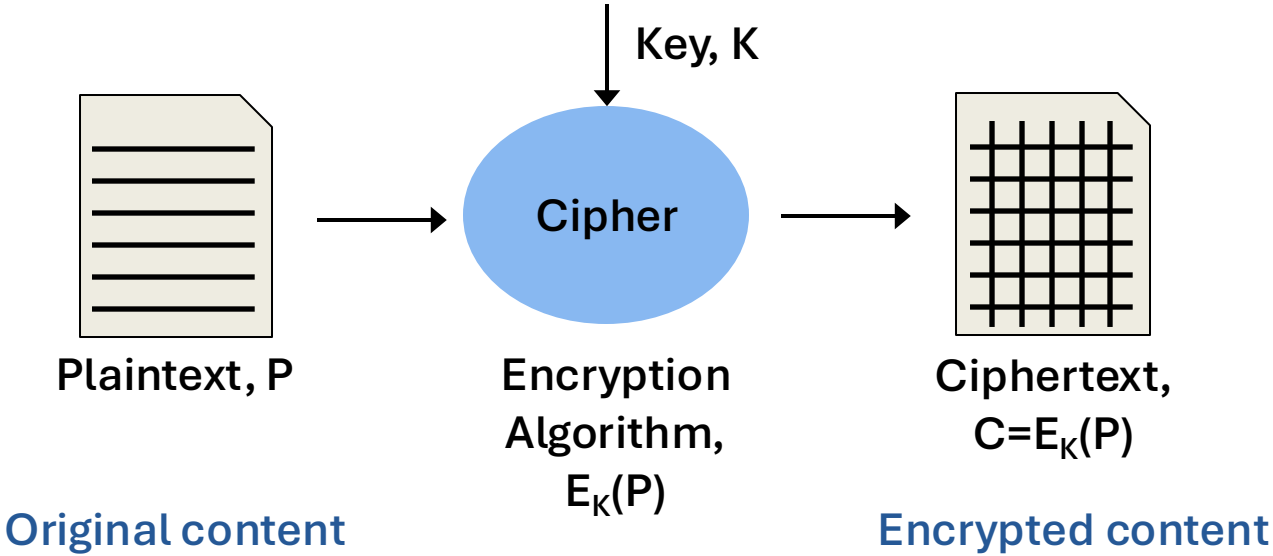

Asymmetric Cryptography,
Data Integrity,
Hybrid Cryptosystems

Symmetric Cryptography: Terms



The same key is used for encryption and decryption

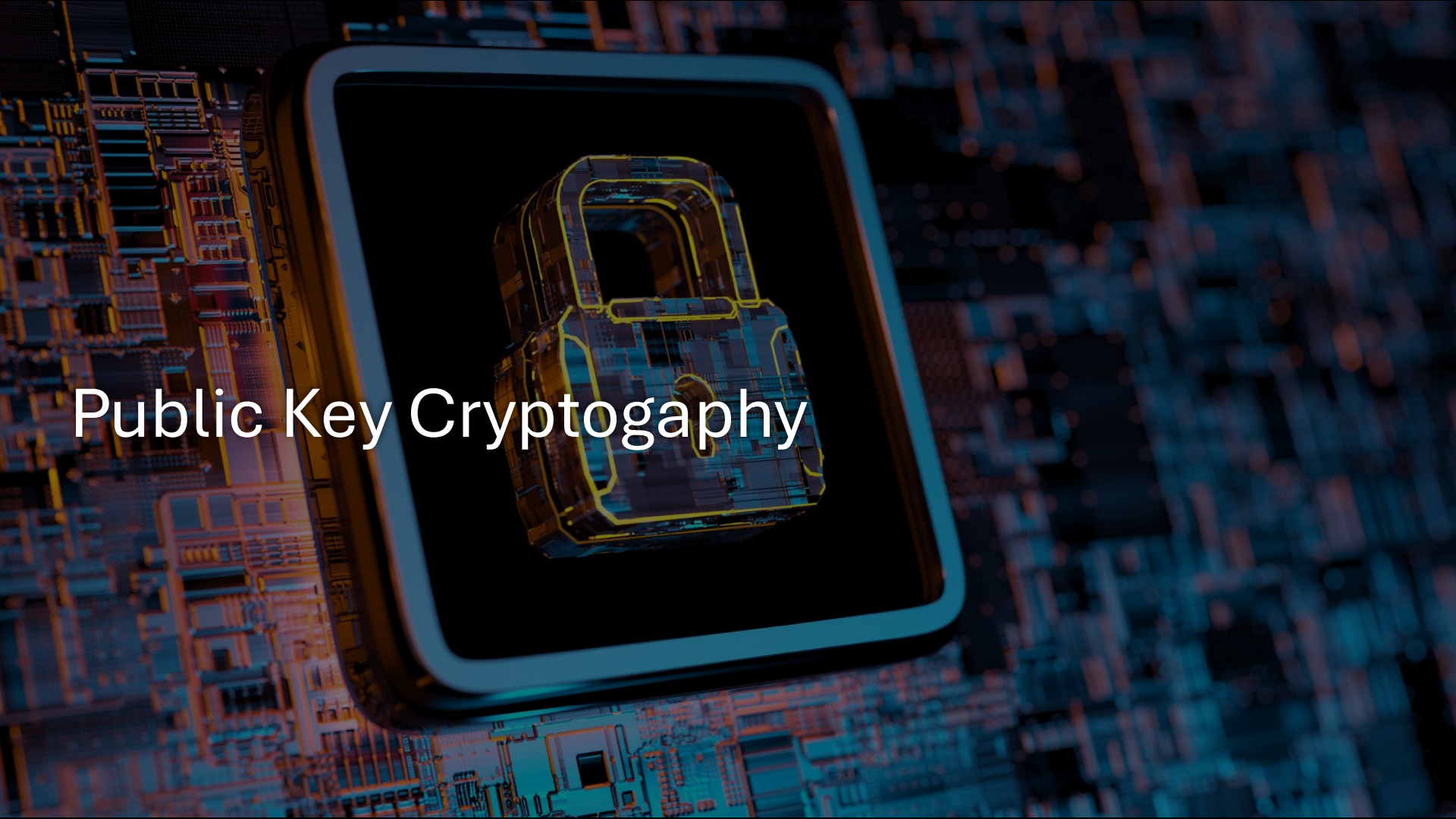
Communicating with symmetric cryptography

Both parties must agree on a secret key, *K*

Key distribution must be secret

Key Distribution

Secure key distribution is the biggest problem with symmetric cryptography


Public Key Cryptogaphy

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 – the 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

Use a one-way function, *B = f (A)*

- Guards get B
  - Enemy cannot compute A if they discover B
- Spies give A, guards compute f(A)
  - If the result is B, the password is correct.

One-way functions

Easy to compute in one direction

Difficult (infeasible) to compute in the other

Example: Middle Squares

A = 18932442986094014771

A^2 = 35843739742170045477960753118 9166182441

Middle square, B = 42170045477960753118

Given A, it is easy to compute B

Given B, it is difficult to compute A

Other One-Way functions

- **Discrete exponentiation – discrete logarithms**
  - $y = g^x \bmod p$
  - Easy to compute for large values of p
  - Hard to find x even when given y, g, and p
- **Elliptic curve multiplication**
  - Given a number k and point P on an elliptic curve and $Q = kP$
  - Easy to compute Q but not feasible to recover k from P and Q

Trapdoor functions

Trapdoor function

- Easy to compute in one direction
- The inverse is **difficult** to compute without extra information

Trapdoor functions

Trapdoor function

- Easy to compute in one direction
- The inverse is **difficult** to compute without extra information

96171919154952919 is the product of two prime #s.

What are they?

Trapdoor functions

Trapdoor function

- Easy to compute in one direction
- The inverse is **difficult** to compute without extra information

96171919154952919 is the product of two prime #s.

If you know one of them is **100225441**

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

Public-key cryptography

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 Cryptography (ECC),
DSS (digital signature standard), Diffie-Hellman

RSA Public Key Cryptography

Ron Rivest, Adi Shamir, Leonard Adleman — 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

RSA algorithm: key generation

1. Choose two random large prime numbers p, q

3, 11

RSA algorithm: key generation

1. Choose two random large prime numbers p, q

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

3, 11

(3-1) x (11-1) = 20

RSA algorithm: key generation

1. Choose two random large prime numbers p, q
2. Compute the product $n = pq$ and $\phi(n) = (p - 1)(q - 1)$
3. Choose the **public exponent**, e, such that:
   $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$

3, 11

(3-1) x (11-1) = 20

Choose e=7

RSA algorithm: key generation

1. Choose two random large prime numbers p, q
2. Compute the product $n = pq$ and $\phi(n) = (p - 1)(q - 1)$
3. Choose the **public exponent**, e, such that:
   $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$
4. Compute the **secret exponent**, d such that:
   $ed = 1 \bmod \phi(n)$
   $d = e^{-1} \bmod ((p - 1)(q - 1))$

3, 11

(3-1) x (11-1) = 20

Choose e=7

Find d: 7d = 1mod 20
7 x 3 = 21 ≡ 1 mod 20
d = 3

RSA algorithm: key generation

1. Choose two random large prime numbers p, q
2. Compute the product $n = pq$ and $\phi(n) = (p - 1)(q - 1)$
3. Choose the **public exponent**, e, such that:
   $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$
4. Compute the **secret exponent**, d such that:
   $ed = 1 \bmod \phi(n)$
   $d = e^{-1} \bmod ((p - 1)(q - 1))$
5. **Public key** = (e, n)
   Private key = (d, n)
   Discard p, q, $\phi(n)$

3, 11

(3-1) x (11-1) = 20

Choose e=7

Find d, 7d = 1mod 20
7 x 3 = 21 ≡ 1 mod 20
d = 3

Pub key = (3, 33)
Pri key = (7, 33)

RSA Encryption

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

Pub key = (3, 33)
Pri key = (7, 33)

Encrypt

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

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

Decrypt

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

Encrypt 18 with public key:
$18^3 \bmod 33 = 24$

Decrypt 24 with private key:
$24^7 \bmod 33 = 18$

Encrypt 29 with private key:
$29^7 \bmod 33 = 17$

Decrypt 17 with public key:
$17^3 \bmod 33 = 29$

RSA security

The security rests on the difficulty of factoring a large integer

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

If you know the public key (3, 33), can you derive the private key?

