COMP 435 Test 3 Study Guide

Computer Security Concepts · UNC Fall 2026 · The cryptographic toolbox

0. What's on the test

SCOPE. L21 through L26 — everything from "what is a cipher" to "how do we get integrity and freshness."

FORMAT. Short answer on paper. Expect small modular-arithmetic computations, figures to complete, attack models to match, and several "why is this insecure" explanations.

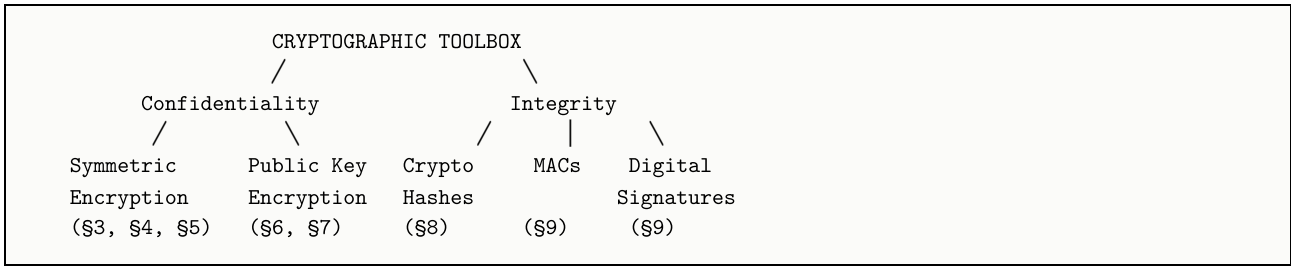
SHAPE. Half definitions and half reasoning. Almost every topic has a paired question: what it gives you, and what it does *not*.

0.1 The published topic list

Topic	Section
Cryptanalysis and attack models (ciphertext-only, known-plaintext, CPA, CCA)	§2
One-time pads: security requirements and key reuse	§3
Substitution, transposition, confusion, and diffusion	§4
Stream ciphers and block cipher modes: ECB, CBC, OFB, CTR	§5
RSA encryption and decryption: key roles and basic modular arithmetic	§6
Diffie-Hellman key exchange, machine-in-the-middle attacks	§7
Cryptographic hashes: collision, pre-image, and second pre-image resistance	§8
MACs: integrity, authentication, and limitations	§9
Encrypt-then-sign: operation order, which keys to use, why other orderings may be insecure	§9.4
Replay attacks and freshness: concepts, definitions, how a replay is carried out	§7.5

0.2 The roadmap

Everything in the course's crypto unit hangs off one diagram. Knowing which branch a tool lives on answers a surprising number of questions:



> **Watch out.** Encryption gives you **confidentiality and nothing else**. The one-time pad is perfectly secret and offers *no* integrity at all. Any answer that says encryption stops an attacker modifying a message is wrong, and that mistake is worth several marks across this test.

0.3 Where each topic came from

Lecture	Title	Core ideas
L21	Intro to Crypto	terminology; Kerckhoffs's principle; exhaustive search vs. cryptanalysis; computational security; the four attack models
L22	Cryptanalysis cont. & One-Time Pad	classical ciphers and frequency analysis; OTP; information-theoretic security
L23	Block Ciphers	substitution and transposition; confusion and diffusion; stream vs. block; AES; ECB, CBC, OFB, CTR
L24	Public Key Encryption	RSA key generation, encryption, decryption; one-way and trapdoor functions; hybrid encryption; nonces
L25	Diffie-Hellman & Hashing	DH key agreement; machine-in-the-middle; cryptographic hash properties; the birthday paradox
L26	MACs & Signatures	MACs vs. bare hashes; RSA signatures; encrypt-then-sign

0.4 How to use this guide

Each section ends with a **Check yourself** set. §10 is a mock test — do it closed-book with a timer before reading §11, which works through every in-class practice problem. §12 is a one-screen cram sheet.

EXAM TIP. Four answer shapes cover most of this test:

- **Compute a small modular power.** Reduce early and use negatives. `33^5 mod 35` is unpleasant; `(-2)^5 mod 35` is not.
- **Match an attack model.** Ask what the attacker *obtains* and whether they *choose* it. Observe only → ciphertext-only. Gets pairs they didn't pick → known-plaintext. Picks plaintexts → CPA. Picks ciphertexts → CCA.
- **Name what a tool does *not* provide.** Encryption → no integrity. A bare hash → no authentication. A MAC → no non-repudiation and no freshness. DH → no authentication.
- **Say which key.** Encrypt for someone → *their* public key. Sign → *your* private key. Verify → *their* public key. Decrypt → *your* private key.

L21 1. Foundations

1.1 Terminology

Encrypt.
Encode a message to obscure its meaning (also: encipher)

Decrypt.
Decode an encrypted message to reveal its original meaning (also: decipher)

Plaintext.
The original message

Ciphertext.
The encrypted message

Cipher.
An algorithm for encrypting or decrypting

```
plaintext ──► Encrypt ──► ciphertext ──► Decrypt ──► plaintext

              c   = Enc_k(msg)
              msg = Dec_k(c)        and so  msg = Dec(Enc(msg))
```

Note which key appears in both lines. In **symmetric** encryption it is the same `k` on both sides; in **public key** encryption (§6) the two differ. What encryption provides is **confidentiality** — and only confidentiality.

CRYPTOSYSTEM. A system for encryption and decryption. It has four parts: an **encryption algorithm**, a **decryption algorithm**, **key generation**, and **key management**.

Two of those four are algorithms and two are about keys, which is a fair reflection of where real systems fail.

1.2 Kerckhoffs's principle

KERCKHOFFS'S PRINCIPLE. The security of a cryptosystem should depend only on the secrecy of the keys.

So publishing your algorithm does not make a system insecure — it is the expected practice. The reasoning is the same as "open design" from the secure design principles: an algorithm cannot be kept secret for long once it is deployed in software anyone can inspect, and a design that has survived public scrutiny is more trustworthy than one that has merely avoided it. Keys, unlike algorithms, can be generated afresh per user and replaced when compromised.

EXAM TIP. "A company publishes its encryption algorithm but keeps each user's key secret — is this insecure?" **No**, and the principle to name is **Kerckhoffs's principle**. Add one clause on why: only the key needs to be secret, and keys can be changed per user while an algorithm cannot.

1.3 Two ways to break encryption

	Exhaustive search	Cryptanalysis
Definition	Try every possible key until the correct one is found	Recover key material or plaintext by exploiting flaws in the cryptosystem
What it attacks	the size of the key space	the structure of the algorithm
Defeated by	a large enough key space	a better-designed cipher

A shift cipher has only **25** keys to try, so exhaustive search breaks it immediately. Mono-alphabetic substitution has $26! \approx 2^{88}$ keys — far too many to search — and yet it falls in minutes to *cryptanalysis*, because letter frequencies survive the substitution. A huge key space is necessary and nowhere near sufficient.

WATCH OUT. If you deduce the key from a plaintext–ciphertext pair, that is **cryptanalysis**, not exhaustive search — you exploited structure (the cipher's fixed, uniform shift) rather than trying candidates. The dividing line is whether you tried keys or reasoned about the algorithm.

1.4 Exhaustive search and key length

The worked case is DES, with a **56-bit key**, against a machine doing 10^{10} to 10^{15} encryptions per second:

```
2^56 = 72,057,594,037,927,936 ≈ 7.2 × 10^16 keys

@ 10^10 /sec :  7.2 × 10^6 s  ≈ 83 days      (course figure: ~75 days)
@ 10^15 /sec :  72 s          ≈ 1 minute
```

ALWAYS HALVE IT. On average the attacker succeeds after searching only $\mathbf{2^{55}}$ keys — half the key space. Whenever you are asked for an expected time, divide by two.

56 bits is much too short for today's computers, which is the whole reason DES was retired.

Computational security.
Secure against an attacker with fixed computational resources — e.g. a 128-bit key is computationally secure against exhaustive search

Sufficient key space principle.
Any secure encryption scheme must have a key space that is not vulnerable to exhaustive search

Computational security is a statement about economics, not impossibility: the cost to break the encryption must greatly exceed the value of the asset, and the time to break it must greatly exceed the lifetime of the asset. Contrast this with the *information-theoretic* security of the one-time pad in §3.5, which holds against unlimited computation.

1.5 Check yourself

1. A shift cipher uses `Enc_k(m) = (m + k) mod 26` with a = 0 … z = 25. You learn that `cat` encrypts to `fdw`. What is the key, and what does `grj` decrypt to? [L21]

> Each letter shifts by the same amount: c(2)→f(5), a(0)→d(3), t(19)→w(22), so **k = 3**. Decrypting `grj`: g(6)−3 = 3 = d, r(17)−3 = 14 = o, j(9)−3 = 6 = g → **dog**.

2. Was that exhaustive search or cryptanalysis? [L21]

> **Cryptanalysis.** No keys were tried. The known plaintext–ciphertext pair plus the structure of the cipher — a single shift applied uniformly to every letter — yields the key directly. Exhaustive search would have meant testing all 25 shifts until one produced readable text.

3. Mono-alphabetic substitution has about 2^{88} keys, far more than DES's 2^{56}. Is it more secure? [L21]

> No. The key space defeats exhaustive search but says nothing about cryptanalysis, and mono-alphabetic substitution preserves letter frequencies exactly — each plaintext letter maps to one ciphertext letter, so `e`'s frequency simply moves to whatever letter represents it. Frequency analysis breaks it with a single ciphertext. The sufficient key space principle is a *necessary* condition, not a sufficient one.

4. A system is "computationally secure." Does that mean it can never be broken? [L21]

> No — it means breaking it costs more than the asset is worth, or takes longer than the asset stays valuable, given an attacker with fixed resources. It is a claim about current economics and technology, and it expires as hardware improves. DES was computationally secure in 1977 and is not now.

L21 · L22 2. Attack models

Four models, in increasing order of attacker power. The first two are **passive** (the attacker only watches); the last two are **active** (the attacker interacts with a service).

2.1 The four models

Model	What the attacker gets	Passive / active
Ciphertext-only	Observes ciphertexts. Nothing else.	passive
Known-plaintext	Learns plaintext–ciphertext pairs *they did not choose*	passive
Chosen-plaintext (CPA)	Obtains the ciphertext for *any plaintext they pick*	active
Chosen-ciphertext (CCA)	Obtains the decryption of *any ciphertext they pick*	active

EXAM TIP. Two questions resolve every matching problem. **(1)** Does the attacker get plaintexts at all? No → ciphertext-only. **(2)** If yes, did they *choose* the input? No → known-plaintext. Yes, they chose the plaintext → CPA. Yes, they chose the ciphertext → CCA.

Each model is strictly stronger than the one above it, so a scheme secure against CCA is secure against all of them. Worked examples of each:

Model	Setup	The break
Ciphertext-only	Mono-alphabetic substitution, and the attacker knows the message is either `banana` or `orange`. Ciphertext: `xmqmqm`.	The repetition pattern x-m-q-m-q-m matches `banana` and not `orange`. No plaintext was ever needed.
Known-plaintext	Shift cipher; the attacker is handed the pair `("abc", "cfg")`.	The pair reveals the key, so every past and future message under that key is readable.
CPA	Key is a 3-letter string; the attacker can request encryptions of `a`, `b`, `c`.	Encrypting chosen messages and comparing outputs — including asking for `Enc("a")` twice — leaks the key.
CCA	8-bit key; the attacker wants to decrypt a target `ct` they are not allowed to submit.	Submit `c1 = ct ⊕ 0000_0001` instead, get back `m1 = Dec(c1)`, and recover the target plaintext from the known relationship.

2.2 Determinism and CPA security

DETERMINISTIC ENCRYPTION. The same plaintext encrypted with the same key always produces the same ciphertext.

To be CPA-secure, a scheme must be non-deterministic.

Why: if encryption is deterministic the attacker can encrypt chosen messages themselves and *compare outputs* to recognise or infer future messages. Suppose they see a ciphertext `c` and suspect the plaintext was "attack at dawn." They encrypt "attack at dawn," compare, and they are certain. The attacker never needs the key — they only need to recognise.

This is the same reasoning that makes **ECB mode insecure** (§5.1) and that breaks textbook RSA when the message space is small (§6.6). It is probably the single most reused idea on this test.

The fix is to inject fresh randomness into every encryption — an IV or a counter — so the same plaintext produces a different ciphertext each time.

