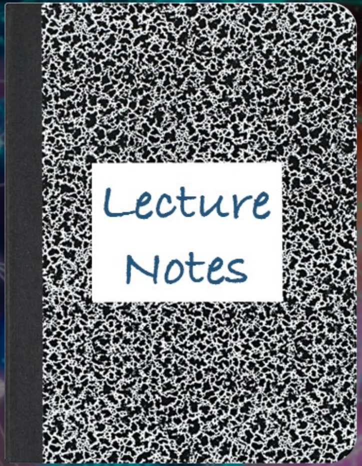


CS 419: Computer Security

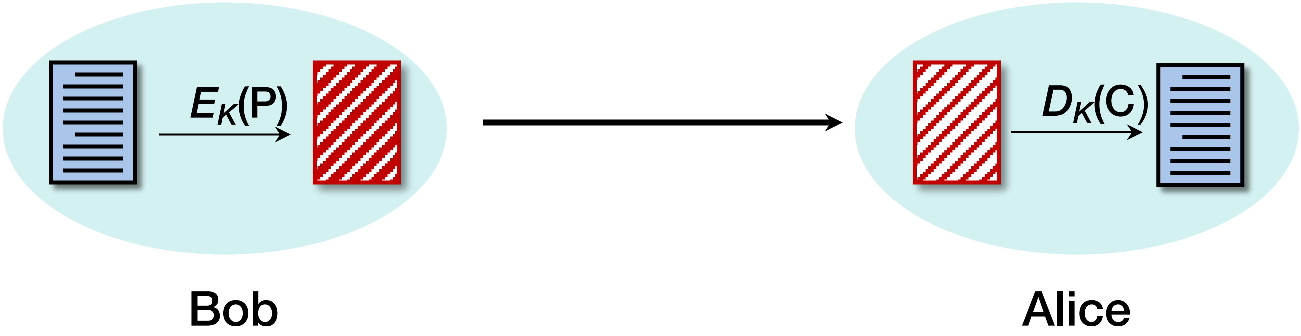
Week 7: Asymmetric Cryptography & Integrity

Paul Krzyzanowski

Key Distribution

Communicating with symmetric cryptography

- Both parties must agree on a secret key, *K*
- Message is encrypted, sent, decrypted at other side



Key distribution must be secret. Otherwise

- Messages can be decrypted by the adversary
- Users can be impersonated

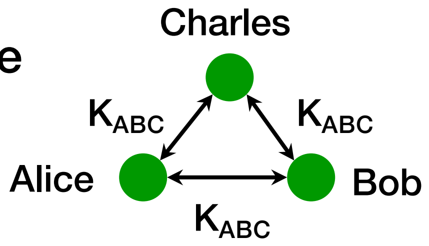
Problems With Keys In Symmetric Cryptography

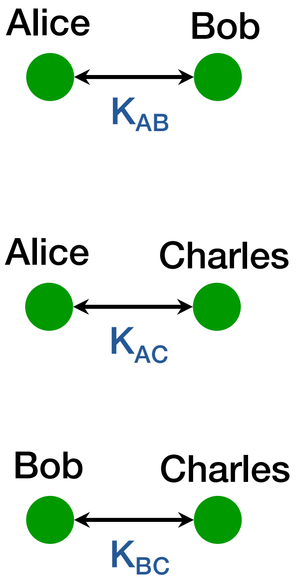
Key Management

- Potentially a lot of keys to track
- Every group of users needs a key

Key Distribution

- How do you communicate with someone you've never met?
- You cannot send them the secret key
  if the communication line is not secure





Key Distribution

Secure key distribution is the biggest problem with symmetric cryptography

Public Key Cryptography

Public-key algorithm

Two related keys:

$$C = E_{K1}(P) \quad P = D_{K2}(C)$$

$$C' = E_{K2}(P) \quad P = D_{K1}(C')$$

K_1 is a public key

K_2 is a private key

Examples:

RSA, Elliptic curve algorithms

DSS (digital signature standard)

Trapdoor functions

- **Public key cryptography relies on trapdoor functions**
- **Trapdoor function**
  - Easy to compute in one direction
  - Inverse is difficult to compute without extra information

- **Example:**

  96171919154952919 is the product of two prime #s. What are they?

  But if you're told that one of them is **100225441**

  … then it's easy to compute the other: **959555959**

RSA Public Key Cryptography

Ron Rivest, Adi Shamir, Leonard Adleman created the first public key encryption algorithm in 1977

Each user generates two keys:

- **Private key** (kept secret)
- **Public key** (can be shared with anyone)

Difficulty of algorithm based on the difficulty of factoring large numbers

- Keys are functions of a pair of large (~300 digits) prime numbers

RSA algorithm: key generation

1. Choose two random large prime numbers p, q

2. Compute the product $n = pq$ and $\phi = (p - 1)(q - 1)$

   n will be presented with the public & private keys. Length(n) is the **key length**

3. Choose the **public exponent**, e, such that:

   $1 < e < \phi$ and $\gcd(e, \phi) = 1$ [e and $(p - 1)(q - 1)$ are relatively prime]

4. Compute the **secret exponent**, d such that:

   $ed = 1 \bmod \phi$

   $d = e^{-1} \bmod ((p - 1)(q - 1))$

5. **Public key** = (e, n)

   Private key = (d, n)

   Discard p, q, ϕ

See https://www.di-mgt.com.au/rsa_alg.html

RSA Encryption

Key pair: public key = (e, n)
private key = (d, n)

Encrypt

- Divide data into numerical blocks $< n$
- Encrypt each block:

$$c = m^e \bmod n$$

Decrypt

$$m = c^d \bmod n$$

RSA security

The security of RSA encryption rests on the difficulty of factoring a large integer

Public key = { *modulus*, *exponent* }, or $\{n, e\}$

- The *modulus* is the product of two primes, p, q
- The private key is derived from the same two primes

Elliptic Curve Cryptography

Alternate approach: elliptic curves

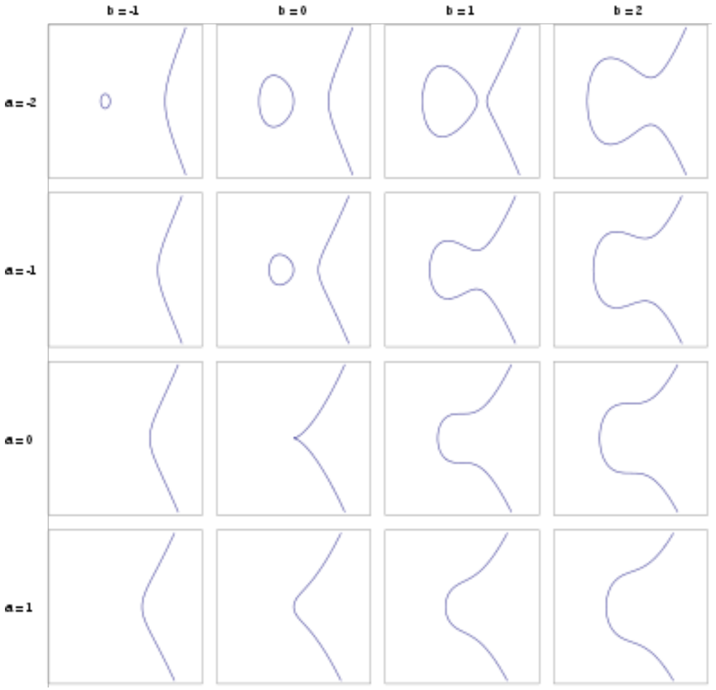
$$y^2 = x^3 + ax + b$$

Using discrete numbers, pick

- A prime number as a maximum (modulus)
- A curve equation
- A public base point on the curve
- A random private key
- Public key is derived from the private key, the base point, and the curve

To compute the private key from the public,

- We would need an elliptic curve discrete logarithm function
- This is difficult and is the basis for ECC's security



Catalog of elliptic curves
https://en.wikipedia.org/wiki/Elliptic_curve

ECC vs. RSA

- **RSA is still the most widely used public key cryptosystem**
  - Mostly due to inertia & widespread implementations
  - Faster for decryption
  - Simpler implementation
- **ECC offers higher security with fewer bits than RSA**
  - ECC is faster for key generation & encryption
  - Uses less memory
  - NIST defines 15 standard curves for ECC
    - But many implementations support only a couple (P-256, P-384)

https://www.keylength.com/en/4/
http://https://www.enisa.europa.eu/publications/algorithms-key-size-and-parameters-report-2014

Key length

Unlike symmetric cryptography, not every number is a valid key with RSA and ECC

Comparable complexity:

