathrm{cert}_{CA \rightarrow CA'}$ to Bob
- Bob does:
  - Uses pk_{CA} and $\mathrm{cert}_{CA \rightarrow CA'}$ to verify that $\mathrm{pk}_{CA'}$ is the public key of CA'
  - Uses $\mathrm{pk}_{CA'}$ and $\mathrm{cert}_{CA' \rightarrow Alice}$ to verify that pk is the public key of Alice

“Roots of trust”

- Bob only needs to securely obtain a *small number* of CA’s public keys
  - Need to ensure secure distribution only for these few, initial public keys

- E.g., distribute as part of an operating system, or web browser
  - Firefox:
    Settings->Privacy & Security->View Certificates
    ->Authorities


Certificate Manager
Your Certificates
People
Servers
Authorities
You have certificates on file that identify these certificate authorities
Certificate Name
Security Device
UniTrust
UCA Global G2 Root
Builtin Object Token
UCA Extended Validation Root
Builtin Object Token
University of Athens
crypto.di.uoa.gr
Software Security Device
University of Maryland
UMD CSD CA
Software Security Device
Unizeto Sp. z o.o.
View...
Edit Trust...
Import...
Export...
Delete or Distrust...
OK

Public Key Info

Algorithm	RSA
Key Size	2048
Exponent	65537
Modulus	DD:84:D4:B9:B4:F9:A7:D8:F3:04:78:9C:DE:3D:DC:6C:13:16:D9:7A:DD:24:51:66:C0: C7:26:59:0D:AC:06:08:C2:94:D1:33:1F:F0:83:35:1F:6E:1B:C8:DE:AA:6E:15:4E:54:27: EF:C4:6D:1A:EC:0B:E3:0E:F0:44:A5:57:C7:40:58:1E:A3:47:1F:71:EC:60:F6:6D:94:C8:1 8:39:ED:FE:42:18:56:DF:E4:4C:49:10:78:4E:01:76:35:63:12:36:DD:66:BC:01:04:36:A 3:55:68:D5:A2:36:09:AC:AB:21:26:54:06:AD:3F:CA:14:E0:AC:CA:AD:06:1D:95:E2:F8: 9D:F1:E0:60:FF:C2:7F:75:2B:4C:CC:DA:FE:87:99:21:EA:BA:FE:3E:54:D7:D2:59:78:DB: 3C:6E:CF:A0:13:00:1A:B8:27:A1:E4:BE:67:96:CA:A0:C5:B3:9C:DD:C9:75:9E:EB:30:9 A:5F:A3:CD:D9:AE:78:19:3F:23:E9:5C:DB:29:BD:AD:55:C8:1B:54:8C:63:F6:E8:A6:EA: C7:37:12:5C:A3:29:1E:02:D9:DB:1F:3B:B4:D7:0F:56:47:81:15:04:4A:AF:83:27:D1:C5: 58:88:C1:DD:F6:AA:A7:A3:18:DA:68:AA:6D:11:51:E1:BF:65:6B:9F:96:76:D1:3D

Public Key Info

Algorithm	RSA
Key Size	4096
Exponent	65537
Modulus	CA:96:6B:8E:EA:F8:FB:F1:A2:35:E0:7F:4C:DA:E0:C3:52:D7:7D:B6:10:C8:02:5E:B3:43: 2A:C4:4F:6A:B2:CA:1C:5D:28:9A:78:11:1A:69:59:57:AF:B5:20:42:E4:8B:0F:E6:DF:5B: A6:03:92:2F:F5:11:E4:62:D7:32:71:38:D9:04:0C:71:AB:3D:51:7E:0F:07:DF:63:05:5C:E 9:BF:94:6F:C1:29:82:C0:B4:DA:51:B0:C1:3C:BB:AD:37:4A:5C:CA:F1:4B:36:0E:24:AB: BF:C3:84:77:FD:A8:50:F4:B1:E7:C6:2F:D2:2D:59:8D:7A:0A:4E:96:69:52:02:AA:36:98: EC:FC:FA:14:83:0C:37:1F:C9:92:37:7F:D7:81:2D:E5:C4:B9:E0:3E:34:FE:67:F4:3E:66:D 1:D3:F4:40:CF:5E:62:34:0F:70:06:3E:20:18:5A:CE:F7:72:1B:25:6C:93:74:14:93:A3:73: B1:0E:AA:87:10:23:59:5F:20:05:19:47:ED:68:8E:92:12:CA:5D:FC:D6:2B:B2:92:3C:20: CF:E1:5F:AF:20:BE:A0:76:7F:76:E5:EC:1A:86:61:33:3E:E7:7B:B4:3F:A0:0F:8E:A2:B9:6 A:6F:B9:87:26:6F:41:6C:88:A6:50:FD:6A:63:0B:F5:93:16:1B:19:8F:B2:ED:9B:9B:C9:9 0:F5:01:0C:DF:19:3D:0F:3E:38:23:C9:2F:8F:0C:D1:02:FE:1B:55:D6:4E:D0:8D:3C:AF:4 F:A4:F3:FE:AF:2A:D3:05:9D:79:08:A1:CB:57:31:B4:9C:C8:90:B2:67:F4:18:16:93:3A:F C:47:D8:D1:78:96:31:1F:BA:2B:0C:5F:5D:99:AD:63:89:5A:24:20:76:D8:DF:FD:AB:4E: A6:22:AA:9D:5E:E6:27:8A:7D:68:29:A3:E7:8A:B8:DA:11:BB:17:2D:99:9D:13:24:46:F 7:C5:E2:D8:9F:8E:7F:C7:8F:74:6D:5A:B2:E8:72:F5:AC:EE:24:10:AD:2F:14:DA:FF:2D:9 A:46:71:47:BE:42:DF:BB:01:DB:F4:7F:D3:28:8F:31:59:5B:D3:C9:02:A6:B4:52:CA:6E:9 7:FB:43:C5:08:26:6F:8A:F4:BB:FD:9F:28:AA:0D:D5:45:F3:13:3A:1D:D8:C0:78:8F:41:6 7:3C:1E:94:64:AE:7B:0B:C5:E8:D9:01:88:39:1A:97:86:64:41:D5:3B:87:0C:6E:FA:0F:C 6:BD:48:14:BF:39:4D:D4:9E:41:B6:8F:96:1D:63:96:93:D9:95:06:78:31:68:9E:37:06:3 B:80:89:45:61:39:23:C7:1B:44:A3:15:E5:1C:F8:92:30:BB