2.3 Malleability and CCA security

Malleable. An attacker can modify a ciphertext in a predictable way that causes a related change in the decrypted plaintext.

Non-malleable. If the adversary modifies a ciphertext, the result is either an *illegal* ciphertext or one that encrypts a plaintext having *no relation* to the original. **To be CCA-secure, a scheme must be non-malleable.**

The concrete attack: the adversary flips the first bit of `c` to get `c'` and sends `c'` to a decryption oracle. For CCA security it must be that `c'` is either rejected outright or decrypts to something completely unrelated to the original message.

Watch out. The one-time pad is **perfectly secret and completely malleable**. Flip bit i of the ciphertext and bit i of the plaintext flips, predictably, without the attacker learning anything about the message. Confidentiality and integrity are genuinely independent properties, and this is the cleanest demonstration of it.

2.4 Check yourself

1. Match each to a model: (a) the attacker submits messages of their choice and receives their encryptions; (b) the attacker observes encrypted messages with no corresponding plaintexts and no access to an encryption or decryption service; (c) the attacker submits modified ciphertexts and receives their decryptions, but cannot request decryption of the target itself; (d) the attacker obtains existing messages and their ciphertexts without choosing them. [L21]

(a) **Chosen-plaintext (CPA)** · (b) **Ciphertext-only** · (c) **Chosen-ciphertext (CCA)** · (d) **Known-plaintext**. The restriction in (c) — not being allowed to submit the target itself — is standard, otherwise the model would be trivial.

2. A scheme is deterministic. Give a concrete attack an eavesdropper can mount with no key and no oracle. [L21]

Traffic analysis by recognition: watch the ciphertexts and note which ones repeat. Identical ciphertexts mean identical plaintexts, so the attacker learns the message structure — which orders are the same, when a known command is reissued. If they can guess a candidate plaintext and have access to the public encryption procedure, they encrypt it and compare directly, confirming the content outright.

3. Why is CCA security strictly stronger than CPA security? [L21]

A CCA attacker has a decryption oracle in addition to everything a CPA attacker can do, so any scheme broken under CPA is broken under CCA. The extra requirement is non-malleability: CPA security only requires the attacker cannot *learn* from chosen encryptions, while CCA security also requires they cannot usefully *tamper*.

4. A scheme is non-deterministic but malleable. Which security does it have? [L21]

It can be CPA-secure but not CCA-secure. Randomised encryption stops the attacker recognising repeated plaintexts, but malleability means they can still transform a ciphertext into one with a predictably related plaintext — which a decryption oracle turns into an attack. CBC mode is exactly this case.

L22 3. Classical ciphers and the one-time pad

3.1 The classical ciphers

Cipher	Rule	Key space	Broken by
Caesar	Shift the alphabet by 3	1 (fixed)	knowing it is Caesar
Shift	`Enc_k(m) = (m + k) mod 26`	25	exhaustive search, or one known pair
Mono-alphabetic substitution	Each letter maps to a fixed, arbitrary other letter	$26! \approx 2^{88}$	letter-frequency analysis
Poly-alphabetic shift (Vigenère)	`Enc_k(m_i) = (m_i + k_i) mod 26` — the shift varies by position, cycling through a keyword	26^{keylen}	finding the key length, then frequency analysis per position

```
Plaintext:   tell them about me
Key:         cafe cafe cafec af
Ciphertext:  veqp vhjq cbtyv mj
```

Vigenère repeats the keyword, so once you know its length the ciphertext splits into that many independent shift ciphers, each of which falls to frequency analysis. The one-time pad is what you get when you remove that weakness entirely by making the key as long as the message.

3.2 The one-time pad (Vernam cipher)

```
c_i = p_i ⊕ k_i          p_i = c_i ⊕ k_i
```

Encryption

```
plaintext    0 1 0 0 0 0 0 1
         ⊕
key          0 1 0 1 0 1 0 1
             ———————————————
ciphertext   0 0 0 1 0 1 0 0
```

Decryption

```
ciphertext   0 0 0 1 0 1 0 0
         ⊕
key          0 1 0 1 0 1 0 1
             ———————————————
plaintext    0 1 0 0 0 0 0 1
```

Encryption and decryption are the same operation, because XOR is its own inverse.

The OTP is classified three ways, and all three are examinable:

Symmetric encryption algorithm.
A single key is used for both encryption and decryption

Stream cipher.
Each unit of plaintext is encrypted individually using the corresponding unit of key

Substitution cipher.
Each unit of plaintext is replaced with a unit of ciphertext

WATCH OUT. The OTP uses **substitution but not transposition**. Nothing is reordered — bit i of the plaintext becomes bit i of the ciphertext. That makes it a clean counterexample to "a secure cipher needs transposition."

3.3 The four key requirements

ALL FOUR MUST HOLD.

- Key material is **as long as the message**
- Key material is **never reused**
- Key material is **kept secret**
- Key material is **chosen uniformly at random**

Drop any one and the guarantee is gone. The two that get examined:

Key reuse. If the same key k encrypts two messages, then

```
c1 ⊕ c2 = (p1 ⊕ k) ⊕ (p2 ⊕ k) = p1 ⊕ p2
```

The key cancels. The attacker now holds the XOR of two plaintexts, with no key involved at all, and that is readily attacked using the redundancy of natural language. This is why it is a *one-time* pad; reusing a pad turns it into a weak classical cipher.

Uniform randomness. If the key is generated by a predictable process, the attacker reconstructs the keystream instead of guessing the message, and the pad becomes a stream cipher with a short effective key (which is exactly what §4.3 describes, minus the security).

3.4 Why frequency analysis fails against the OTP

Against a shift cipher, frequency analysis works because the cipher is a *fixed relabelling*: every `e` becomes the same letter, so the frequency distribution is preserved and merely permuted. Count the most common ciphertext symbol and you have found `e`.

WITH A ONE-TIME PAD THERE IS NO SUCH RELABELLING. Each position uses an independent, uniformly random key symbol, so the same plaintext letter in two positions almost certainly produces two different ciphertext symbols. The ciphertext distribution is uniform regardless of the plaintext, so counting frequencies yields nothing.

Said another way: for *any* candidate plaintext of the right length there exists a key that maps the ciphertext to it, and every such key is equally likely. The ciphertext genuinely carries no information about which plaintext was sent.

3.5 Information-theoretic security

INFORMATION-THEORETIC SECURITY. Security derives from information theory rather than from computational cost. The OTP is information-theoretically secure: the attacker cannot recover the plaintext without the key, it is not susceptible to cryptanalysis, and it is not susceptible to exhaustive search.

"Not susceptible to exhaustive search" is the striking part. An attacker *can* try every key — and every key produces a different, perfectly plausible plaintext of the right length, with nothing to distinguish the real one. Exhaustive search succeeds and tells them nothing.

This is a strictly stronger guarantee than the computational security of §1.4: it holds against unlimited computation, forever.

3.6 What the OTP does not give you

Property	Status
Perfect secrecy of the message content	**Yes**
Integrity	**No.** The attacker cannot read the message but can modify it in a predictable way — flipping ciphertext bit i flips plaintext bit i. The OTP is maximally malleable.
Hiding the **length** of the message	**No.** The ciphertext is exactly as long as the plaintext.
Hiding metadata	**No.** That a message was sent, when, and between whom are all visible.
Practicality	**Poor.** Key distribution and key storage are hard — you must pre-share as much secret key material as you will ever send.

EXAM TIP. "An attacker sees a one-time-pad ciphertext on the wire. Using the structure of the ciphertext, what can they learn?" The answer is **the length of the plaintext** — plus the surrounding metadata. Perfect secrecy is a statement about message *content*, nothing more.

3.7 Check yourself

1. A mono-alphabetic substitution produces the ciphertext `AOPP`. Which of `DRSS`, `TREE`, `THAT` is *not* a possible decryption? [L22]

THAT. A mono-alphabetic substitution is a bijection, so equal ciphertext letters must come from equal plaintext letters and vice versa. `AOPP` has its last two letters identical, so the plaintext must too. `DRSS` (x y z z) and `TREE` (x y z z) fit; `THAT` is T H A T — its last two letters differ, and its first and last are the same while `A` and `P` differ. Both ways it is impossible.

2. Same ciphertext `AOPP`, but encrypted with a one-time pad. Which of those three is a possible decryption? [L22]

> **All of them.** Each position has an independent key symbol, so there is a key mapping `AOPP` to any 4-character string you like — including ones with a completely different repetition pattern. This is perfect secrecy stated concretely: the ciphertext rules nothing out.

3. Alice encrypts two different messages with the same OTP key. What can Mallory compute, and why is that bad? [L22]

> `c1 ⊕ c2 = p1 ⊕ p2` — the key cancels entirely. Mallory now has the XOR of two natural-language plaintexts and no longer needs the key at all; the redundancy of language makes that readable with standard techniques. Perfect secrecy depends on the key being used exactly once.

4. True or false: substitution without transposition is insufficient for a secure symmetric cipher. [L22]

> **False.** The one-time pad is pure substitution with no transposition and is information-theoretically secure. Transposition is a useful *practical* building block for ciphers with short keys — it breaks up patterns a short key would otherwise leave visible — but it is not a requirement for security.

L23 4. Building blocks of symmetric ciphers

4.1 Substitution and transposition

Substitution.
Each symbol of plaintext is replaced with a new symbol

Transposition.
Symbols of plaintext are rearranged and reordered in the ciphertext

They do different jobs, and the examinable point is which does which:

	Substitution	Transposition
What it hides	the original **symbols**	the original **patterns**
Examples	Caesar, shift, mono-alphabetic, poly-alphabetic, OTP	rail fence and column transpositions; the ShiftRows step of AES
What survives it	word lengths and repetition structure	the letters themselves, and so their frequencies

Neither alone is enough for a practical cipher with a short key. Substitution alone leaves patterns — `security` enciphered with a fixed substitution still looks like an eight-letter word with a repeated letter. Transposition alone leaves the symbols — `security → cresiytu` has exactly the same letter frequencies as the plaintext. Real block ciphers alternate both, many times.

> WATCH OUT. "Transposition is necessary for any secure symmetric cipher" is **false**: the one-time pad is a counterexample. What *is* true is that transposition obscures patterns, and that ciphers with keys shorter than the message need it.

4.2 Confusion and diffusion

Confusion.
Obscure the relationship between the **key** and the **ciphertext**

Diffusion.
Spread the plaintext statistics across the ciphertext

> THE DISTINCTION, IN ONE LINE. Confusion is about the **key**. Diffusion is about the **plaintext**. If a question mentions the key, it is confusion; if it mentions statistics, patterns, or one change affecting many output bits, it is diffusion.

Diffusion is what makes a single plaintext bit affect many ciphertext bits, so that statistical structure in the message is smeared out and cannot be recovered block by block. Confusion is what stops an attacker who sees ciphertext from reasoning backwards to the key.

EXAM TIP. "All stream ciphers, including the one-time pad, rely solely on _____." The answer is **confusion**. A stream cipher encrypts each symbol independently — `c_i = p_i ⊕ k_i` — so changing one plaintext bit changes exactly one ciphertext bit. That is no diffusion at all. Its security rests entirely on the keystream's relationship to the key being unrecoverable.

4.3 Stream ciphers vs. block ciphers

	Stream cipher	Block cipher
Definition	Each symbol of plaintext (bit, byte, char, digit) is encrypted separately	Groups of symbols are encrypted together as a single block
Examples	Caesar, poly-alphabetic shift, OTP, RC4	AES, DES, 3DES
Diffusion	none within the cipher itself	yes, within a block
Padding	not needed — ciphertext length equals plaintext length	needed, to fill the last block

The line blurs in practice: OFB and CTR modes (§5) use a block cipher to generate a keystream, turning it into a stream cipher.

4.4 AES

Four operations per round, repeated:

Step	What it does	Which building block
SubBytes	Each byte is replaced using an S-box	substitution → confusion
ShiftRows	Row i of the state is rotated left by i	transposition → diffusion
MixColumns	Each column is multiplied by a fixed matrix, so every output byte in a column depends on all four input bytes	diffusion
AddRoundKey	The state is XORed with a round key derived from the main key	mixes the key in → confusion

You will not be asked to perform AES. You should be able to say that it is a **block cipher** that alternates substitution and transposition across many rounds to achieve both confusion and diffusion.

