Symmetric Encryption and Public-Key Cryptography

CS 465 · Introduction to Security and Privacy · Fall 2026

BYU


SCA SUSANNA
SANCTA CATARINA
SCA Margarita

The big idea

We use symmetric cryptography to encrypt data. We use public keys to exchange session keys and sign. Real systems combine the two with vetted designs.

Symmetric encryption

AES-GCM

One shared key. Fast enough for bulk data, and GCM adds integrity.

Public-key cryptography

RSA and signatures

Each person has a key pair. Used to deliver a session key and to sign messages.

Combined in practice

Hybrid encryption

A public key delivers the session key; AES-GCM protects the data. Use TLS instead of building your own.

Use vetted cryptography and assume attackers know your design

Don’t design your own

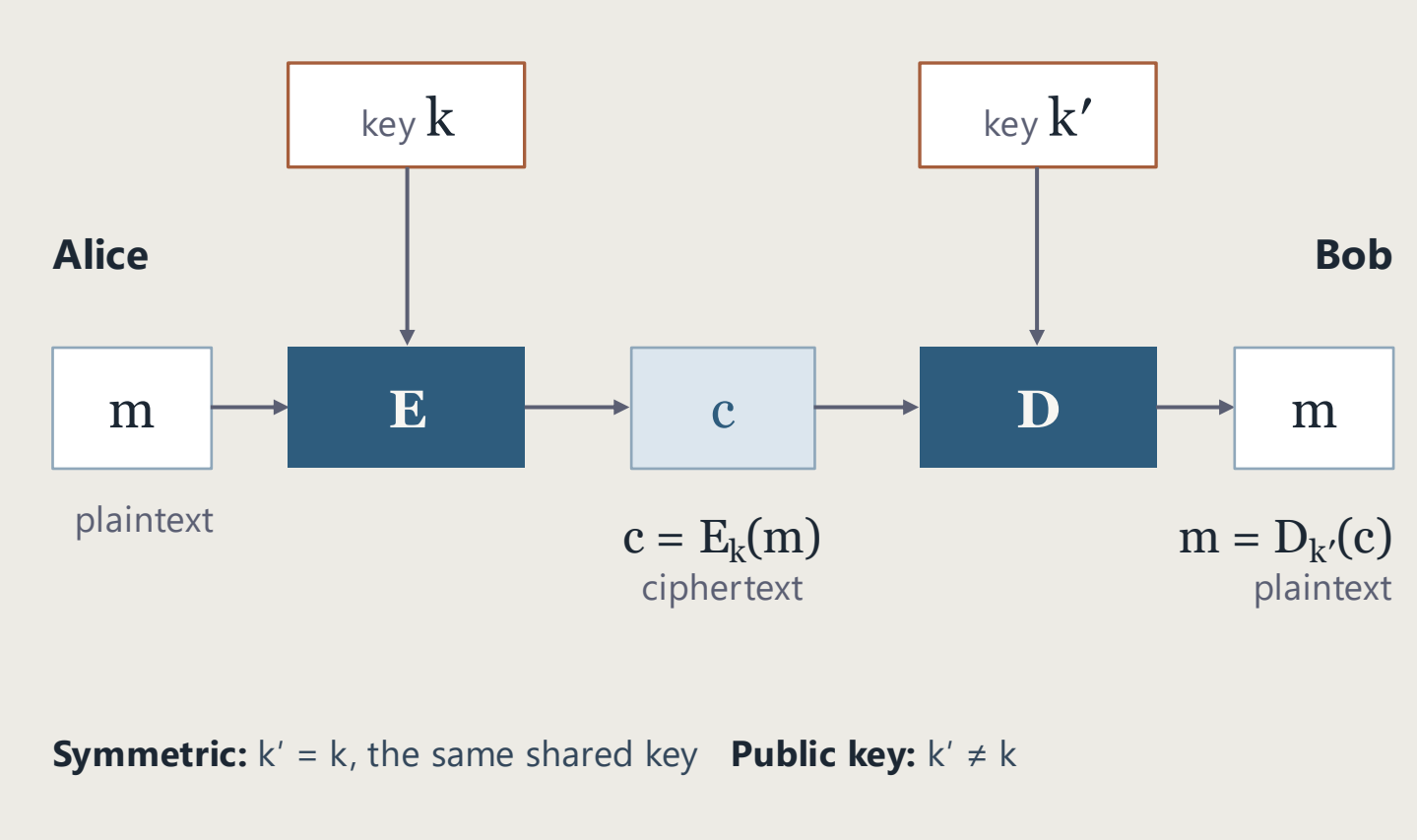
Don’t design your own cryptographic algorithms or protocols, unless you are doing PhD-level research or have become a cryptography expert.

Use well-studied standards and libraries instead.

Kerckhoffs’s principle

> “Design your system assuming that your opponents know it in detail.”
>
> Steven Bellovin’s restatement
>
> Security should rest on the secrecy of the key alone.

Notation shows which key each operation uses



$$c = E_k(m)$$

$$m = D_{k'}(c)$$

m: plaintext
c: ciphertext
E, D: encrypt, decrypt
k, k': encryption, decryption keys

Notation varies by author.

Papers and websites define their symbols before using them, so read the definitions first.

Redrawn after van Oorschot, Computer Security and the Internet, Fig. 2.1

Brute force broke 56-bit DES in under a day

≈ 20 hours

to try every possible 56-bit DES key at a trillion keys per second

$2^{56} \approx 7.2 \times 10^{16}$ keys

$\div 10^{12}$ keys per second $\approx 7.2 \times 10^{4}$ s

In January 1999, distributed.net and the Electronic Frontier Foundation publicly broke a DES key in 22 hours 15 minutes.

The defense against brute force attacks is a large key space.

Attacks are classified by what the attacker can see and do

Attack model	What the attacker has
Ciphertext-only	Only ciphertext; must recover the plaintext or key
Known-plaintext	Examples of matching plaintext and ciphertext
Chosen-plaintext	Can pick plaintext and see its ciphertext
Chosen-ciphertext	Can pick ciphertext and see its decryption