– 3072-bit RSA = 256-bit elliptic curve = 128-bit symmetric cipher

– 15360-bit RSA = 521-bit elliptic curve = 256-bit symmetric cipher

For long-term security

The European Union Agency for Network and Information Security (ENISA) and the National Institute for Science & Technology (NIST) recommend:

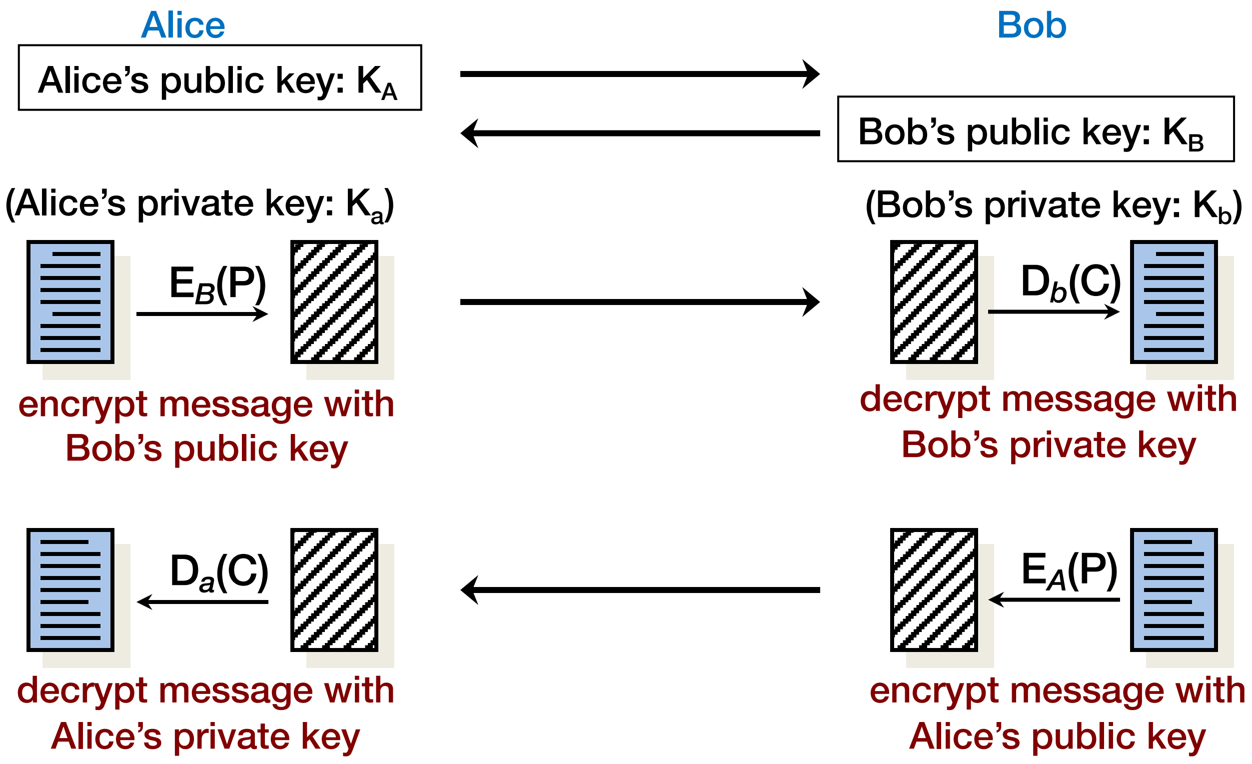
AES: 256-bit keys RSA: 15,360-bit keys ECC: 512 bit-keys

Communication with public key algorithms

Different keys for encrypting and decrypting

- No need to worry about key distribution

Communication with public key algorithms



RSA isn't good for communication

Calculations are very expensive relative to symmetric algorithms

Common speeds:

Algorithm	Bytes/sec
AES-128-ECB	148,000,000
AES-128-CBC	153,000,000
AES-256-ECB	114,240,000
RSA-2048 encrypt	3,800,000
RSA-2048 decrypt	96,000

AES ~1500x faster to decrypt; 40x faster to encrypt than RSA

If anyone learns your private key, they can read all your messages

Key Exchange

Diffie-Hellman Key Exchange

Key distribution algorithm

- Allows two parties to share a secret key over a non-secure channel
- *Not* public key encryption
- Based on difficulty of computing discrete logarithms in a finite field compared with ease of calculating exponentiation

Allows us to negotiate a secret common key without fear of eavesdroppers

Diffie-Hellman Key Exchange

- **All arithmetic performed in a field of integers modulo some large number**
- **Both parties agree on**
  - a large prime number p
  - and a **number $\alpha < p$**
- **Each party generates a public/private key pair**

Private key for user i: X_i

Public key for user i: $Y_i = \alpha^{X_i} \bmod p$

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B

K = (Bob's public key) $^{\textit{(Alice's private key)}}$ ***mod p***

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B
- Bob computes

$$K = Y_A^{X_B} \bmod p$$

K' = (Alice's public key) $^{\textbf{(Bob's private key)}}$ ***mod p***

Diffie-Hellman exponential key exchange

- Alice has secret key X_A
- Alice sends Bob public key Y_A
- Alice computes

$$K = Y_B^{X_A} \bmod p$$

- expanding:

$$\begin{aligned} K &= Y_B^{X_A} \bmod p \\ &= (\alpha^{X_B} \bmod p)^{X_A} \bmod p \\ &= \alpha^{X_B X_A} \bmod p \end{aligned}$$

- Bob has secret key X_B
- Bob sends Alice public key Y_B
- Bob computes

$$K = Y_A^{X_B} \bmod p$$

- expanding:

$$\begin{aligned} K &= Y_B^{X_B} \bmod p \\ &= (\alpha^{X_A} \bmod p)^{X_B} \bmod p \\ &= \alpha^{X_A X_B} \bmod p \end{aligned}$$

K* = *K'

K is a ***common key***, known *only* to Bob and Alice

Diffie-Hellman simple example

Assume p=1151, α=57

- Alice's secret key X_A = 300
- Alice's public key Y_A = 57^{300} mod p = 282
- Alice computes

$$K = Y_B^{X_A} \bmod p \quad = 1046^{300} \bmod p$$

$$K = 105$$

- Bob's secret key X_B = 25
- Bob's public key Y_B = 57^{25} mod p = 1046
- Bob computes

$$K = Y_A^{X_B} \bmod p \quad = 282^{25} \bmod p$$

$$K = 105$$

Given p=1151, α=57, Y_A=282, Y_B=1046, you cannot get 105

Hybrid Cryptosystems

Hybrid Cryptosystems

- **Session key**: randomly-generated key for one communication session
- **Use a public key algorithm to send the session key**
- **Use a symmetric algorithm to encrypt data with the session key**

Public key algorithms are almost never used to encrypt messages

- MUCH slower; vulnerable to *chosen-plaintext attacks*
- RSA-2048 approximately 55x slower to encrypt and 2,000x slower to decrypt than AES-256