RSA Security

Large keys make it difficult to find factors via an exhaustive search

Example: a 2048-bit modulus (n) and secret exponent (d):

n =
0xa709e2f84ac0e21eb0caa018cf7f697f774e96f8115fc2359e9cf60b1dd8d4048d974cdf8422bef6be3c162b04b916f7ea2133f0e3e4e0eee164859bd9c1e0ef0357c142f4f633b4add4aab86c8f8895cd33fbf4e024d9a3ad6be6267570b4a72d2c34354e0139e74ada665a16a2611490debb8e131a6cffc7ef25e74240803dd71a4fcd953c988111b0aa9bbc4c57024fc5e8c4462ad9049c7f1abed859c63455fa6d58b5cc34a3d3206ff74b9e96c336dbacf0cdd18ed0c66796ce00ab07f36b24cbe3342523fd8215a8e77f89e86a08db911f237459388dee642dae7cb2644a03e71ed5c6fa5077cf4090fafa556048b536b879a88f628698f0c7b420c4b7

d =
0x10f22727e552e2c86ba06d7ed6de28326eef76d0128327cd64c5566368fdc1a9f740ad8dd221419a5550fc8c14b33fa9f058b9fa4044775aaf5c66a999a7da4d4fdb8141c25ee5294ea6a54331d045f25c9a5f7f47960acbae20fa27ab5669c80eaf235a1d0b1c22b8d750a191c0f0c9b3561aaa4934847101343920d84f24334d3af05fede0e355911c7db8b8de3bf435907c855c3d7eeede4f148df830b43dd360b43692239ac10e566f138fb4b30fb1af0603cfcf0cd8adf4349a0d0b93bf89804e7c2e24ca7615e51af66dccfdb71a1204e2107abbee4259f2cac917fafe3b029baf13c4dde7923c47ee3fec248390203a384b9eb773c154540c5196bce1

Elliptic Curve Cryptography (ECC)

Key Generation

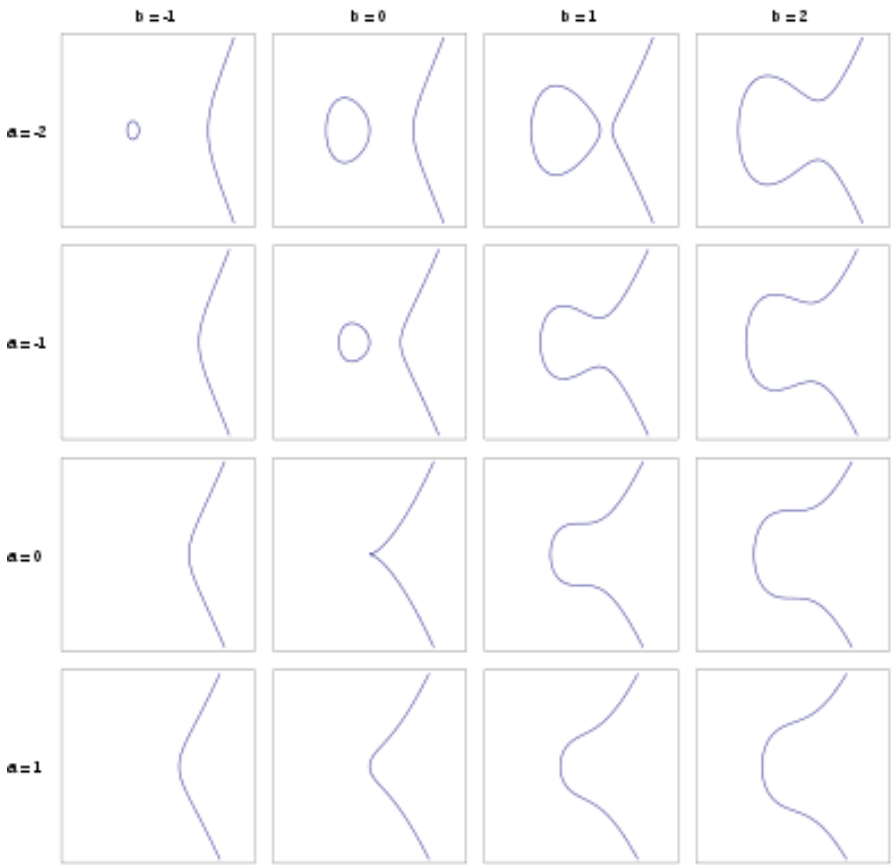
Using discrete numbers, pick

- A prime number as a maximum (modulus)
- A curve equation in the family

$$y^2 = x^3 + ax + b \pmod{p}$$

where p is a large prime number

- A public base point on the curve, G
- **Private key**: random integer, d
- **Public key**: computed from the private key, the base point, and the curve: dG



Catalog of elliptic curves

ECC vs. RSA

- **RSA is still a widely used public key cryptosystem (but fading)**
  - Inertia & widespread implementations
  - 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)

Key length

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

Comparable complexity:

- 3072-bit RSA = 256-bit elliptic curve = 128-bit symmetric cipher
- 15360-bit RSA = 512-bit elliptic curve = 256-bit symmetric cipher

For long-term security, ENISA (EU) and NIST (US) recommend:

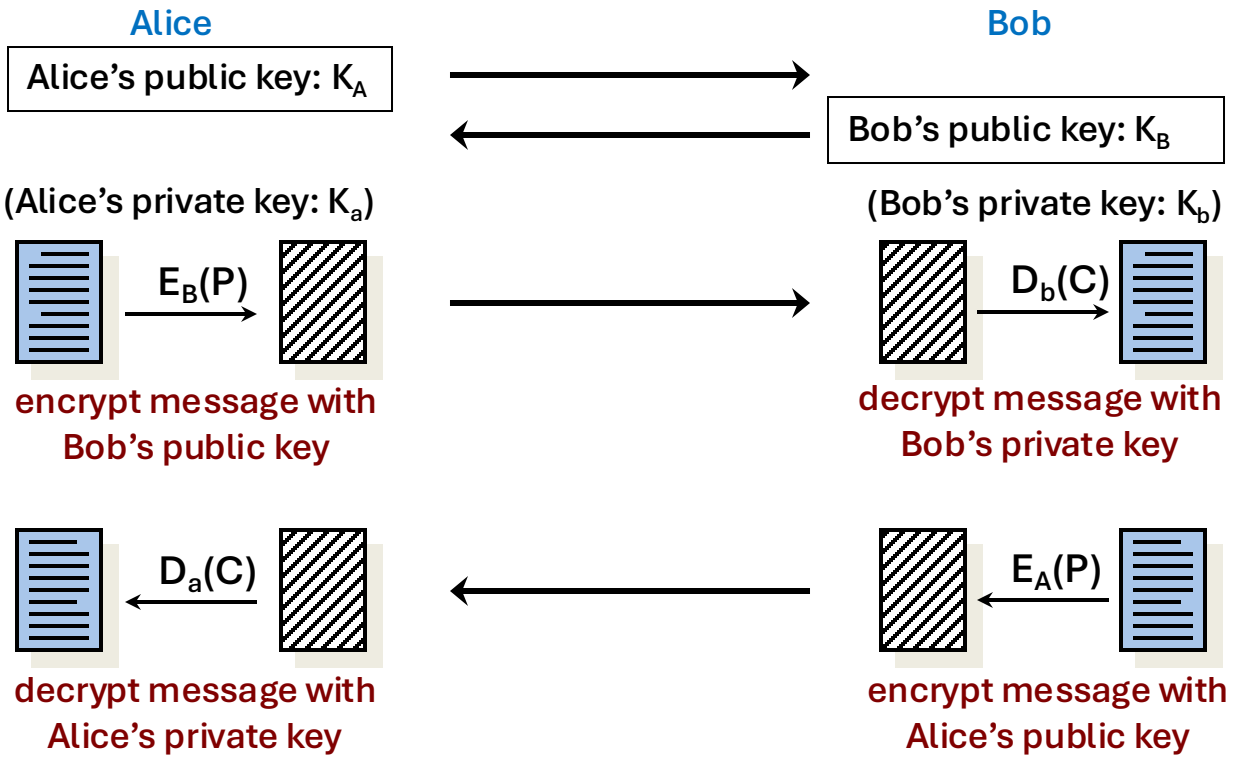
AES: 256-bit keys **RSA: 15,360-bit keys** **ECC: 512 bit-keys**

