Cryptography Fundamentals

A Complete 12-Chapter Security Course

Topics covered:

Goals & Kerckhoffs's Principle · Classical ciphers & frequency analysis

The Enigma machine & how it was broken · Symmetric ciphers & AES · Modes of operation

Hash functions & MACs · RSA, Diffie-Hellman & elliptic curves · Digital signatures

Key management in practice · Post-quantum cryptography

Exercises: 36 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · from Caesar's cipher to lattice-based cryptography

Table of Contents

CHAPTER 1 OF 12

What Is Cryptography? Goals, Terminology & Kerckhoffs's Principle

Cryptography Fundamentals

Chapter 1 · What Is Cryptography? Goals, Terminology & Kerckhoffs's Principle

The **HTTPS/TLS Fundamentals** course introduced enough cryptography to explain how a browser padlock actually works — symmetric/asymmetric crypto, hashing, and key exchange, covered just deeply enough to get through a TLS handshake (`https1-2`, `https1-3`). This course goes underneath that: the primitives themselves, why they're built the way they are, and — because it's genuinely a great story with real technical substance — a proper look at classical cryptography's most famous machine, the Enigma, and how it was broken.

The Four Goals of Cryptography

"Cryptography" is often shorthand for just one thing — keeping messages secret. That's only a quarter of the picture. Every cryptographic system exists to provide some combination of four distinct guarantees:

GOAL	QUESTION IT ANSWERS	TYPICAL TOOL
Confidentiality	Can anyone but the intended recipient read this?	Encryption (Ch.5, Ch.9)
Integrity	Was this message altered in transit?	Hash functions, MACs (Ch.7, Ch.8)
Authenticity	Did it really come from who it claims to?	Digital signatures, MACs (Ch.8, Ch.10)
Non-repudiation	Can the sender later deny sending it?	Digital signatures specifically (Ch.10)

ENCRYPTION ALONE ONLY BUYS YOU ONE OF THESE FOUR

Encrypting a message keeps it confidential — but an attacker who can't read a message can often still *tamper* with it, and a recipient decrypting it has no guarantee it actually came from the claimed sender. Confusing "encrypted" with "secure" is one of the most common real-world cryptography mistakes; a genuinely secure system almost always needs confidentiality, integrity, and authenticity working together, not encryption standing in for all three.

The Vocabulary You'll Need

- **Plaintext** — the original, readable message before any transformation.

- **Ciphertext** — the transformed, unreadable-without-the-key output.
- **Cipher** — the algorithm that transforms plaintext into ciphertext and back.
- **Key** — the secret input that controls exactly how a cipher transforms the data; the same cipher with a different key produces completely different ciphertext.
- **Encrypt / decrypt** — apply the cipher forward (plaintext → ciphertext) or in reverse (ciphertext → plaintext).
- **Cryptography** — the practice of designing these systems; **cryptanalysis** — the practice of breaking them; **cryptology** — the umbrella term covering both.

That last distinction matters more than it looks — Chapter 4's entire subject is cryptanalysis, not cryptography: how a well-designed cipher's real-world implementation and operation, not its underlying mathematics, ended up being its weakness.

Kerckhoffs's Principle — The Algorithm Isn't the Secret

Formulated by Auguste Kerckhoffs in 1883, the principle states: **a cryptographic system should remain secure even if everything about it — except the key — is public knowledge.** Only the key is secret; the algorithm itself can be published, studied, and attacked by the entire world without that alone breaking the system.

This runs directly against instinct. Surely keeping the algorithm itself secret adds a layer of protection? In practice, the opposite holds:

- A secret algorithm has never been reviewed by anyone outside the small group that built it — flaws go unnoticed until an attacker finds them first.
- A public algorithm like AES (Ch.5) has been attacked for decades by the world's best cryptanalysts and has held up — that scrutiny *is* the evidence of its strength.
- Relying on secrecy of design rather than secrecy of key is called **security through obscurity**, and it's considered a red flag rather than a defence in modern cryptographic engineering.

> **THIS IS WHY CHAPTER 3 MATTERS FOR MORE THAN HISTORY**
>
> The Enigma machine's design was *not* secret — the Allies had working replica machines. What stayed secret, and what actually mattered, was the daily key settings (rotor order, starting positions, plugboard wiring). Kerckhoffs's Principle, formulated over 50 years before Enigma was ever built, predicts exactly this outcome: the machine's fall came from key-management and operational failures (Ch.4), not from anyone reverse-engineering its mechanism.

Symmetric vs. Asymmetric — A First Look

Two fundamentally different approaches to using a key, both covered in real depth later in this course:

	SYMMETRIC	ASYMMETRIC (PUBLIC-KEY)
Keys involved	One shared key, used for both encrypt and decrypt	A key pair — public key encrypts, private key decrypts (or vice versa for signing)
Speed	Fast — used for the bulk of real data	Much slower — used sparingly, often just to exchange a symmetric key
The hard problem	Getting the shared key to both parties securely	No shared secret ever needs to travel at all
Covered in depth	Chapter 5 (AES), Chapter 6 (modes)	Chapter 9 (RSA, Diffie-Hellman, ECC)

Every classical cipher this course covers next — including Enigma — is symmetric: the same setting that encrypts a message also decrypts it. Asymmetric cryptography is a 20th-century invention (Chapter 9), and understanding why it took so long to invent is itself a genuinely interesting question, revisited once Chapter 9 lays out the "hard problem" symmetric crypto never solved.

Where This Course Is Headed

CHAPTER	TOPIC
2	Classical Ciphers — Substitution, Transposition & Frequency Analysis
3	The Enigma Machine — Mechanism & Daily Operation
4	Breaking Enigma — Cribs, the Bombe & the Fatal Flaw
5	Symmetric-Key Cryptography — Block & Stream Ciphers, AES
6	Modes of Operation — ECB, CBC, CTR/GCM
7	Hash Functions
8	Message Authentication — MACs, HMAC & AEAD
9	Public-Key Cryptography — RSA, Diffie-Hellman & Elliptic Curves
10	Digital Signatures & the Trust Chain
11	Key Management in Practice
12	Capstone: Cryptography in the Real World

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: what actually breaks a cryptographic system in practice? Rarely is it the underlying mathematics — it's almost always key management, implementation choices, or operational

discipline. That thread runs from Enigma's daily settings (Ch.3–4) all the way to Chapter 11's real-world key management, and it's worth watching for throughout.

Hands-On Exercises

EXERCISE 1

A colleague says "we encrypted the data, so it's secure." Using this chapter's four goals, explain what encryption alone does and doesn't guarantee, and give a concrete scenario where encrypted data is still successfully attacked.

View solution

EXERCISE 2

A company builds its own secret, unpublished encryption algorithm and refuses to disclose how it works, arguing that keeping the algorithm secret makes it stronger. Explain, using Kerckhoffs's Principle, why this reasoning is considered a red flag rather than good practice.

View solution

EXERCISE 3

Classify each of the following as primarily a confidentiality, integrity, authenticity, or non-repudiation problem: (a) an attacker reads a private email in transit, (b) a signed software update is altered after signing without detection, (c) an attacker forges a message that appears to come from your bank, (d) a sender later claims they never sent a message they actually did send.

View solution

CHAPTER 1 QUICK REFERENCE

- **Four goals:** confidentiality, integrity, authenticity, non-repudiation — encryption alone only covers the first
- **Vocabulary:** plaintext/ciphertext/cipher/key; cryptography (build) vs. cryptanalysis (break) vs. cryptology (both)
- **Kerckhoffs's Principle:** security must come from the secrecy of the key, not the secrecy of the algorithm — "security through obscurity" is a red flag
- **Symmetric** = one shared key, fast, hard problem is key distribution; **asymmetric** = key pair, slower, no shared secret needs to travel
- This course's throughline: real-world breaks are almost always about key management and operational discipline, not the underlying math

- **Next chapter:** Classical Ciphers — substitution and transposition ciphers, and why frequency analysis eventually defeats all of them

CHAPTER 2 OF 12

Classical Ciphers — Substitution, Transposition & Frequency Analysis

Cryptography Fundamentals

Chapter 2 · Classical Ciphers — Substitution, Transposition & Frequency Analysis

Chapter 1 introduced the vocabulary — plaintext, ciphertext, cipher, key — in the abstract. This chapter puts real, historical ciphers behind those words, and builds toward a genuinely important point: every classical cipher, no matter how clever, eventually falls to the same weapon — **statistics**. Understanding exactly why sets up Chapter 3's Enigma machine as something more than a historical curiosity — it was a serious, partially successful attempt to defeat this exact weakness mechanically.