Communication with a hybrid cryptosystem

Alice

Bob

Bob's public key: K_B

Pick a random <u>session key</u>, K

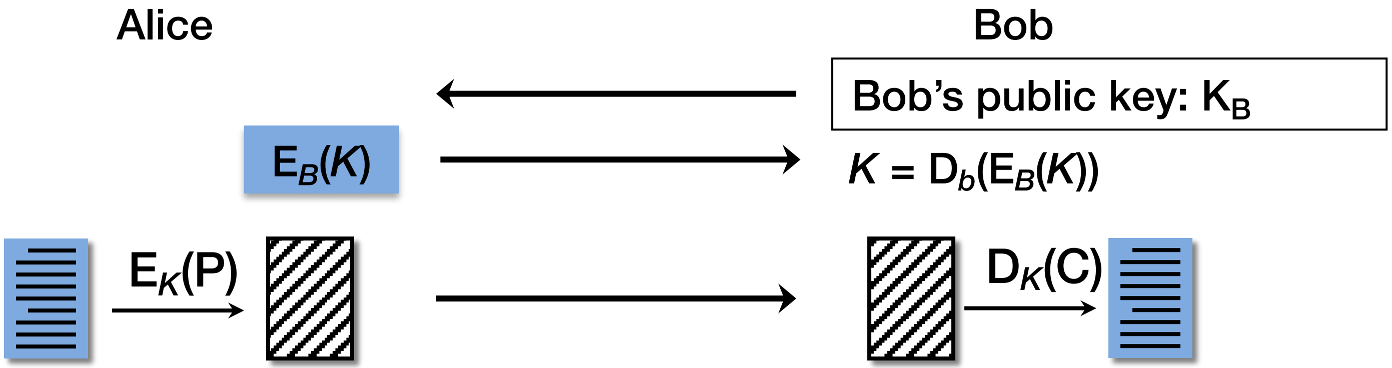
$K = D_b(E_B(K))$

encrypt session key with
Bob's public key

Bob decrypts K with
his private key

Now Bob knows the secret session key, K

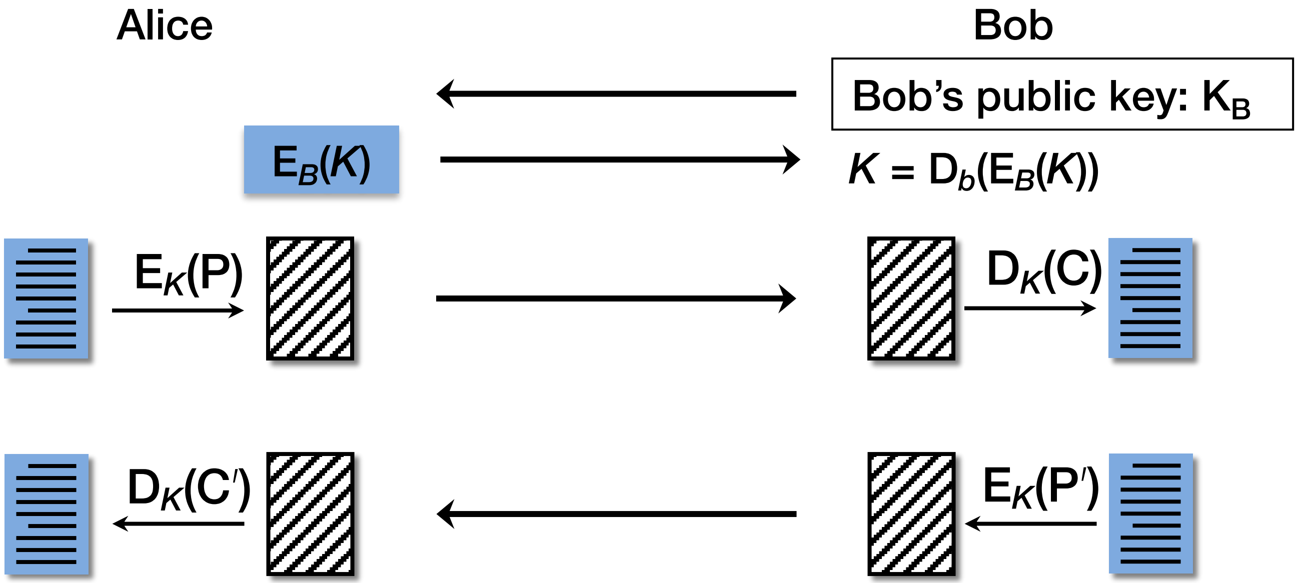
Communication with a hybrid cryptosystem



encrypt message using a symmetric algorithm and key *K*

decrypt message using a symmetric algorithm and key *K*

Communication with a hybrid cryptosystem



decrypt message using a symmetric algorithm and key *K*

encrypt message using a symmetric algorithm and key *K*

Forward Secrecy

Private keys need to be protected

Pick a session key & encrypt it with the Bob's public key → Bob decrypts the session key with his private key

Suppose an attacker steals Bob's private key

- Future messages can be compromised
- The attacker can also go through past messages & decrypt old session keys

Security rests entirely on the secrecy of Bob's private key

- If Bob's private key is compromised, all recorded past traffic can be decrypted

Forward Secrecy

Forward secrecy

- Compromise of long-term keys does not compromise past session keys
- There is no one secret to steal that will compromise multiple messages

Achieving Forward Secrecy

Use <u>ephemeral keys</u> for key exchange + <u>session keys</u> for communication

Diffie-Hellman key exchange is commonly used for key exchange

- Generate a set of keys per session
- Use the derived common key as the encryption/decryption key … or as a key to encrypt a session key
- Not recoverable as long as private keys are thrown away

  Unlike RSA keys, key generation in Diffie-Hellman is extremely efficient

Keys must be ephemeral

Client & server will generate new Diffie-Hellman parameters for each session – all will be thrown away after the session

Diffie-Hellman is preferred over RSA for key exchange to achieve forward secrecy.
Generating Diffie-Hellman keys is a rapid, low-overhead process.

Cryptographic systems: summary

- **Symmetric ciphers**
  - Based on SP-networks (usually) = substitution & permutation sequences
- **Asymmetric ciphers – public key cryptosystems**
  - Based on trapdoor functions: easy to compute in one direction, difficult to compute in the other direction without special information (the trapdoor)
- **Hybrid cryptosystem**
  - Pick a random session key + public key algorithm for key exchange
  - Use a symmetric key algorithm to encrypt traffic back & forth
  - **Forward secrecy**: establish session key via ephemeral keys
- **Key exchange algorithms (more to come later)**
  - Diffie-Hellman
  - Public key

  Enables secure communication without knowledge of a shared secret
- **Perfect secrecy**
  - Ephemeral keys + Session key

Looking ahead

RSA cryptography in the future

- **Based on the difficulty of factoring products of two large primes**
- **Factoring algorithms get more efficient as numbers get larger**
  - As the ability to decrypt numbers increases, the key size must therefore grow even faster
  - This is not sustainable (especially for embedded devices)
- **ECC is a better choice for most applications**

Quantum Computers & Cryptography

Once (if) useful quantum computers can be built, they can

- Factor efficiently
  - Shor's algorithm factors numbers exponentially faster
  - RSA will not be secure anymore
- Find discrete logarithms & elliptic curve discrete logarithms efficiently
  - Diffie-Hellman key exchange & ECC will not be secure

Not all is bad

Symmetric cryptography is largely immune to attacks

Some optimizations are predicted (Grover's algorithm): crack a symmetric cipher in time proportional to the square root of the key space size: $2^{n/2}$ vs. 2^n

– Use 256-bit AES to be safe

2016: NSA called for a migration to "post-quantum cryptographic algorithms"
… but no agreement yet on what those will be

July 2020: Narrowed submissions down to 7 finalists & 8 alternates

Quantum-resistant standard expected to be announced in 2022 (delayed by COVID-19)

https://csrc.nist.gov/projects/post-quantum-cryptography

https://www.nist.gov/news-events/news/2020/07/nists-post-quantum-cryptography-program-enters-selection-round

Quantum-proofing cryptography

Quantum computing is not faster at everything

Only four types of problems are currently identified where quantum computing offers an advantage

Researchers are developing algorithms that are cannot be made more efficient with quantum computing

31108953	1190018662
104910828	2598220447
3027417464	3006531459
2376520867	804531264
2430217482	1122428373

Example: Add 3 out of a set of 10 numbers

- **Give the sum to a friend and ask them to determine which numbers were added**
- **Try this if someone picks 500 out of 1,000 numbers with 1,000 digits each**

Which 3 numbers add up to 5656746864?

https://www.scientificamerican.com/article/new-encryption-system-protects-data-from-quantum-computers/

Stay tuned…

- **2016: NSA called for a migration to "post-quantum cryptographic algorithms"**
- **July 2020: Narrowed submissions down to 7 finalists & 8 alternates**
- **Solution families**
  1. Lattice-based
  2. Code-based
  3. Multivariate

Quantum-resistant standard expected to be announced in 2022

Message Integrity

McCarthy's Spy Puzzle (1958)

The setting:

- Two countries are at war
- One country sends spies to the other country
- To return safely, spies must give the border guards a password

Conditions

- Spies can be trusted
- Guards chat – information given to them may leak

McCarthy’s Spy Puzzle

Challenge

- How can a border guard authenticate a person without knowing the password?
- Enemies cannot use the guard’s knowledge to introduce their own spies

Solution to McCarthy's puzzle

Michael Rabin, 1958

- **Use a one-way function, *B = f (A)***
  - Guards get B
    - Enemy cannot compute A if they know A
  - Spies give A, guards compute f(A)
    - If the result is B, the password is correct.
- **Example function:**
  - Middle squares
    - Take a 100-digit number (A), and square it
    - Let B = middle 100 digits of 200-digit result

One-way functions

- Easy to compute in one direction
- Difficult to compute in the other

Examples:

Factoring:

$pq = N$	EASY	Basis for RSA
find p,q given N	DIFFICULT	

Discrete Log:

$a^b \bmod c = N$	EASY	Basis for Diffie-Hellman & Elliptic Curve
find b given a, c, N	DIFFICULT	