4.5 Check yourself

1. Which is true of transposition: (a) it obscures the symbols of the plaintext, (b) it obscures the patterns of the plaintext, (c) it is necessary for any secure symmetric cipher? [L23]

Only **(b)**. Obscuring symbols is what substitution does — transposition keeps every letter and merely moves it, so letter frequencies are untouched. And (c) is false because the OTP is secure with substitution alone.

2. A cipher XORs each plaintext byte with a keystream byte. One plaintext byte is changed. How many ciphertext bytes change, and which property does that answer name? [L23]

Exactly **one**. That is the absence of **diffusion** — the hallmark of a stream cipher. It is also precisely why stream ciphers are malleable: a predictable, localised change to the ciphertext makes the same predictable change to the plaintext.

3. Which AES step provides diffusion, and how? [L23]

MixColumns primarily — multiplying each column by a fixed matrix makes every output byte depend on all four input bytes of that column, so one input bit influences many output bits. **ShiftRows** contributes by moving bytes between columns, so that over successive rounds the influence spreads across the whole block rather than staying in one column.

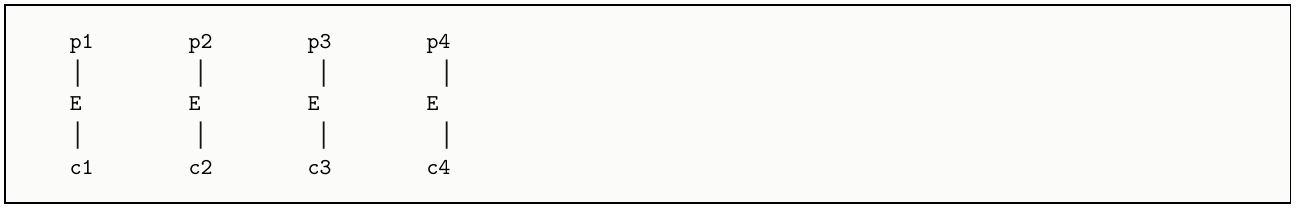
4. A designer proposes a cipher that is pure transposition with a huge key space. Why is it insecure? [L23]

Transposition preserves the multiset of symbols, so the ciphertext has exactly the same letter frequencies as the plaintext. A ciphertext-only attacker can confirm the language, identify the plaintext character set, and attack the ordering with anagramming and known-structure techniques — all without searching the key space. Another case where a large key space defeats exhaustive search and does nothing against cryptanalysis.

L23 5. Block cipher modes of operation

A block cipher encrypts one fixed-size block. A **mode** says how to use it on a message longer than one block. Four modes, and the comparison table in §5.5 is the thing to memorise.

5.1 ECB — Electronic Code Book



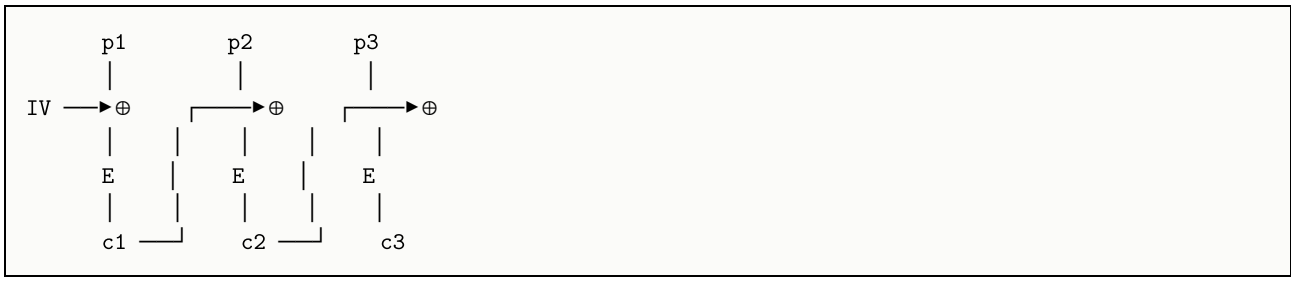
Each block is encrypted independently with the same key. Decryption is the same picture with `D` in place of `E`.

> ECB IS NOT SECURE AND YOU SHOULD BE ABLE TO SAY WHY IN ONE SENTENCE. Encryption is **deterministic** — the same plaintext block always produces the same ciphertext block — so it **cannot be CPA secure**. Identical blocks are visible in the ciphertext, which is why the famous encrypted-penguin image is still recognisably a penguin.

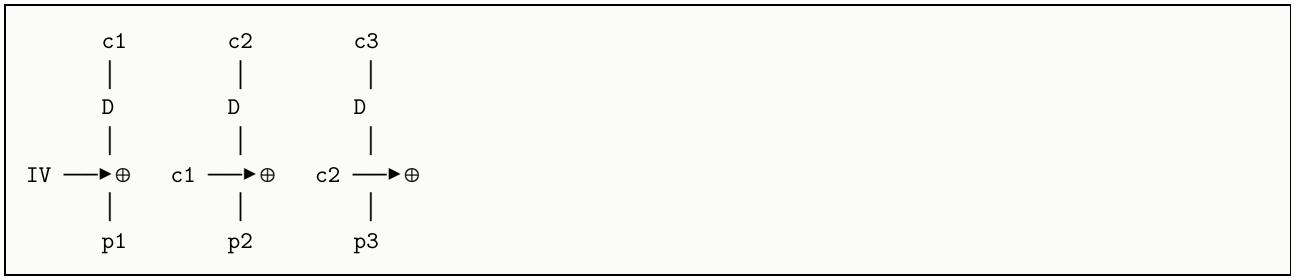
ECB has no IV, nothing is chained, and so it is the only one of the four modes that is deterministic.

5.2 CBC — Cipher Block Chaining

Encryption



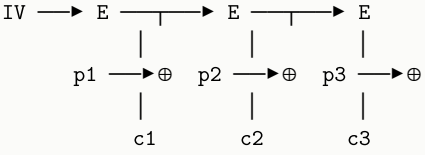
Decryption



- The **IV** is a random initialization vector of length n, where n is the block size. It must be chosen at random **each time** — that is what makes the algorithm probabilistic — and it is sent in the clear.
- **Chaining adds diffusion:** a change to the plaintext in one block changes the output of all subsequent ciphertext blocks.
- The scheme is **CPA secure**.
- **Drawback: encryption must be done sequentially** — block 1 has to be encrypted before block 2 can start.

MUST CBC DECRYPTION BE SEQUENTIAL? NO. Look at the decryption picture: recovering `p_i` needs `D(c_i)` and `c_{i-1}`, and *all* the ciphertext blocks are already in hand. So every block can be decrypted in parallel. Encryption is sequential because it depends on the previous *ciphertext*, which does not exist yet; decryption is parallel because that ciphertext already exists.

5.3 OFB — Output Feedback

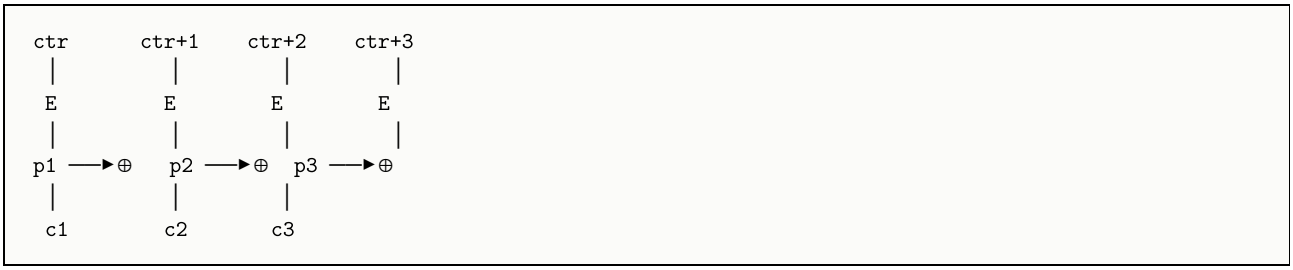


- The block cipher is used to generate a **stream of pseudorandom key bits**, which are XORed with the message — so a block cipher is being used as a **stream cipher**.
- Random IV, sent in the clear. Probabilistic and **CPA secure**.
- The keystream must be generated sequentially, since each block cipher output feeds the next.
- **Pre-processing is possible for the sender:** the keystream depends only on the IV and the key, not on the message, so it can be generated in advance. Once the plaintext arrives, encryption is just a fast XOR.

DECRYPTION IS THE IDENTICAL PICTURE. The same keystream is generated and XORed with the ciphertext. OFB needs no decryption function at all — only `E`.

EXAM TIP. "Why is pre-processing not an option for the receiver?" Because the receiver does not learn the **IV** until the message arrives, and the keystream cannot be generated without it. The sender chooses the IV and so can start early; the receiver is handed it at the last moment.

5.4 CTR — Counter mode



- Encrypting a counter produces a pseudorandom stream of bits, XORed with the plaintext — again a block cipher used as a stream cipher.
- The counter is sent in the clear and must be chosen uniformly at random. Probabilistic and **CPA secure**.
- **Keystream generation is parallelizable** for both encryption and decryption — block i only needs `ctr+i`, which is known immediately.
- **Random access:** you can decrypt the i-th block without decrypting anything else.

Decryption is the same picture as encryption, for the same reason as OFB.

5.5 The comparison table

	ECB	CBC	OFB	CTR
Deterministic?	deterministic	probabilistic	probabilistic	probabilistic
Randomness	none	IV chosen uniformly at random	IV chosen uniformly at random	counter chosen uniformly at random
CPA secure?	**No**	Yes	Yes	Yes
Encryption	parallel	**sequential**	sequential	**parallel**
Decryption	parallel	**parallel**	sequential	**parallel**
Builds a stream cipher?	no	no	yes	yes
Random access?	yes	no	no	**yes**

EXAM TIP. Three facts carry most of the marks: **ECB is deterministic and therefore not secure**; **CBC encrypts sequentially but decrypts in parallel**; **CTR does both in parallel and supports random access**. If you remember only those, you can reconstruct the rest from the diagrams.

5.6 Check yourself

1. Why can ECB never be CPA secure, no matter how strong the underlying block cipher? [L23]

> Because it is deterministic. A CPA attacker requests the encryption of two chosen messages that share a block, sees the identical ciphertext block, and distinguishes them. The weakness is in the mode, not the cipher — AES in ECB mode is insecure despite AES itself being fine.

2. One ciphertext block is corrupted in transit. How many plaintext blocks are damaged under CBC, and under CTR? [L23]

> **CBC: two.** Block i is garbled (it passes through `D`), and block $i+1$ has predictable bit errors, since it is XORed with the corrupted `c_i`. Blocks after that are fine. **CTR: one.** Each block's keystream comes from the counter alone, so damage does not propagate.

3. A developer wants to encrypt a large file stored on disk so that any single record can be read quickly. Which mode, and why? [L23]

> **CTR.** It supports random access — the i-th block can be decrypted on its own from `ctr+i` without touching the rest — and keystream generation is parallelizable. CBC would force decryption of the preceding block to recover the XOR input, and OFB would require generating the keystream from the IV all the way up to the target block.

4. OFB and CTR both turn a block cipher into a stream cipher. What breaks if the IV or counter repeats across two messages under the same key? [L23]

> The same keystream is produced twice, so you have the one-time pad key-reuse problem from §3.3: `c1 ⊕ c2 = p1 ⊕ p2` and the key cancels. This is why the IV or counter must be chosen uniformly at random each time — the mode's entire CPA security rests on never repeating a keystream.

L24 6. Public key encryption and RSA

6.1 The idea

```
plaintext ──► Encrypt ──► ciphertext ──► Decrypt ──► plaintext
                 ▲                          ▲
                 Ke                         Kd
           made public                kept private

           c   = Enc_Ke(msg)
           msg = Dec_Kd(c)
```

Bob publishes `Ke`. Anyone can encrypt to Bob; only Bob, holding `Kd`, can decrypt. The sender and receiver never have to meet.

	Symmetric	Public key
Keys for n parties	`n(n-1)/2` — a key per pair	`2n` — a keypair per person
Key distribution	hard — every pair needs a pre-shared secret	easy — publish the public key
Speed	**fast** algorithms	**slow** algorithms

That trade-off is exactly why real systems use both, which is hybrid encryption in §6.7.