The Caesar Cipher — Simplest Substitution

Named for Julius Caesar, who reportedly used it for military messages: shift every letter of the alphabet forward by a fixed amount. A shift of 3 turns A into D, B into E, and so on, wrapping around at the end.

```
Plaintext:  HELLO WORLD
Shift:      +3
Ciphertext: KHOOR ZRUOG
```

The **key** here is just the shift amount — a single number between 1 and 25. ROT13 (shift 13) is the same idea, still occasionally used today for puzzle spoilers, precisely *because* it offers no real security — its own reversal is trivial to spot.

A 25-KEY KEYSPACE IS NOT A REAL DEFENCE

With only 25 possible shifts, an attacker can simply try every single one — a "brute-force" attack that takes seconds by hand and is instantaneous by computer. The Caesar cipher's weakness isn't subtle; it's the entire keyspace.

Substitution Ciphers, Generalized

A **monoalphabetic substitution cipher** generalizes Caesar's idea: instead of a fixed shift, use any arbitrary one-to-one mapping from plaintext letters to ciphertext letters. There are **26! (about 4×10^{26})** possible mappings — brute force is completely infeasible here, unlike Caesar's 25 keys.

And yet monoalphabetic substitution is still comprehensively broken — not through brute force, but through a completely different kind of attack.

Frequency Analysis — Breaking Substitution Ciphers

Every natural language has a distinctive statistical fingerprint. In English text, **E** is by far the most common letter (roughly 12.7% of all letters), followed by T, A, O, I, N — often remembered by the mnemonic **ETAOIN SHRDLU**. A monoalphabetic substitution cipher maps each plaintext letter to *always* the same ciphertext letter — which means it preserves this frequency fingerprint perfectly, just relabeled.

Given enough ciphertext, an analyst counts how often each ciphertext symbol appears, matches the most frequent symbol to E, the next most frequent to T, and so on — then refines using common short words ("THE", "AND"), double letters, and known digraphs/trigraphs. No matter how large the keyspace, the underlying language's statistics leak straight through the substitution.

ENGLISH LETTER	APPROX. FREQUENCY
E	12.7%
T	9.1%
A	8.2%
O	7.5%
I / N	~7%
Z / Q	<0.1%

Transposition Ciphers

A completely different family: instead of *replacing* letters, **rearrange** their order. The rail fence cipher writes the message in a zigzag across several "rails" and reads them off row by row; columnar transposition writes the message into a grid and reads the columns out in a key-defined order.

```
# Rail fence, 3 rails:
Plaintext:  W E A R E D I S C O V E R E D
Rail 1:     W . . . E . . . C . . . R . . . D
Rail 2:     . E . R . D . S . O . E . E . D .
Rail 3:     . . A . . . I . . . V . . . . . .
Ciphertext: WECRDIOVEEREDAI...
```

Because a transposition cipher never changes *which* letters appear — only their order — the letter frequencies of the ciphertext are **identical** to the plaintext's, which defeats a naive frequency-count attack outright. But it

introduces a different weakness: the ciphertext is a pure anagram of the plaintext, and short, structured messages are often recoverable by testing plausible word fragments and grid dimensions — a real weakness, just a different one than substitution's.

The Vigenère Cipher — "Le Chiffre Indéchiffrable"

For roughly 300 years, the Vigenère cipher (popularized in the 16th century) was considered unbreakable — French for "the indecipherable cipher." It's a **polyalphabetic** substitution cipher: instead of one fixed shift (Caesar) or one fixed mapping (monoalphabetic substitution), it cycles through *multiple* different shifts, determined by a repeating keyword.

```
Plaintext:   A T T A C K A T D A W N
Keyword:     L E M O N L E M O N L E    # repeated to match length
Shift by:    L=11 E=4 M=12 O=14 N=13 L=11 ...
Ciphertext:  L X F O P V E F R N H R
```

Each letter of the keyword selects a different Caesar shift for the letter beneath it — "ATTACK" gets encrypted with a *different* effective cipher at nearly every position, so the same plaintext letter (the two T's in ATTACK) can produce two completely different ciphertext letters. That single property is exactly what defeats a direct frequency count: the ciphertext's letter frequencies are smeared across multiple shifted alphabets rather than reflecting English's fingerprint directly.

Breaking Vigenère — the Kasiski Examination

Vigenère's real weakness is that the keyword **repeats**. If a keyword is 5 letters long, then every 5th letter of ciphertext was shifted by the same keyword letter — meaning the ciphertext is actually 5 separate, interleaved Caesar ciphers, each individually breakable by ordinary frequency analysis once you know that 5 is the key length.

Friedrich Kasiski published the practical break in 1863: look for **repeated sequences** of 3+ letters in the ciphertext. If the same plaintext fragment happens to align with the same part of the repeating keyword twice, it produces the same ciphertext fragment twice — and the **distance** between those two repeats must be a multiple of the key length. Find several such repeated fragments, take the greatest common divisor of their distances, and the key length falls out.

> **ONCE THE KEY LENGTH IS KNOWN, VIGENÈRE COLLAPSES BACK TO CHAPTER 1'S PROBLEM**
>
> Split the ciphertext into that many interleaved groups (every 5th letter together, for a length-5 key), and each group is now a simple single-shift Caesar cipher — solvable by the exact frequency analysis covered earlier in this chapter. "Unbreakable" polyalphabetic substitution turns out to reduce to a handful of the easiest cipher in this chapter, once the repeating structure is found.

Why All Classical Ciphers Eventually Fall

Every cipher in this chapter shares one structural weakness: each is a **fixed, repeating** transformation — a single shift, a single substitution alphabet, or a substitution alphabet that repeats every few letters. Fixed and repeating transformations always leave some statistical fingerprint of the underlying language intact, and given enough ciphertext, that fingerprint is always recoverable.

BRIDGING TO CHAPTER 3

Enigma is, underneath everything, *also* a polyalphabetic substitution cipher — mechanically, it's not a different category of cipher from Vigenère at all. Its genuine innovation was generating a completely new substitution alphabet on **every single keystroke**, via rotors that physically advance after each letter, making its "keyword" length effectively many millions of characters before any repetition occurs. That's precisely why Chapter 3 matters technically and not just historically — Enigma was a serious mechanical attempt to eliminate the repeating structure that broke Vigenère.

NONE OF THIS BELONGS ANYWHERE NEAR REAL SECURITY TODAY

Every cipher in this chapter is broken by hand, by a single person, often within minutes for short messages. They're taught here for the ideas they establish — substitution, transposition, keyspace size, frequency analysis — not as anything to actually rely on. Chapter 5 onward covers the modern ciphers (AES and friends) that real systems use instead.

Hands-On Exercises

EXERCISE 1

Decrypt the Caesar-ciphertext `WKLV LV IXQ`. You don't know the shift in advance — explain your approach and give the shift amount and the plaintext.

View solution

EXERCISE 2

You have a 300-letter monoalphabetic substitution ciphertext. The letter `Q` appears 38 times (by far the most), and `F` appears 27 times (second most). Using this chapter's frequency table, what would you guess Q and F map to first, and what would you do next to confirm or refute the guess?