Example of a one-way function: middle squares

Example with a 20-digit number

A = 18932442986094014771

A^2 = 358437397421700454779607531189166182441

Middle square, B = 42170045477960753118

Given A, it is easy to compute B

Given B, it is difficult to compute A

"Difficult" = no known short-cuts; requires an exhaustive search

Cryptographic hash functions

Cryptographic hash functions

Also called *digests* or *fingerprints*

Properties

– Arbitrary length input → **fixed-length output**

– **Deterministic**: you always get the same hash for the same message

– **One-way function** (**pre-image resistance**, or ***hiding***)
  - Given ***H***, it should be difficult to find ***M*** such that ***H=hash(M)***

– **Collision resistant**
  - Infeasible to find any two different strings that hash to the same value:
    Find ***M***, ***M'*** such that ***hash(M) = hash(M')***

– **Output should not give any information about any of the input**
  - Like cryptographic algorithms, relies on ***diffusion***

– **Efficient**
  - Computing a hash function should be computationally efficient

Hash functions are the basis of integrity

- **Not encryption**
- **Can help us to detect:**
  - **Masquerading**:
    - Insertion of message from a fraudulent source
  - **Content modification**:
    - Changing the content of a message
  - **Sequence modification**:
    - Inserting, deleting, or rearranging parts of a message
  - **Replay attacks**:
    - Replaying valid sessions

Hash Algorithms

Use iterative structure like block ciphers do … but use no key

- **Example:**
  - Secure Hash Algorithm, **SHA-1**
    - Designed by the NSA in 1993; revised in 1995
    - US standard for use with NIST Digital Signature Standard (DSS)
    - Produces 160-bit hash values
    - Chosen prefix collision attacks demonstrated May 2019
- **Successors**
  - **SHA-2** (2001)
    - Produces 224, 256, 384, or 512-bit hashes
    - Approved for use with the NIST Digital Signature Standard (DSS)
  - **SHA-3** (2015)
    - Can be substituted for SHA-2
    - Improved robustness

Example: SHA-1 Overview

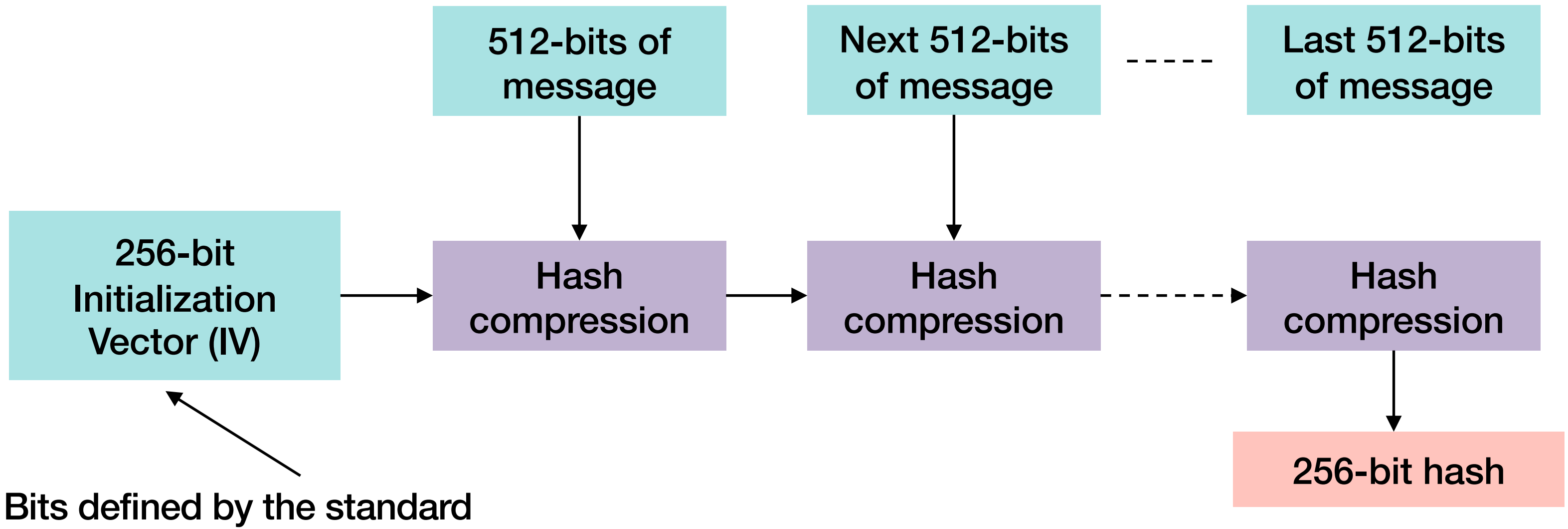
- **Prepare the message**
  - Append the bit 1 to the message
  - Pad message with 0 bits so its length = 448 mod 512
  - Append length of message as a 64-bit big endian integer
- **Use an Initialization Vector (IV) = 5-word (160-bit) buffer:**

```
a = 0x67452301  b = 0xefcdab89  c = 0x98badcfe
d = 0x10325476  e = 0xc3d2e1f0
```

- **Process the message in 512-bit chunks**
  - Expand the 16 32-bit words into 80 32-bit words via XORs & shifts
  - Iterate 80 times to create a hash for this chunk
    - Various sets of ORs, XORs, ANDs, shifts, and adds
  - Add this hash chunk to the result so far

See https://www.saylor.org/site/wp-content/uploads/2012/07/SHA-1-1.pdf

SHA-2 Overview



Popular (& formerly popular) Hash Functions

MD5	• 128 bits • Linux passwords used to use this • Rarely used now since weaknesses were found
SHA-1	• 160 bits – was widely used: checksum in Git & torrents • Google demonstrated a *collision attack* in Feb 2017 … Google had to run >9 quintillion SHA-1 computations to complete the attack … but already being phased out since weaknesses were found earlier • Used for message integrity in GitHub
SHA-2	*Believed to be secure* • Designed by the NSA; published by NIST • Variations: SHA-224, SHA-256, SHA-384, SHA-512 • Linux passwords use SHA-512 • Bitcoin uses SHA-256
SHA-3	*Believed to be secure* • 256 & 512 bit
bcrypt	• Blowfish cipher used for *bcrypt* password hashing in OpenBSD since 1997 • Phased out in 2023: *scrypt* and *Argon2* are replacements
3DES	• Linux passwords used to use this

Designed to be slow!

Linux commands

sha1sum: create a SHA-1 hash

```
echo "hello, world!" | sha1sum
e91ba0972b9055187fa2efa8b5c156f487a8293a  -
```

sha3sum: create a 256-bit SHA-3 hash

```
echo "hello, world!" | sha3sum
c3d69513b79e0cdf3aa2b4afa38a5ffde144310109029e0e1aa57eb6  -
```

md5sum: create an MD5 hash

```
echo "hello, world!" | md5sum
910c8bc73110b0cd1bc5d2bcae782511  -
```

openssl: create a 512-bit SHA-3 hash (many other options available, run `openssl -?`)

```
echo "hello, world!" | openssl dgst -sha3-512
(stdin)=8fc33b84ff22559082893fdc73f6877e590eb67533441fe5e48cd6d8a11aaf8d6270f82ef437
c2c758000d65b09b45116b9c0eb3f3162149b13ca98c8cc8c90f
```

Hash Collisions

Hashes are *collision resistant*, but collisions can occur

Pigeonhole principle

- If you have 10 pigeons & 9 compartments, at least one compartment will have more than one pigeon
- A hash is a fixed-size small number of bits (e.g., 256 bits = 32 bytes)
- Every possible permutation of an arbitrary number of bytes cannot fit into every permutation of 32 bytes!

wikipedia

Collisions: The Birthday Paradox

How many people need to be in a room such that the probability that two people will have the same birthday is > 0.5?

Your guess before you took a probability course: 183

This is true to the question of "how many people need to be in a room for the probability that someone else will have the same birthday as *one specific student*?"

Answer: 23

$$p(n) = 1 - \frac{n! \cdot \binom{365}{n}}{365^n}$$

Approximate solution for # people required to have a 0.5 chance of a shared birthday, where m = # days in a year

$$n \approx \sqrt{2 \times m \times 0.5}$$

The Birthday Paradox: Implications

- **Searching for a collision with a pre-image (known message) is *A LOT* harder than searching for two messages that have the same hash**

- **Strength of a hash function is approximately ½ (# bits)**
  - 256-bit hash function has a strength of approximately 128 bits
  - But that's a huge space!

    $2^{128} = 3.4 \times 10^{38}$
  - It's not feasible to try that many messages in the hope of finding a collision
    - BTW … the odds of winning the Powerball lottery are only $1{:}2.9 \times 10^{8}$

Message Integrity

How do we detect that a message has been tampered?