Adversaries

Passive

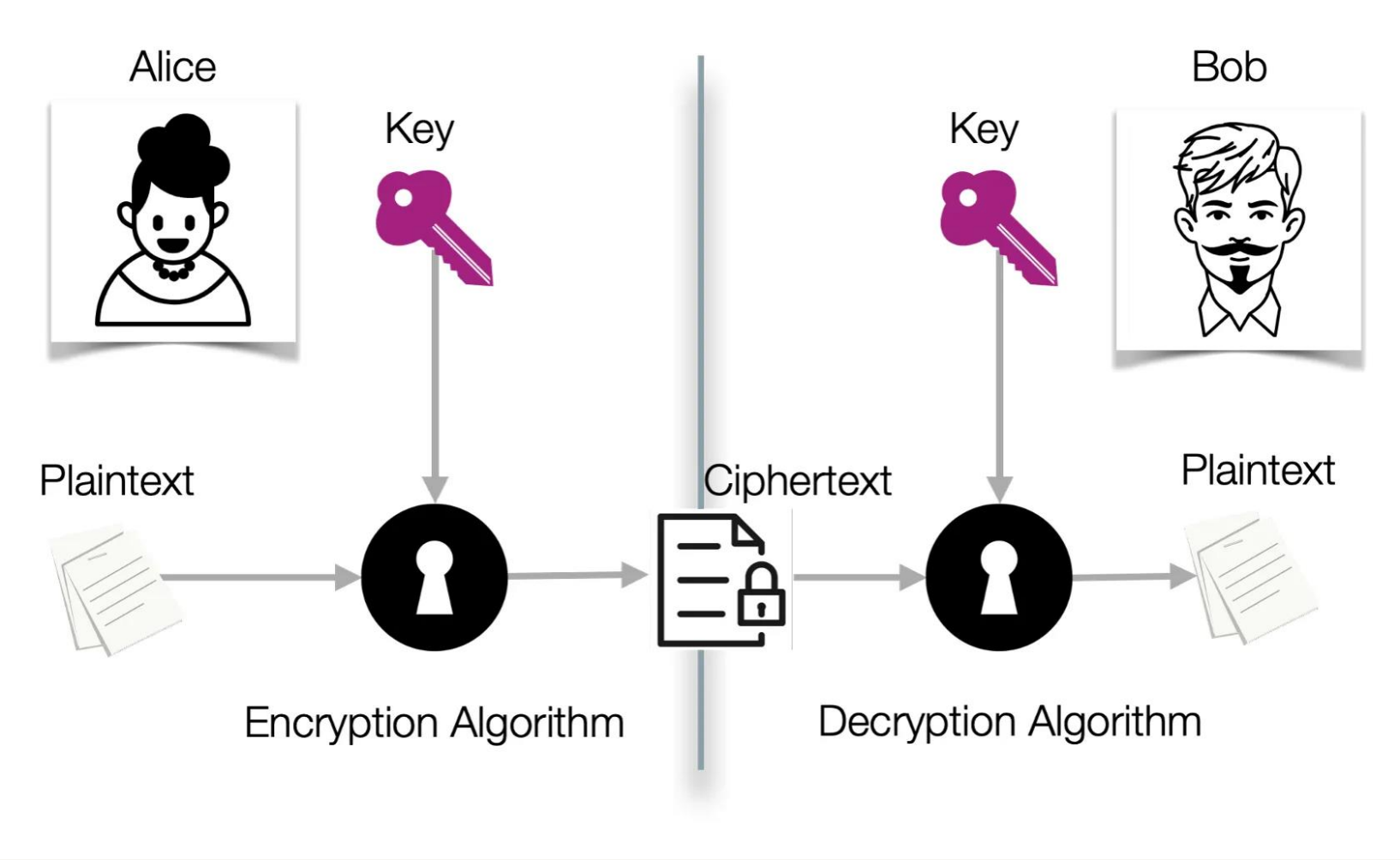
Can only observe messages

Active

Can inject or alter messages

Symmetric encryption

Symmetric encryption uses one shared key to encrypt and decrypt



Same key both ways

Alice encrypts with k; Bob decrypts with the same k.

The catch

Both sides need k before they start. Public-key encryption covers how they get it.

Caesar cipher shifts every letter by the same amount

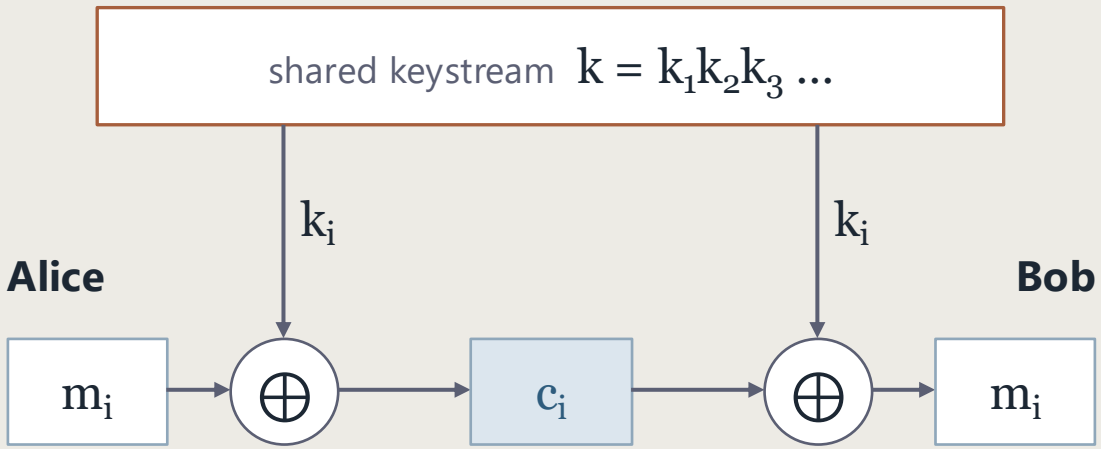
Plain	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cipher	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

Plaintext `THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG`

Ciphertext `QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD`

The key is the shift. With only 25 useful shifts, trying every key takes seconds.

A one-time pad is unbreakable but impractical, and has no integrity



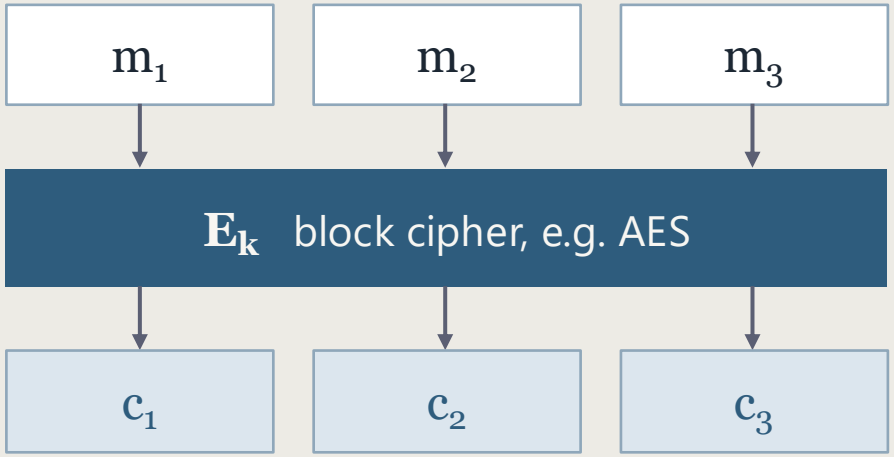
One bit at a time: Alice and Bob repeat this for i = 1, 2, 3, …, using a different key bit k_i for each message bit.

- **One-time pad:** a truly random key stream, used once and never reused.
- **Information-theoretically secure:** even unlimited computing power and time can't recover the plaintext without the key.
- **No integrity:** an attacker who flips a ciphertext bit flips the same plaintext bit, undetected.
- **In practice:** we aim for computational security; an attacker with fixed resources can't search a large key space.

Redrawn after van Oorschot, Computer Security and the Internet, Fig. 2.2

Stream ciphers encrypt bit by bit; block ciphers encrypt fixed-size chunks

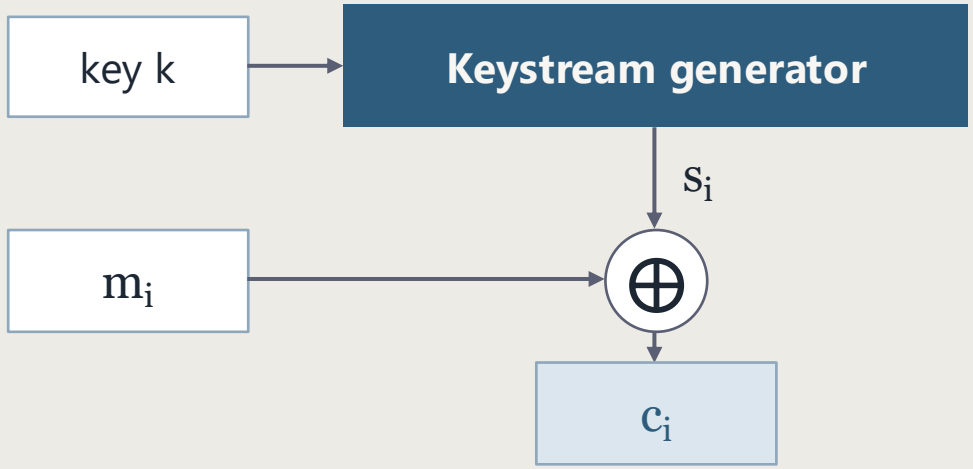
Block cipher



Encrypts fixed-size chunks, e.g. 128 bits.

A short last block is padded, e.g. with a 1, then zeros.

Stream cipher

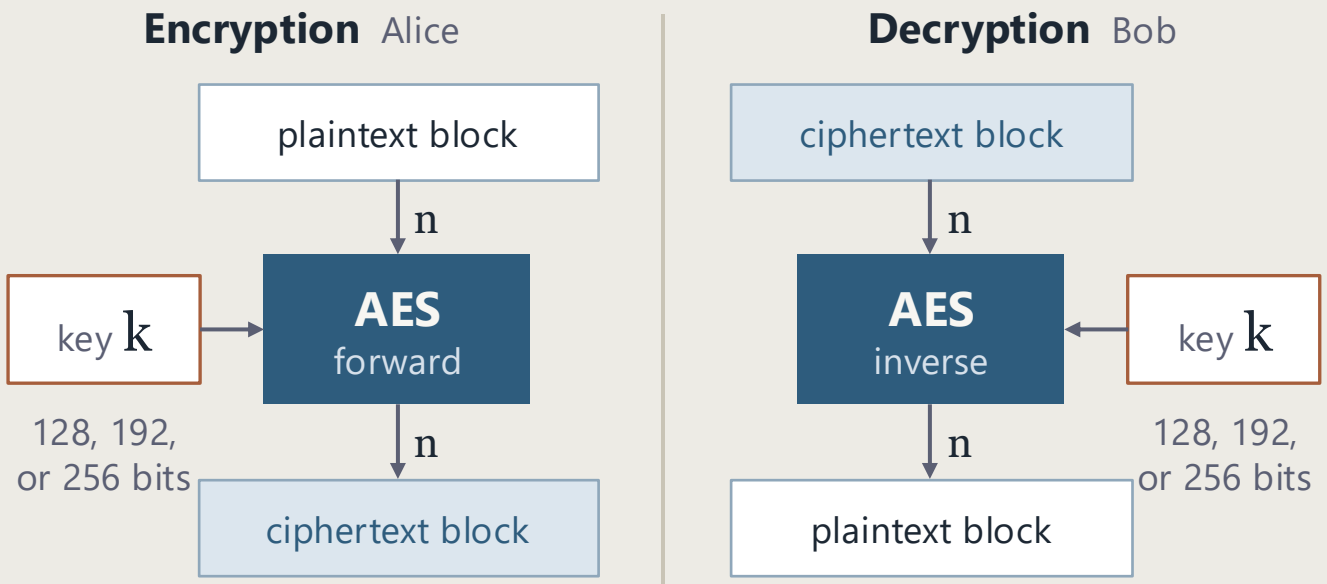


Encrypts one bit or one character at a time.