6.2 RSA key generation

Rivest, Shamir and Adleman, 1978; Turing Award 2002; based on the hardness of factoring; keys are 1024 or 2048 bits.

```
1. Given a length l > 2 and an odd e > 2:
2. Generate a prime p, l bits long, with gcd(e, p-1) = 1
3. Generate a prime q ≠ p, l bits long, with gcd(e, q-1) = 1
4. N := pq                                        ← the RSA modulus
5. Find d such that  e·d ≡ 1 mod (p-1)(q-1)   ← the private exponent

   public key  Ke = (N, e)
   private key Kd = (N, d)
```

The quantity `(p-1)(q-1)` is **Euler's totient** `φ(N)` — the number of positive integers less than N that are relatively prime to N. The requirement `gcd(e, φ(N)) = 1` is what guarantees `d` exists.

> WATCH OUT. `p` and `q` must stay secret forever, even though `N = pq` is published. Anyone who factors `N` recovers `φ(N)`, then `d`, and the private key is theirs.

6.3 Encryption and decryption

```
Encryption:  c   = msg^e mod N        (with the public key)
Decryption:  msg = c^d mod N          (with the private key)
```

Correctness is the theorem that for distinct primes p, q with $N = pq$, if `ed ≡ 1 (mod (p-1)(q-1))` then `x^(ed) ≡ x (mod N)` for every integer x. So decrypting re-raises the ciphertext to a power that cancels the encryption.

THE SECURITY ASSUMPTION. Given `N`, `e` and `msg^e`, it is difficult to compute `msg`. Given `N`, `e`, `msg^e` *and* `msg`, it is still difficult to compute `d`. **If the attacker can factor N, they win.**

6.4 One-way and trapdoor functions

One-way function.
A function $F: X \to Y$ that is easy to compute but hard to invert

Trapdoor function.
A one-way function for which there exists a *secret* that makes inversion easy

Easy direction	Hard direction	Trapdoor
multiplying two large primes: `pq = N`	factoring `N` to find `p, q`	—
computing `x^e mod N`	inverting `x^e mod N`	`d` — knowing it makes `x^(ed) mod N` easy

The same vocabulary returns for Diffie-Hellman, where modular exponentiation is the one-way function and the discrete logarithm is the hard direction.

6.5 Worked example

Take `p = 5`, `q = 7`, so `N = 35` and `φ(N) = 4 × 6 = 24`. Choose `e = 5`; then `d = 5` works, since `5 × 5 = 25 ≡ 1 mod 24`.

```
Encrypt m = 2:   c = 2^5 mod 35 = 32 mod 35 = 32
Decrypt c = 32:  m = 32^5 mod 35

   32 ≡ -3 (mod 35)
   (-3)^5 = -243
   -243 + 245 = 2        (since 245 = 7 × 35)
   → m = 2  ✓
```

And with `m = 3`:

```
Encrypt:  c = 3^5 mod 35 = 243 mod 35 = 243 - 210 = 33
Decrypt:  m = 33^5 mod 35

   33 ≡ -2 (mod 35)
   (-2)^5 = -32 ≡ 3 (mod 35)
   → m = 3  ✓
```

EXAM TIP. Reduce to a **small negative** before exponentiating. `33^5` is a sixteen-digit number; `(-2)^5` is −32. Any base above `N/2` should immediately become `base - N`. Watch the sign: an odd exponent keeps it negative, so add a multiple of N at the end.

6.6 Why textbook RSA leaks

A scenario worth working through, because it ties this section back to §2.2. A professor has every student generate an RSA keypair and post the public key `(e, N)` publicly. To distribute grades, the professor encrypts each student's grade g under that student's public key, computing `c ≡ g^e mod N`, and posts `c` next to the student's name. Only the student has the private key, so only the student can decrypt — or so the argument goes.

WHAT IS WRONG. Textbook RSA is **deterministic**, and the space of grades is **tiny**. Anyone can take a student's *public* key, encrypt every possible grade — A, B, C, D, F, or every integer 0–100 — and compare each result against the posted ciphertext. One match reveals the grade. No private key, no factoring, no cryptanalysis: just a few dozen encryptions anyone is able to perform.

This is a CPA-style break in its purest form, and the fix is the one from §2.2: **randomised padding**, so that the same grade encrypts to a different ciphertext every time. Real RSA always pads (OAEP); the bare `m^e mod N` taught here is for understanding the mathematics, not for deployment.

6.7 Hybrid encryption

KEY ESTABLISHMENT. A means of agreeing on a shared secret — for example, two parties establishing the symmetric key to be used for AES encryption.

Public key encryption is slow; symmetric encryption is fast but needs a pre-shared key. Hybrid encryption uses each for what it is good at:

```
Alice:   c1 = PK-Enc_Ke(k)      ← encrypt a fresh session key with Bob's PUBLIC key
         c2 = S-Enc_k(msg)      ← encrypt the actual message symmetrically under k
         send (c1, c2)

Bob:     k   = PK-Dec_Kd(c1)    ← recover the session key with his PRIVATE key
         msg = S-Dec_k(c2)
```

The expensive public-key operation runs once, on a short key; the bulk of the data goes through fast symmetric encryption. This is what TLS does.

6.8 Check yourself

1. With `N = 35`, `e = 5`, `d = 5`, encrypt `m = 3` and decrypt the result. [L24]

`c = 3^5 mod 35 = 243 mod 35 = 33`. Then `m = 33^5 mod 35`; since `33 ≡ -2`, this is `(-2)^5 = -32 ≡ 3 mod 35`. Back to **3**.

2. 200 people want to communicate pairwise. How many keys with symmetric encryption, and how many with public key? [L24]

Symmetric: `n(n-1)/2 = 200 × 199 / 2 = 19,900` shared keys, each needing a secure out-of-band exchange. Public key: `2n = 400` keys — 200 keypairs — and the public halves can simply be published. That gap is the practical argument for public key cryptography.

3. If an attacker factors `N`, exactly what can they compute, and in what order? [L24]

Factoring gives `p` and `q`. From those, `φ(N) = (p-1)(q-1)`. Since `e` is public and `ed ≡ 1 mod φ(N)`, they invert `e` modulo `φ(N)` to get `d` — the private key. They can now decrypt everything ever sent under that key and forge signatures. This is why RSA's security is exactly the hardness of factoring.

4. In hybrid encryption, why not just encrypt the whole message with RSA? [L24]

Public key algorithms are far slower than symmetric ones, and RSA can only encrypt messages smaller than the modulus, so a long message would have to be split into many blocks and each would need a separate expensive modular exponentiation. Encrypting one short session key and doing the bulk work with AES gives the key-distribution benefit of public key cryptography at the speed of symmetric cryptography.

L24 · L25

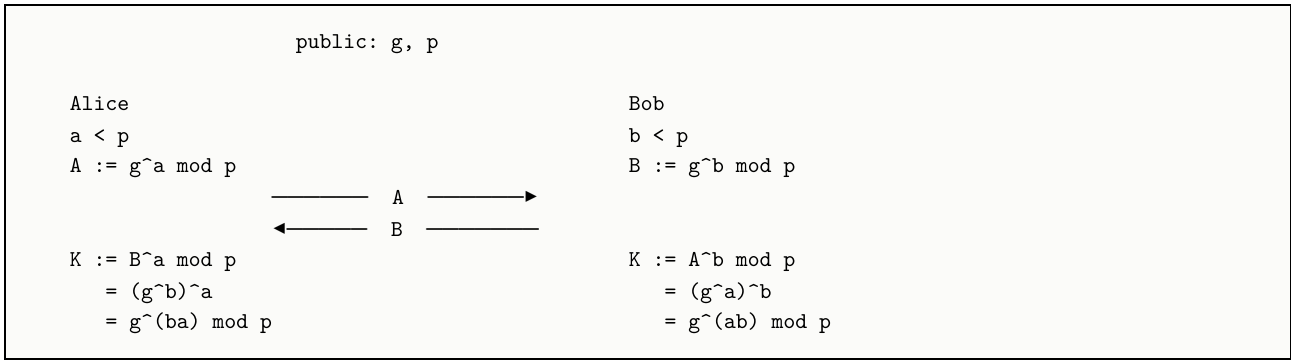
7. Diffie-Hellman, replay and freshness

7.1 The setup

Two public parameters: a prime `p`, and `g`, a **primitive root** of `p`.

> PRIMITIVE ROOT. `g` is a primitive root of `p` if for every `n` with `1 ≤ n < p` there exists an integer `i` such that `g^i mod p = n`. In other words, powers of `g` run through every value in the group — nothing is unreachable.

7.2 The protocol



Correctness: `K = A^b = (g^a)^b = g^(ab) = (g^b)^a = B^a mod p`. Both sides compute the same value, and neither `a` nor `b` ever crosses the wire.

> SECURITY. The attacker sees `g`, `p`, `A` and `B` and must find `K`. **If the attacker can compute the discrete log of A or B, they win** — recovering `a` from `A = g^a mod p` lets them compute `K = B^a` exactly as Alice does.

The one-way function here is **modular exponentiation**: computing `g^a mod p = A` is easy, while computing `dlog_{g,p}(A) = a` is hard. Note the contrast with RSA — there is no trapdoor, because nobody is supposed to be able to invert it.

7.3 Worked example

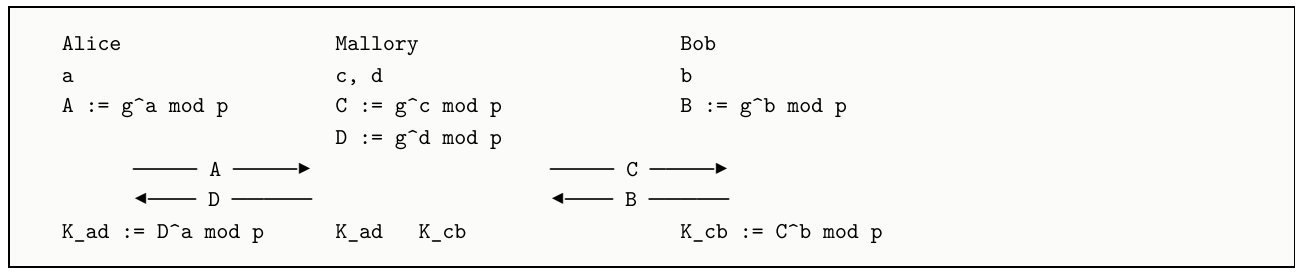
Take `g = 2`, `p = 11`, with Alice choosing `a = 3` and Bob choosing `b = 5`.

```
A = 2^3 mod 11 = 8  mod 11 = 8
B = 2^5 mod 11 = 32 mod 11 = 10

Alice: K = B^a mod 11 = 10^3 mod 11 = 1000 mod 11 = 10
Bob:   K = A^b mod 11 = 8^5  mod 11

       8^2 = 64  ≡ 9  (mod 11)
       8^4 = 9^2 = 81 ≡ 4  (mod 11)
       8^5 = 8^4 · 8 = 4 · 8 = 32 ≡ 10 (mod 11)

Shared key K = 10  ✓
```

EXAM TIP. Reduce at every step and use repeated squaring — `8^5` via `8^2 → 8^4 → 8^4·8` keeps every number under 100. For `10^3 mod 11`, note `10 ≡ -1`, so `(-1)^3 = -1 ≡ 10`. The negative trick works here too.

7.4 Machine-in-the-middle

MACHINE-IN-THE-MIDDLE (MITM), ALSO ADVERSARY-IN-THE-MIDDLE. Two parties, believing that they are communicating with each other, are in fact communicating with an attacker.



Mallory runs two separate exchanges. Alice ends up sharing `K_ad` with Mallory while believing she shares it with Bob; Bob shares `K_cb` with Mallory while believing he shares it with Alice. Mallory decrypts everything from one side, reads or alters it, re-encrypts under the other key, and forwards. Neither party notices.

THE ROOT CAUSE, AND THE PHRASE TO WRITE: THERE IS NO AUTHENTICATION. Plain Diffie-Hellman establishes a shared secret with *somebody*, and proves nothing about who that somebody is. The fix is to authenticate the exchange — sign the DH values, or run it inside a channel already authenticated by certificates.

7.5 Replay attacks and freshness

Recall hybrid encryption (§6.7). Alice sends: "I'm Alice. Let's use `Enc_Ke(S)`" — a session key encrypted under Bob's public key.

> REPLAY ATTACK. The attacker records the setup message and sends it again later, claiming to be Alice. Bob may accept the old key as a new session key. **Encryption alone does not establish the sender's identity or the message's freshness.**

Note what the attacker does *not* need: they never decrypt anything and never learn `S`. They simply resend bytes they captured. The message is valid, correctly encrypted and correctly formed — it is just *old*. That is why confidentiality is no defence against it.