- **A cryptographic hash acts as a checksum**
- **Associate a hash with a message**
  - we're not encrypting the message
  - we're concerned with *integrity*, not *confidentiality*
- **If two messages hash to different values, we know the messages are different**

$$H(M) \neq H(M')$$

Tamperproof Integrity: Message Authentication Codes and Digital Signatures

Message Integrity: MACs

We rely on hashes to assert the integrity of messages

But an attacker can create a new message & a new hash
and replace *H(M)* with *H(M′)*

So, let's create a checksum that relies on a key for validation

Message Authentication Code (MAC)

Two forms: hash-based & block cipher-based

Hash-based MAC

We can create a MAC from a cryptographic hash function

HMAC = Hash-based Message Authentication Code

$$HMAC(m, k) = H((opad \oplus k) \,\|\, H(ipad \oplus k) \,\|\, m))$$

where

H = cryptographic hash function

opad = outer padding 0x5c5c5c5c … (01011100…)

ipad = inner padding 0x36363636… (00110110…)

k = secret key

m = message

$\oplus$ = XOR, $\|$ = concatenation

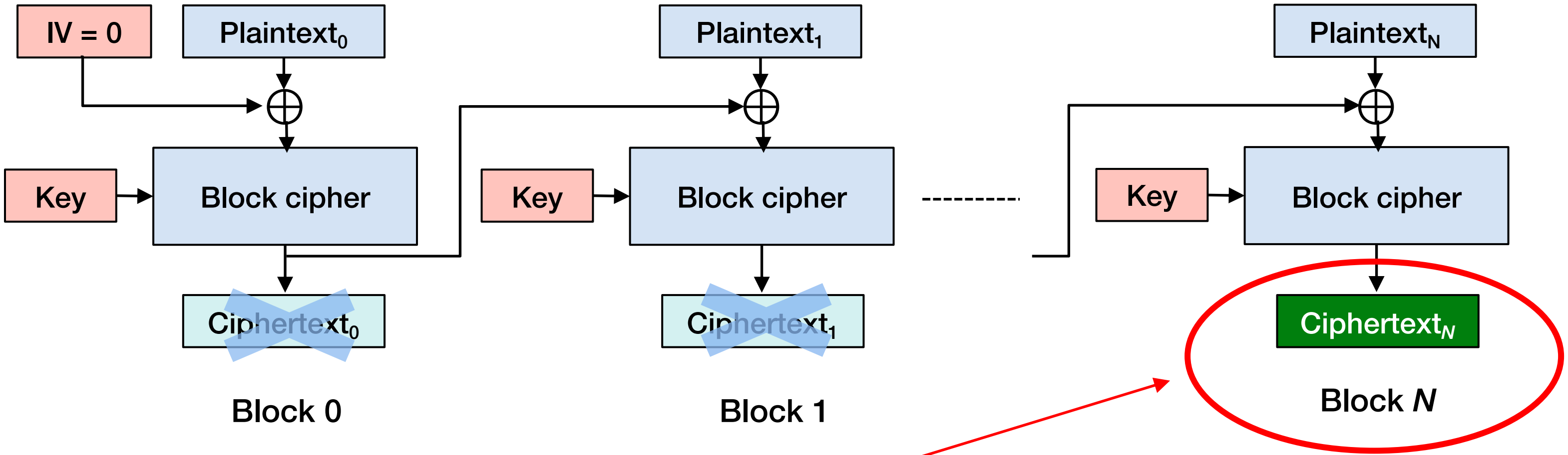
Basically, incorporate a key into the message before hashing it

See RFC 2104

Block Cipher Based MAC: CBC-MAC

Cipher Block Chaining (CBC) ensures that every encrypted block is a function of all previous blocks

CBC-MAC uses an initialization vector = 0



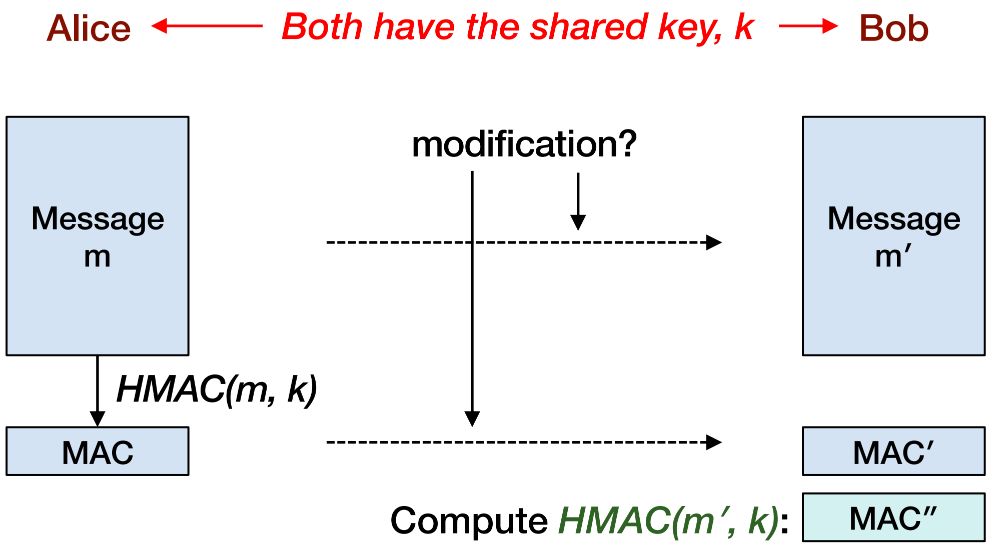
MAC = final ciphertext block – others are discarded

Examples: AES-CBC-MAC, DES-MAC

Don't use the same key for the MAC as for encrypting the message

If an adversary gets one of the keys, she will be unable to create either a valid message or a valid hash

Using a MAC



1. Bob receives the Message m’ and a MAC.
2. Knowing the key, k, he generates a MAC for the message: MAC″ = HMAC(m′, k)
3. If MAC′ = MAC″, he’s convinced that the message has not been modified

Digital Signatures

- **MACs rely on a shared key**
  - Anyone with the key can modify and re-sign a message

- **Digital signature properties**
  - Only you can sign a message, but anyone can validate it
  - You cannot cut and paste the signature from one message to another
  - If the message is modified, the signature will be invalid
  - An adversary cannot forge a signature
    - Even after inspecting an arbitrary number of signed messages

Digital Signature Primitives

1. **Key generation**

   { secret_key, verification_key } := **gen_keys**(key_size)

2. **Signing**

   signature := **sign**(message, secret_key)

3. **Validation**

   Isvalid := **verify**(verification_key, message, signature)

We sign *hash(message)* instead of the *message*

- We'd like the signature to be a small, fixed size
- We may not need to hide the contents of the message
- We trust hashes to be collision-free

Digital Signatures & Public Key Cryptography

Public key cryptography enables digital signatures

secret_key = private key

verification_key = public key

- Alice encrypts a message with her private key

$$S = E_a(M)$$

- Anyone can decrypt it using her public key

$$D_A(S) = D_A(E_a(M)) = M$$

- Nobody but Alice can create S

Popular Digital Signature Algorithms

- **DSA: Digital Signature Algorithm**
  - NIST standard – Uses SHA-1 or SHA-2 hash
  - Key pair based on difficulty of computing discrete logarithms
- **ECDSA: Elliptic Curve Digital Signature Algorithm**
  - Variants of DSA that uses elliptic curve cryptography
  - Used in bitcoin
- **EdDSA: Edwards-curve Digital Signature Algorithm**
  - Slightly faster than ECDSA

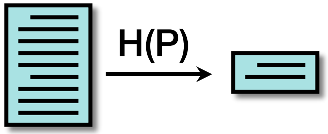
Digital Signature Algorithms combine hashing + encryption into one step

- **signature:** $S := E_{pri\_key}(H(\mathrm{M}))$
- **verification =** $H(\mathrm{M}) \stackrel{?}{=} D_{pub\_key}(S)$

Digital signatures

Alice

Bob

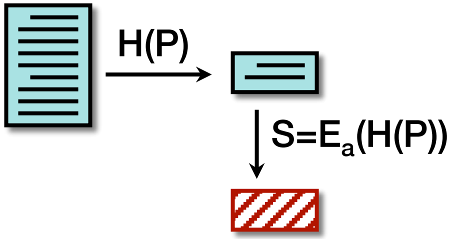


Alice generates a hash of the message, H(P)