Each plaintext bit is XORed with a keystream bit.

Redrawn after van Oorschot, Computer Security and the Internet, Fig. 2.6

AES is the standard block cipher: 128-bit blocks, keys up to 256 bits



Block length n = 128 bits. For a fixed key, AES maps each 128-bit input block to a unique output block; the inverse maps it back.

128 bits
block size, fixed

128 · 192 · 256
key length in bits

Most common
block cipher in use today

Redrawn after van Oorschot, Computer Security and the Internet, Fig. 2.3

Block cipher modes let AES encrypt messages longer than 128 bits

AES encrypts exactly 128 bits. A mode says how to apply it to longer data. Wherever a subsequent diagram says “block cipher encryption,” that is where AES goes.

ECB

Electronic Code Book

Encrypts each block on its own.

Not recommended

CBC

Cipher Block Chaining

Chains each block to the previous ciphertext.

Needs a random IV and padding

CTR

Counter

Encrypts a counter and XORs it with the plaintext.

Never reuse a nonce

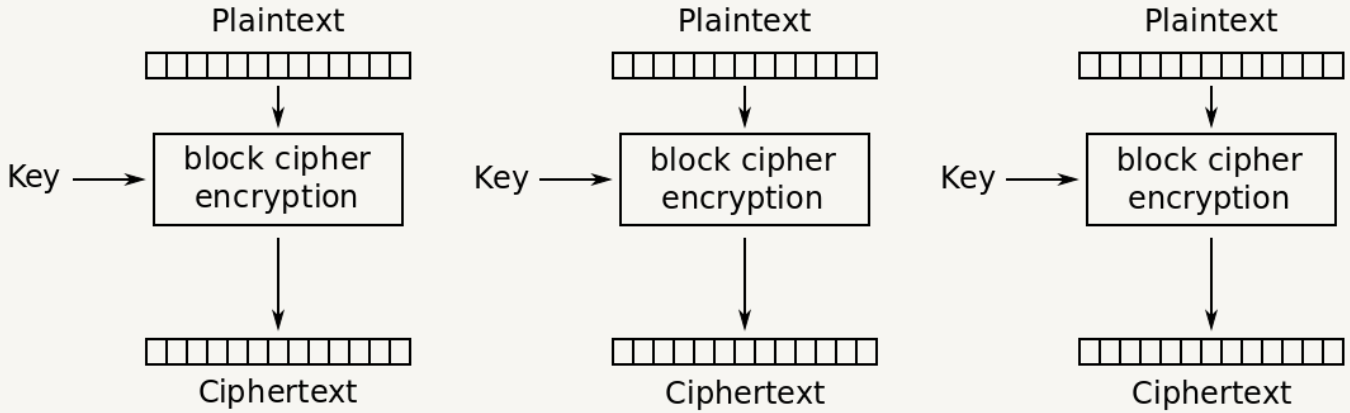
GCM

Galois/Counter Mode

CTR mode plus authentication.

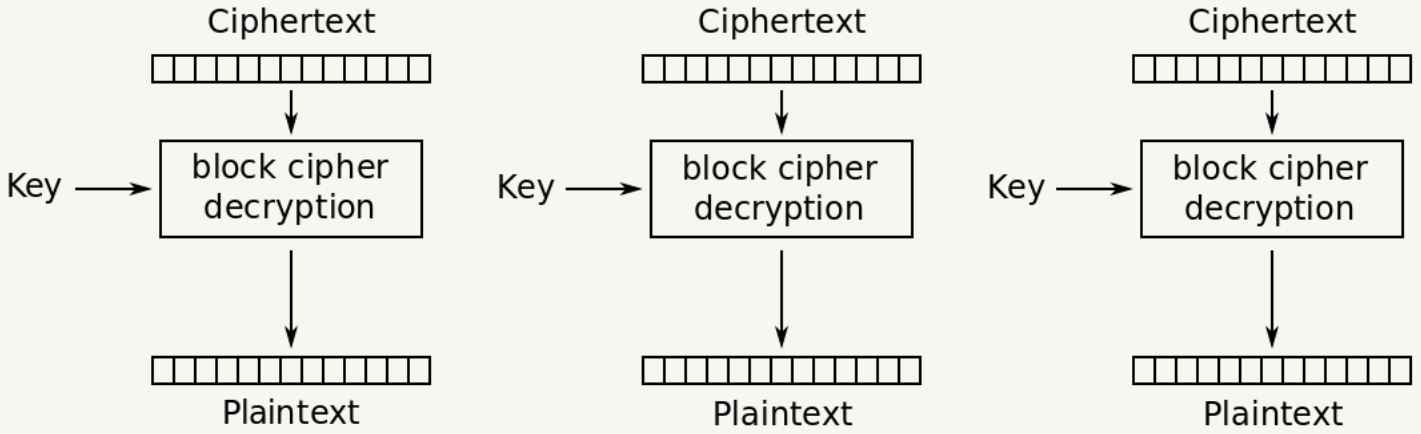
Use this one

ECB encrypts every block independently



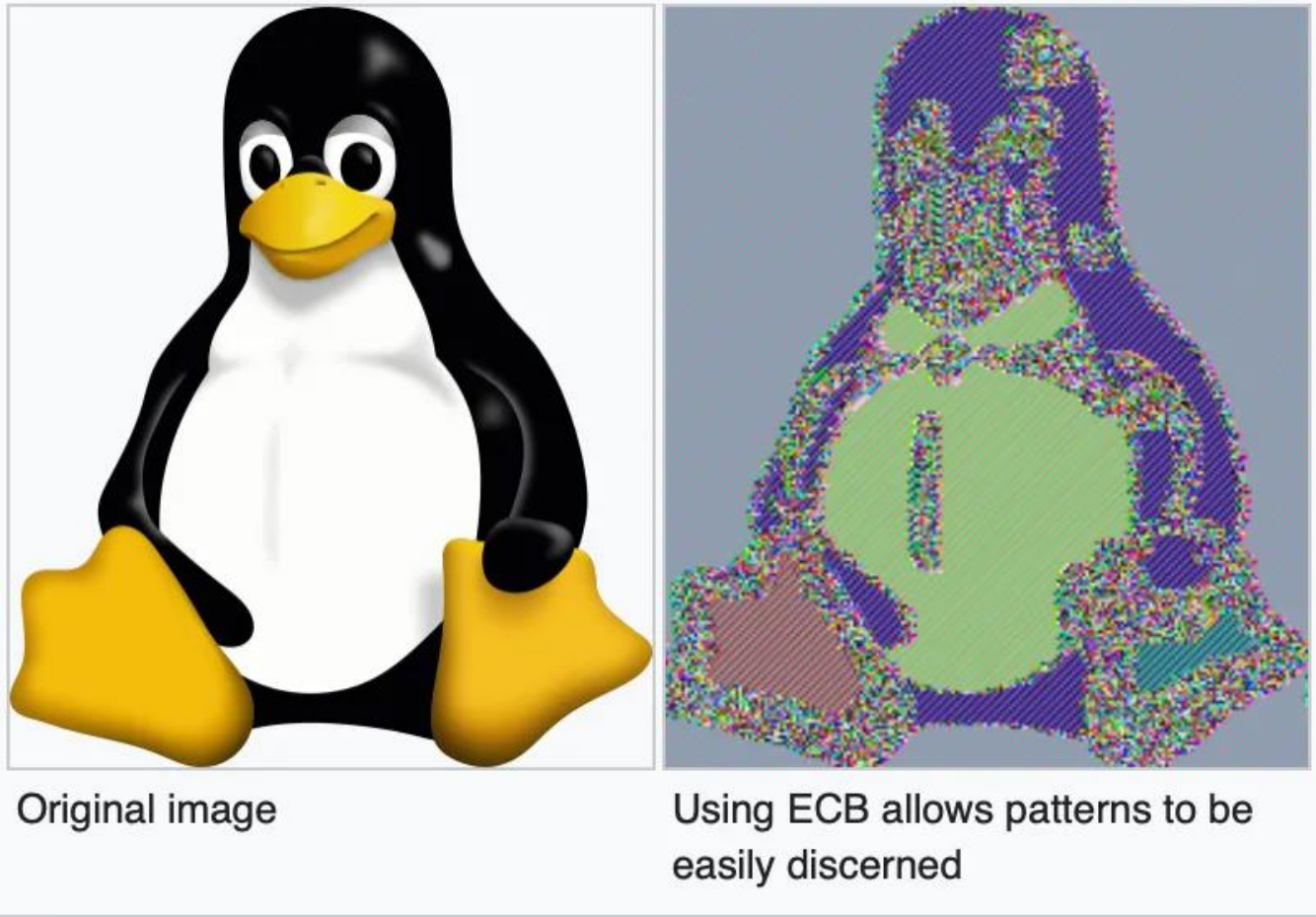
Electronic Codebook (ECB) mode encryption

Same key, same block in, same block out. ECB is not recommended because of this weakness.