Why replaying an old session key is dangerous: if Mallory ever recovers one old session key — through a later compromise, a weak cipher, or a leaked log — she can force Bob back onto that key at will and read everything in the new session.

> NONCE. A number used only once as part of a cryptographic protocol. It may be a **random number** providing freshness and message continuity, a **sequence number** demonstrating uniqueness, or a **timestamp** demonstrating timeliness. The IVs used in CBC, OFB and CTR modes are nonces.

Kind of nonce	What it proves	What it needs
Random challenge	freshness — the response was computed after the challenge was issued	a round trip, and state to remember the challenge
Sequence number	uniqueness — this message has not been seen before	both sides to keep a counter
Timestamp	timeliness — the message is recent	synchronised clocks and an acceptance window

Bind the nonce into the authenticated part of the message and a replayed copy is detectably stale — the sequence number has already been used, the timestamp has expired, or the challenge does not match the one just issued.

7.6 Check yourself

1. With `g = 2`, `p = 11`, `a = 4`, `b = 3`, compute `A`, `B` and `K`. [L25]

`A = 2^4 mod 11 = 16 mod 11 = 5`. `B = 2^3 mod 11 = 8`. Alice: `K = B^a = 8^4 mod 11`; `8^2 = 64 ≡ 9`, so `8^4 ≡ 9^2 = 81 ≡ 4`. Bob: `K = A^b = 5^3 = 125 mod 11 = 125 - 121 = 4`. Both get $\mathbf{K = 4}$.

2. Mallory records every message of a DH exchange but cannot modify anything. Can she compute `K`? [L25]

> No. She has `g`, `p`, `A` and `B`, and computing `K = g^(ab)` from those requires solving the discrete logarithm problem, which is believed hard. DH is secure against a **passive** attacker; it is the **active** attacker who defeats it, and only because there is no authentication.

3. Bob receives a correctly encrypted, well-formed session-key setup message from "Alice." What has he actually established? [L24 · L25]

> Almost nothing. He knows *someone* encrypted a value under his public key — which anyone can do, since the key is public — and he has no evidence of who sent it or when. The message could be a replay of one Alice sent last year. Encryption gives confidentiality; identity needs authentication and freshness needs a nonce.

4. Would adding a timestamp to the setup message stop the replay? What does it cost? [L24]

> Yes, if the timestamp is inside the authenticated portion and Bob rejects anything outside an acceptance window — a replay carries its original, now-expired timestamp. The cost is **synchronised clocks**, plus a window that is a genuine trade-off: too wide and replays succeed inside it, too narrow and legitimate messages are rejected by clock drift. A sequence number or a random challenge avoids the clock dependency at the cost of keeping state or adding a round trip.

L25 8. Cryptographic hash functions

The move from confidentiality to **integrity**.

> MESSAGE INTEGRITY. The message has not been modified.

Alice sends "Do not trust Mallory"; Mallory changes it to "You can trust Mallory." Encryption does not help — Mallory does not need to read the message to alter it. The tool for detecting the change is a cryptographic hash.

8.1 What a hash function is

> HASH FUNCTION. A function that takes an arbitrary-length input and produces a fixed-length output.

The output goes by several names: **hash value**, **hash**, **message digest**, **digital fingerprint**. Even `m mod 10` is a hash function by this definition — `3 % 10 = 3` and `3590224723940253 % 10 = 3` — which is exactly why "hash function" and "cryptographic hash function" are different things.

> COLLISIONS MUST EXIST. By the **pigeonhole principle**, infinitely many inputs map to finitely many outputs, so some inputs must share a hash. A collision-resistant hash function is *not* one with no collisions — it is one where collisions are computationally infeasible to *find*.

It follows that hash functions are **not invertible**: inverting one would require a single hash value to map back to many messages, which is not a function. A hash can therefore **reveal** modifications but can never **guarantee** that none exist.

8.2 The three properties

Listed in **decreasing order of strength** — collision resistance is the strongest:

Property	Definition	What the attacker gets to choose
Collision resistant	Hard to find *any* `msg`, `msg'` with `msg ≠ msg'` and `h(msg) = h(msg')`	**Both** messages. Nothing is fixed in advance.
Second pre-image resistant	*Given* `msg`, hard to find `msg'` with `msg ≠ msg'` and `h(msg) = h(msg')`	Only the **second** message — the first is handed to them.
Pre-image resistant	Given `H = h(msg)`, hard to find *any* `msg'` with `h(msg') = H`	Neither message. They see only the hash.

EXAM TIP. Match by **what is fixed in advance**. Nothing fixed → collision resistance. One message fixed → second pre-image. Only the hash value is given → pre-image. The common error is swapping the first two: "create a different contract with the same hash as *this* contract" fixes one message, so it is **second pre-image**; "create any two documents with the same hash" fixes nothing, so it is **collision**.

CRYPTOGRAPHIC HASH FUNCTION. A hash function strong enough to resist collisions — collision resistant, second pre-image resistant, and pre-image resistant.

8.3 The birthday paradox

Collision resistance is harder to achieve than it looks, because finding *some* collision is much easier than hitting a *specific* target.

Number of people	Probability of a shared birthday
5	3%
20	41%
50	97%
100	99.99%

Fifty people out of 365 days gives a 97% chance of a collision, because the number of *pairs* grows quadratically while the number of people grows linearly. The consequence for hashing: an n-bit hash needs about $2^{n/2}$ work to find a collision, not 2^n. A 256-bit hash gives roughly 128 bits of collision resistance, which is why hash outputs are twice as long as you might first expect.

8.4 The real hash functions

Function	Status
MD5	deprecated — collisions are findable in seconds
SHA-1	deprecated — collisions demonstrated in practice
SHA-2	in use; SHA-256 is the common variant
SHA-3	in use; multiple variants

MD5 and SHA-1 both failed on *collision* resistance first — the strongest of the three properties, and so the first to break.

8.5 What a bare hash does not give you

Alice sends `(msg, H)` with `H = h(msg)`, and Bob recomputes `h(msg)` and compares. What has Bob learned?

AGAINST AN ACTIVE ATTACKER: NOTHING. The hash function is public and unkeyed, so Mallory replaces `msg` with `msg'`, computes `H' = h(msg')` herself, and sends `(msg', H')`. Bob's check passes. A bare hash detects **accidental** corruption — transmission errors, disk rot — and provides no defence against deliberate modification.

What is missing is a **secret**. Fixing that is the MAC in §9.

8.6 Check yourself

1. A hash function takes messages of any length and produces a 256-bit output. Which is correct: (a) a collision-resistant hash has no collisions; (b) collisions must exist but finding one should be infeasible; (c) any two messages of the same length have the same hash; (d) different messages always have different hashes? [L25]

(b). There are infinitely many possible inputs and only 2^{256} outputs, so by the pigeonhole principle collisions must exist. Collision resistance is about the computational difficulty of *finding* one, not about their absence.

2. Match each goal to the property it defeats: (a) given only `H`, find any message hashing to `H`; (b) given a particular contract, create a different contract with the same hash; (c) create any two different documents with the same hash, neither fixed in advance. [L25]

(a) **pre-image resistance** · (b) **second pre-image resistance** · (c) **collision resistance**. The tell in each case is how much the attacker is given: just a hash, one fixed message, or complete freedom.

3. Let `h(m)` be the sum of the decimal digits of `m`, modulo 10, so `h(123) = 6`. Give a different integer with the same hash. Is this collision resistant? [L25]

Any integer whose digits sum to 6 mod 10 works — **321** (a permutation), or **15**, or **6**, or **114**. It is **not** collision resistant at all: there are only 10 possible outputs, digit order is irrelevant, and collisions can be constructed by inspection. It fails all three properties, since inverting it is also trivial.

4. A download page lists a SHA-256 hash next to the file. What attack does this stop, and what does it not? [L25]

It stops **accidental corruption** and catches a tampered *mirror*, provided the hash itself came over a channel the attacker does not control. It does **not** stop an attacker who controls the page: they replace the file and the listed hash together, and the check passes. The hash is only as trustworthy as the channel carrying it — which is why signed releases exist.

L26 9. MACs, signatures and encrypt-then-sign

A bare hash fails against an active attacker because it has no secret in it (§8.5). Adding a secret gives a MAC; using a *private* key instead gives a digital signature.

9.1 Message authentication codes

MAC. Alice and Bob share a secret key `k`. Alice sends `(msg, tag)` with `tag = MAC_k(msg)`. Bob recomputes `MAC_k(msg)` and compares. Mallory cannot forge a valid tag because she does not have `k`.

Comparing the two protocols directly:

Alice sends	What Bob gets when the check passes
`(msg, H)` with `H = h(msg)`	**Nothing, against an active attacker.** The hash is public and unkeyed, so Mallory substitutes `msg'` and recomputes `H' = h(msg')`. Only accidental corruption is detected.
`(msg, tag)` with `tag = MAC_k(msg)`	**Integrity** — the message was not modified in transit. **Authentication** — it came from someone who holds `k`.

What a MAC does not give you.

- **No confidentiality.** `msg` travels in the clear; a MAC authenticates, it does not hide.
- **No non-repudiation.** Alice and Bob share `k`, so Bob could have produced the tag himself. He cannot prove to a third party that Alice sent it.
- **No freshness.** A recorded `(msg, tag)` pair stays valid forever, so replay still works unless a nonce or sequence number is inside the MACed data.

That second limitation is the entire reason digital signatures exist.

9.2 Digital signatures

Signing inverts the key roles of encryption. Alice signs with her **private** key; anyone verifies with her **public** key.

```
sig = m^s mod N          ← s is Alice's private signing exponent
verify: sig^v mod N = m  ← v is Alice's public verification exponent
```

	MAC	Digital signature
Key type	shared secret	signer's private key
Who can verify	anyone holding `k` — so, the other party	anyone at all

Non-repudiation	**no**	**yes**
Speed	fast	slow

9.3 Worked signature example

Alice's private signing exponent `s = 3`, her public verification exponent `v = 7`, `N = 33`, message `m = 2`.

```
Sign:    sig = m^s mod N = 2^3 mod 33 = 8

Verify:  sig^v mod N = 8^7 mod 33

   8^2 = 64  ≡ 31 (mod 33)
   8^4 = 31^2 = 961 ≡ 4  (mod 33)        [961 − 957 = 4, 957 = 29 × 33]
   8^7 = 8^4 · 8^2 · 8 = 4 · 31 · 8
       = 124 · 8 ≡ 25 · 8 = 200 ≡ 2 (mod 33)

   → 2 = m, so the signature verifies ✓
```

EXAM TIP. Does sending `(m, sig)` keep the message confidential? **No**, twice over. First, `m` is right there in the clear. Second, even if it were omitted, `v` is public, so anyone computes `sig^v mod N = m` and recovers it. **Signing is not encrypting.** Signatures give integrity, authentication and non-repudiation — never confidentiality.

9.4 Encrypt-then-sign

To get confidentiality *and* authenticity, combine both — and the order and key choices matter.

Step	Key used
Alice encrypts the message to get `c`	**Bob's public key**
Alice signs `c` to get `tag`	**Alice's private key**
Bob verifies the signature on `c`	**Alice's public key**
Bob decrypts `c`	**Bob's private key**

THE RULE THAT GENERATES THE WHOLE TABLE. You use **other people's public keys** to send to them and to check them — encrypt with the recipient's public key, verify with the sender's public key. You use **your own private key** for the two things only you may do — sign and decrypt.

Alice transmits `(c, tag)`. Now suppose Mallory replaces `c` with a different ciphertext `c'` but keeps Alice's original tag. **Bob's signature check fails**, because the signature was computed over `c` and Bob verifies it against `c'`. Any change to the ciphertext is detected before Bob ever decrypts it.

That last part is a practical benefit worth naming: Bob verifies *first* and only decrypts data that has already been authenticated, so he never runs his private key over attacker-chosen input. An attacker who cannot get past the signature check cannot use Bob as a decryption oracle, which is what makes a chosen-ciphertext attack possible.

WHY THE OTHER ORDERING IS RISKY. Under **sign-then-encrypt**, Alice signs `m` and then encrypts `(m, sig)` to Bob. Bob decrypts, and now holds a message signed by Alice with *nothing binding it to him.* He can re-encrypt `(m, sig)` under Carol's public key and forward it; Carol verifies Alice's signature and concludes Alice sent her the message. The signature authenticates the *content* but not the *intended recipient.*

Worth knowing: encrypt-then-sign has a mirror-image limitation. Mallory can strip Alice's signature from `c` and sign the same `c` with her own key. She learns nothing — she cannot decrypt `c` — but Bob may conclude the message came from Mallory. Neither ordering is complete on its own; real protocols name the sender and recipient *inside* the encrypted payload so the binding cannot be stripped or redirected.