View solution

EXERCISE 3

While Kasiski-examining a Vigenère ciphertext, you find the trigram `QVX` repeated at two positions 24 letters apart, and the trigram `MPZ` repeated at two positions 18 letters apart. What key lengths does this evidence support, and why?

View solution

CHAPTER 2 QUICK REFERENCE

- **Caesar cipher** — fixed shift, only 25 keys, broken by brute force alone
- **Monoalphabetic substitution** — 26! keys (brute force infeasible), broken by frequency analysis (ETAOIN SHRDLU) instead
- **Transposition** — rearranges letters rather than replacing them; preserves letter frequency but is a pure anagram, breakable by structure
- **Vigenère** — polyalphabetic, cycles through shifts via a repeating keyword; broken by the **Kasiski examination** (repeated ciphertext fragments reveal the key length), then reduces to several Caesar ciphers
- Shared weakness across every cipher here: a **fixed, repeating** transformation always leaks the underlying language's statistics eventually
- **Next chapter:** The Enigma Machine — a polyalphabetic cipher that generates a new substitution alphabet on every keystroke, mechanically attacking exactly this weakness

CHAPTER 3 OF 12

The Enigma Machine — Mechanism & Daily Operation

Cryptography Fundamentals

Chapter 3 · The Enigma Machine — Mechanism & Daily Operation

Chapter 2 closed on a bridge: Enigma is, structurally, a polyalphabetic substitution cipher — the same family as Vigenère — but instead of cycling through a short repeating keyword, it mechanically generates a **new substitution alphabet on every single keystroke**. This chapter is about exactly how it did that: the physical components, the electrical path a signal actually takes, and the daily settings that — per Kerckhoffs's Principle (Chapter 1) — were the only real secret in the whole system.

The Physical Machine, Component by Component

A military Enigma (the German Wehrmacht/Luftwaffe 3-rotor variant this chapter focuses on) is built from five functional parts, wired together in series:

COMPONENT	ROLE
Keyboard	26 keys, A-Z; pressing one starts an electrical signal
Plugboard (Steckerbrett)	Swaps pairs of letters before and after the rotor stack
Rotors (3, chosen from 5 or later 8)	Each is an internally-wired substitution alphabet that also physically rotates
Reflector (Umkehrwalze)	Sends the signal back through the rotors a second time, in reverse
Lampboard	26 lamps, A-Z; whichever one lights up is the ciphertext letter

The Electrical Pathway — Tracing a Keypress

Pressing a single key completes a circuit that travels through the entire machine in one direction, then back through most of it in the other direction, before finally lighting a lamp. In order:

```
1. Key pressed          -> signal enters the plugboard
2. Plugboard            -> swapped if this letter is one of the ~10 wired pairs
3. Rotor 1 (rightmost)  -> substituted per its internal wiring
4. Rotor 2 (middle)     -> substituted per its internal wiring
5. Rotor 3 (leftmost)   -> substituted per its internal wiring
6. Reflector            -> sends the signal back the OTHER way, through a fixed pairing
7. Rotor 3 (leftmost)   -> substituted again, in reverse
8. Rotor 2 (middle)     -> substituted again, in reverse
9. Rotor 1 (rightmost)  -> substituted again, in reverse
10. Plugboard           -> swapped again if applicable
11. Lampboard           -> a single lamp lights — the ciphertext letter
```

Every keypress passes through the rotor stack **twice** — once outward toward the reflector, once back — which is exactly what makes Enigma *reciprocal*: with identical settings, encrypting a letter and decrypting a letter are literally the same operation run through the same circuit. That's why the same machine, same daily settings, both encrypted and decrypted — a genuinely elegant piece of engineering, and (as Chapter 4 shows) also the source of its most damaging flaw.

The Rotors — Wiring & Stepping

Each rotor is a disc with 26 electrical contacts on each face, internally wired so that entering on contact A might exit on contact F, entering on B might exit on U, and so on — a fixed, physical monoalphabetic substitution baked into the metal (echoing Chapter 2's substitution ciphers, just implemented as wiring rather than a lookup table).

What makes it more than a fixed substitution is that **the rightmost rotor physically advances one position after every keypress**, like the units wheel on an odometer — so the substitution it performs on the very next letter is different again. When the rightmost rotor completes a full revolution, it "kicks" the middle rotor forward one position too (with a quirky, historically fiddly detail called **double-stepping**, where the middle rotor can occasionally step twice in a row); the leftmost rotor advances only rarely. The practical effect: the combined substitution performed by all three rotors together doesn't meaningfully repeat until many thousands of letters have been typed — nothing like Vigenère's short, repeating keyword from Chapter 2.

> **THIS IS THE DIRECT MECHANICAL ANSWER TO CHAPTER 2'S WEAKNESS**
>
> A Vigenère cipher's polyalphabetic protection breaks down because its "alphabet-changing" mechanism (the keyword) is short and repeats predictably — exactly what the Kasiski examination exploited. Enigma's rotor stepping is also polyalphabetic substitution, but its effective "keyword" is enormously long before it cycles — which is precisely why naive frequency analysis, straight out of Chapter 2, does not work against Enigma ciphertext at all.

The Reflector (Umkehrwalze) — Symmetric But Flawed

The reflector is wired as 13 fixed pairs covering all 26 letters (A always reflects to, say, Y and vice versa), and critically, it has one structural property: **no letter can ever reflect back to itself**. That single design choice — which is what makes the machine reciprocal at all — turns out to be Enigma's single most exploitable weakness, covered in full in Chapter 4.

The Plugboard (Steckerbrett) — Extra Scrambling

Before the German military added it, Enigma's civilian/commercial version relied on the rotors and reflector alone. The plugboard sits at both the very start and very end of the electrical path (steps 2 and 10 above) and swaps pairs of letters via physical cables — typically **10 pairs** swapped, out of 26 letters, on a standard military setup. It doesn't change the fundamental structure of the cipher at all — it's still the same rotor-and-reflector substitution underneath — but it multiplies the number of possible daily configurations enormously, which mattered a great deal for the keyspace calculation below.

The Daily Key Settings — What Was Actually Secret

Per Chapter 1's Kerckhoffs's Principle, the machine's design was not secret — the Allies had working Enigma machines. What was secret, distributed via monthly codebooks to every operator, was the **daily key**:

- **Rotor choice & order (Walzenlage)** — which 3 of the available rotors were installed, and in what left-to-right order.
- **Ring settings (Ringstellung)** — an offset between each rotor's internal wiring and its outer letter ring.
- **Initial rotor positions (Grundstellung)** — the starting letter each rotor was turned to before the message was typed.
- **Plugboard pairs (Steckerverbindungen)** — which ~10 letter pairs were physically cabled together that day.

Every operator sending or receiving a message that day used the identical settings — get any one piece wrong, and the decrypted output is unreadable garbage from the very first letter.

Enigma's Keyspace — Astronomically Large...

Combining rotor choice and order, ring settings, starting positions, and plugboard pairings, the total number of possible daily configurations for a 3-rotor military Enigma has been estimated at roughly $\mathbf{10^{114}}$ — a number so large that brute-forcing every possible setting, even with modern computing power, remains completely infeasible. On paper, this looks like an unbreakable system.

> **THAT KEYSPACE CALCULATION ASSUMES SOMETHING CHAPTER 4 SHOWS DIDN'T HOLD**

10^{114} possible settings assumes an attacker has to search the *entire* space blindly, with no shortcuts and no help from the system's own structure or the humans operating it. Chapter 4 covers exactly how that assumption failed in practice: the reflector's "no letter maps to itself" property (this chapter, above) turned out to eliminate huge swaths of that search space instantly, and real operators made real, exploitable mistakes in how they used the machine day to day.

Hands-On Exercises

EXERCISE 1

Describe, in order, the full electrical path a single keypress takes through an Enigma machine, from the key being pressed to a lamp lighting up. Be specific about how many times the signal passes through the rotor stack, and in which directions.