Communication with public key algorithms

Different keys for encrypting and decrypting

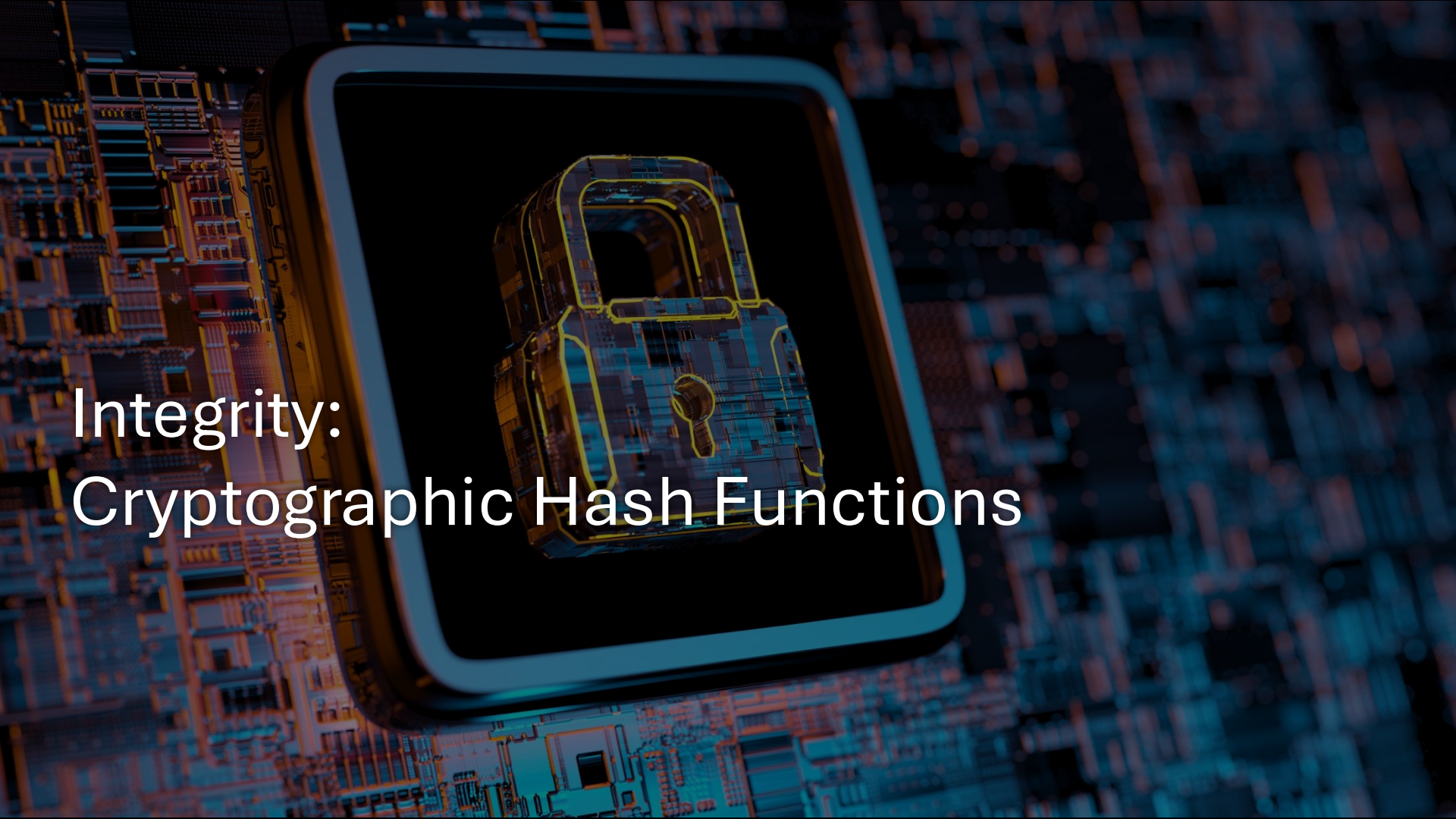
No need to worry about key distribution!

Communication with public key algorithms



Why Not Use Public Key Algorithms for All Encryption?

- **Slow Performance**
- **Ciphertext expansion**
- **Vulnerability to chosen plaintext attacks (or guessing)**
- **Some algebraic relationships may be preserved**



Integrity:
Cryptographic Hash Functions

Cryptographic hash functions - Properties

- **Fixed-length output**
- **Deterministic**
- **Preimage resistant**: Given **H**, it should be infeasible to find **M** such that **H=hash(M)**
- **Second preimage resistant**:
  Given $\mathbf{M_1}$, it should be infeasible to find $\mathbf{M_2}$ such that $\mathbf{hash(M_1)=hash(M_2)}$
- **Collision resistant**: It should be infeasible to find **M**, **M'** such that **hash(M) = hash(M')**
- **Uniform**
- **Avalance effect**
- **Efficient**

Also called digests or fingerprints

Hash functions are the basis of integrity

- **Not encryption**

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**
    - US standard for use with NIST Digital Signature Standard (DSS) – 160-bit hash

Hash Algorithms

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

- **Example:**
  - **Secure Hash Algorithm, SHA-1**
    - Produces 160-bit hash values
- **Successors**
  - **SHA-2** (2001) – **SHA-224, SHA-256, SHA-384, SHA-512**
    - Approved for use with the NIST Digital Signature Standard (DSS)
  - **SHA-3** (2015)
    - Can be substituted for SHA-2