9.5 Check yourself

1. Alice sends `(msg, H)` with `H = h(msg)` over a network with an active attacker. Bob's check passes. What does he know? [L26]

Essentially nothing about deliberate tampering. `h` is public and unkeyed, so Mallory can substitute any `msg'` and recompute `H' = h(msg')`, and Bob's check still passes. He learns only that the message was not *accidentally* corrupted. There is no secret anywhere in the protocol, so there is nothing an attacker lacks.

2. Same setup, but with `tag = MAC_k(msg)` under a shared key. Now what does Bob know — and what can he still not prove? [L26]

He gets **integrity** (the message was not modified) and **authentication** (it came from a holder of `k`). He cannot prove to a **third party** that Alice sent it, because he holds `k` too and could have generated the tag himself — no non-repudiation. He also has no **freshness** guarantee: a replayed old pair verifies perfectly. And `msg` was never confidential.

3. Fill in the keys: Alice encrypts to get `c`; Alice signs `c`; Bob verifies; Bob decrypts. [L26]

Encrypt → **Bob's public key**. Sign → **Alice's private key**. Verify → **Alice's public key**. Decrypt → **Bob's private key**. Public keys for acting on someone else's behalf (sending to them, checking them); your own private key for the two operations only you can perform.

4. Mallory replaces `c` with `c'` but keeps Alice's tag. Does Bob's check pass? [L26]

No. The signature is over `c`; verifying it against `c'` fails. This is the point of signing the ciphertext — any modification is caught before decryption, so Bob never applies his private key to attacker-supplied data.

5. A protocol MACs every message with a shared key but includes no nonce. Which attack still works? [L24 · L26]

Replay. The recorded `(msg, tag)` pair remains valid indefinitely, so Mallory resends it and Bob accepts — the tag is genuine and the message is unmodified, it is simply old. A MAC gives integrity and authentication but says nothing about *when*. Fix it by MACing a nonce, sequence number or timestamp along with the message.

10. Mock test

Ten questions, closed-book, 50-minute timer, on paper. Do this before you read §11.

Question 1: Attack models. Match each to ciphertext-only, known-plaintext, CPA or CCA.
1.1. A web server encrypts whatever a user types into a search box and the attacker can read the result.
1.2. An attacker captures Wi-Fi traffic and has nothing else.
1.3. An attacker intercepts a file whose standard 40-byte header they already know, together with its ciphertext.
1.4. A padding-oracle attacker sends altered ciphertexts to a server and learns whether each decrypts validly. [L21]

> **1.1. CPA** — the attacker chooses the plaintext and obtains its encryption.
> **1.2. Ciphertext-only** — observation with no plaintext and no service access.
> **1.3. Known-plaintext** — they have a plaintext–ciphertext pair but did not choose the plaintext; it was always going to be that header.
> **1.4. CCA** — they submit ciphertexts of their choosing and learn something about the decryption.

Question 2: Key reuse. Alice encrypts `p1` and `p2` with a one-time pad, accidentally using the same key `k` for both.
2.1. Write the expression Mallory can compute from `c1` and `c2`, showing the cancellation.
2.2. Why is that dangerous when Mallory does not have `k`?
2.3. Which of the four OTP key requirements was violated, and does violating it affect the other three? [L22]

> **2.1.** `c1 ⊕ c2 = (p1 ⊕ k) ⊕ (p2 ⊕ k) = p1 ⊕ p2`. XOR is associative and commutative and `k ⊕ k = 0`, so the key cancels.
> **2.2.** She now holds the XOR of two natural-language plaintexts, with the key gone from the problem entirely. English is highly redundant, so guessing a crib in one message reveals the corresponding stretch of the other, and the two can be peeled apart. The key being secret no longer helps, because it is no longer involved.
> **2.3. Key material is never reused.** The other three can all still hold — the key can be message-length, secret and uniformly random, and the scheme is still broken. Perfect secrecy needs all four; each is independently necessary.

Question 3: Modes. A developer encrypts a database of 10,000 records, one block each, with AES in ECB mode, and notes that AES is unbroken.
3.1. What can an attacker learn without any key?
3.2. Name the property ECB lacks and the security notion it therefore fails.
3.3. They switch to CBC. Can they still decrypt record 7,000 on its own? [L23]

3.1. Which records are *identical*. ECB maps equal plaintext blocks to equal ciphertext blocks, so repeated values — a status field, a salary band, a diagnosis code — are visible as repeated ciphertexts, along with their frequencies. Combined with any outside knowledge of the distribution, individual values can often be identified.
3.2. It is **deterministic** (no IV, no chaining), so it **cannot be CPA secure**. The weakness is in the mode, not in AES.
3.3. Yes, but not *on its own*: decrypting block i under CBC needs `D(c_i)` XORed with `c_{i-1}`. Both ciphertext blocks are already available, so no other decryption is required and CBC decryption parallelizes — but you do have to fetch the preceding ciphertext block. CTR mode would give true random access from the counter alone.

Question 4: RSA. Use `p = 3`, `q = 11`, so `N = 33` and `φ(N) = 20`. Take `e = 3`.
4.1. Verify that `d = 7` is a valid private exponent.
4.2. Encrypt `m = 5`.
4.3. Decrypt your answer to confirm. [L24]

4.1. Need `ed ≡ 1 mod φ(N)`: `3 × 7 = 21 = 20 + 1 ≡ 1 mod 20` ✓. Also `gcd(3, 20) = 1`, so `e` is valid.
4.2. `c = 5^3 mod 33 = 125 mod 33 = 125 − 99 = 26`.
4.3. `m = 26^7 mod 33`. Use `26 ≡ −7 (mod 33)`: `(−7)^2 = 49 ≡ 16`; `(−7)^4 ≡ 16^2 = 256 ≡ 256 − 231 = 25 ≡ −8`; so `(−7)^7 = (−7)^4 · (−7)^2 · (−7) = (−8)(16)(−7)`. Then `(−8)(16) = −128 ≡ −128 + 132 = 4`, and `4 × (−7) = −28 ≡ 5 (mod 33)`. Back to **5** ✓.

Question 5: Diffie-Hellman. Public parameters `g = 3`, `p = 7`. Alice picks `a = 2`; Bob picks `b = 4`.
5.1. Compute `A`, `B`, and the shared key `K` from both sides.
5.2. An eavesdropper records `g`, `p`, `A` and `B`. What problem must they solve?
5.3. An attacker who can also modify traffic mounts a different attack. Name it and state the single missing property that enables it. [L25]

5.1. `A = 3^2 mod 7 = 9 mod 7 = 2`. `B = 3^4 mod 7`: `3^2 ≡ 2`, so `3^4 ≡ 2^2 = 4`. Alice: `K = B^a = 4^2 = 16 mod 7 = 2`. Bob: `K = A^b = 2^4 = 16 mod 7 = 2`. $\mathbf{K = 2}$ ✓
5.2. The **discrete logarithm problem** — recovering `a` from `A = g^a mod p`. Modular exponentiation is one-way, so a passive eavesdropper is stuck.
5.3. Machine-in-the-middle. The missing property is **authentication**: plain DH establishes a shared secret with *somebody* and proves nothing about who. Mallory runs one exchange with each party and relays.

Question 6: Hash properties. For each, name the property being attacked.
6.1. A forger is shown a signed purchase order for $100 and wants a different order with the same hash.
6.2. A researcher produces two PDFs with identical SHA-1 digests, having written both.
6.3. An attacker has a stolen password digest and wants any string that hashes to it.
6.4. Which of the three is strongest, and what does "strongest" mean here? [L25]

6.1. Second pre-image resistance — one message is fixed in advance.
6.2. Collision resistance — neither document was fixed; the attacker chose both.
6.3. Pre-image resistance — only the hash value is given.
6.4. Collision resistance is strongest, meaning it is the hardest to achieve and the first to fall: an attacker with total freedom over both inputs has the easiest job (helped by the birthday bound, $\sim 2^{n/2}$ work). A function that is collision resistant is necessarily second pre-image resistant, but not conversely — which is why MD5 and SHA-1 broke on collisions first.

Question 7: Integrity. Alice sends Bob `(msg, h(msg))` over a network with an active attacker.
7.1. Describe the attack in two steps.
7.2. She switches to `(msg, MAC_k(msg))`. Name the two properties Bob now gets.
7.3. Name two properties he still does not get. [L25 · L26]

7.1. (1) Mallory intercepts and replaces `msg` with `msg'`. (2) She computes `H' = h(msg')` — the hash function is public and takes no key — and forwards `(msg', H')`. Bob's check passes.
7.2. Integrity and **authentication** (it came from a holder of `k`).
7.3. Any two of: **confidentiality** (`msg` is in the clear), **non-repudiation** (Bob shares `k`, so he cannot prove to anyone else that Alice sent it), **freshness** (a replayed pair verifies perfectly).

Question 8: Encrypt-then-sign. Alice sends Bob a confidential, verifiable message. Both know each other's correct public keys.
8.1. Give the four operations in order with the key used for each.
8.2. Mallory intercepts `(c, tag)` and forwards `(c', tag)` for some other ciphertext `c'`. What happens, and why does this matter before decryption?
8.3. Why is signing first and then encrypting the pair a weaker design? [L26]

8.1. Alice encrypts `m` → **Bob's public key**, giving `c`. Alice signs `c` → **Alice's private key**, giving `tag`. Bob verifies `tag` over `c` → **Alice's public key**. Bob decrypts `c` → **Bob's private key**.
8.2. The verification **fails**: the signature was computed over `c`, not `c'`. It matters that this happens first because Bob rejects the message without ever applying his private key to attacker-chosen input — denying Mallory the decryption oracle a chosen-ciphertext attack requires.
8.3. Under sign-then-encrypt, Bob decrypts and holds `(m, sig)`, a validly signed message with nothing binding it to him as the intended recipient. He can re-encrypt it to Carol, who verifies Alice's signature and believes Alice wrote to her. The signature authenticates the content but not the destination.

Question 9: Freshness. A smart lock accepts an "unlock" command encrypted under a shared key. The cipher is strong and correctly used.
9.1. Name and describe the attack available to someone who can record radio traffic.
9.2. Why doesn't encryption prevent it?
9.3. Give two fixes and state the cost of each. [L24 · L26]

9.1. A **replay attack**. The attacker records the ciphertext of a legitimate unlock and retransmits it later; the lock decrypts a perfectly valid command and opens.
9.2. Because nothing in the message is *new*. Encryption hides content, and the attacker never needs the content — they resend the exact bytes. Confidentiality says nothing about identity or recency.
9.3. Sequence number inside the authenticated message: the lock rejects any counter it has already seen. Cost: both sides keep state, and they can desynchronise. **Random challenge** from the lock, which the fob must incorporate into its response: cost is an extra round trip and the lock must remember the outstanding challenge. (A **timestamp** is a third option, costing synchronised clocks and an acceptance window inside which replays still succeed.)

Question 10: Synthesis. A developer encrypts every message with AES-CTR under a key established by unauthenticated Diffie-Hellman, and argues the system is secure because AES is unbroken and DH is based on a hard problem. Give three distinct failures.
[L23 · L25 · L26]

(1) No authentication of the key exchange. Unauthenticated DH falls to a machine-in-the-middle: Mallory shares one key with each party and relays, reading and altering everything. The hardness of discrete log is irrelevant, because she never solves it.
(2) No integrity. CTR is a stream cipher, so it is maximally malleable — flipping ciphertext bit i flips plaintext bit i predictably. Encryption provides confidentiality only; the design needs a MAC or signature over the ciphertext.
(3) No freshness. Nothing binds a message to a point in time, so recorded ciphertexts can be replayed and will decrypt correctly. A nonce or sequence number inside the authenticated data is required.

A fourth, if the counter is mismanaged: reusing a counter value under the same key repeats the keystream, giving `c1 ⊕ c2 = p1 ⊕ p2` and the one-time-pad key-reuse break. "The primitives are strong" is never an argument that a protocol is secure.

11. The in-class practice problems, worked

Every in-class practice question from L21 through L26, with answers and notes on where marks get lost. If you only have twenty minutes left, read this.