View solution

EXERCISE 2

The reflector guarantees that a letter can never encrypt to itself. Without looking ahead to Chapter 4, reason about why this specific property — on its own — could plausibly give a codebreaker useful information about a message, even without knowing the daily key.

View solution

EXERCISE 3

A 3-rotor military Enigma selects 3 rotors, in a specific left-to-right order, from a set of 5 available rotors. How many distinct rotor choice-and-order combinations are possible? Show your work.

View solution

CHAPTER 3 QUICK REFERENCE

- **Signal path:** keyboard → plugboard → 3 rotors (forward) → reflector → 3 rotors (reverse) → plugboard → lampboard
- **Rotors** are fixed internal substitutions that also physically step after each keypress — the source of the "new alphabet every keystroke" property
- **Reflector** makes the machine reciprocal (same settings encrypt and decrypt) but guarantees no letter ever maps to itself — Chapter 4's key weakness
- **Plugboard** swaps ~10 letter pairs at the start and end of the path, multiplying the keyspace without changing the underlying structure

- **Daily key** = rotor choice/order + ring settings + starting positions + plugboard pairs — the only real secret, per Kerckhoffs's Principle (Ch.1)
- Full keyspace ≈ 10^{114} — looks unbreakable by brute force alone
- **Next chapter:** Breaking Enigma — how the reflector's flaw, cribs, and operational mistakes made that astronomical keyspace irrelevant in practice

CHAPTER 4 OF 12

Breaking Enigma — Cribs, the Bombe & the Fatal Flaw

Cryptography Fundamentals

Chapter 4 · Breaking Enigma — Cribs, the Bombe & the Fatal Flaw

Chapter 3 ended with a warning: the ~10^{114} keyspace calculation assumed an attacker searching blindly, with no shortcuts and no help from the machine's own structure or the humans operating it. Neither assumption held. This chapter covers how Enigma was actually broken — first by Polish mathematicians before the Second World War even started, then at a much larger, sustained scale at Britain's Bletchley Park — and why the real story is less "a genius found one clever trick" and more "a systematic, industrial-scale process exploited several genuine weaknesses at once."

The Reflector's Fatal Flaw, Formalized

Chapter 3 introduced the reflector's guarantee that no letter can ever encrypt to itself, and asked you to reason about why that might help a codebreaker even without knowing the daily key. Here's the formal technique built on exactly that property, known as **crib-dragging**.

A **crib** is a guessed or known fragment of plaintext believed to appear somewhere in a message — more on where these guesses actually came from in the next section. To test a crib against a stretch of ciphertext:

1. Slide the crib along the ciphertext, testing one starting position (offset) at a time.
2. At each offset, compare the crib letter-by-letter against the ciphertext letters directly beneath it.
3. If *any* letter of the crib matches the ciphertext letter in the same position, that offset is **impossible** — Enigma could never have produced a letter mapping to itself — and can be discarded immediately, with zero knowledge of the actual daily key required.

This alone doesn't reveal the key — but it dramatically narrows down which positions in the ciphertext a guessed crib could actually start at, which is the essential first step every other technique in this chapter builds on.

Cribs — Where the Guesses Came From

Crib-dragging is only useful if you have a good guess for what plaintext to test. In practice, German military communication was remarkably predictable:

- Routine weather reports opened with the same standard phrase, *Wettervorhersage* ("weather forecast"), nearly every single day.
- Uneventful daily status reports commonly used the stock phrase *Keine besonderen Ereignisse* ("nothing to report").
- Formal messages followed rigid formatting and titles — dates, ranks, and unit names appearing in predictable positions.
- Some phrases were considered so security-sensitive their use was actually restricted — Hitler's name, for instance, was avoided in encrypted traffic specifically because its predictable spelling made it a dangerously strong crib.

> **THE LESSON ISN'T REALLY ABOUT ENIGMA SPECIFICALLY**
>
> Predictable message structure undermining an otherwise-strong cipher is a pattern that reappears constantly in real cryptography, right up to the present — Chapter 6's discussion of why ECB mode leaks patterns in structured data is a direct modern descendant of this exact idea.

The Polish Contribution — Rejewski & the Bomba

The first real break came years before the war, from the Polish Cipher Bureau. Mathematician **Marian Rejewski** exploited a specific procedural weakness in how Germany used Enigma in the early-to-mid 1930s: each operator's message began with the day's starting position, *enciphered twice in a row* as a check against transmission errors. That doubled repetition, combined with rigorous permutation-group mathematics, let Rejewski deduce the internal rotor wiring and build the **Bomba kryptologiczna** — an electromechanical device that automated testing candidate rotor settings against this specific weakness.

In July 1939, with war looming, Poland shared its Enigma-breaking methods and even reconstructed machines with British and French intelligence — a critical head start that the much larger Bletchley Park effort built directly on top of.

Turing, Welchman & the British Bombe

Once Germany fixed the doubled-key weakness the Poles had exploited, the Polish method stopped working. At Bletchley Park, **Alan Turing** designed a new electromechanical device — also called the **Bombe**, distinct from and considerably more powerful than the Polish Bomba — built around crib-dragging rather than the doubled-key flaw. **Gordon Welchman** then added a critical improvement, the *diagonal board*, which dramatically increased the number of plugboard-related contradictions the machine could detect, cutting the effective search time enormously.

How the Bombe Actually Worked, Conceptually

Starting from a crib believed to align with a stretch of ciphertext, codebreakers built a **menu** — a diagram linking each ciphertext letter to its corresponding crib letter at that position, forming a chain (and sometimes closed loops, which were especially valuable). The Bombe then electrically simulated many linked Enigma rotor-sets simultaneously, each wired according to the menu, and tested candidate rotor starting positions one at a time — extremely fast, since it was doing this electromechanically rather than by hand.

For each candidate position, the machine checked whether the wiring was **self-consistent**. A contradiction — most directly, a chain implying some letter would have to map to itself, violating the reflector's own guarantee from Chapter 3 — meant that starting position was impossible and could be discarded instantly, without ever needing to determine the plugboard or ring settings for it at all. Only candidate positions surviving this contradiction check were passed on for further, more detailed testing. The overwhelming majority of the $\sim 10^{114}$-sized keyspace was never touched directly; entire families of settings were ruled out the moment a single contradiction appeared.

Human & Operational Mistakes That Helped

Beyond the machine's own structural properties, operators themselves handed codebreakers real, repeated advantages:

- **Lazy or predictable settings** — some operators chose starting positions or plugboard pairs following obvious patterns (keyboard sequences, initials, or repeatedly reusing a personal favorite), rather than genuinely random choices, nicknamed "cillies" by Bletchley staff.
- **Message-key reuse** — sending the same or a very similar message twice under different daily keys (for instance, retransmitting a message that failed to arrive) gave codebreakers two independently encrypted versions of the same plaintext, dramatically easier to attack together than either alone.
- **"Kisses"** — cases where the same message content was transmitted both via Enigma and via a separately-broken, lower-grade cipher, letting codebreakers use the already-known plaintext as a perfect crib against the Enigma version.
- **Rigid message formatting** — the same predictability the cribs section above already covered, but worth restating here as a genuinely operational (not mathematical) failure: discipline about varying message structure was simply never enforced.

> **NOT A SINGLE "AHA" MOMENT**
>
> Breaking Enigma is often told as a single dramatic insight, usually credited entirely to Turing. The real story is a sustained, years-long industrial effort combining Polish mathematics, British engineering, thousands of Bletchley Park staff processing traffic daily, and a steady stream of real operational sloppiness — several genuinely independent weaknesses, exploited together, none of which alone would have been sufficient.

What This Teaches About Cryptography Generally

Enigma's underlying cipher mathematics were genuinely sound for their era — the ~10^{114} keyspace from Chapter 3 was real. Every practical avenue of attack covered in this chapter — the reflector's structural guarantee, predictable message content, procedural shortcuts, and operator carelessness — came from **how the system was used**, not from the core cipher design being fundamentally broken.