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

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
```

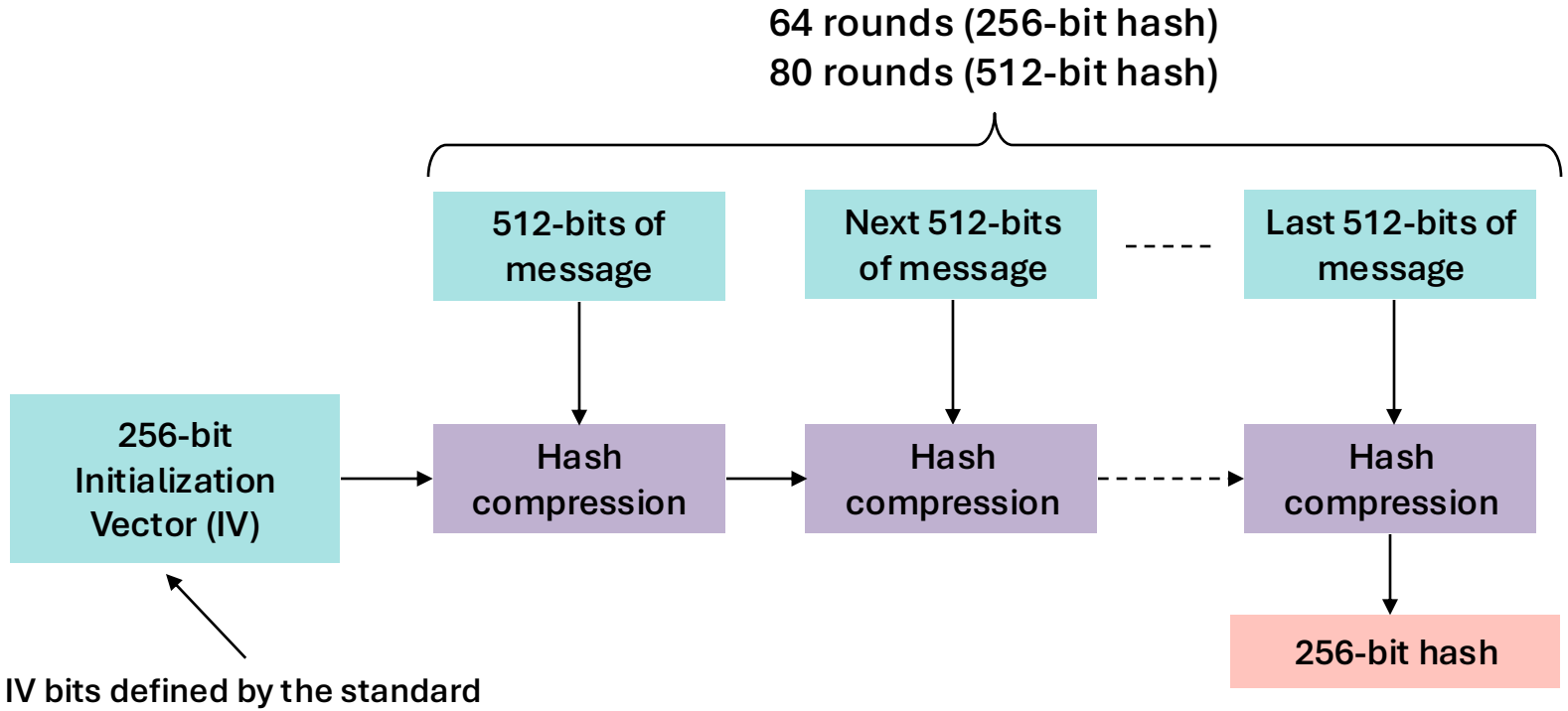
Example: SHA-1 Overview

- **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 – 80 rounds**
  - Expand the 16 32-bit words into 80 32-bit words via XORs & shifts
  - Iterate 80 times to create a hash for this chunk
  - Add this hash chunk to the result so far

SHA-2 Overview



Popular (& formerly popular) Hash Functions

MD5
- 128 bits
- Linux passwords used to use this

SHA-1
- 160 bits – was widely used: checksum in Git & torrents

Git also supports SHA-256

SHA-2
- Designed by the NSA; published by NIST
- Variations: SHA-224, SHA-256, SHA-384, SHA-512
- Linux passwords (SHA-512)
- Bitcoin (SHA-256)

Believed to be secure

SHA-3
- 256 & 512 bit

Believed to be secure

bcrypt
- Blowfish cipher used for bcrypt password hashing in OpenBSD

3DES
- Linux passwords used to use this

Designed to be slow!

Hash Collisions

Hashes are *collision resistant*, but collisions can occur

Pigeonhole principle

- A hash is a fixed-size number of bits
- Every possible permutation of an arbitrary number of bytes cannot fit into every permutation of 32 bytes!

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: 365 ÷ 2 = 183

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: 365 ÷ 2 = 183

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)

- For SHA-256, # operations =

$$2^{128} = 3.4 \times 10^{38}$$

This shows why collisions are guaranteed in theory but practically unachievable in secure hash functions

Data Integrity

How do we detect that a message has been tampered?

- A cryptographic hash acts as a strong 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

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

But an attacker can create a new hash for a modified message

Integrity: Message Authentication Codes and Digital Signatures

MAC (also called a Keyed Hash)

Create a checksum that relies on a key for validation

Message Authentication Code (MAC)

Two forms:

hash-based & block cipher-based

HMAC: Hash-based MAC – RFC 2104

A MAC can be created 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…)

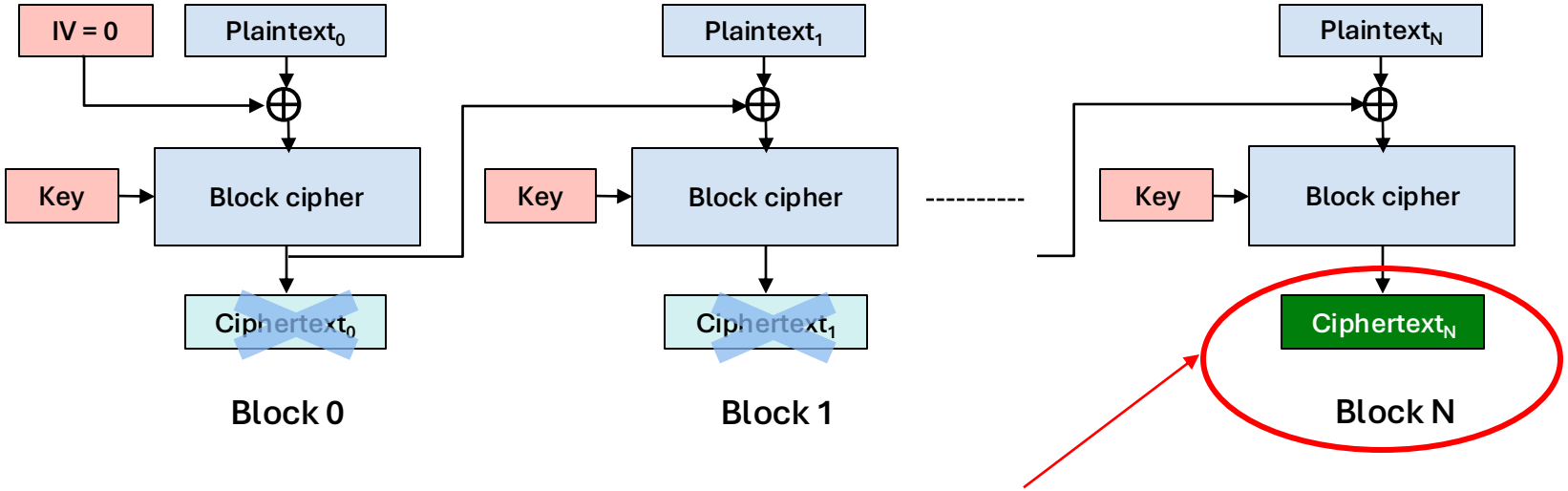
k = secret key

m = message

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

Basically, incorporate a key into the message before hashing it

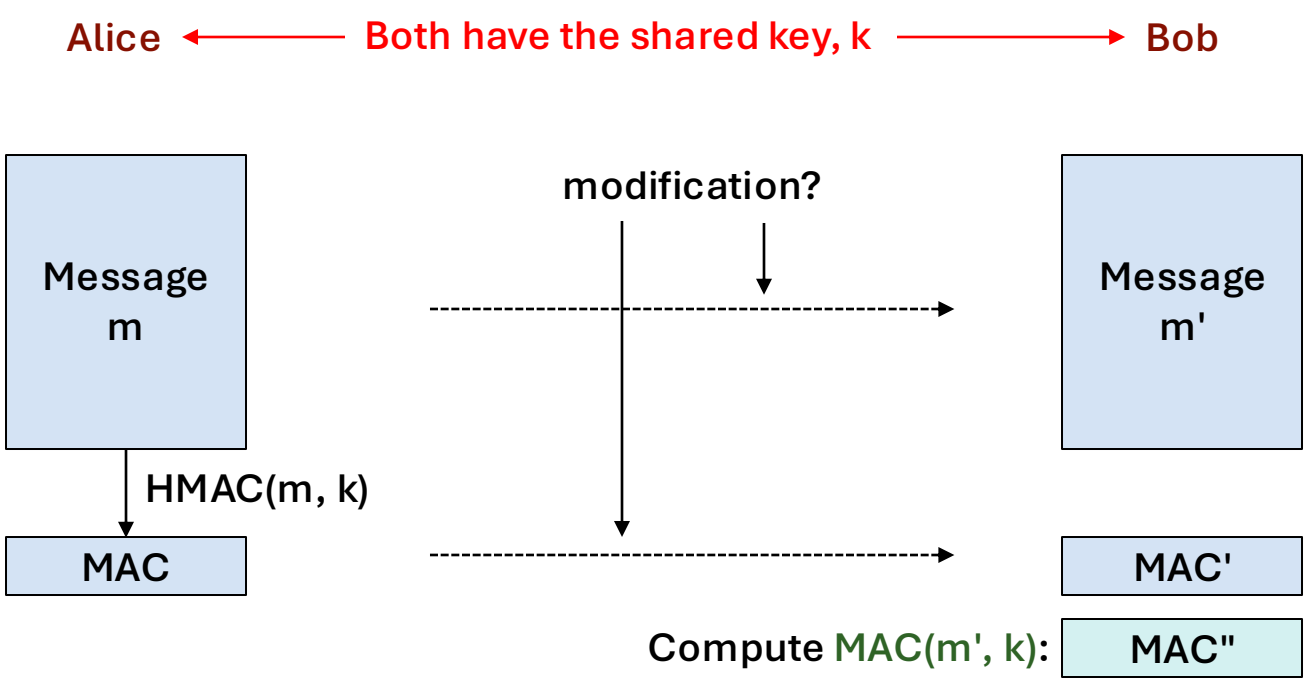
Block Cipher Based MAC: CBC-MAC and CMAC



MAC = final ciphertext block – others are discarded

CMAC – Cipher-based Message Authentication Code

Using a MAC


Alice
Both have the shared key, k
Bob
Message
m
HMAC(m, k)
MAC
modification?
Message
m'
MAC'
Compute MAC(m', k):
MAC"