Digital signatures: public key cryptography

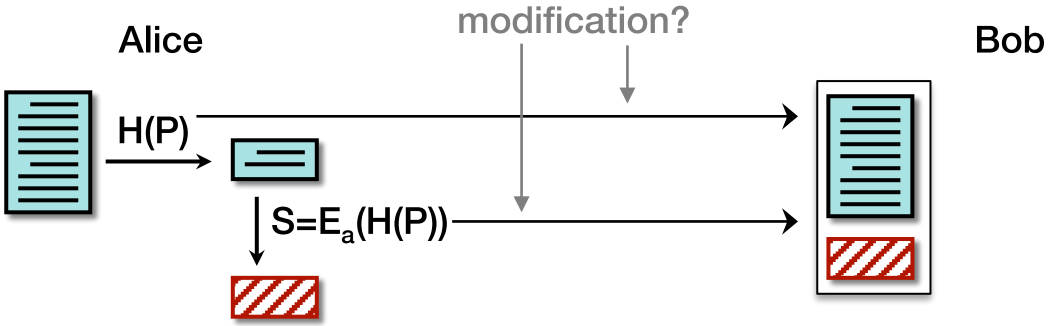
Alice

Bob



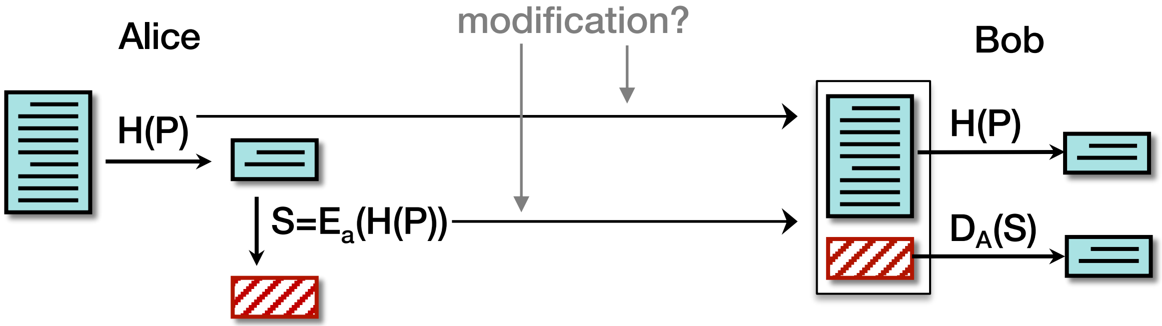
Alice encrypts the hash with her private key
This is her signature.

Using Digital Signatures



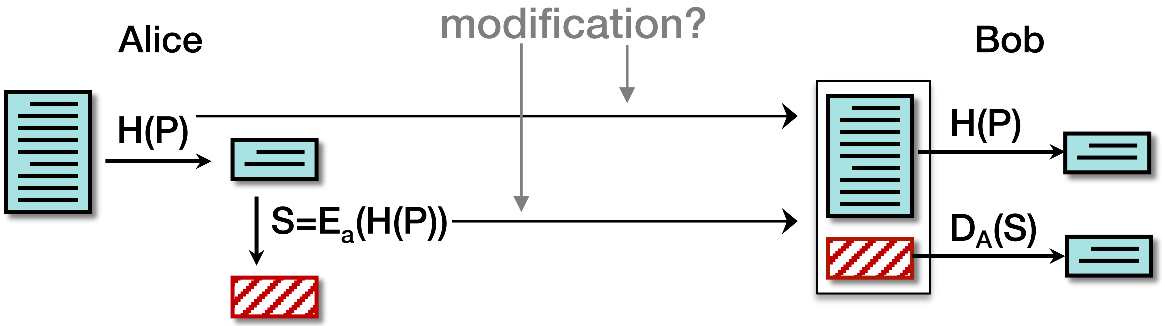
Alice sends Bob the message & the encrypted hash

Using Digital Signatures



1. Bob decrypts the hash using Alice's public key
2. Bob computes the hash of the message sent by Alice

Using Digital Signatures

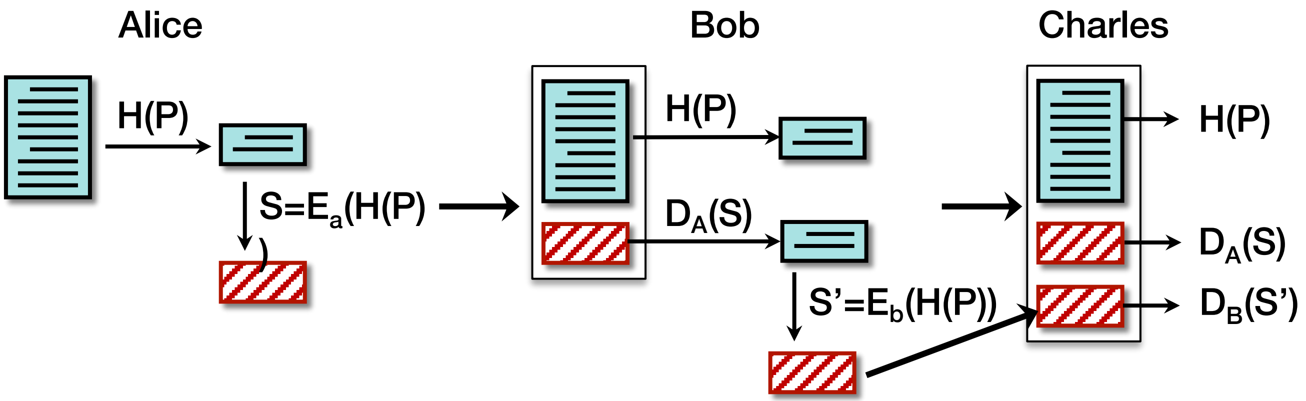


If the hashes match, the signature is valid
⇒ the encrypted hash *must* have been generated by Alice

Digital signatures & non-repudiation

- **Digital signatures provide non-repudiation**
  - Only Alice could have created the signature because only Alice has her private key
- **Proof of integrity**
  - The hash assures us that the original message has not been modified
  - The encryption of the hash assures us that an attacker could not have re-created the hash

Digital signatures: multiple signers



Charles:

- Generates a hash of the message, H(P)
- Decrypts Alice's signature with Alice's public key
  - Validates the signature: $D_A(S) \stackrel{?}{=} H(P)$
- Decrypts Bob's signature with Bob's public key
  - Validates the signature: $D_B(S) \stackrel{?}{=} H(P)$

Covert AND authenticated messaging

If we want to keep the message secret

– combine encryption with a digital signature

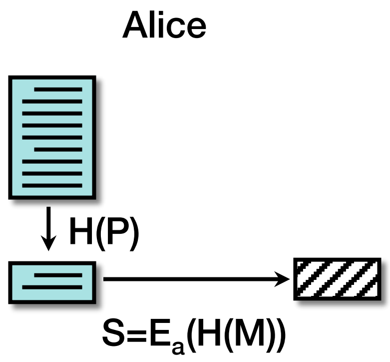
Use a session key:

– Pick a random key, *K*, to encrypt the message with a symmetric algorithm

– Encrypt *K* with the public key of each recipient

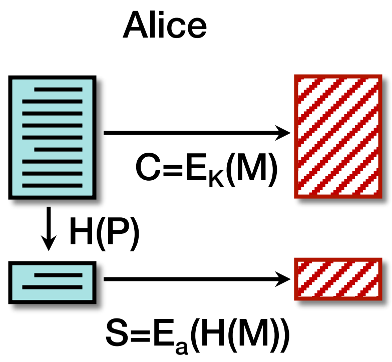
– For signing, encrypt the hash of the message with sender's private key

Covert and authenticated messaging



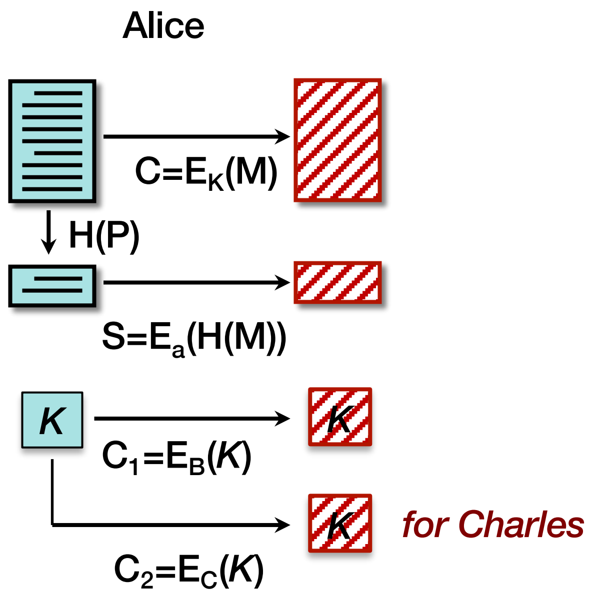
Alice generates a digital signature by
encrypting the message with her private key