> **THIS CHAPTER'S REAL THROUGHLINE**
>
> This is the same lesson Chapter 1 named as this entire course's throughline: real-world cryptographic failures are almost always about key management and operational discipline, not the underlying math. Chapter 11 (Key Management in Practice) picks this exact thread back up in a fully modern context — password reuse, predictable key generation, and poor rotation practices are the direct 21st-century descendants of "cillies" and reused message keys.

Hands-On Exercises

EXERCISE 1

Given the 12-letter ciphertext `RWEFTBERXTQY` and the crib `WETTER` (6 letters), test starting offsets 1, 4, and 7 (1-indexed) using crib-dragging. Which offset(s), if any, are eliminated by the "no letter maps to itself" rule, and which survive?

View solution

EXERCISE 2

Explain why a "kiss" — the same message content sent both via Enigma and via a separately broken, weaker cipher — could compromise Enigma's security for that message, even though the daily key changed every day and the weaker cipher's own key had nothing to do with Enigma's.

View solution

EXERCISE 3

Explain, conceptually, why testing rotor settings for logical contradictions (the Bombe's approach) was so much faster than brute-forcing all ~10^{114} possible daily keys directly, one at a time.

View solution

CHAPTER 4 QUICK REFERENCE

- **Crib-dragging** — slide a guessed plaintext fragment along the ciphertext; any position producing a letter matching itself is impossible and eliminated, with no knowledge of the key needed

- **Cribs** came from predictable German message structure — standard weather-report phrasing, routine "nothing to report" messages, rigid formatting
- **Rejewski's Bomba** (Poland, pre-war) exploited a doubled-key procedural flaw; shared with Britain/France in July 1939
- **Turing/Welchman's Bombe** (Bletchley Park) used crib-derived "menus" and tested rotor positions for logical contradictions, discarding entire families of settings instantly
- **Operational mistakes** — "cillies" (predictable settings), message-key reuse, and "kisses" (the same content sent via a weaker, already-broken cipher) all gave codebreakers real advantages
- The core lesson: Enigma's math was sound; every real attack came from **how it was used** — the same throughline this whole course keeps returning to
- **Next chapter:** Symmetric-Key Cryptography — leaving classical/historical ciphers behind for the modern block and stream ciphers real systems use today

CHAPTER 5 OF 12

Symmetric-Key Cryptography — Block & Stream Ciphers, AES

Cryptography Fundamentals

Chapter 5 · Symmetric-Key Cryptography — Block & Stream Ciphers, AES

Chapters 2 through 4 covered ciphers no one would trust with real data today — but the underlying ideas (substitution, permutation, and the danger of a repeating key) turn out to be exactly the ideas modern symmetric cryptography is built from, just executed with vastly more rigor and mathematical scrutiny. This chapter leaves history behind and covers the ciphers actually protecting real systems right now — including the ones briefly introduced in `https1-2`, covered here in real depth.

Block Ciphers vs. Stream Ciphers — Two Different Approaches

Modern symmetric cryptography splits into two families, based on how they handle data:

	BLOCK CIPHERS	STREAM CIPHERS
Unit of operation	Fixed-size chunks ("blocks," e.g. 128 bits)	One bit or byte at a time
Partial data	Needs padding to fill out the last block	No padding needed at all
How it works	Complex, keyed transformation applied to each block	Generates a pseudorandom keystream, XORed with plaintext
Examples	DES, 3DES, AES	RC4, ChaCha20

Block ciphers also need a **mode of operation** to handle messages longer than one block — that's genuinely its own topic, covered fully in Chapter 6.

The Data Encryption Standard (DES) — A Brief History

Developed by IBM and adopted as the US federal encryption standard in 1977, DES was the first widely-used, publicly-specified block cipher — a real-world application of Chapter 1's Kerckhoffs's Principle, published openly rather than kept secret. It operates on 64-bit blocks using a **56-bit key** and a structure called a Feistel network, which splits each block in half and repeatedly mixes the halves together across 16 rounds.

56 bits gives roughly **7.2×10^{16}** possible keys — enormous by 1970s standards, but computing power caught up. By 1998, the Electronic Frontier Foundation's purpose-built "Deep Crack" machine brute-forced a DES key in under a day, publicly and decisively demonstrating that 56 bits was no longer remotely adequate.

Triple DES (3DES) — The Stopgap

Rather than design a wholly new cipher immediately, the industry's first response was to apply DES **three times in a row**, typically with two or three different keys, extending the effective key length without changing the underlying algorithm at all.

> **3DES DOESN'T TRIPLE THE SECURITY YOU MIGHT EXPECT**
>
> A "meet-in-the-middle" attack means two-key 3DES provides roughly 80 bits of effective security, not the 112 bits naive doubling might suggest — and 3DES is also roughly three times slower than plain DES, since it genuinely runs the cipher three times. It was always understood as a stopgap while a proper replacement was developed, and it still lingers in some legacy systems today, but it is not a recommended choice for new designs.

The AES Competition & Rijndael

Rather than have a single government agency design DES's replacement behind closed doors, the US National Institute of Standards and Technology (NIST) ran a genuinely open, public competition from 1997 to 2000 — every candidate algorithm was published in full and attacked publicly by cryptographers worldwide, the largest real-world demonstration of Kerckhoffs's Principle this course covers.

The winner, submitted by Belgian cryptographers Joan Daemen and Vincent Rijmen under the name **Rijndael**, became the **Advanced Encryption Standard (AES)** in 2001 — and remains, over two decades later, the world's dominant symmetric cipher, with no practical attack against the full algorithm ever found.

AES Internals — A Conceptual Overview

AES operates on 128-bit blocks, with a choice of 128-, 192-, or 256-bit keys (correspondingly using 10, 12, or 14 rounds). Each round applies four distinct transformations in sequence:

- **SubBytes** — a byte-for-byte substitution through a fixed, cryptographically-designed lookup table (the S-box) — conceptually Chapter 2's substitution ciphers, but nonlinear and specifically engineered to resist frequency-style analysis.
- **ShiftRows** — cyclically shifts bytes within each row of the block's internal grid — conceptually Chapter 2's transposition ciphers, rearranging position without changing values.
- **MixColumns** — a matrix operation that spreads each byte's influence across an entire column, ensuring a single-bit change in the input cascades into a large, unpredictable change in the output (the "avalanche

effect," revisited properly in Chapter 7).

- **AddRoundKey** — XORs the block with a subkey, freshly derived from the main key for that specific round via a key schedule.

> **AES IS CHAPTER 2'S IDEAS, ENGINEERED PROPERLY**
>
> Every classical cipher in Chapter 2 used exactly one of substitution or transposition, applied once, with a small, brute-forceable or statistically-leaky key. AES combines both, repeated across 10-14 rounds, with a rigorously designed nonlinear S-box and a 128-256 bit key — the same basic ingredients, but engineered specifically to eliminate the very weaknesses Chapter 2 spent an entire chapter exploiting.

AES Security Today

AES-128's keyspace is roughly **3.4×10^{38}** — utterly infeasible to brute-force with any foreseeable computing power, classical or otherwise. AES-256 is sometimes preferred for long-term data or particularly high-value secrets, partly as a margin against future cryptanalytic advances, and partly because of quantum computing: Grover's algorithm can theoretically halve a symmetric cipher's effective key strength, meaning AES-256 still offers roughly AES-128-equivalent (128-bit) security even against a quantum attacker — a topic Chapter 12's post-quantum preview returns to.

Stream Ciphers

A stream cipher takes a key (plus, critically, a **nonce** — a number used once) and generates a long pseudorandom keystream, then simply XORs that keystream with the plaintext, one bit or byte at a time. Decryption is identical: XOR the same keystream against the ciphertext.

RC4, designed in 1987, was for years the most widely deployed stream cipher — used in early WEP Wi-Fi security and early TLS. Statistical biases discovered in its keystream output, combined with real misuse in WEP's key-scheduling, eventually made it thoroughly broken and deprecated. **ChaCha20** is its modern successor — fast, carefully designed, and one of the two cipher families (alongside AES) used throughout TLS 1.3 today.