11.1 L21 — Foundations and attack models

Q1. A company publishes its encryption algorithm but keeps each user's key secret. Does publishing the algorithm necessarily make the system insecure? Name the principle. [L21]

No. The principle is **Kerckhoffs's principle**: the security of a cryptosystem should depend only on the secrecy of the keys. Publishing the algorithm invites public scrutiny, which strengthens confidence in it, and keys can be generated per user and replaced when compromised — an algorithm cannot.

Where marks get lost. Answering "no" without naming the principle. The question asks for it explicitly.

Q2. A shift cipher uses `Enc_k(m) = (m + k) mod 26`. You learn `cat` encrypts to `fdw`. (2.1) What is the key, and what does `grj` decrypt to? (2.2) Is this exhaustive search or cryptanalysis? [L21]

2.1. c(2)→f(5), a(0)→d(3), t(19)→w(22): every letter shifts by 3, so **k = 3**. Decrypting `grj` by subtracting 3: g(6)→d(3), r(17)→o(14), j(9)→g(6) = **dog**.
2.2. Cryptanalysis. No keys were tried; the key was deduced from a known plaintext–ciphertext pair using the structure of the cipher (one uniform shift for every letter). Exhaustive search would mean testing all 25 shifts and reading the output.

Where marks get lost. Calling it exhaustive search because the key space happens to be small. The distinction is about *method*, not size — you exploited structure, so it is cryptanalysis.

Q3. Match each scenario to A. ciphertext-only, B. known-plaintext, C. chosen-plaintext, D. chosen-ciphertext. (3.1) Submits messages of their choice, receives their encryptions. (3.2) Observes encrypted messages, no plaintexts, no service access. (3.3) Submits modified ciphertexts and receives decryptions, but not of the target. (3.4) Obtains existing messages and their ciphertexts without choosing them. [L21]

3.1 → C (chosen-plaintext) · **3.2 → A** (ciphertext-only) · **3.3 → D** (chosen-ciphertext) · **3.4 → B** (known-plaintext).

Where marks get lost. Confusing 3.1 and 3.4. Both involve plaintext–ciphertext pairs; the difference is *who picked the plaintext*. Chosen means the attacker selected it; known means they were simply handed it.

11.2 L22 — Substitution, transposition and the OTP

Q1. A mono-alphabetic substitution produced the ciphertext `AOPP`. Which is *not* a possible decryption: DRSS, TREE, THAT, or none? [L22]

THAT. A mono-alphabetic substitution is a one-to-one mapping, so the repetition pattern is preserved exactly. `AOPP` has the shape x-y-z-z, with the last two letters equal and the first three distinct. DRSS and TREE both have that shape. THAT is T-H-A-T: its last two letters differ (so it cannot produce `PP`) and its first and last are equal (so it would require `A` and `P` to be the same). Impossible either way.

Where marks get lost. Checking only one direction. Say explicitly that the mapping is a bijection, so equal plaintext letters force equal ciphertext letters *and* vice versa.

Q2. Same ciphertext `AOPP`, under a one-time pad. Which is a possible decryption? [L22]

All of the above. Each position is XORed with an independent key symbol, so there is a key mapping `AOPP` to any four-character string whatsoever — the repetition pattern carries no information. This is perfect secrecy made concrete: the ciphertext rules out nothing.

Where marks get lost. Carrying the pattern-matching reasoning over from Q1. The two questions are deliberately paired to test whether you noticed that the OTP removes exactly that structure.

Q3. A Caesar ciphertext is susceptible to letter-frequency analysis. Is the same true of a one-time pad ciphertext? [L22]

No. Caesar applies one fixed shift to every letter, so the frequency distribution survives intact and is merely relabelled — the most common ciphertext letter is `e`. A one-time pad uses an independent, uniformly random key symbol at every position, so the same plaintext letter in two places generally produces two different ciphertext symbols, and the ciphertext distribution is uniform no matter what the plaintext was. There is no frequency structure left to count.

Where marks get lost. Saying "because the key is longer." The length matters only because it allows the key to be *independent at every position* — that independence is what destroys the frequency signal.

Q4. Mallory sees an OTP ciphertext on the wire. Using the structure of the ciphertext, what can she learn about the plaintext? [L22]

Its **length** — the ciphertext is exactly as long as the plaintext. Beyond the ciphertext itself she also learns the metadata: that a message was sent, when, between whom, and how often. Perfect secrecy is a guarantee about message *content* only.

Q5–Q9. Transposition obscures _____ · Substitution without transposition is insufficient for a secure cipher (T/F) · Substitution obscures _____ · The OTP uses _____ · All stream ciphers rely solely on _____ [L22]

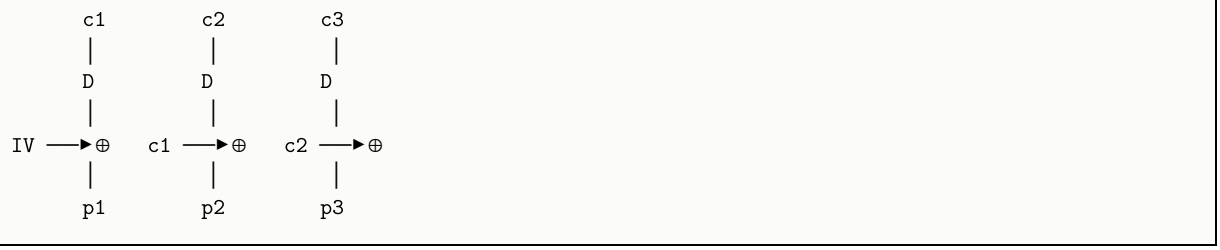
Q5. Transposition obscures **the patterns** of the original plaintext. (It does *not* obscure the symbols — every letter survives, merely moved — and it is not necessary for a secure cipher.)
Q6. False. The one-time pad is pure substitution with no transposition and is information-theoretically secure.
Q7. Substitution obscures **the symbols** of the original plaintext. (It is certainly used in modern ciphers — AES's SubBytes step.)
Q8. The OTP uses **substitution, but not transposition**.
Q9. Stream ciphers rely solely on **confusion**. Each symbol is encrypted independently, so one plaintext bit affects exactly one ciphertext bit — that is the absence of diffusion. Security rests entirely on the keystream's relationship to the key being unrecoverable.

Where marks get lost. Q6 catches people who learned "you need both building blocks" as a rule. It is a good heuristic for ciphers with short keys and false as a general claim, and the OTP is the counterexample the question is built around.

11.3 L23 — Mode diagrams

Q1. Draw CBC mode decryption. Must decryption be done sequentially? [L23]

Each ciphertext block goes through `D`, and the result is XORed with the *previous ciphertext block* (the IV for the first):

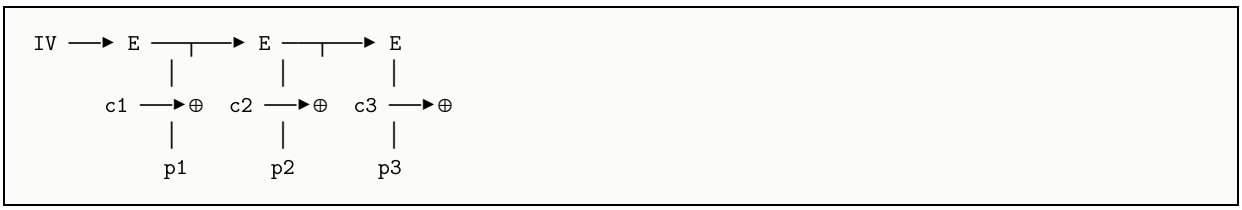


No, decryption need not be sequential. Recovering `p_i` requires `D(c_i)` and `c_{i-1}`, and every ciphertext block is already in hand, so all blocks can be processed in parallel. Encryption is sequential only because it feeds on the previous *ciphertext*, which does not exist until the previous block has been encrypted.

Where marks get lost. Assuming symmetry — "encryption is sequential, so decryption must be too." State the reason: the dependency is on data the decryptor already has.

Q2. Draw OFB mode decryption. Why is pre-processing not an option for the receiver? [L23]

Identical to encryption — the IV is repeatedly encrypted to generate a keystream, which is XORed with the ciphertext:



OFB needs no decryption function at all, only `E`.

Pre-processing is not available to the receiver because the keystream depends on the IV, and the receiver does not learn the IV until the message arrives. The sender chooses the IV and can therefore generate the keystream in advance, then encrypt with a fast XOR the moment the plaintext is ready. The receiver has nothing to work from until the transmission starts.

Where marks get lost. Answering "because decryption depends on the ciphertext." It does not — the keystream depends only on the key and the IV. The IV's arrival time is the whole answer.

11.4 L24 — RSA and Diffie-Hellman

Q1. With `N = 35`, `e = 5`, `d = 5`: encrypt `m = 3`, then decrypt it. [L24]

Encrypt: `c = 3^5 mod 35 = 243 mod 35`. Since `35 × 6 = 210`, `c = 33`.
Decrypt: `m = 33^5 mod 35`. Note `33 ≡ -2 (mod 35)`, so this is `(-2)^5 = -32 ≡ 3 (mod 35)`.
Recovered $\mathbf{m = 3}$ ✓

Where marks get lost. Trying to compute `33^5 = 39,135,393` by hand under time pressure. Reduce to a small negative first — it turns the whole thing into one line.

Q2. Prof. Reckless has each student post an RSA public key `(e, N)`, then encrypts each student's grade under their public key and posts the ciphertext next to their name. Only the student has the private key, so only they can decrypt. What is wrong? [L24]

Textbook RSA is **deterministic** and the grade space is **tiny**. Anyone can take a student's public key — it is published — encrypt every possible grade, and compare each result against the posted ciphertext. The single match reveals that student's grade. No private key, no factoring, no cryptanalysis: a handful of encryptions anyone is able to perform.

The fix is **randomised padding**, so the same grade encrypts to a different ciphertext every time. This is the same requirement as "to be CPA-secure a scheme must be non-deterministic," and the same flaw as ECB mode.

Where marks get lost. Attacking the key-verification step — the professor meeting each student is actually the *sound* part of the scheme, preventing a public-key substitution attack. The flaw is determinism plus a small message space.

Q3. Diffie-Hellman with `g = 2`, `p = 11`, `a = 3`, `b = 5`. Find `A`, `B` and `K`. [L24 · L25]

`A = 2^3 mod 11 = 8`. `B = 2^5 mod 11 = 32 mod 11 = 10`.
Alice: `K = B^a mod 11 = 10^3 mod 11`. Using `10 ≡ −1`: `(−1)^3 = −1 ≡ 10`.
Bob: `K = A^b mod 11 = 8^5 mod 11`. `8^2 = 64 ≡ 9`; `8^4 ≡ 9^2 = 81 ≡ 4`; `8^5 ≡ 4 × 8 = 32 ≡ 10`.
Both sides get $\mathbf{K = 10}$ ✓

Where marks get lost. Computing `8^5 = 32,768` directly. Square repeatedly and reduce at each step, and use `10 ≡ −1` wherever it appears.

11.5 L25 — Hash functions

Q1. Complete: a hash function `h` is collision resistant if it is computationally infeasible to find messages `m` and `m'` such that... [L25]

…`m ≠ m'` **and** `h(m) = h(m')`.

Where marks get lost. Omitting `m ≠ m'`. Without it the statement is trivially satisfiable by taking the same message twice. Both clauses are required.

Q2. A hash accepts messages of any length and produces a 256-bit output. Which is correct? [L25]

Collisions must exist, but finding one should be computationally infeasible. By the pigeonhole principle, infinitely many inputs map into 2^{256} outputs, so collisions are unavoidable. Collision *resistance* is about the difficulty of finding one, never about their absence.

Q3. Match: A. collision resistance, B. second pre-image resistance, C. pre-image resistance. (3.1) Given only a hash `H`, find any message hashing to `H`. (3.2) Given a particular contract, create a different contract with the same hash. (3.3) Create any two different documents with the same hash, neither fixed in advance. [L25]

3.1 → C (pre-image) · **3.2 → B** (second pre-image) · **3.3 → A** (collision).

Where marks get lost. Swapping 3.2 and 3.3. The deciding question is how much is fixed before the attacker starts: nothing at all → collision; one specific message → second pre-image; only a hash value → pre-image.