Electronic Codebook (ECB) mode decryption

Identical plaintext blocks give identical ciphertext, so ECB leaks patterns



Original image

Using ECB allows patterns to be easily discerned

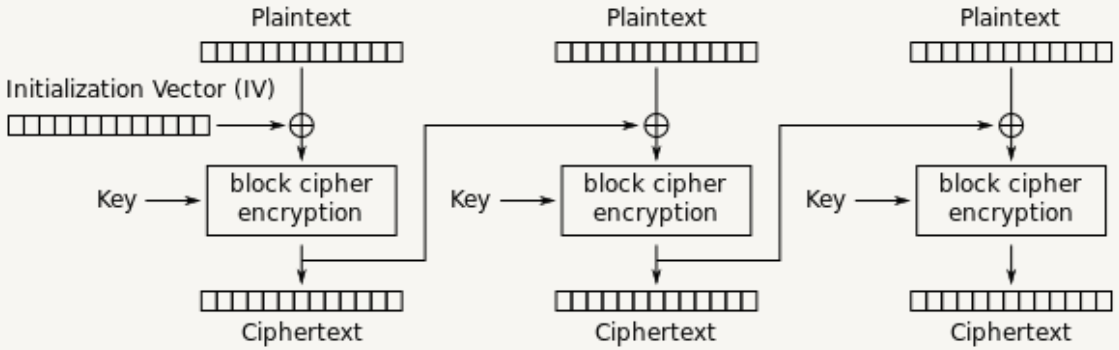
What you see

Large areas of one color encrypt to the same ciphertext blocks, so the outline survives encryption.

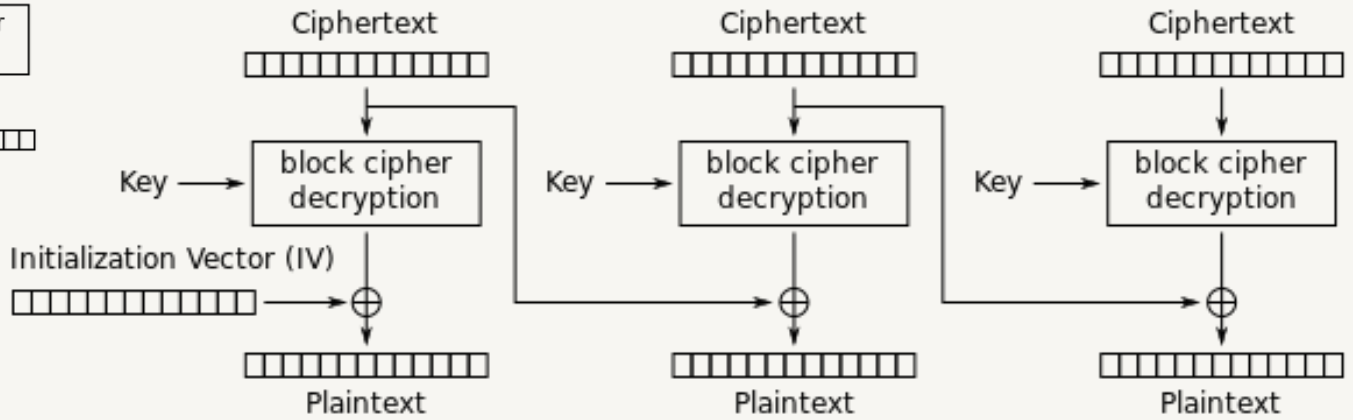
The fix

Use a mode that makes each block depend on more than its own plaintext: CBC, CTR, or GCM.

CBC chains each block to the one before it, starting from a random IV



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

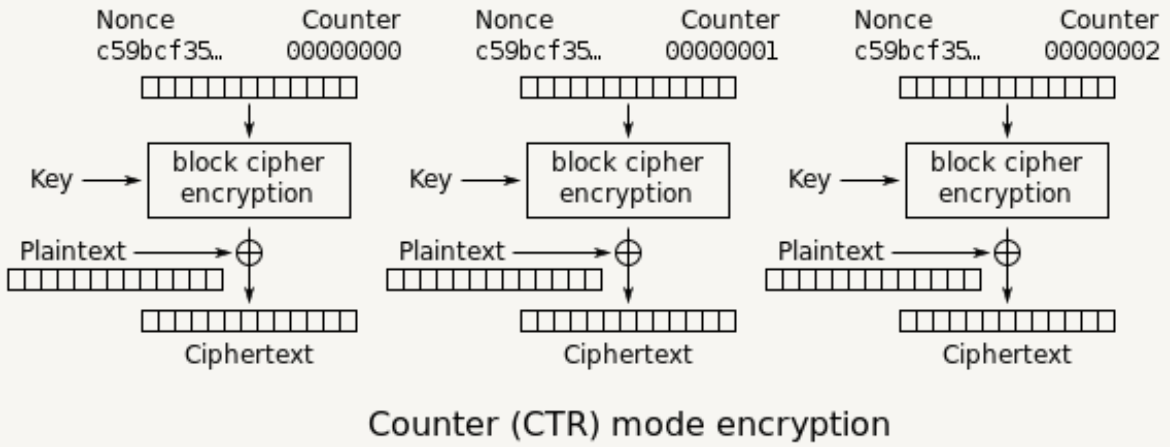
Chained
Each plaintext block is XORed with the previous ciphertext block.

Random IV
The initialization vector makes each message unique.

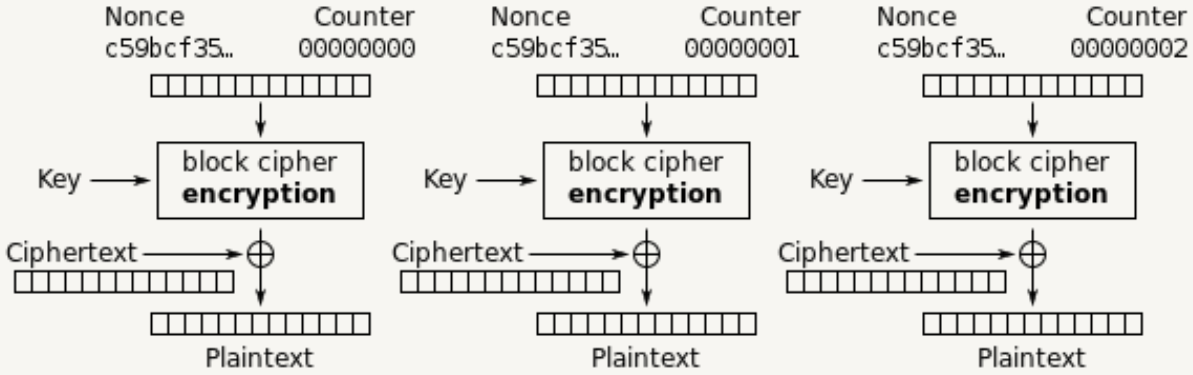
Sequential
Encryption must go block by block, in order.

Padded
The message must fill a whole number of blocks.

CTR mode encrypts a counter and XORs it with the plaintext



Counter (CTR) mode encryption



Counter (CTR) mode decryption

Nonce + counter
The nonce acts as the IV; the counter never repeats.

Never reuse a nonce
with the same AES key.

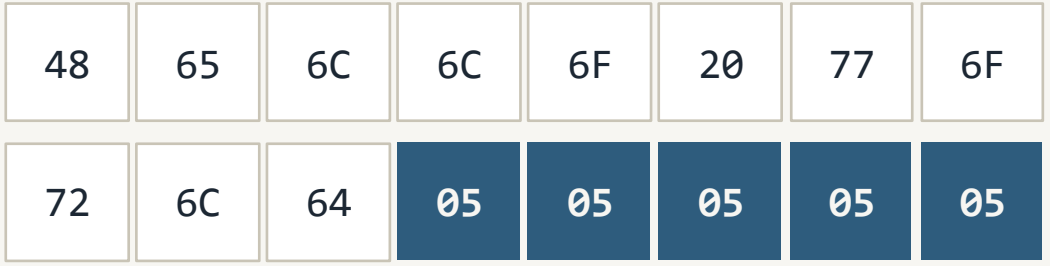
Stream cipher
Each plaintext bit is XORed with the encrypted counter.