> **A STREAM CIPHER IS VIGENÈRE, DONE PROPERLY**
>
> Structurally, a stream cipher is exactly Chapter 2's Vigenère cipher: combine plaintext with a repeating-or-not keystream, letter by letter (or bit by bit). Vigenère's fatal flaw was that its "keystream" — the repeating keyword — was short and predictable, which the Kasiski examination exploited directly. A modern stream cipher's keystream is cryptographically generated, effectively non-repeating for any realistic message length, and never reused — solving Vigenère's exact weakness rather than sidestepping it.

> **NEVER ROLL YOUR OWN CIPHER**

DES took the combined effort of IBM and NSA review to reach 1970s-adequate security, and still needed replacing within two decades. AES took a multi-year, worldwide public competition to validate. A custom, unreviewed cipher — no matter how clever it feels — has had none of that scrutiny, and history (this entire course, so far) consistently shows that unreviewed cryptographic designs fail in ways their designers never anticipated. Always use a vetted, standard implementation (AES, ChaCha20) rather than inventing a new one.

Hands-On Exercises

EXERCISE 1

DES's 56-bit key (≈7.2×10^{16} possibilities) was brute-forced in under a day by 1998. AES-128's key is 128 bits (≈3.4×10^{38} possibilities). Roughly how many times larger is the AES-128 keyspace than DES's, and what does that imply about brute-forcing AES-128 with similar-generation hardware?

View solution

EXERCISE 2

Explain, in your own words, the structural difference between a block cipher and a stream cipher, and describe one practical scenario where a stream cipher's lack of padding requirement would make it the more natural choice.

View solution

EXERCISE 3

A developer accidentally reuses the exact same key and nonce to stream-encrypt two different messages. Using Chapter 2's Vigenère weakness as a guide, explain what an attacker who intercepts both ciphertexts could do, and why this happens.

View solution

CHAPTER 5 QUICK REFERENCE

- **Block ciphers** process fixed-size chunks and need a mode of operation (Ch.6); **stream ciphers** generate a keystream and XOR it with data, byte by byte
- **DES** (1977, 56-bit key) — brute-forced in under a day by 1998; **3DES** — a stopgap, only ~80-bit effective security despite 3x the computation
- **AES** (2001, Rijndael) — won an open, public NIST competition; 128/192/256-bit keys, 10/12/14 rounds of SubBytes/ShiftRows/MixColumns/AddRoundKey
- AES combines Chapter 2's substitution (SubBytes) and transposition (ShiftRows) ideas, engineered to resist the exact weaknesses that broke classical ciphers

- **RC4** — broken, deprecated; **ChaCha20** — its modern replacement, used throughout TLS 1.3
- A stream cipher is structurally Vigenère (Ch.2) with a cryptographically strong, effectively non-repeating keystream — reusing a keystream reintroduces Vigenère's exact weakness
- **Next chapter:** Modes of Operation — how block ciphers handle messages longer than one block, and why the choice of mode matters enormously

CHAPTER 6 OF 12

Modes of Operation — ECB, CBC, CTR/GCM & Why Mode Choice Matters

Cryptography Fundamentals

Chapter 6 · Modes of Operation — ECB, CBC, CTR/GCM & Why Mode Choice Matters

Chapter 5 covered AES as a cipher that transforms exactly one 128-bit block at a time. Real messages are almost never exactly 128 bits — so every real use of a block cipher needs a **mode of operation**: a defined way of chaining repeated block-cipher operations together to handle messages of any length. This turns out to matter as much as the underlying cipher itself — a message encrypted with rock-solid AES, in the wrong mode, can still be catastrophically insecure.

ECB — Electronic Codebook

The most obvious approach: split the message into blocks, and encrypt each block independently with the same key.

```
Block 1  --AES(key)-->  Ciphertext 1
Block 2  --AES(key)-->  Ciphertext 2
Block 3  --AES(key)-->  Ciphertext 3
```

This has one devastating property: **identical plaintext blocks always produce identical ciphertext blocks**. Any structure or repetition in the original data — rows of a spreadsheet, repeated fields in a record, or the flat colored regions of an image — survives directly into the ciphertext, just relabeled.

THE "ECB PENGUIN"

The canonical illustration is a bitmap image of a penguin, encrypted block-by-block with ECB mode. The output is still visibly, unmistakably a penguin — every flat-colored region of the original image, being made of many identical or near-identical blocks, maps to identical or near-identical ciphertext blocks in exactly the same layout. The image is technically "encrypted" and yet its shape is completely legible.

This is structurally the exact same weakness Chapter 2 spent an entire chapter exploiting in monoalphabetic substitution: a **fixed mapping** — there, letter-to-letter; here, block-to-block — always leaks the structure of whatever it's applied to, no matter how strong the underlying transformation is.

A REAL, WIDELY REPORTED CONSEQUENCE

> Adobe's 2013 breach is widely cited as a real-world example of exactly this failure: rather than hashing passwords (Chapter 7), reports indicate Adobe encrypted them with a block cipher in ECB mode. Because identical plaintexts produce identical ciphertexts, security researchers examining the leaked data were able to identify large clusters of users who shared the exact same password, without ever needing to actually decrypt anything.

CBC — Cipher Block Chaining

CBC breaks ECB's fixed-mapping problem directly: before encrypting each block, XOR it with the *previous block's ciphertext* first. This makes every block's encryption depend on everything that came before it, so identical plaintext blocks no longer produce identical ciphertext.

```
Ciphertext 1 = AES( Plaintext 1 XOR IV )
Ciphertext 2 = AES( Plaintext 2 XOR Ciphertext 1 )
Ciphertext 3 = AES( Plaintext 3 XOR Ciphertext 2 )
```

The very first block has no "previous ciphertext" to XOR against, so CBC needs an **initialization vector (IV)** — a random, unpredictable value used only once, playing a role reminiscent of Chapter 5's stream-cipher nonce. CBC encryption is inherently sequential (each block depends on the last), though decryption can actually be parallelized, since every ciphertext block is already available up front.

> **CBC HAS ITS OWN REAL PITFALLS**
> A reused or predictable IV reintroduces exactly the kind of pattern leakage ECB has, at least for the first block of a message. CBC's padding handling has also historically enabled "padding oracle" attacks, where an application's error-handling behavior around invalid padding leaks information an attacker can exploit to decrypt data without the key — a reminder that a mode's mathematical design and its real-world implementation are two separate things to get right.

CTR — Counter Mode

Counter mode takes a completely different approach: rather than encrypting the plaintext directly, it encrypts a **counter** (a nonce combined with an incrementing number) to generate a keystream, then XORs that keystream with the plaintext — turning a block cipher into a stream cipher, in the same spirit as Chapter 5's ChaCha20.

```
Keystream 1 = AES( Nonce || Counter=1 )
Keystream 2 = AES( Nonce || Counter=2 )
Keystream 3 = AES( Nonce || Counter=3 )

Ciphertext N = Plaintext N XOR Keystream N
```

Because each block's keystream depends only on the nonce and counter — not on any other block — CTR mode is fully **parallelizable** in both directions, unlike CBC, and needs no padding at all, since it's really a stream cipher underneath.

> **THE EXACT DANGER FROM CHAPTER 5, AGAIN**
>
> If the same nonce+counter combination is ever reused with the same key, CTR mode regenerates the identical keystream — reintroducing precisely the two-time-pad vulnerability Chapter 5's closing exercise walked through in full. Nonce management is CTR mode's single most important operational discipline.

The Missing Piece — None of These Provide Integrity

ECB, CBC, and CTR all provide **confidentiality only** — none of them protect integrity or authenticity, Chapter 1's other two goals. An attacker who intercepts CBC or CTR ciphertext can flip specific bits without ever needing the key, causing controlled, predictable changes in the decrypted plaintext — with no built-in mechanism to detect that tampering occurred at all. This is the exact scenario Chapter 1's very first exercise asked you to reason about in the abstract; here it is made concrete.