Authenticated Encryption with Associated Data (AEAD)

Encryption + Integrity in one step

AEAD adds an authentication tag to the ciphertext

Two popular types

– AES-GCM: CTR mode + hash

– ChaCha20-Poly1305: 128-bit tag – *f*(message and derived key)

Digital Signatures

MACs rely on a shared key

Anyone with the key can modify the message and create the correct MAC

Digital Signatures

MACs rely on a shared key

Anyone with the key can modify the message and create the correct MAC

Digital signature properties

- Only you can sign a message, but anyone can validate it
- You cannot copy the signature from one message to another
- If the message is modified, the signature will be invalid
- An adversary cannot forge a signature

Digital Signature Primitives

1. **Key generation**

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

2. **Signing**

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

3. **Validation**

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

Digital Signature Primitives

1. **Key generation**

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

2. **Signing**

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

3. **Validation**

   is_valid := **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

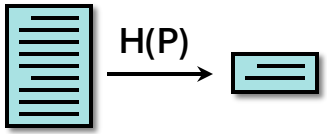
$$D_A(S) = D_A(E_a(M)) = M$$

Nobody but Alice can create S

Conceptual View of Using Digital Signatures

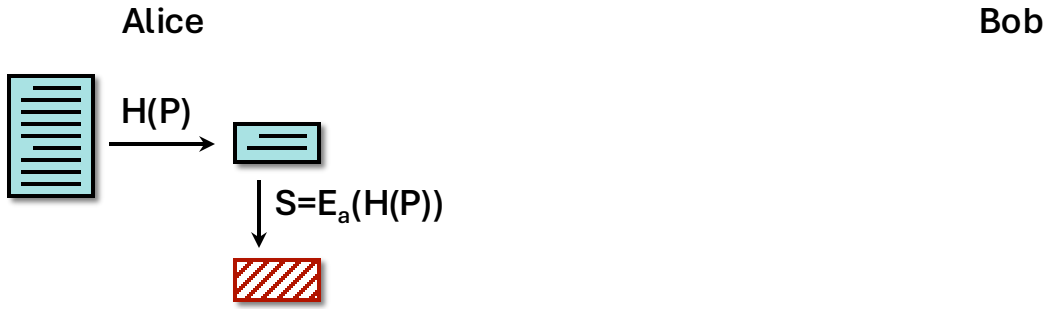
Alice

Bob



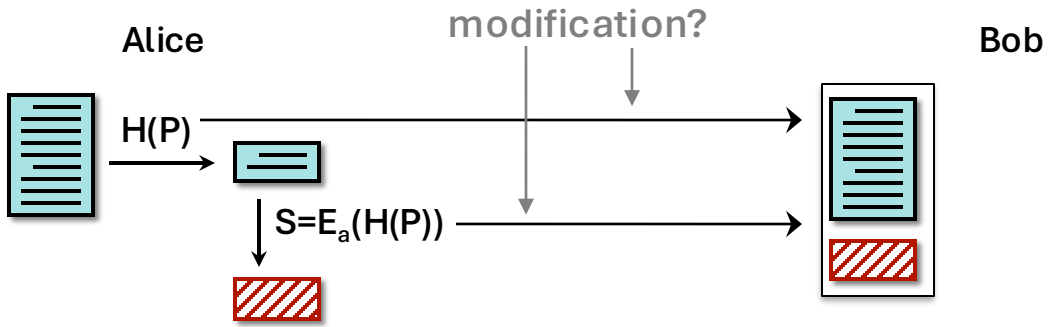
Alice generates a hash of the message, H(P)

Conceptual View of Using Digital Signatures



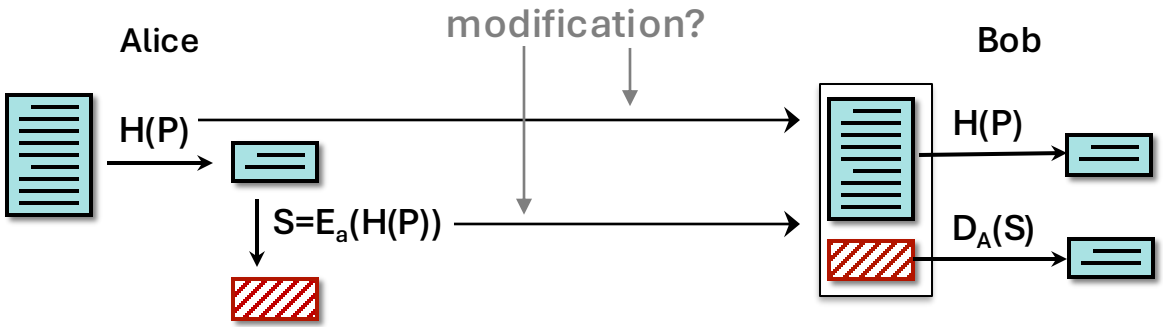
Alice encrypts the hash with her private key
This is her signature

Conceptual View of Using Digital Signatures



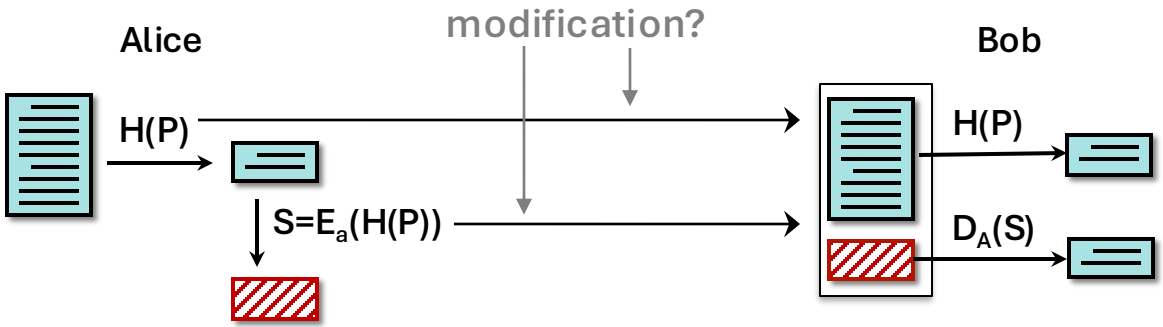
Alice sends Bob the message & the encrypted hash

Conceptual View of Using Digital Signatures



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

Conceptual View of Using Digital Signatures



If the hashes match, the signature is valid

⇒ the encrypted hash must have been generated by Alice

Popular Digital Signature Algorithms

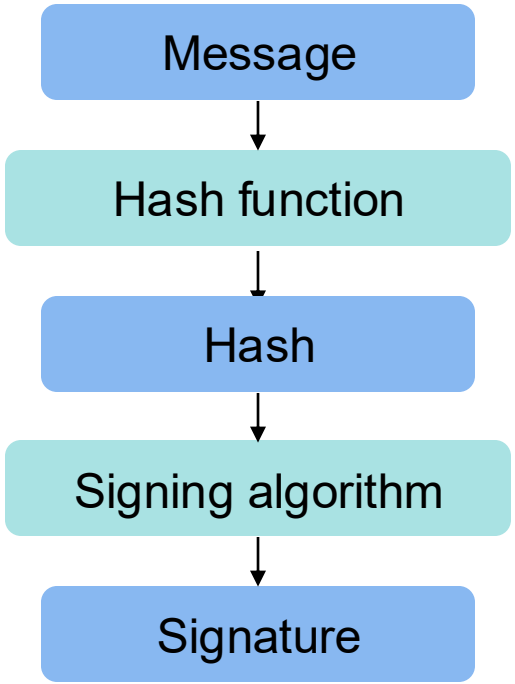
Digital Signature Algorithms improve security vs. public key encryption

DSA: Digital Signature Algorithm

ECDSA: Elliptic Curve Digital Signature Algorithm

EdDSA: Edwards-curve Digital Signature Algorithm

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



Digital signatures & non-repudiation

Digital signatures provide

- **Non-repudiation**
- **Proof of integrity**

Public Keys as Identities

A public signature verification key can be treated as an identity

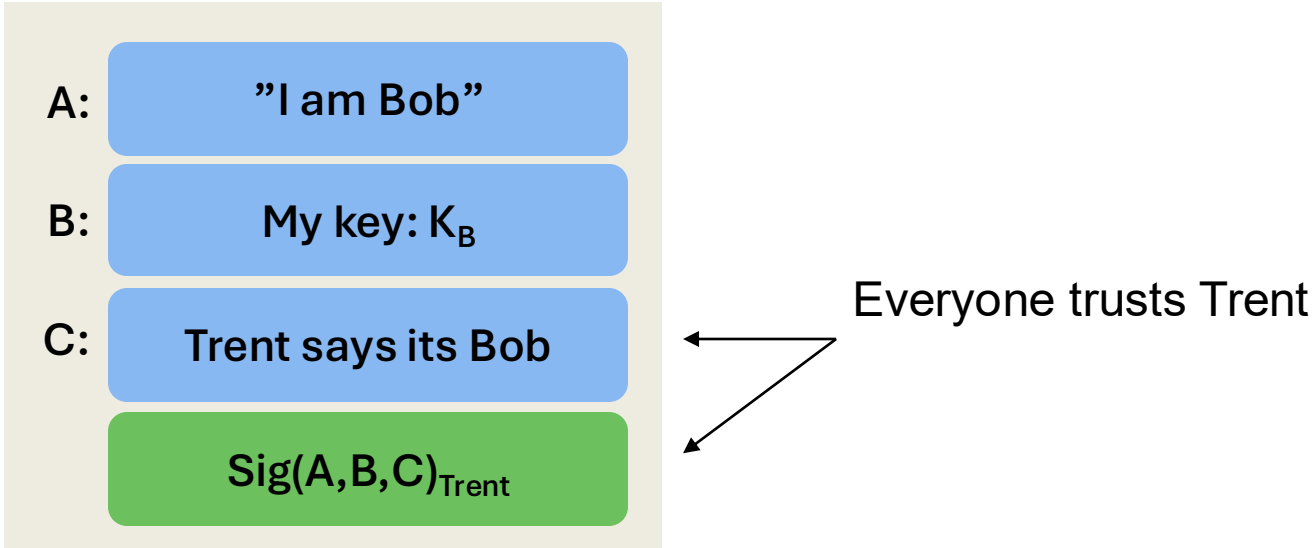
Certificates: Identity Binding

Identity Binding