Parallel
Blocks can be encrypted and decrypted at the same time.

Padding fills out the last block for ECB and CBC

- ECB and CBC need the plaintext to be a multiple of the block size
- Padding fills the last block so every block is full
- Both sides must agree on the padding scheme
- PKCS #7 is a common scheme
- Other schemes exist, e.g. a 1 bit, then zeros

PKCS #7 example: “Hello world” in one 16-byte block



Add N bytes, each with the value N.

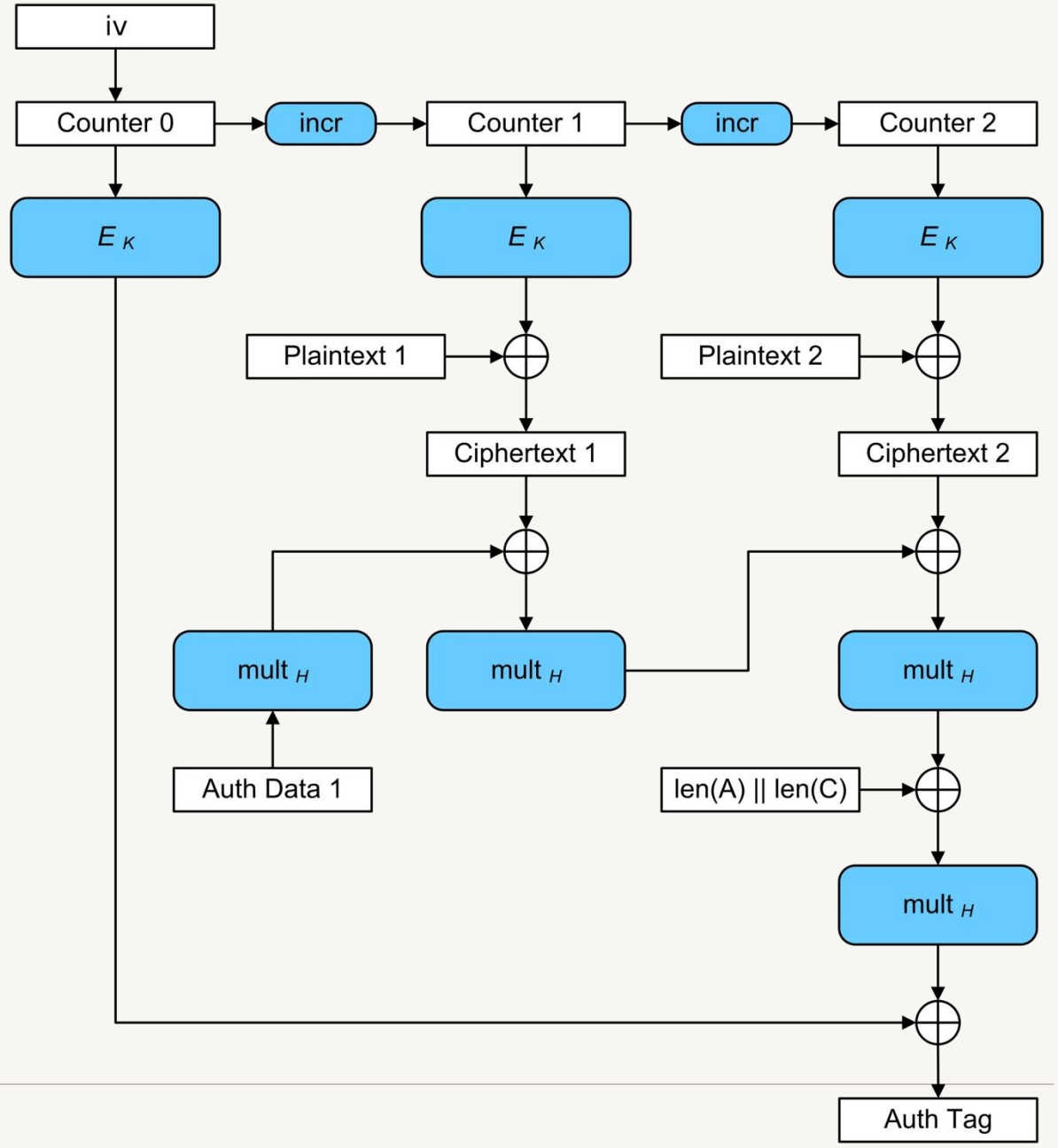
Here 5 bytes are missing, so the pad is five 0x05 bytes. The receiver reads the last byte and strips that many.

A full last block still gets padded: a whole extra block of sixteen 0x10 bytes, so the receiver can always strip the pad.

AES-GCM adds integrity and authenticates unencrypted header data

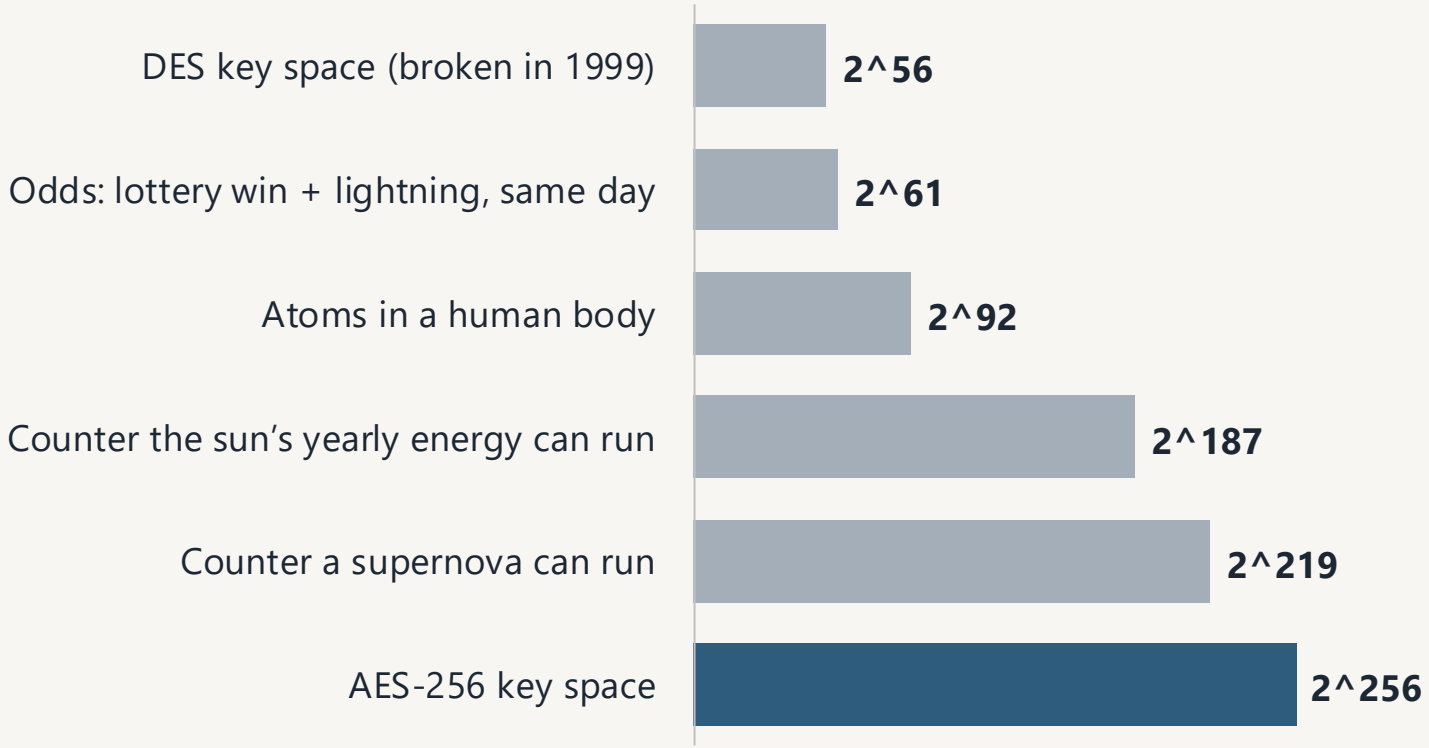
AEAD
Authenticated Encryption with Associated Data: confidentiality, integrity, and authentication, with integrity covering the unencrypted header too.

- **CTR mode plus auth data,** mixed in with XOR and multH (Galois field multiplication)
- **Fast, patent-free, online,** and parallelizable
- **Needs a unique IV** for every encryption with the same key
- **Used in** TLS, SSH, IPsec, and OpenVPN



Physics puts a 256-bit key search out of reach

Size of each quantity, as a power of 2



Thermodynamic limit

Setting or clearing one bit costs at least kT. At 3.2 K, kT = 4.4×10^{-16} erg.