GCM — Galois/Counter Mode (Authenticated Encryption)

GCM solves the missing piece directly. It's built on CTR mode's own encryption approach, but adds a built-in **authentication tag**, computed over the ciphertext using a technique called GMAC — bundling confidentiality, integrity, *and* authenticity into a single primitive, a category called **AEAD** (Authenticated Encryption with Associated Data). "Associated data" lets an application authenticate extra context — like a message header — without needing to encrypt it at all.

Chapter 8 covers message authentication and AEAD ciphers in full depth; this chapter's job is just to establish GCM as the mode-of-operation-level answer to the gap the previous section identified. GCM (or an equivalent AEAD mode) is the default in TLS 1.3 today, precisely because it closes the confidentiality-only gap that ECB, plain CBC, and plain CTR all share.

Choosing a Mode Today

MODE	VERDICT
ECB	Never use for more than a single block — leaks structural patterns directly
CBC	Legacy but workable if IVs are handled correctly and padding errors are never exposed to an attacker
CTR	A solid building block, but confidentiality-only — needs a separate integrity mechanism (Ch.8) if used alone
GCM (AEAD)	The modern recommended default for nearly everything — confidentiality, integrity, and authenticity together

This is the mode-choice version of Chapter 5's "never roll your own cipher" warning: the cipher can be perfectly strong AES, and the system can still be broken by choosing the wrong mode around it. Reach for a vetted AEAD mode by default rather than assembling confidentiality and integrity by hand.

Hands-On Exercises

EXERCISE 1

Explain why encrypting a bitmap image with ECB mode still reveals the image's overall shape, connecting your answer back to the specific weakness of monoalphabetic substitution from Chapter 2.

View solution

EXERCISE 2

Two different messages are encrypted with CBC mode using the same key AND the same IV. If the first block of plaintext happens to be identical in both messages, what will an attacker observe by comparing the two ciphertexts, and what does that leak?

View solution

EXERCISE 3

An application uses plain CTR mode (no authentication) to encrypt a bank transfer's amount field, embedded within a larger ciphertext at a known byte position. Describe how an attacker who cannot decrypt the ciphertext could still tamper with the transferred amount, and explain how switching to GCM would prevent this specific attack.

View solution

CHAPTER 6 QUICK REFERENCE

- **ECB** — encrypts each block independently; identical plaintext blocks → identical ciphertext blocks; never use beyond one block
- **CBC** — XORs each block with the previous ciphertext; needs a random, non-reused IV; sequential encryption, parallel decryption
- **CTR** — encrypts a nonce+counter to build a keystream, turning a block cipher into a stream cipher; fully parallelizable; nonce+counter must never repeat
- ECB/CBC/CTR are all **confidentiality-only** — none detect tampering; bit-flipping attacks are possible against all three without the key
- **GCM (AEAD)** — CTR-based encryption plus a built-in authentication tag (GMAC); the modern recommended default

- **Next chapter:** Hash Functions — properties, SHA-2/SHA-3, and why MD5/SHA-1 were retired

CHAPTER 7 OF 12

Hash Functions — Properties, SHA-2/SHA-3 & Why MD5/SHA-1 Fell

Cryptography Fundamentals

Chapter 7 · Hash Functions — Properties, SHA-2/SHA-3 & Why MD5/SHA-1 Fell

Chapter 6 mentioned GCM's authentication tag without explaining how anything like it is actually built. Hash functions are the primitive that makes it possible — and they're a genuinely different tool from everything covered so far: no key in their basic form, and deliberately, permanently **not reversible**.

What Is a Hash Function?

A cryptographic hash function takes an input of *any* length — a single character or an entire multi-gigabyte file — and deterministically produces a **fixed-length output**, called a digest or hash. The same input always produces the same output, but there's no way to run the process backward: given only the digest, there's no operation that recovers the original input.

Hash ≠ Encryption

This is one of the most common mix-ups in casual security conversation — "we encrypted your password" almost always actually means "we hashed it," and the two are fundamentally different operations:

	ENCRYPTION (CH.5-6)	HASHING
Reversible?	Yes — with the right key	No, by design — one-way only
Needs a key?	Yes	No, in its basic form
Output length	Roughly matches input length	Always fixed, regardless of input length
Goal	Confidentiality (Ch.1)	Integrity verification, and — with care — one-way storage

"Decrypting a hash" isn't a meaningful operation at all — there's no key to decrypt it with, and the transformation was never designed to be reversed.

The Three Security Properties

A cryptographically secure hash function needs to resist three distinct kinds of attack:

- **Preimage resistance** — given only a hash output H, it should be computationally infeasible to find *any* input that produces H.
- **Second-preimage resistance** — given a specific input M1, it should be infeasible to find a *different* input M2 that produces the same hash as M1.
- **Collision resistance** — it should be infeasible to find *any* two different inputs M1 and M2 (neither one given in advance) that happen to produce the same hash.

Collision resistance is strictly the hardest of the three to achieve — and, thanks to a genuinely counterintuitive piece of probability, it needs considerably more output bits than the other two properties to hold up.

The Birthday Paradox & Why Collision Resistance Needs More Bits

The "birthday paradox" observes that in a room of just 23 people, there's already a better-than-even chance two of them share a birthday — far fewer people than the 366 you might naively expect, because you're comparing every pair of people against each other, not just one person against everyone else.

The same math applies to hash collisions. Finding a *specific* preimage for an n-bit hash takes roughly $\mathbf{2^n}$ attempts on average. But finding *any* collision at all — any two inputs that happen to match — takes only about $\mathbf{2^n/^2}$ attempts, because every new attempt is being checked against every previous attempt, not just one fixed target.

> **THIS IS EXACTLY WHY HASH OUTPUTS NEED TO BE SO LARGE**
>
> A 128-bit hash sounds enormous — but its real collision resistance, thanks to the birthday paradox, is only about 2^{64}, not 2^{128}. That's precisely why every modern hash function this chapter recommends uses 256 bits or more: it's not overcautious rounding, it's a direct, calculable consequence of the birthday paradox halving the effective exponent.

The Avalanche Effect

A well-designed hash function exhibits the **avalanche effect**: changing even a single bit of the input should completely and unpredictably change the output digest, with roughly half of the output bits flipping on average. Two near-identical inputs should never produce even remotely similar-looking hashes — this is the same underlying idea Chapter 5 introduced for AES's MixColumns step, now the defining property of an entire class of primitive.

MD5 — Broken

Designed in 1992 with a 128-bit output, MD5 was for years the internet's default hash function. Its 128-bit output already gives only ~2^{64} collision resistance per the birthday paradox above — a real weak point even before anything else went wrong. In 2004, researchers (Wang et al.) demonstrated **practical** collision attacks, dramatically faster than the theoretical 2^{64} figure, exploiting real structural weaknesses in MD5's design. The consequences were not theoretical: the 2012 Flame malware used a forged MD5 collision to fake a legitimate Microsoft code-signing certificate. MD5 is still sometimes used for non-security checksums (verifying a file wasn't corrupted in transit) but must never be used anywhere security depends on it.

SHA-1 — Also Broken

Designed in 1995 with a 160-bit output, SHA-1 became the internet's next default — used for years in TLS certificates, and still used today inside Git for commit hashing (a non-security use, more on that distinction shortly). In 2017, Google and CWI Amsterdam publicly demonstrated **SHAttered**, a practical SHA-1 collision, using an approach echoing MD5's own 2004 break. Browsers and certificate authorities deprecated SHA-1 for TLS certificates around the same time.

SHA-2 — The Current Workhorse

Published in 2001, SHA-2 is actually a family of functions — SHA-256, SHA-384, SHA-512, named for their output size in bits. Despite being designed by the NSA (a detail that draws periodic suspicion), SHA-2 has been public and openly scrutinized for over two decades, per Chapter 1's Kerckhoffs's Principle, with no practical attack ever found. It remains the dominant hash function in TLS, Bitcoin, and countless other systems today.