- **How does Alice know Bob's public key is really his?**
- **Get it from a trusted server?**

Identity Binding – Another Option

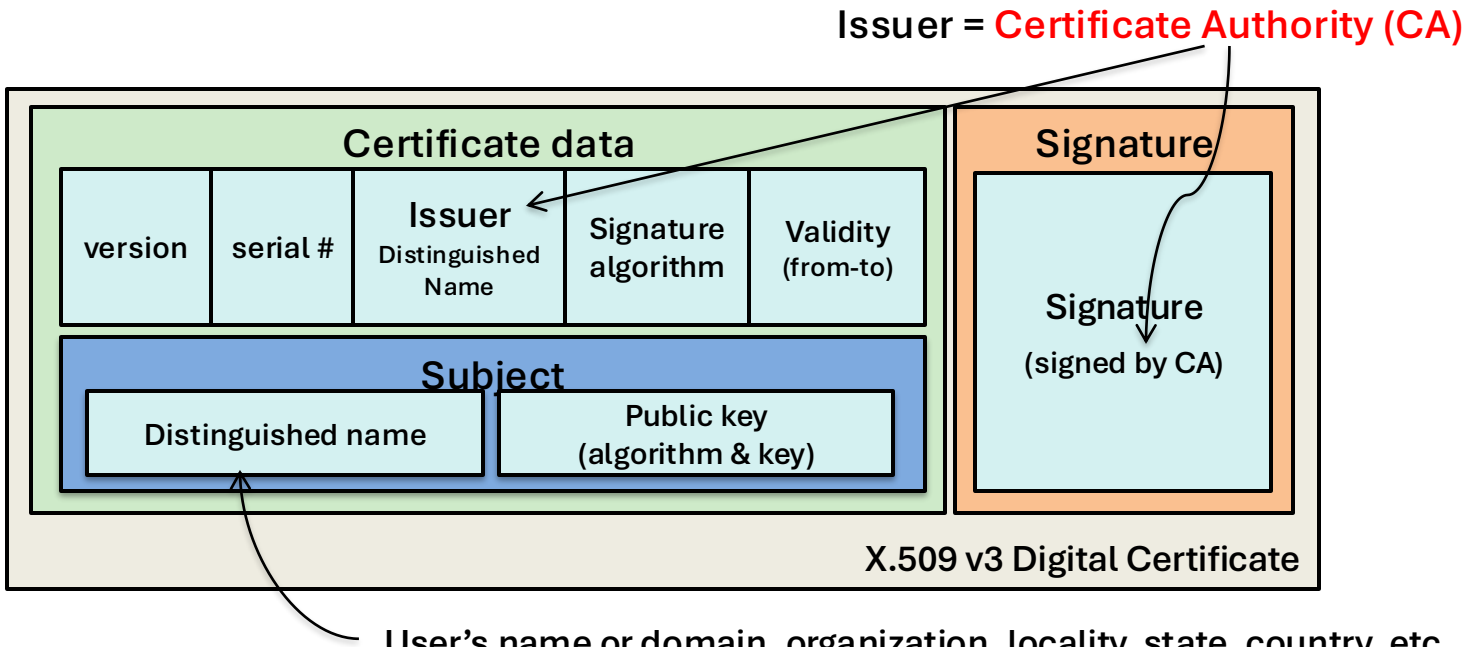
- **Have a trusted party sign Bob's public key**
- **Once signed, it is tamper-proof**
- **But we need to know it's Bob's public key and who signed it**
  - Create & sign a data structure



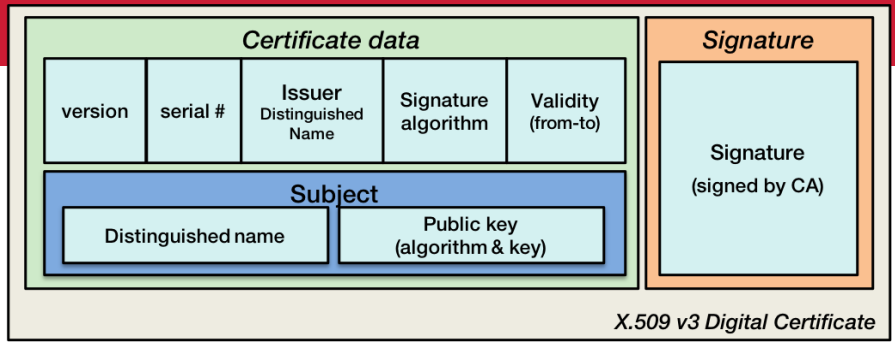
X.509 Certificates

ISO introduced a set of authentication protocols

X.509: Structure for public key certificates:



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 (SHA-256)
   - Use CA's public key to validate the CA's signature

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

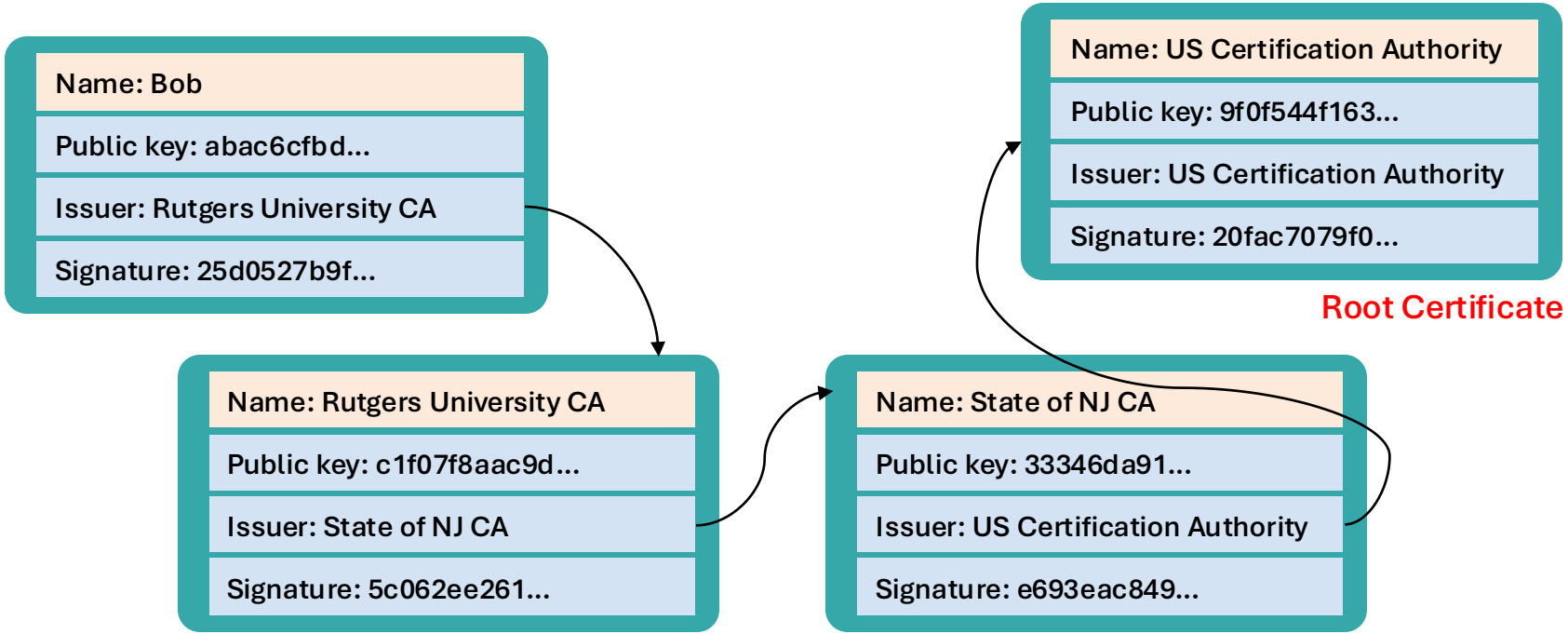
Certificates prevent someone from using a phony public key to masquerade as another person/company

…if you trust the CA

Certificate Authorities (CAs)

How do you know the public key of the CA?

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



Certificate Authorities (CAs)

But trust must start somewhere

You need a public key you can trust – this is the root certificate

- **Apple Keychain**
- Windows **Certificate Store** via the Microsoft Management Console (mmc)
- Android **Credential Storage**

Key revocation

- **Used to invalidate certificates before expiration time**
- **Certificate revocation list (CRL)**
- **Problems**
  - Authorization
  - Delivery/synchronization
  - Client attention

Code Integrity

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**

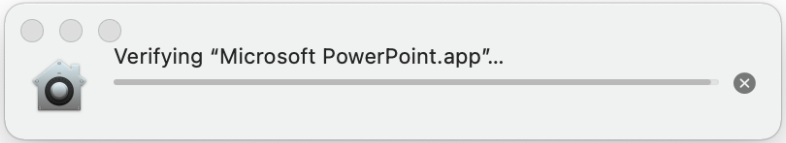
- **Windows since XP: Microsoft Authenticode**
  - **SignTool** command
  - Hashes stored in system catalog or signed & embedded in the file
  - Microsoft-tested drivers are signed
- **macOS**
  - **codesign** command
  - Hashes & certificate chain stored in file
- **Also Linux, Android, & iOS**

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

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

Diffie-Hellman Key Exchange (DHKE)

Diffie-Hellman Key Exchange (DHKE)

Key distribution algorithm

- Share a secret key over a non-secure channel
- Based on the difficulty of computing discrete logarithms in a finite field vs. the ease of calculating exponents

Negotiate a secret common key without fear of eavesdroppers

Diffie-Hellman Key Exchange (DHKE)

- **All arithmetic performed in a field of integers modulo some large number**

- **Both parties agree on**
  1. a **large prime number *p***
  2. 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 Key Exchange (DHKE)

- 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) (Alice's private key) mod p

Diffie-Hellman Key Exchange (DHKE)