The sun's yearly output, 1.21×10^{41} erg, runs a 187-bit counter; 32 years reaches 192 bits.

A supernova, about 10^{51} erg, reaches 219 bits.

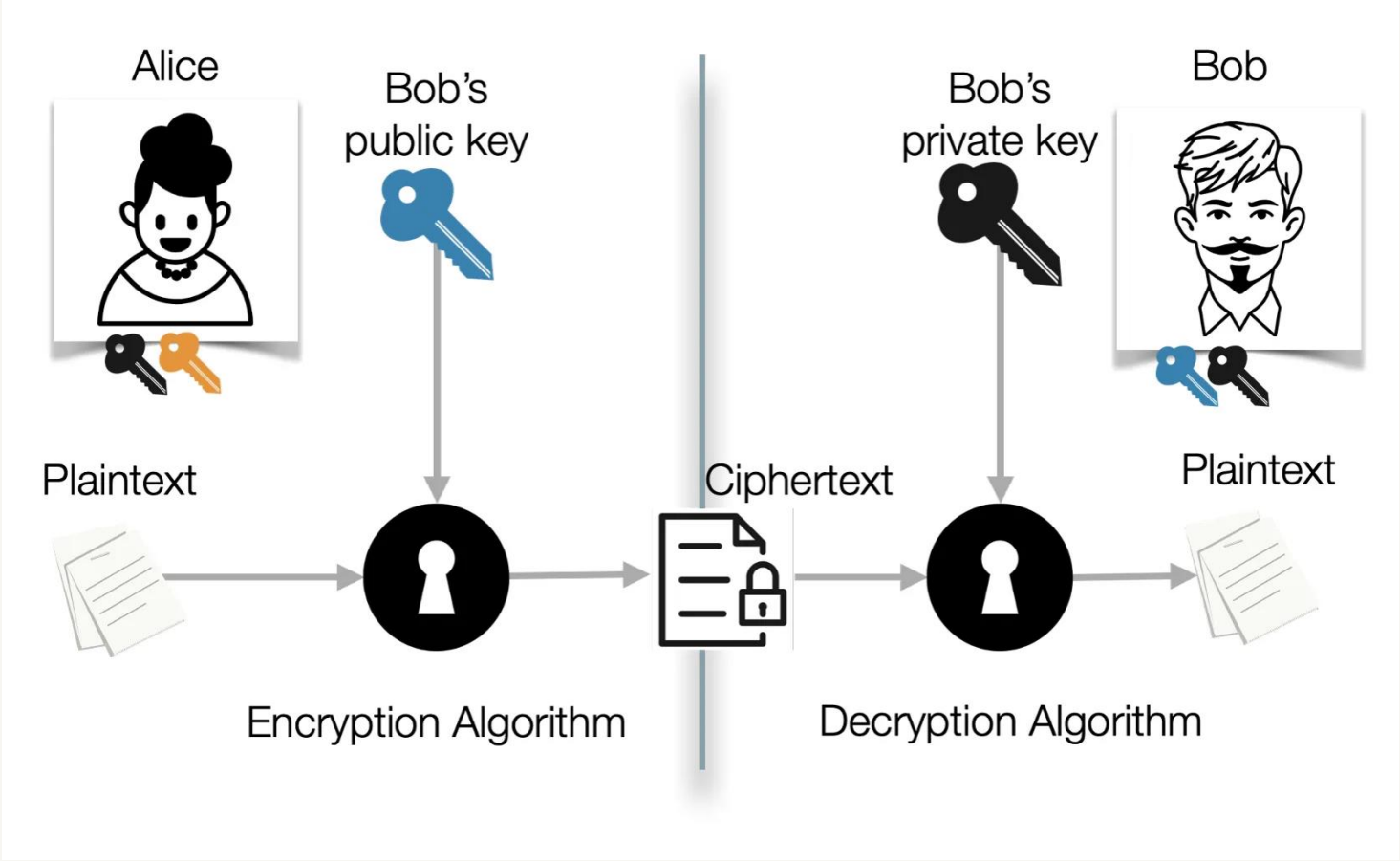


Class exercise: AES-GCM in Rust

- **Crates:** Rust Crypto aes-gcm
- **Code example:** AES-GCM in the class Rust Cryptography repo

Public-key cryptography

Anyone can encrypt to Bob with his public key; only Bob can decrypt



$$c = E_{eB}(m)$$

$$m = D_{dB}(c)$$

e_B Bob's public key

The encryption key. Bob publishes it; anyone can use it to encrypt to him.

d_B Bob's private key

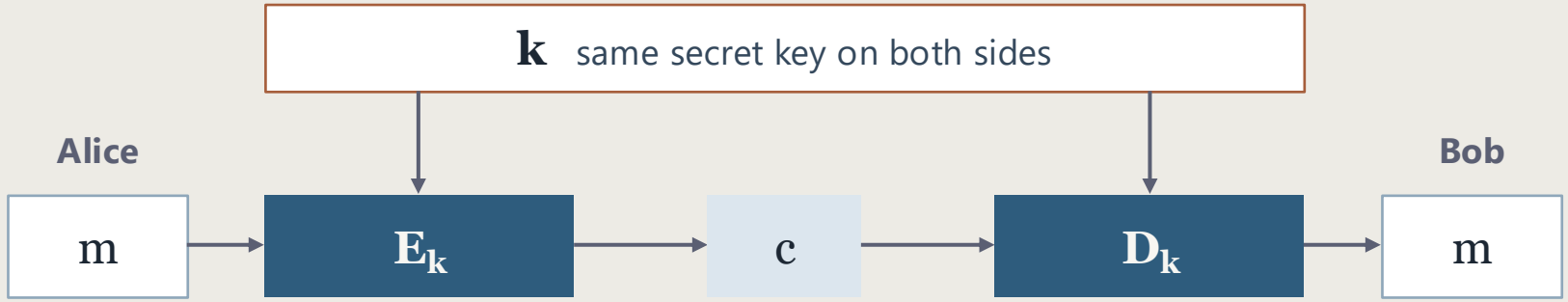
The decryption key. Bob must keep it secret, or anyone could decrypt messages sent to him.

The subscript names the owner: Bob's key pair is (e_B, d_B).

Symmetric crypto shares one key; public-key crypto uses a key pair

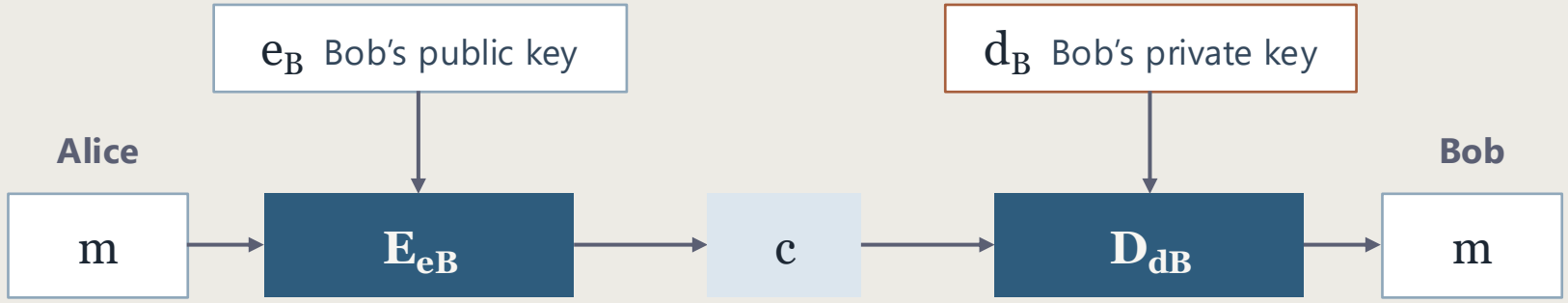
Symmetric key

Alice and Bob share one secret key.



Public key

Bob publishes one key and keeps the other private.



Redrawn after van Oorschot, Computer Security and the Internet, Fig. 2.7

RSA is the standard public-key algorithm, and it needs padding

- **RSA** provides computational security: breaking it is infeasible, not impossible.
- **Raw RSA is deterministic:** the same message always encrypts the same way.
- **Padding adds randomness.** A common standard is PKCS #1 v2.

Extra reading: the Marvin attack

Stop using RSA PKCS #1 v1.5 encryption.

In TLS, disable RSA key-exchange cipher suites and use ECDHE (TLS 1.3 already does).

RSA-OAEP (PKCS #1 v2) and RSA signatures remain acceptable.