Public Key Info

Algorithm	Elliptic Curve
Key Size	256
Curve	P-256
Public Value	04:29:97:A7:C6:41:7F:C0:0D:9B:E8:01:1B:56:C6:F2:52:A5:BA:2D:B2:12:E8:D2:2E:D7: FA:C9:C5:D8:AA:6D:1F:73:81:3B:3B:98:6B:39:7C:33:A5:C5:4E:86:8E:80:17:68:62:45: 57:7D:44:58:1D:B3:37:E5:67:08:EB:66:DE

Miscellaneous

Serial Number	06:6C:9F:D5:74:97:36:66:3F:3B:0B:9A:D9:E8:9E:76:03:F2:4A
Signature Algorithm	ECDSA with SHA-256

“Web of trust”

- Obtain public keys *in person*
    - “Key-signing parties”

- Obtain “certificates” on your public key from people who know you

- If A knows pk_B, and B issued a certificate for C, then C can send that certificate to A
    - What trust assumptions are being made here?

Public repository

- Store certificates in a central repository
  - E.g., OpenPGP keyserver

- To find Alice’s public key
  - Get all public keys for “Alice,” along with certificates on those keys
  - Look for a certificate signed by someone you trust whose public key you already have

PKI in practice…

- Does not work quite as well as in theory…
  - Proliferation of root CAs
    - Compromises of CAs
  - Revocation can be difficult
  - Users/browsers may not verify certificates properly

SSL/TLS

- How can you securely send your credit card number to Amazon?

- SSL/TLS
  - Secure Socket Layer (Netscape, mid-’90s)
  - Transport Layer Security
    - TLS 1.0 (1999)
    - TLS 1.2 (2008)
    - TLS 1.3 (2018)
  - Used by every web browser for https connections

TLS 1.3

- Goals
  - Understand (at a high level) a real-world crypto protocol
  - Pull together everything learned in this course

- Not goals
  - Understanding low-level details/implementation
  - Defining or proving security

TLS 1.3

- Two phases
  - Handshake protocol
    - Establish shared keys between two entities
    - Server-to-client authentication only
  - Record-layer protocol
    - Use shared keys for secure communication

- Note: high-level details only
  - Actual implementation is (even) more complex

Handshake protocol

pk_{CA}

$sk, pk, cert_{CA \rightarrow Bank}$

$\xrightarrow{\text{https://bank.com}, g^x, N_C}$

$\xleftarrow{g^y, N_S}$

$mk = (g^x)^y$

$k_C, k'_C, k_S, k'_S = \text{Derive}(mk)$

$\xleftarrow{c \leftarrow \text{Enc}_{k'_S}(\text{Sign}_{sk}(\text{trans}), pk, cert)}$

$mk = (g^y)^x$

$k_C, k'_C, k_S, k'_S = \text{Derive}(mk)$

$(\sigma, pk, cert) = \text{Dec}_{k'_S}(c)$

Verify!

$\xrightarrow{\text{Mac}_{k'_C}(\text{trans}')}$

Verify!

Record-layer protocol

- Parties now share *session keys* k_C, k_S
- Client uses k_C for authenticated encryption of all messages it sends
- Server uses k_S for authenticated encryption of all messages it sends
  - Prevents reflection attacks
- Sequence numbers used to prevent replay attacks