- 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) (Bob's private key) mod p

Diffie-Hellman Key Exchange (DHKE)

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

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

- expanding:

$$K = Y_B^{X_A} \bmod p$$
$$= (\alpha^{X_B} \bmod p)^{X_A} \bmod p$$
$$= \alpha^{X_B X_A} \bmod p$$

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

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

- expanding:

$$K = Y_B^{X_B} \bmod p$$
$$= (\alpha^{X_A} \bmod p)^{X_B} \bmod p$$
$$= \alpha^{X_A X_B} \bmod p$$

$$\boldsymbol{K = K'}$$

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 $Y_A = 57^{300} \bmod p = 282$
- 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 $Y_B = 57^{25} \bmod p = 1046$
- 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

Why are we sill looking at this?

Why not just use RSA or ECC to encrypt a random key?

CS 419: Computer Security

Hybrid Cryptosystems

Strengths & Weaknesses

- **Public key algorithms**:
  - Good at exchanging secrets
  - Bad at encrypting large amounts of data

- **Symmetric algorithms**:
  - Good at bulk encryption but require a shared key

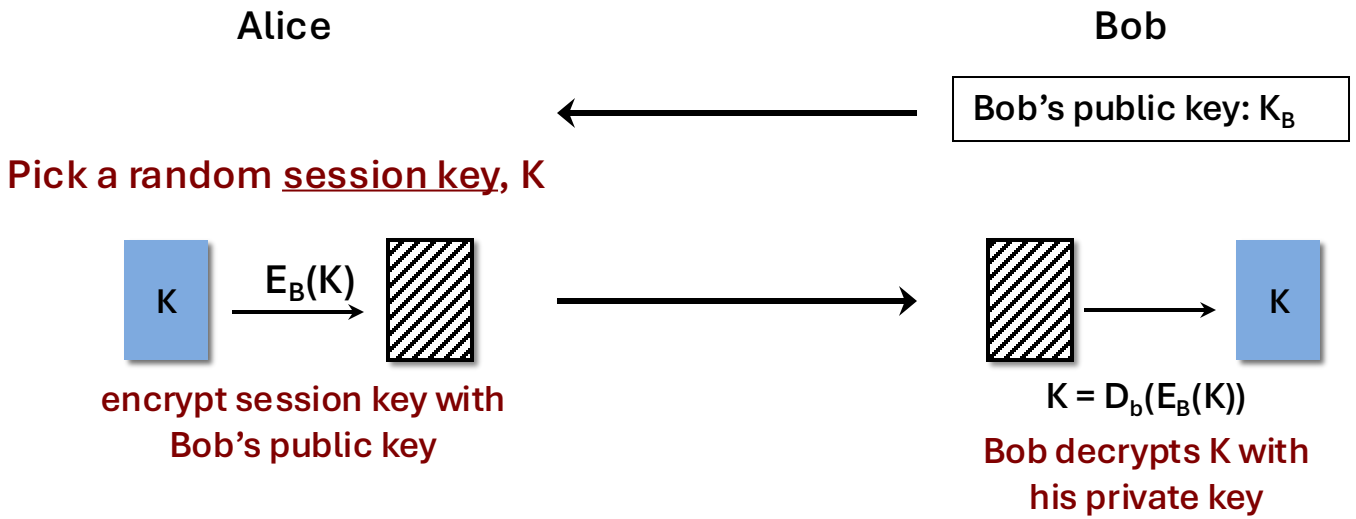
Hybrid Cryptosystems

- **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 never used to encrypt messages

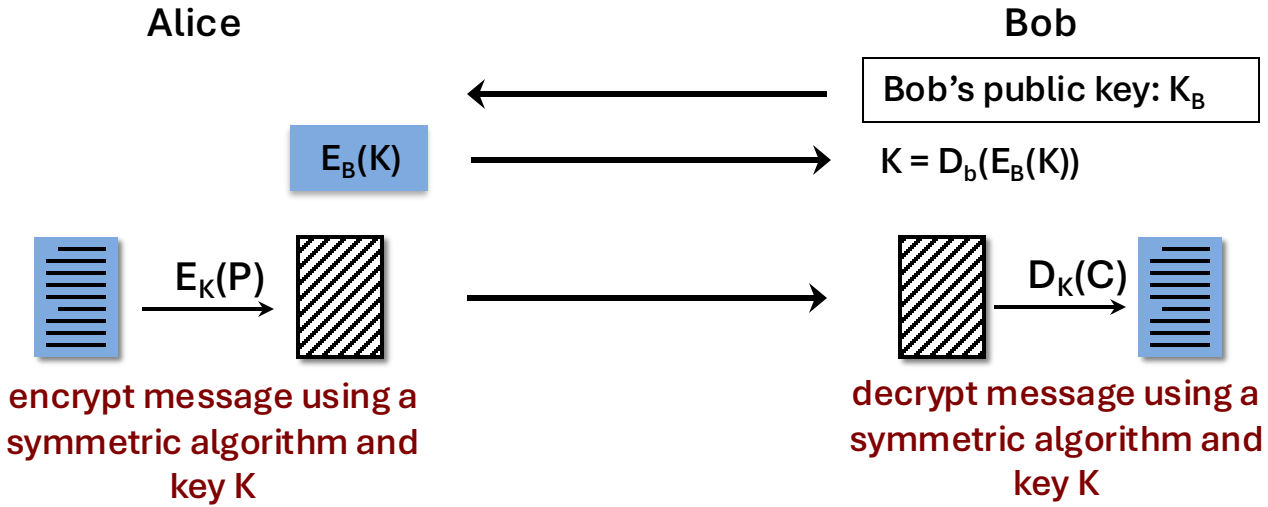
– MUCH slower; vulnerable to *chosen-plaintext* and *algebraic attacks*

Communication with a hybrid cryptosystem

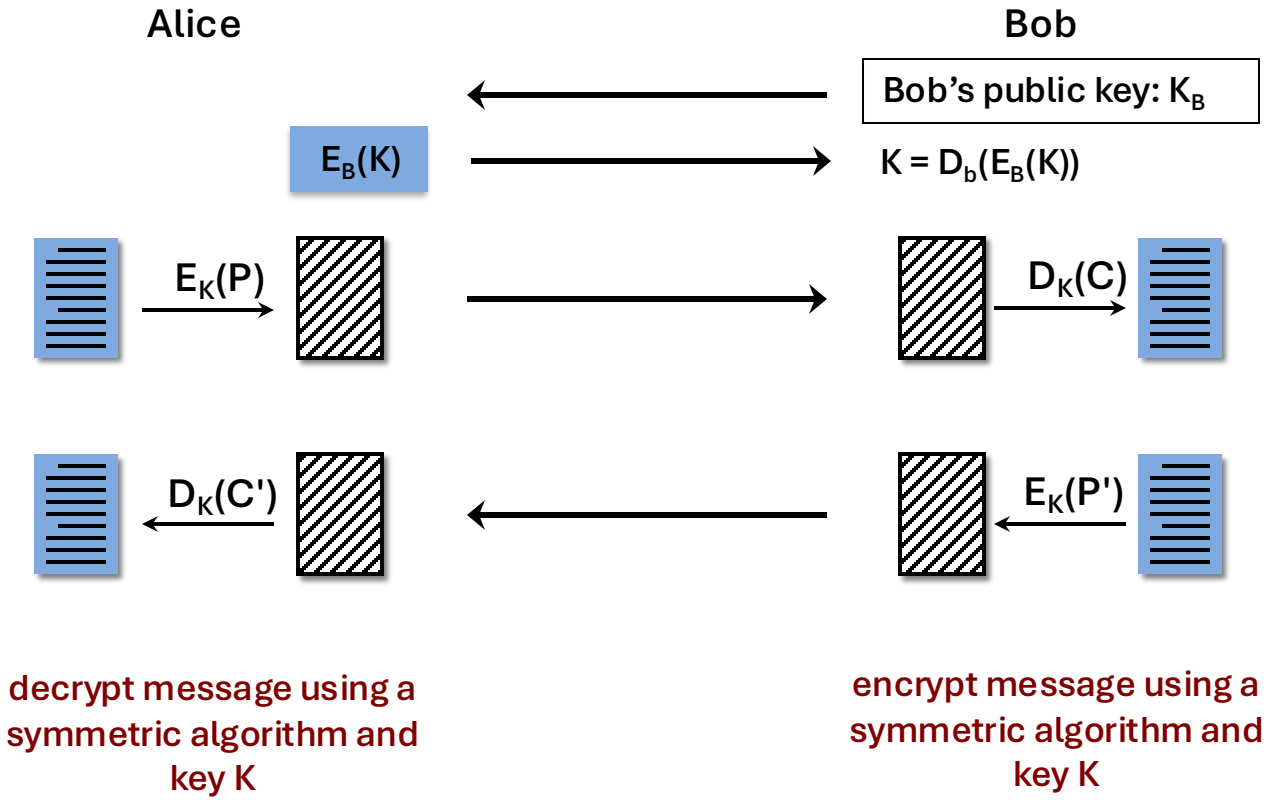


Now Bob knows the secret session key, K

Communication with a hybrid cryptosystem

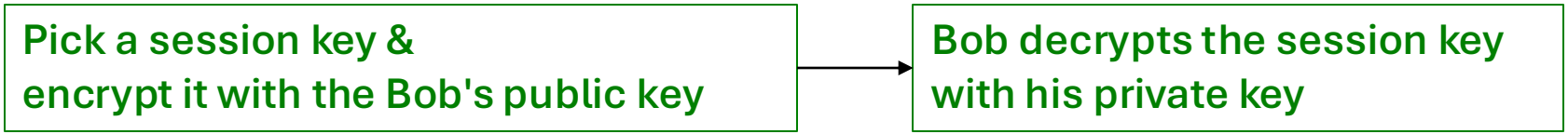


Communication with a hybrid cryptosystem



Forward Secrecy

Private keys need to be protected



Suppose an attacker steals Bob's private key

- Future & past messages can be compromised

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

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 ephemeral keys for key exchange
+ session keys for communication

Diffie-Hellman key exchange is commonly used for key exchange

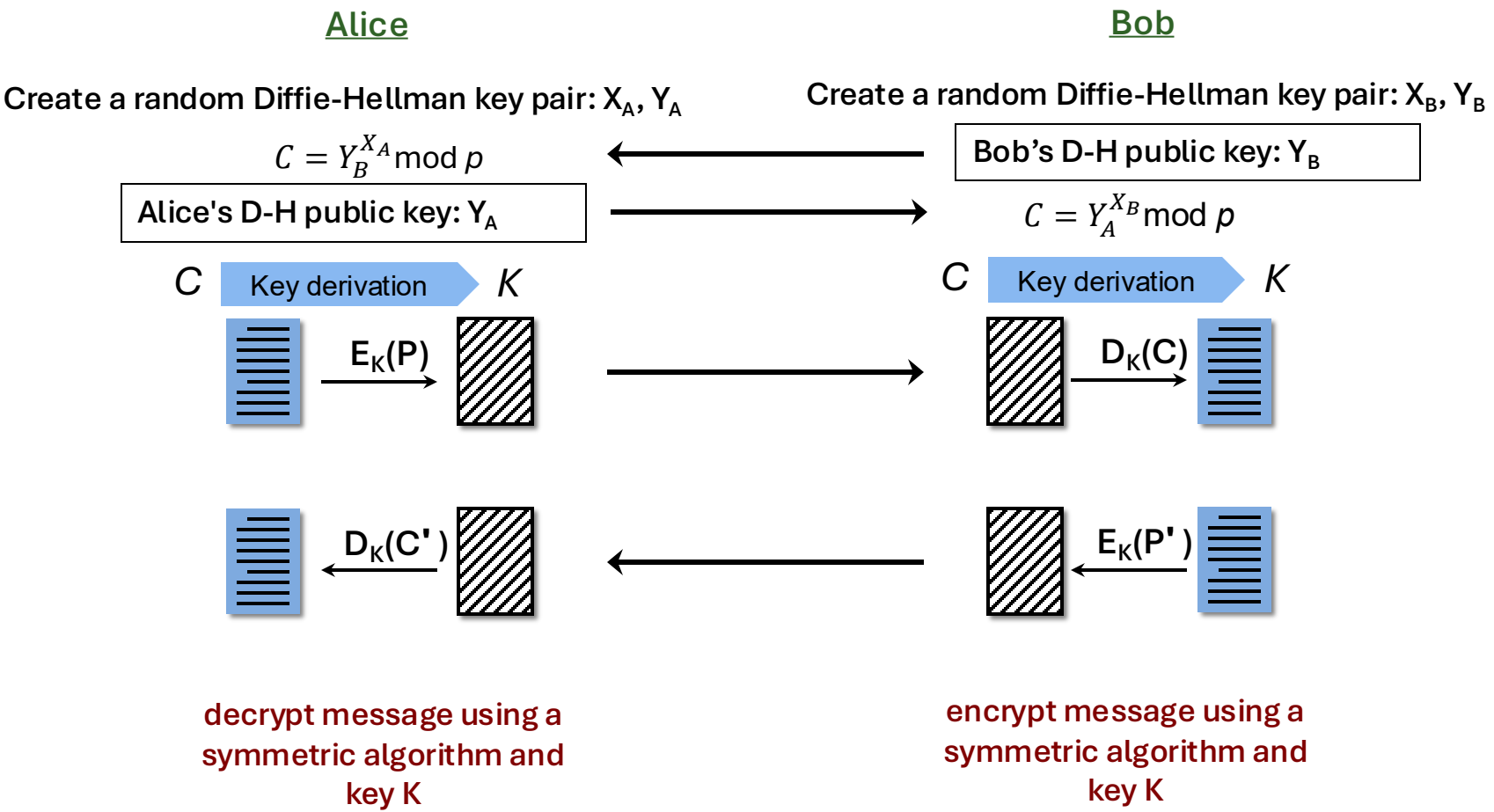
- Generate a set of keys per session
- **Not recoverable as long as private keys are thrown away**

Long-term key: Used for identity verification, authentication

Ephemeral key:
Used for establishing a session key – thrown away immediately

Session key: Used to encrypt data for a single session

Communication with a hybrid cryptosystem (DHKE)



Cryptographic systems: summary

- **Symmetric ciphers**
  - Based on SP-networks (usually) = substitution & permutation sequences

Cryptographic systems: summary

- **Symmetric ciphers**
  - Based on SP-networks (usually) = substitution & permutation sequences
- **Asymmetric ciphers – public key cryptosystems**