Covert and authenticated messaging



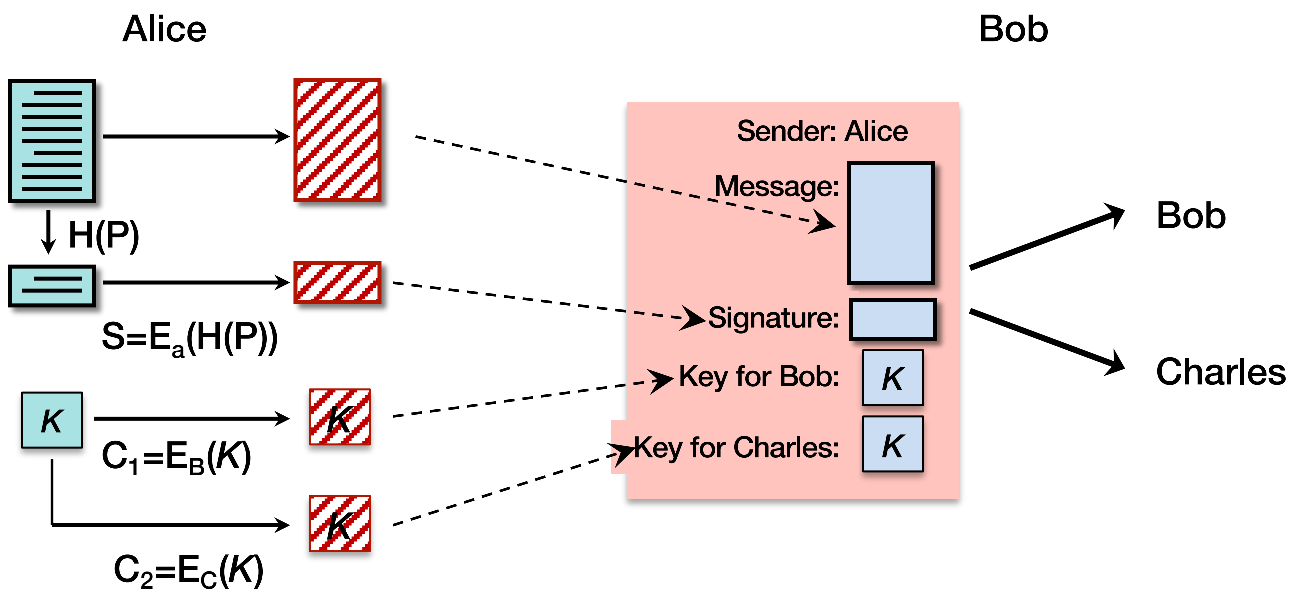
Alice picks a random key, *K*, and encrypts the message *P* with it using a symmetric cipher

Covert and authenticated messaging



Alice encrypts the session key for each recipient of this message using their public keys

Covert and authenticated messaging



The aggregate message is sent to Bob & Charles

Note: we do not have forward secrecy by doing this

Certificates: Identity Binding

Public Keys as Identities

- **A public signature verification key can be treated as an identity**
  - Only the owner of the corresponding private key will be able to create the signature
- **New identities can be created by generating new random {private, public} key pairs**
- **Anonymous identity – no identity management**
  - A user is known by a random-looking public key
  - Anybody can create a new identity at any time
  - Anybody can create as many identities as they want
  - A user can throw away an identity when it is no longer needed
  - Example: your Bitcoin identity = hash(public key)

Identity Binding

- **How does Alice know Bob's public key is really his?**
- **Get it from a trusted server?**
  - What if the enemy tampers with the server?
  - Or intercepts Alice's query to the server (or the reply)?
  - What set of public keys does the server manage?
  - How do you find it in a trustworthy manner?

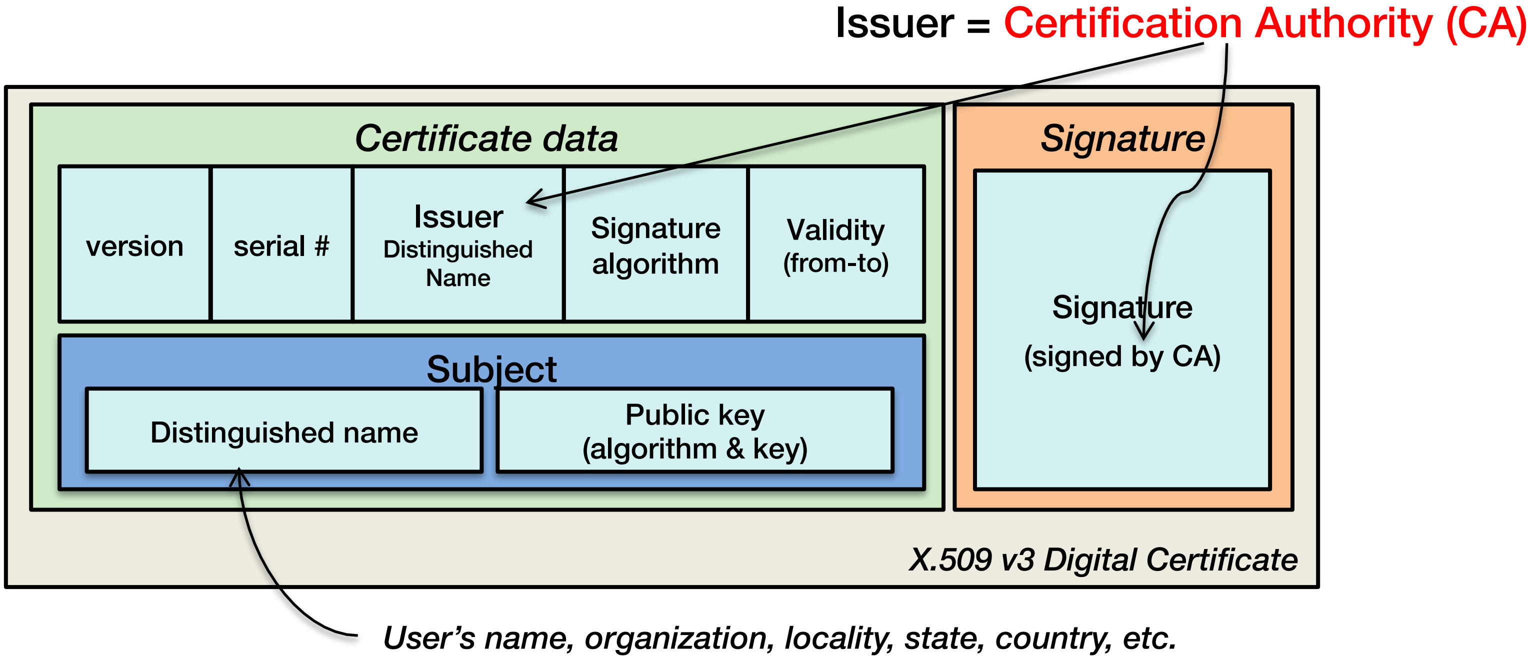
Identity Binding – Another Option

- **Have a trusted party sign Bob's public key**
- **Once signed, it is tamper-proof**
  - An attacker cannot generate the signature after modifying the key
- **But we need to know it's Bob's public key and who signed it**
  - Create & sign a data structure that
    - Identifies Bob
    - Contains his public key
    - Identifies who is doing the signing

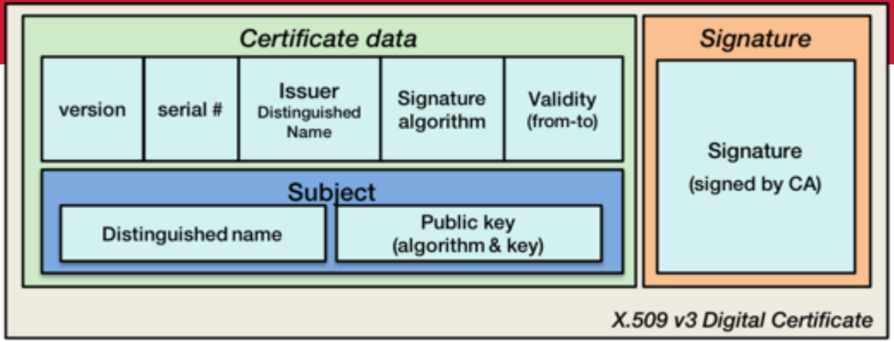
X.509 Certificates

ISO introduced a set of authentication protocols

X.509: Structure for public key <u>certificates</u>:



X.509 certificates



To validate a certificate

Verify its signature:

1. Get the issuer (CA) from the certificate
2. Validate the certificate's signature against the issuer's public key
   – Hash contents of certificate data
   – Decrypt CA's signature with CA's public key

Obtain CA's public key (certificate) from trusted source

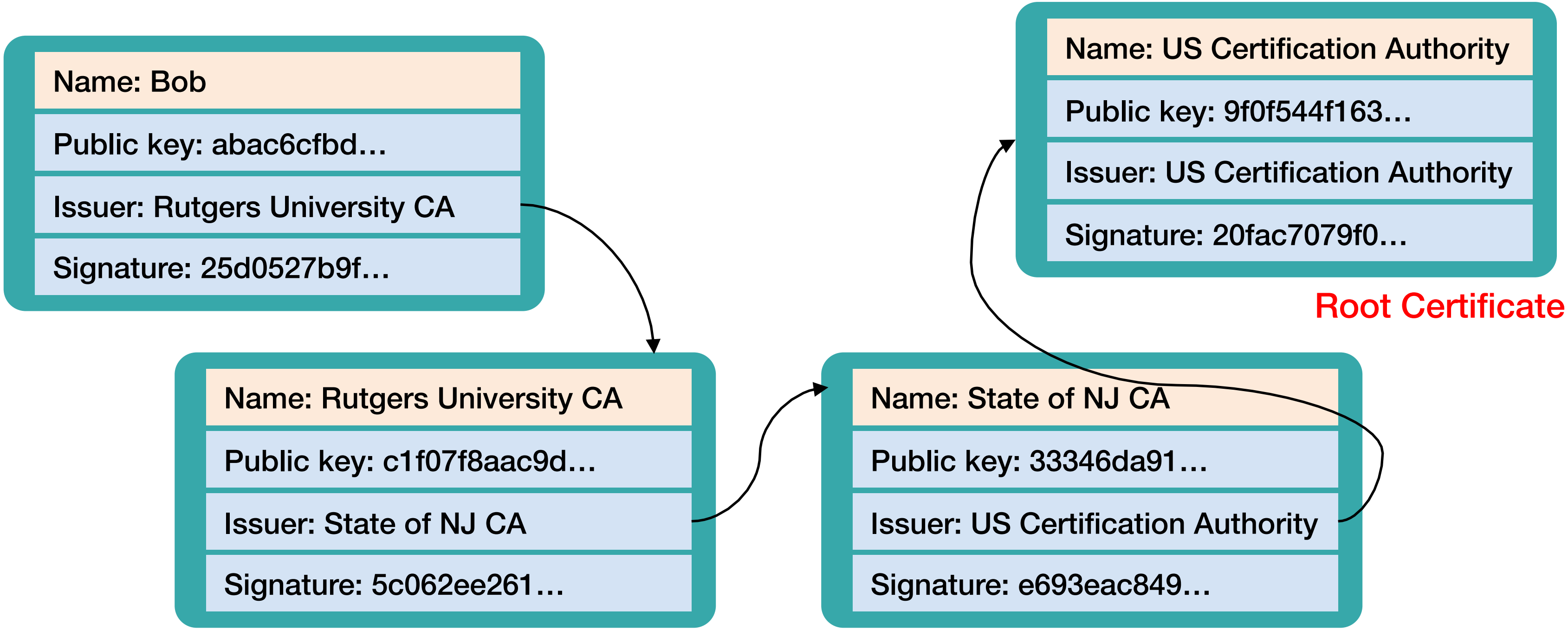
Certificates prevent someone from using a phony public key to masquerade as another person

…if you trust the CA

Certification Authorities (CAs)

How do you know the public key of the CA?

– You can get it from another certificate! ⇒ this is called certificate chaining



Certification Authorities (CAs)

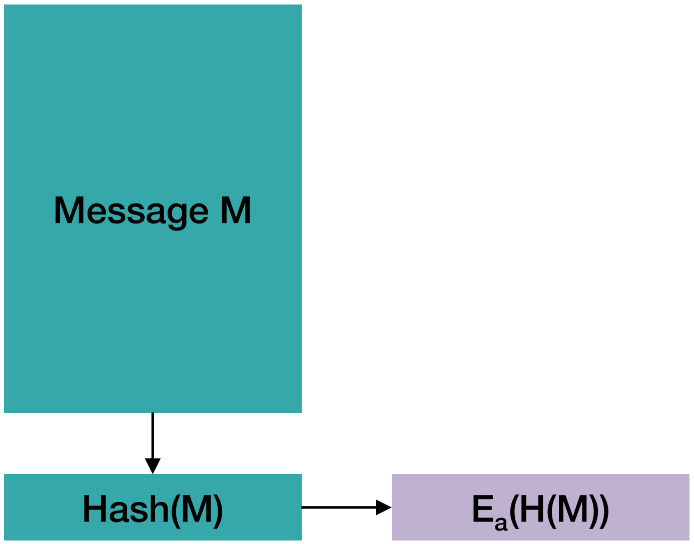
- **But trust must start somewhere**
  You need a public key you can trust – this is the root certificate
  - Apple's keychain is pre-loaded with hundreds of CA certificates
  - Windows stores them in the Certificate Store and makes them accessible via the Microsoft Management Console (mmc)
  - Android stores them in Credential Storage
- **Can you trust a CA?**
  - Maybe…
    check their reputation and read their Certification Practice Statement (CPS)
  - Even trustworthy ones might get hacked (e.g., VeriSign in 2010)

Key revocation

- **Used to invalidate certificates before expiration time**
  - Usually because of a compromised key
  - Or policy changes (e.g., someone leaves a company)
- **Certificate revocation list (CRL)**
  - Lists certificates that are revoked
  - Only certificate issuer can revoke a certificate
- **Problems**
  - Need to make sure that the entity issuing the revocation is authorized to do this
  - Revocation information may not circulate quickly enough
    - Dependent on dissemination mechanisms, network delays & infrastructure
  - Some systems may not have been coded to process revocations

Code Integrity

Review: signed messages



Encrypt with Alice's private key
= digital signature

We can sign code as well

- **Validate integrity of the code**
  - If the signature matches, then the code has not been modified
- **Enables**
  - Distribution from untrusted sources
  - Distribution over untrusted channels
  - Detection of modifications by malware
- **Signature = encrypted hash signed by trusted source**
  - Does *not* validate the code is good … just where it comes from

Code Integrity: **signed software**

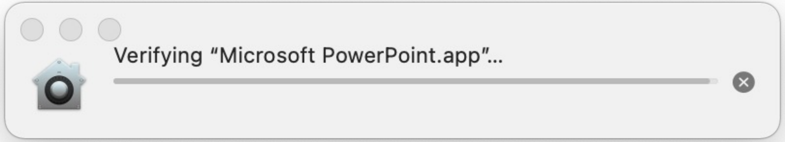
- **Windows since XP: Microsoft Authenticode**
  - *SignTool* command
  - Hashes stored in system catalog or signed & embedded in the file
  - Microsoft-tested drivers are signed
- **macOS**
  - *codesign* command
  - Hashes & certificate chain stored in file
- **Also Android & iOS**

Code signing: Microsoft Authenticode

- **A format for signing executable code (dll, exe, cab, ocx, class files)**
- **Software publisher:**
  - Generate a public/private key pair
  - Get a digital certificate from a certification authority (CA) that is enrolled in the *Microsoft Trusted Root Certificate Program*
  - Generate a hash of the code to create a fixed-length digest
  - Encrypt the hash with your private key
  - Combine digest & certificate into a Signature Block
  - Embed Signature Block in executable
- **Microsoft SmartScreen:**
  - Manages reputation based on download history, popularity, anti-virus results
- **Recipient:**
  - Call *WinVerifyTrust* function to validate:
    - Validate certificate, decrypt digest, compare with hash of downloaded code

Per-page hashing

- **Integrity check when program is first loaded**



- **Per-page signatures – improved performance**
  – Check hashes for every page upon loading (demand paging)
- **Per-page hashes can be disabled optionally on both Windows and macOS**

Windows code integrity checks

- **Implemented as a file system driver**
  - Works with demand paging from executable
  - Check hashes for every page as the page is loaded
- **Hashes stored in system catalog or embedded in file along with X.509 certificate**
- **Check integrity of boot process**
  - Kernel code must be signed or it won't load
  - Drivers shipped with Windows must be certified or contain a certificate from Microsoft

The End