Optional: the math behind RSA

Key setup

1. Pick secret large primes p, q
2. $n = pq$, $\varphi(n) = (p-1)(q-1)$
3. Pick e with $\gcd(e, \varphi(n)) = 1$
4. Compute d with $ed \equiv 1 \pmod{\varphi(n)}$
5. Public key (e, n); private key (d, n)

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

RSA is broken if n can be factored.

Toy example: p = 3, q = 11

$n = 33$, $\varphi(n) = 2 \cdot 10 = 20$

$e = 3$: $\gcd(3, 20) = 1$

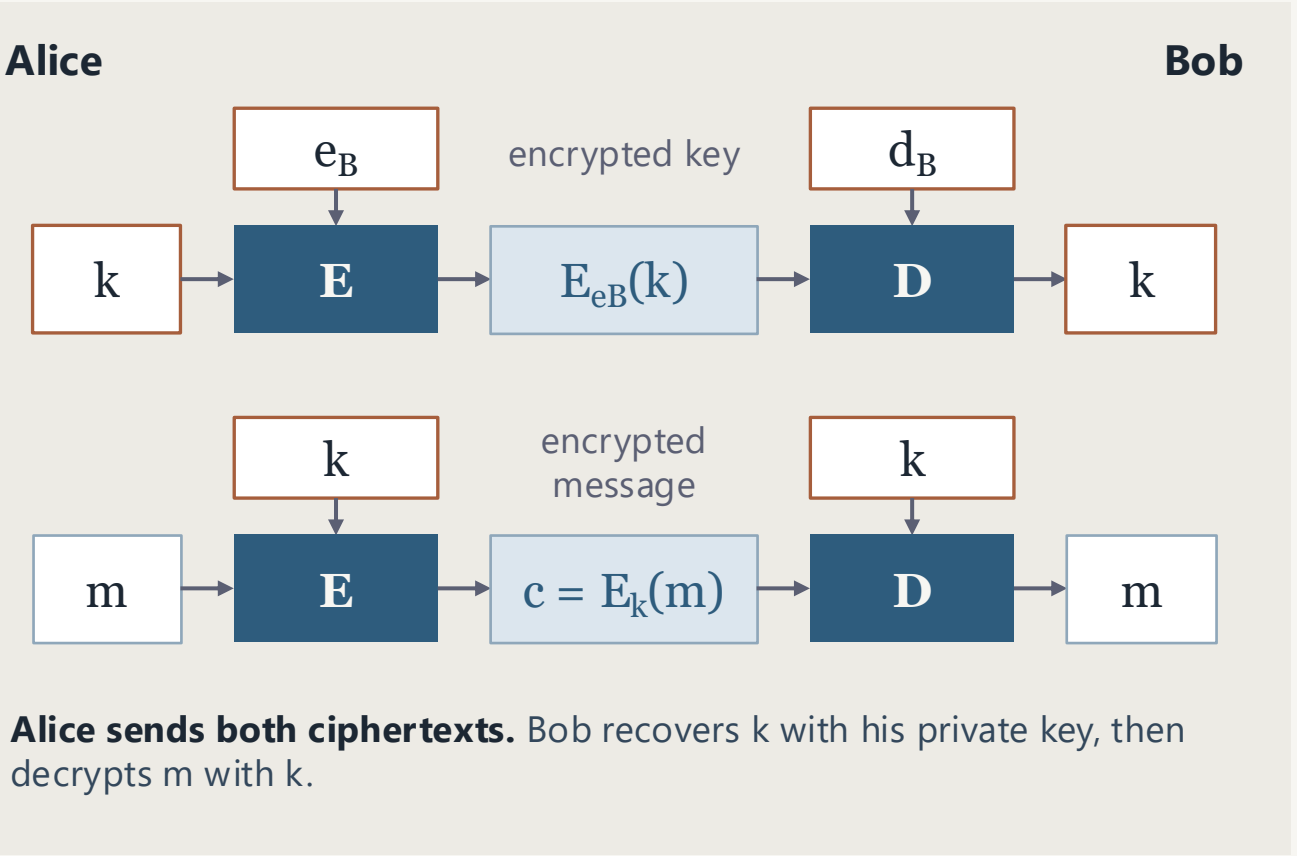
$d = 7$: $3 \cdot 7 = 21 \equiv 1 \pmod{20}$

Encrypt $m = 4$: $4^3 = 64 \bmod 33 \rightarrow c = 31$

Decrypt: $31^7 \bmod 33 = 4$ ✓

Public key (3, 33) and private key (7, 33) differ. Real keys use primes of about 1,000 bits each.

Hybrid encryption uses public keys to share a fast symmetric session key

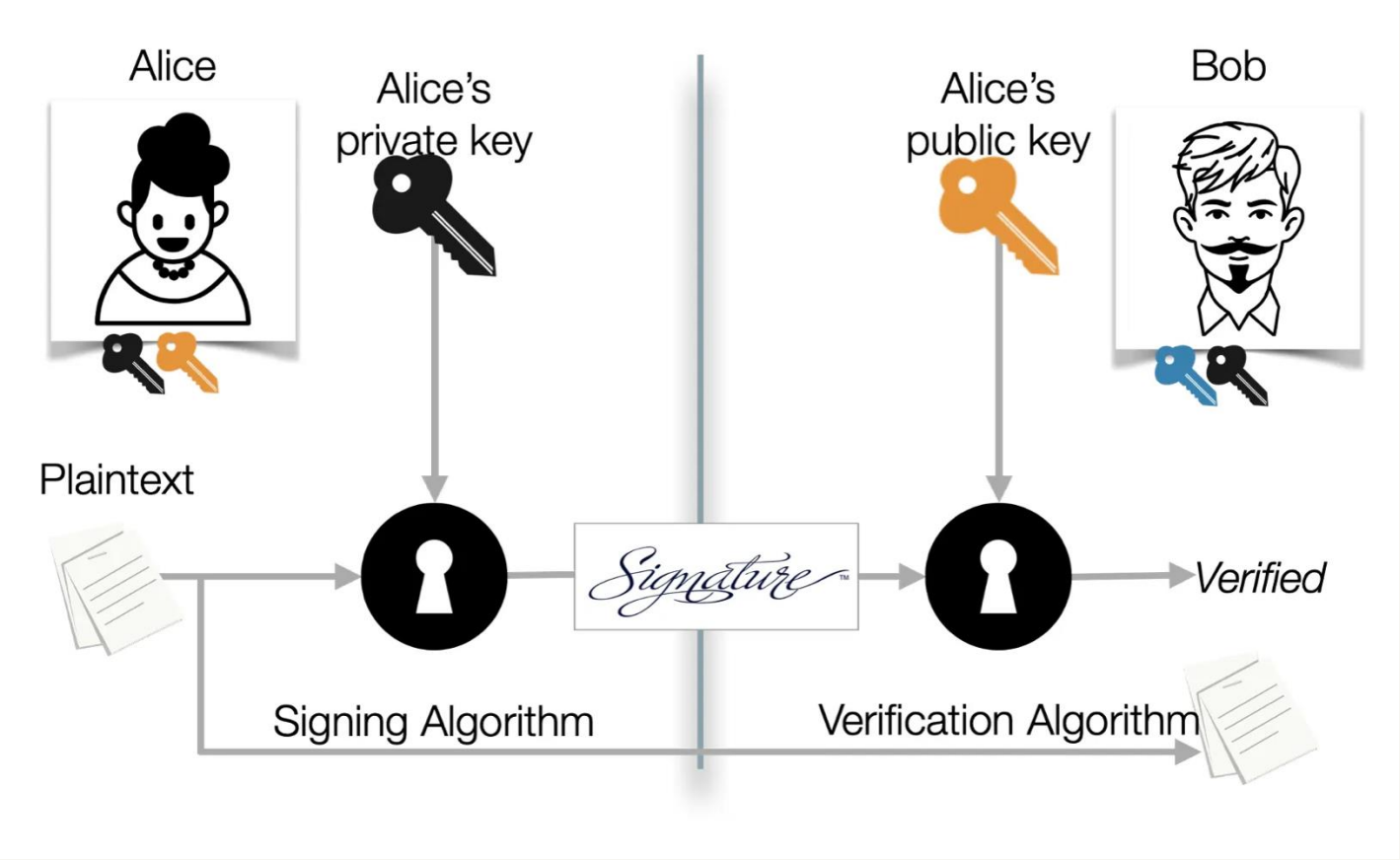


Alice sends both ciphertexts. Bob recovers k with his private key, then decrypts m with k.

1. Alice picks a fresh symmetric key k: the session key.
2. She encrypts k with Bob's public key and sends it.
3. She encrypts the message with k, which is much faster than public-key encryption.
4. Bob decrypts k with his private key, then the message with k.