  - Based on trapdoor functions

Cryptographic systems: summary

- **Symmetric ciphers**
  - Based on SP-networks (usually) = substitution & permutation sequences
- **Asymmetric ciphers – public key cryptosystems**
  - Based on trapdoor functions
- **Hybrid cryptosystem**
  - Public key algorithm for key exchange, symmetric algorithm for messages

Cryptographic systems: summary

- **Symmetric ciphers**
  - Based on SP-networks (usually) = substitution & permutation sequences
- **Asymmetric ciphers – public key cryptosystems**
  - Based on trapdoor functions
- **Hybrid cryptosystem**
  - Public key algorithm for key exchange, symmetric algorithm for messages
- **Key establishment algorithms**
  - Diffie-Hellman
  - Public key

} **Enables secure communication without knowledge of a shared secret**

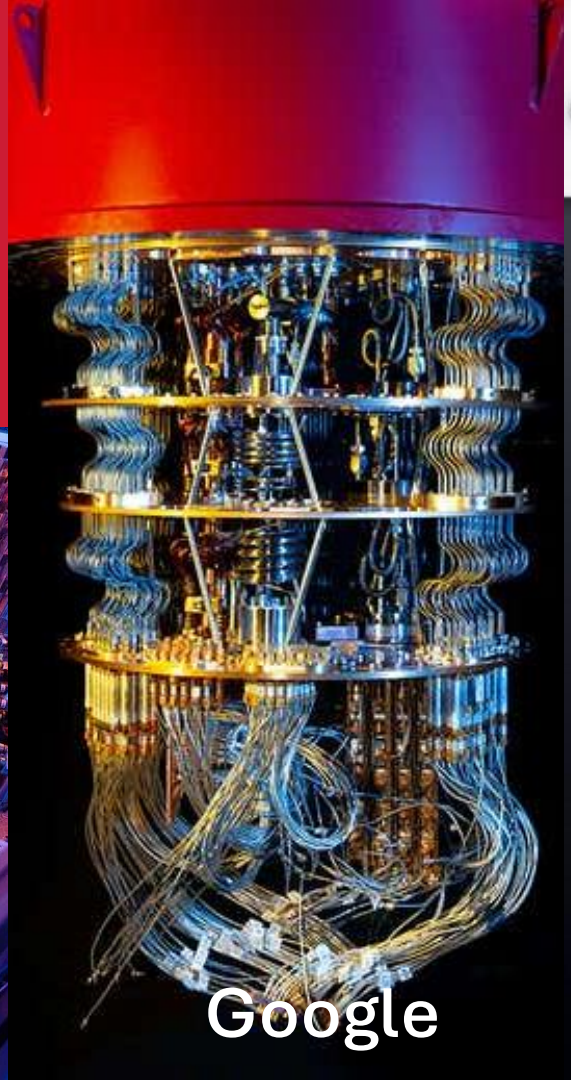
Cryptographic systems: summary

- **Symmetric ciphers**
  - Based on SP-networks (usually) = substitution & permutation sequences
- **Asymmetric ciphers – public key cryptosystems**
  - Based on trapdoor functions
- **Hybrid cryptosystem**
  - Public key algorithm for key exchange, symmetric algorithm for messages
- **Key establishment algorithms**
  - Diffie-Hellman
  - Public key

  } **Enables secure communication without knowledge of a shared secret**
- **Forward secrecy**
  - Establish session key via ephemeral keys

Looking ahead

Quantum Computers



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 efficiently**
  - Diffie-Hellman key exchange & ECC will not be secure

Not all is bad

Symmetric cryptography is largely immune to attacks

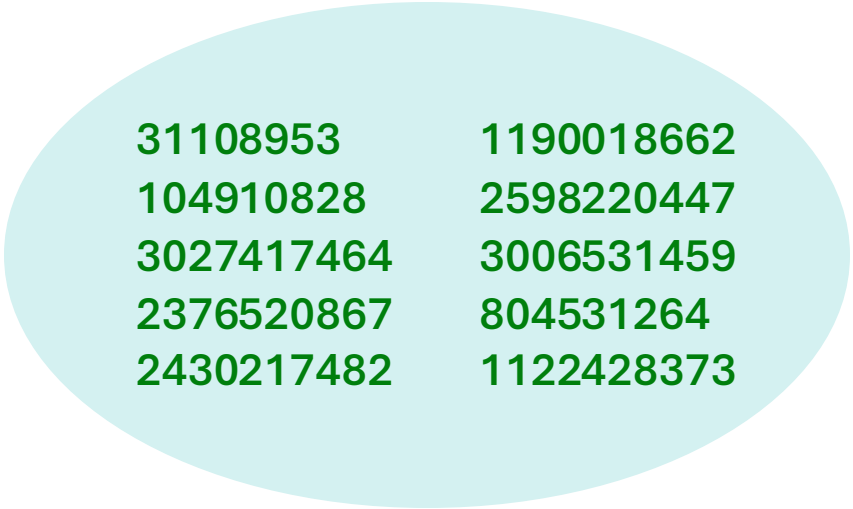
Some optimizations are predicted:

Crack a symmetric cipher in $2^{n/2}$ vs. 2^{n} iterations

Quantum-proofing cryptography

Quantum computing is not faster at everything

Only a few algorithms currently identified where quantum computing offers an advantage



Which 3 numbers sum to 5656746864?

NIST Releases First Post-Quantum Encryption Standards

August 13, 2024: Releases first set of standards:
CRYSTALS-Kyber, CRYSTALS-Dilithium, Sphincs+ and FALCON

1. **FIPS 203 - ML-KEM**
   Module-Lattice-Based Key-Encapsulation Mechanism,
   based on CRYSTALS-Kyber algorithm

2. **FIPS 204 – ML-DS**
   Module-Lattice-Based Digital Signature Algorithm,
   based on the CRYSTALS-Dilithium algorithm

3. **FIPS 205 – SLH-DSA – *backup to ML-DS***
   Stateless Hash-Based Digital Signature Algorithm)

4. **(draft) FIPS 206 – FN-DSA**
   Fast-Fourier Transform over NTRU-Lattice-Based Digital Signature Algorithm

5. **(2025 proposal) HQC – *backup to ML-KEM***
   Key Encapsulation Mechanism based on error correcting codes

The End

The End

The End

The End