Q4. `h(m)` is the sum of the decimal digits of `m`, modulo 10, so `h(123) = 6`. Give a different integer with the same hash. Is this collision resistant? [L25]

Any integer whose digits sum to 6 modulo 10: **321** (just a permutation of the digits), or **15**, **6**, **114**, **2004**.
Not collision resistant, for three independent reasons: there are only 10 possible outputs, so collisions are everywhere; digit order is ignored, so every permutation collides; and collisions can be constructed to order by choosing digits that sum correctly. It also fails pre-image and second pre-image resistance, since inverting it is trivial.

11.6 L26 — MACs and signatures

Q1. Over a network with an active attacker: (1) Alice sends `(msg, H)` with `H = h(msg)`; (2) Alice and Bob share a key `k` and she sends `(msg, tag)` with `tag = MAC_k(msg)`. For each, what does Bob get when the check passes? [L26]

Protocol 1: nothing, against this attacker. The hash is public and unkeyed, so Mallory replaces `msg` with `msg'`, recomputes `H' = h(msg')`, and sends the pair. Bob's check passes on a forged message. A bare hash detects only *accidental* corruption.
Protocol 2: integrity and authentication. The message was not modified, and it came from someone holding `k`. Still absent: **confidentiality** (`msg` is in the clear), **non-repudiation** (Bob holds `k` too, so he cannot prove to a third party that Alice sent it), and **freshness** (a replayed pair verifies).

Where marks get lost. Saying protocol 1 gives integrity. It gives integrity only against noise, not against an adversary — and the question specifies an active attacker.

Q2. Toy RSA signatures: `s = 3`, `v = 7`, `N = 33`, `m = 2`. (2.1) Compute `sig = m^s mod N`. (2.2) Verify that `sig^v mod N = m`. (2.3) Does sending `(m, sig)` keep the message confidential?
[L26]

2.1. `sig = 2^3 mod 33 = 8`.
2.2. `8^7 mod 33`. `8^2 = 64 ≡ 31`; `8^4 = 31^2 = 961 ≡ 4` (since `29 × 33 = 957`); then `8^7 = 8^4 · 8^2 · 8 = 4 × 31 × 8`. Take `4 × 31 = 124 ≡ 25`, then `25 × 8 = 200 ≡ 2` (since `6 × 33 = 198`). Result **2 = m**, so the signature verifies ✓
2.3. No. `m` is transmitted in the clear. And even if it were not, `v` is public, so anyone computes `sig^v mod N = m` and recovers it. **Signing is not encrypting** — signatures give integrity, authentication and non-repudiation, never confidentiality.

Where marks get lost. In 2.2, arithmetic. Reduce after every multiplication and follow the suggested decomposition rather than expanding `8^7 = 2,097,152`.

Q3. Encrypt-then-sign. (3.1) For each step, whose key and public or private? (3.2) Mallory replaces `c` with a different ciphertext but keeps Alice's tag. Should Bob's check pass?
[L26]

3.1.

Step	Key used
Alice encrypts the message to get `c`	Bob's **public** key
Alice signs `c` to get `tag`	Alice's **private** key
Bob verifies the signature on `c`	Alice's **public** key
Bob decrypts `c`	Bob's **private** key

3.2. No, the check should fail. The signature was computed over `c`; Bob verifies it against `c'`, and a secure signature scheme will not validate a tag against different data. The tamper is caught *before* Bob decrypts, so he never applies his private key to attacker-chosen input.

Where marks get lost. In 3.1, mixing up whose key. The generating rule: you use **other people's public keys** to send to them and to check them; you use **your own private key** for the two things only you can do — sign and decrypt.

12. One-screen cram sheet

Everything the test covers, in the order you would want it in the last ten minutes before the room opens.

Terminology. Encrypt/decrypt · plaintext · ciphertext · cipher (the algorithm). `c = Enc_k(msg)`, `msg = Dec_k(c)`. Encryption provides **confidentiality only.** .

Cryptosystem. Four parts: encryption algorithm · decryption algorithm · key generation · key management.

Kerckhoffs's principle. Security should depend only on the secrecy of the **keys.** . Publishing the algorithm is fine — keys can be per-user and replaced; algorithms cannot.

Two ways to break. **Exhaustive search.** = try every key; defeated by a big key space. **Cryptanalysis.** = exploit flaws in the cryptosystem; defeated only by better design. Deducing a key from a known pair is cryptanalysis.

Key length. DES: 56-bit key, $2^{56} \approx 7.2 \times 10^{16}$. At 10^{10}/s ≈ 83 days; at 10^{15}/s ≈ 1 minute. **Average = half the key space.** , 2^{55}.

Computational security. Secure against an attacker with fixed resources. Cost to break ≫ value of asset; time to break ≫ lifetime of asset. **Sufficient key space principle:.** necessary, not sufficient — 26! keys didn't save mono-alphabetic substitution.

Four attack models. **Ciphertext-only.** (observes) · **known-plaintext.** (gets pairs it didn't pick) · **CPA.** (picks plaintexts) · **CCA.** (picks ciphertexts). First two passive, last two active, each strictly stronger.

CPA ⇒ non-deterministic. Deterministic = same plaintext + same key → same ciphertext. The attacker encrypts guesses and compares. Breaks ECB and textbook RSA alike.

CCA ⇒ non-malleable. Malleable = modify the ciphertext for a *predictable* change in the plaintext. Non-malleable = any modification yields an illegal ciphertext or an unrelated plaintext.

Classical ciphers. Shift (25 keys) · mono-alphabetic (26! ≈ 2^{88}, falls to frequency analysis) · Vigenère (poly-alphabetic; find the key length, then it's several shift ciphers).

One-time pad. `c_i = p_i ⊕ k_i`, and decryption is the same operation. Symmetric · stream cipher · substitution cipher. **Substitution, no transposition.**.

OTP's four rules. Key is **as long as the message.** · **never reused.** · **kept secret.** · **chosen uniformly at random.** . All four needed; each independently necessary.

Key reuse kills it. `c1 ⊕ c2 = (p1⊕k) ⊕ (p2⊕k) = p1 ⊕ p2` — the key cancels and the attacker never needed it.

Information-theoretic security. Holds against unlimited computation. Not susceptible to cryptanalysis *or* exhaustive search — every key yields a plausible plaintext, so trying them all tells you nothing.

What OTP doesn't give. No integrity. (maximally malleable — flip ciphertext bit i, flip plaintext bit i) · doesn't hide **length.** or metadata · key distribution is impractical.

Substitution vs. transposition. Substitution. hides the **symbols.** . **Transposition.** hides the **patterns.** . Neither is required for security (OTP is substitution-only), but short-key ciphers need both.

Confusion vs. diffusion. Confusion. = obscure the relation between **key.** and ciphertext. **Diffusion.** = spread **plaintext statistics.** across the ciphertext. Mentions the key → confusion; mentions patterns → diffusion.

Stream vs. block. Stream = each symbol separately (Caesar, Vigenère, OTP, RC4); **relies solely on confusion.** , no diffusion. Block = groups together (AES, DES, 3DES).

AES. SubBytes (substitution) · ShiftRows (transposition) · MixColumns (diffusion) · AddRoundKey (mixes in the key). Many rounds of both building blocks.

ECB. Each block encrypted independently. **Deterministic → cannot be CPA secure..** Identical blocks visible — the encrypted penguin.

CBC. XOR with previous ciphertext, then encrypt. Random IV each time, sent in clear. Chaining adds diffusion. **Encrypt sequentially, decrypt in parallel..** CPA secure.

OFB. Repeatedly encrypt the IV to make a keystream, XOR with the message — block cipher as stream cipher. Sequential. **Sender can pre-process; receiver can't.** (no IV until the message arrives).

CTR. Encrypt ctr, ctr+1, … for a keystream. Counter sent in the clear, chosen uniformly at random. **Encrypt and decrypt both parallel; random access..** CPA secure.

IV/counter reuse. Same keystream twice → `c1 ⊕ c2 = p1 ⊕ p2`. The OTP key-reuse break, inside a modern mode.

Public key. `Ke` public, `Kd` private. Keys for n parties: symmetric `n(n-1)/2`, public key `2n`. Public key is slow; symmetric is fast.

RSA KEYGEN. Primes `p, q` → `N = pq` · `φ(N) = (p-1)(q-1)` · pick `e` with `gcd(e, φ(N)) = 1` · find `d` with `ed ≡ 1 mod φ(N)`. Public `(N, e)`, private `(N, d)`.

RSA ENC/DEC. `c = m^e mod N` · `m = c^d mod N`. **IF THE ATTACKER FACTORS N THEY WIN.** — `p, q` → `φ(N)` → `d`.

MODULAR ARITHMETIC TRICK. Reduce to a small **NEGATIVE.** before exponentiating: `33^5 mod 35` → `(-2)^5 = -32 ≡ 3`. Square repeatedly and reduce at every step.

ONE-WAY / TRAPDOOR. One-way = easy forward, hard to invert (multiplying primes; modular exponentiation). **TRAPDOOR.** = a secret makes inversion easy — `d` for RSA. DH has *no* trapdoor.

TEXTBOOK RSA LEAKS. Deterministic + small message space = encrypt every candidate under the public key and compare. Fix: randomised padding.

HYBRID ENCRYPTION. `c1 = PK-Enc_Ke(k)`, `c2 = S-Enc_k(msg)`. Public key for the short session key, symmetric for the bulk. This is what TLS does.

DIFFIE-HELLMAN. Public `g, p` (`g` a primitive root). `A = g^a`, `B = g^b`; both compute `K = g^(ab) mod p`. Secure against a **PASSIVE.** attacker — needs the discrete log.

MITM ON DH. Mallory runs two exchanges and relays. Root cause: **NO AUTHENTICATION.** . DH gives a shared secret with *somebody* and proves nothing about who.

REPLAY ATTACK. Record a valid message, resend it later. The attacker never decrypts anything. **ENCRYPTION ALONE ESTABLISHES NEITHER IDENTITY NOR FRESHNESS..**

NONCE. A number used only once. **RANDOM.** → freshness · **SEQUENCE NUMBER.** → uniqueness · **TIMESTAMP.** → timeliness. IVs in CBC/OFB/CTR are nonces.

HASH FUNCTION. Arbitrary-length input → fixed-length output. **COLLISIONS MUST EXIST.** (pigeonhole); hashes are not invertible. A hash reveals modification but cannot guarantee its absence.

THREE HASH PROPERTIES. Decreasing strength: **COLLISION.** (find any two, nothing fixed) → **SECOND PRE-IMAGE.** (one message fixed) → **PRE-IMAGE.** (only the hash given). Match by what's fixed in advance.

BIRTHDAY PARADOX. 50 people → 97% chance of a shared birthday. An n-bit hash needs only $\sim 2^{n/2}$ work for a collision. MD5 and SHA-1 fell on collisions first.

BARE HASH IS USELESS VS. AN ATTACKER. `(msg, h(msg))`: Mallory swaps the message and recomputes the hash. No secret anywhere → catches accidents only.

MAC. `tag = MAC_k(msg)` with a shared key. Gives **INTEGRITY + AUTHENTICATION.** . **NO.** confidentiality, **NO.** non-repudiation (Bob has `k` too), **NO.** freshness.

SIGNATURE. Sign with **YOUR PRIVATE.** key, verify with **YOUR PUBLIC.** key. Adds **NON-REPUDIATION.** ; anyone can verify. **SIGNING IS NOT ENCRYPTING.** — `sig^v mod N = m` recovers the message.

ENCRYPT-THEN-SIGN. Encrypt → **BOB'S PUBLIC.** · Sign → **ALICE'S PRIVATE.** · Verify → **ALICE'S PUBLIC.** · Decrypt → **BOB'S PRIVATE.** . Tamper with `c` and verification fails *before* decryption.

KEY RULE. Other people's **PUBLIC.** keys to send to them and check them. Your own **PRIVATE.** key for the two things only you can do: sign and decrypt.

WHY NOT SIGN-THEN-ENCRYPT. Bob decrypts, gets `(m, sig)` bound to nothing, and re-encrypts it to Carol — who believes Alice wrote to her. The signature authenticates content, not recipient.

Built October 30, 2026 from the L21–L26 lecture decks, the in-class practice sheets and the published topic list for Test 3 of COMP 435 (Fall 2026, UNC). Personal study material.
© 2026 Deven Jadhav. Not affiliated with or endorsed by UNC-Chapel Hill; course material © its respective owners. Willful large-scale copyright infringement may be a federal crime (17 U.S.C. §506).