Redrawn after van Oorschot, Computer Security and the Internet, Fig. 2.8

Alice signs with her private key; anyone can verify with her public key



$t = S_{sA}(m)$

$V_{vA}(m, t) \rightarrow \text{valid?}$

(v, s) = (public key, private key)

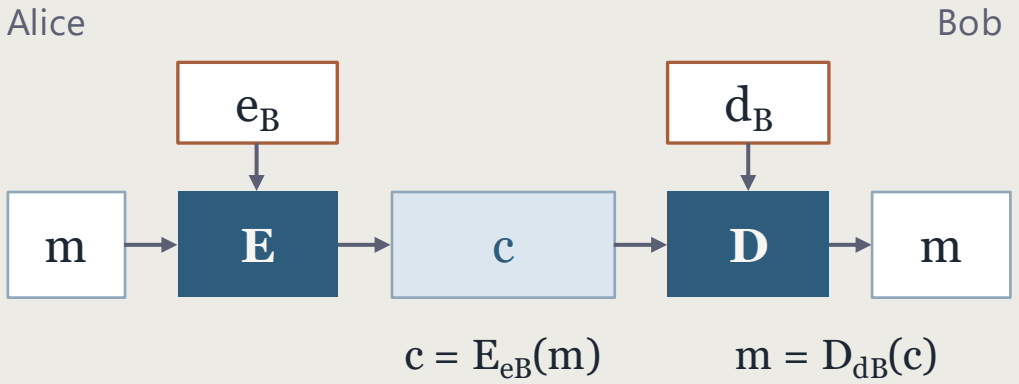
With RSA, signing resembles encrypting with the private key, and verifying resembles decrypting with the public key.

Keep separate key pairs

Use one key pair for encryption and decryption and a different one for signing and verifying.

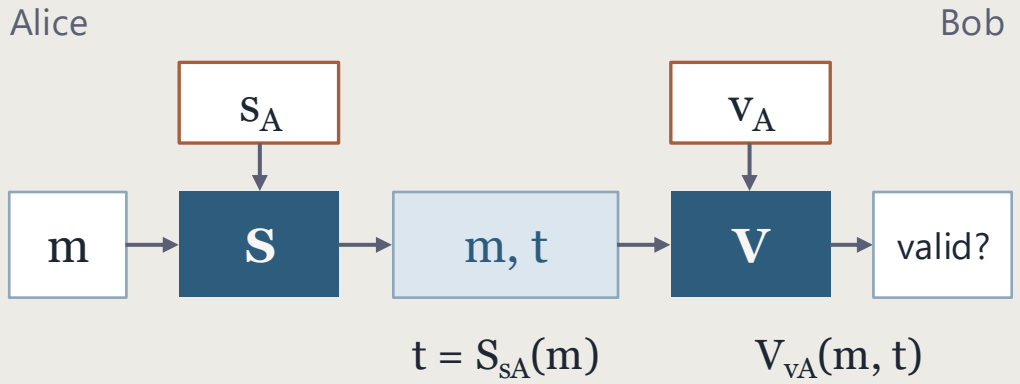
Encrypting uses the recipient's keys; signing uses the sender's

Public-key encryption



Bob's keys: public e_B encrypts; private d_B decrypts

Public-key signature



Alice's keys: private s_A signs; public v_A verifies

To encrypt for Bob

Alice uses Bob's public key e_B

Only Bob's private key decrypts it.

To sign for Bob

Alice uses her own private key s_A

Anyone with Alice's public key verifies it.

Redrawn after van Oorschot, Computer Security and the Internet, Fig. 2.9

Signatures prove who signed and that nothing changed

Data origin authentication

You can verify who signed the message.

Data integrity

You can verify the message is exactly what was signed.

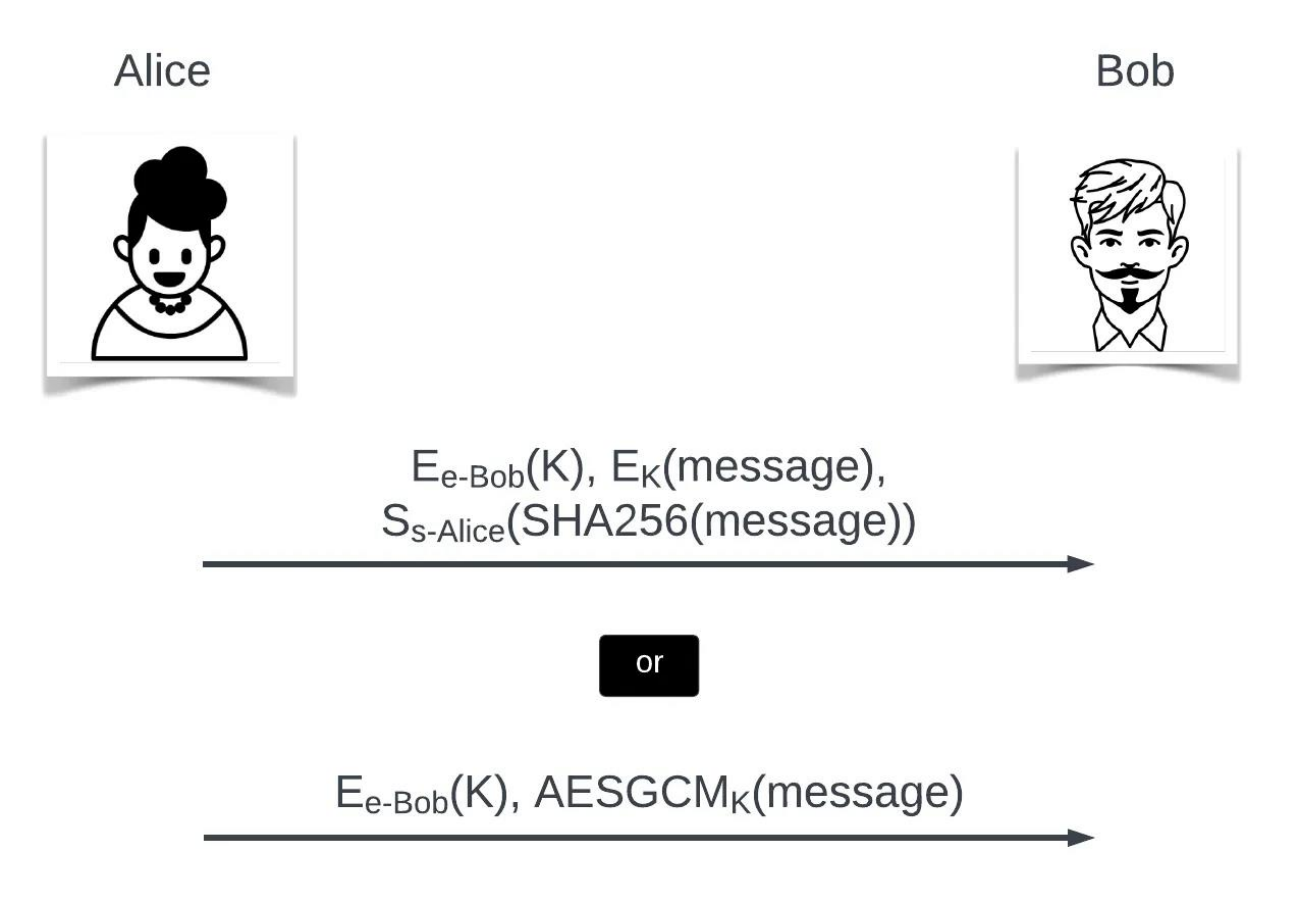
Non-repudiation

The signer can’t later deny signing, unless their private key was stolen.

The catch: key management

- **Private keys must stay private.** Non-repudiation fails if someone steals the signing key.
- **Public keys must be distributed reliably,** so others know which key is really yours.
- **A valid signature doesn’t prove freshness:** an attacker can replay an old signed message.

Signing plus hybrid encryption gives confidentiality and integrity



- **A signature on the message hash** tells Bob it came from Alice and wasn't altered.
- **AES-GCM adds integrity,** and if only Alice and Bob know k, Bob knows Alice sent it.

> **Getting every piece right is tricky.**
> See *Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML*. Use TLS where possible.

Class exercise: RSA in Rust

- **Crates:** Rust Crypto rsa

- **Code examples:** RSA in the Rust Cryptography repo

Takeaways

We use symmetric cryptography to encrypt data. We use public keys to exchange session keys and sign. Real systems combine the two with vetted designs.

1. **Don’t design your own crypto.** Assume attackers know your system; only the key is secret.
2. **Use AES-GCM for data.** It gives confidentiality and integrity. Never reuse an IV with the same key.
3. **Use public keys to deliver session keys and to sign.** Keep private keys private, and use separate pairs for each job.
4. **Combine them through vetted protocols like TLS.** Hand-assembled sign-and-encrypt schemes have a history of flaws.