SHA-3 — A Structurally Different Backup

MD5, SHA-1, and SHA-2 all share the same underlying internal design, called the Merkle-Damgård construction. SHA-3, standardized by NIST in 2015 after another open public competition (again mirroring the AES competition from Chapter 5), uses a completely different internal design called the **sponge construction**, based on an algorithm called Keccak. SHA-3 isn't a replacement for SHA-2 — SHA-2 remains fully secure — it exists specifically as a structural hedge: if some future breakthrough attack targets a weakness shared by the Merkle-Damgård construction itself, SHA-3's genuinely different internals wouldn't be affected the same way.

What Hash Functions Are Actually Used For

- **Integrity verification** — confirming a downloaded file or received message wasn't altered, by comparing hashes.

- **Digital signatures** — signing algorithms (Chapter 10) actually sign a message's *hash*, not the raw message itself, since hashes are small and fixed-size regardless of the original message's length.
- **Content addressing** — Git identifies every commit by its hash; blockchain systems chain blocks together the same way.
- **Password storage** — with an important, easily-missed caveat below.

NEVER USE A RAW FAST HASH FOR PASSWORD STORAGE

SHA-256 being completely uncollided and cryptographically sound says nothing about whether it's the *right tool* for hashing passwords. SHA-256 is deliberately, extremely fast — a property that's exactly right for verifying a large file's integrity quickly, and exactly wrong for password storage, since it lets an attacker with stolen password hashes try billions of guesses per second on modern GPU hardware. Real password storage needs a deliberately *slow*, purpose-built function — bcrypt, scrypt, or Argon2id — a distinction covered in full in Chapter 11, and already covered from the authentication side in this site's own `bc1-2` chapter.

Hands-On Exercises

EXERCISE 1

In your own words, explain the difference between second-preimage resistance and collision resistance. Give a concrete example of an attack scenario each property is specifically designed to prevent.

View solution

EXERCISE 2

MD5's 128-bit output gives roughly 2^{64} ($\approx 1.8 \times 10^{19}$) collision resistance due to the birthday paradox. Is 2^{64} still a safe number of attempts to require today? Explain your reasoning, and connect it to why MD5 is considered fully broken rather than merely "a bit weak."

View solution

EXERCISE 3

A developer argues: "SHA-256 has never been broken, so hashing user passwords with plain SHA-256 is perfectly secure." Explain what's wrong with this reasoning.

View solution

CHAPTER 7 QUICK REFERENCE

- **Hash function** — deterministic, one-way, fixed-length output from any-length input; NOT reversible, NOT the same as encryption
- Three properties: **preimage** (can't find any input for a given hash), **second-preimage** (can't find a second input matching a given one), **collision** (can't find any two matching inputs at all)
- **Birthday paradox** — collision resistance is only ~$2^{n/2}$ for an n-bit hash, which is why modern hashes use 256+ bits
- **Avalanche effect** — a tiny input change should completely, unpredictably change the output
- **MD5** and **SHA-1** — both practically broken (2004, 2017) and must not be used for security purposes
- **SHA-2** — the current secure workhorse; **SHA-3** — a structurally different hedge, not a replacement
- Fast hashes (SHA-256) are wrong for password storage — use deliberately slow bcrypt/scrypt/Argon2id instead (Ch.11, `bc1-2`)
- **Next chapter:** Message Authentication — MACs, HMAC & AEAD, building the integrity/authenticity piece hash functions alone don't provide

CHAPTER 8 OF 12

Message Authentication — MACs, HMAC & AEAD

Cryptography Fundamentals

Chapter 8 · Message Authentication — MACs, HMAC & AEAD

Chapter 7 introduced hash functions as tools for integrity checking — but a plain hash, sent alongside a message with no other protection, turns out to guarantee almost nothing on its own. This chapter covers the missing ingredient: binding an integrity check to a **secret**, so only someone who holds it can produce a valid one — and finally explains, properly, what GCM's authentication tag from Chapter 6 was actually doing the whole time.

Why a Plain Hash Isn't Enough

Imagine a sender transmits a message along with its SHA-256 hash, so the receiver can verify nothing was altered in transit. An attacker intercepting this in transit can simply:

1. Tamper with the message however they like.
2. Recompute the SHA-256 hash of their *tampered* message — this requires no secret at all, anyone can run SHA-256.
3. Forward the tampered message along with this newly-recomputed, perfectly valid hash.

The receiver checks the hash, finds it matches, and accepts the tampered message as genuine. The hash was never wrong — it correctly hashed the (tampered) message it was given. A plain hash verifies "this message wasn't altered *after I computed this specific hash*," which is completely useless against an attacker who can just recompute a fresh one.

MACs — Message Authentication Codes

A **MAC** is a hash-like function that *also* takes a secret key: `MAC(key, message)`. Only someone who holds the key can produce a valid MAC for a given message. The sender computes the MAC and sends it alongside the message; the receiver, who also holds the key, recomputes the MAC independently and checks it matches.

This directly closes the gap from the previous section: an attacker who tampers with the message can no longer just recompute a matching value, because doing so requires the secret key they don't have. A mismatched MAC means either the message was altered, or it wasn't produced by someone holding the real key — either way, it's rejected.

HMAC — Hash-based MAC

Rather than design an entirely new primitive for MACs, **HMAC** is the standard way of building a MAC out of an existing hash function you already trust (Chapter 7's SHA-256, for instance). Conceptually:

```
HMAC(key, message) =
    Hash( (key XOR outerPad) || Hash( (key XOR innerPad) || message ) )
```

The nested, double-hashing structure isn't decoration — it exists specifically to defend against a real weakness in naive approaches like simply computing `Hash(key || message)` directly. Some hash constructions (including the Merkle-Damgård structure behind SHA-1 and SHA-2, from Chapter 7) allow an attacker who knows a hash's output — without knowing the original input — to compute a *valid* hash for that input with extra data appended to it. This is called a **length-extension attack**, and it's a real, demonstrated weakness in the naive construction. HMAC's specific nested design was engineered to resist it.

Verifying a MAC — Constant-Time Comparison

A SUBTLE BUT VERY REAL IMPLEMENTATION PITFALL

A naive byte-by-byte comparison that stops and returns "false" the instant it finds a mismatched byte leaks **timing information** — a comparison that fails on the very first byte returns fractionally faster than one that fails on the tenth byte. An attacker who can measure response timing precisely enough can exploit this to guess a valid MAC one byte at a time, dramatically faster than brute-forcing the whole thing at once. Correct implementations always use a **constant-time comparison** function, which takes exactly the same amount of time regardless of where (or whether) a mismatch occurs — a good reminder, echoing Chapter 5's "never roll your own crypto," that even comparing two values correctly is a real, easy-to-get-wrong detail.

Encrypt-then-MAC vs. MAC-then-Encrypt vs. Encrypt-and-MAC

Combining a cipher (Chapters 5-6) with a MAC can be done in three different orders, and the choice genuinely matters:

SCHEME	HOW IT WORKS	VERDICT
Encrypt-then-MAC	Encrypt the plaintext, then compute the MAC over the ciphertext	**Recommended** — the MAC can be checked before decryption is ever attempted
MAC-then-Encrypt	Compute the MAC over the plaintext, then encrypt both together	Historically used by early TLS versions; a contributing factor in several real TLS vulnerabilities
Encrypt-and-MAC	Encrypt the plaintext and MAC the plaintext independently, send both	Used by SSH; the MAC is computed over unencrypted data, which can leak information in some cases

Encrypt-then-MAC's advantage is concrete, not theoretical: because the MAC covers the *ciphertext*, the receiver can verify it and reject a tampered message **before ever decrypting attacker-controlled data**. This is exactly the standard defense against Chapter 6's padding oracle attacks — if a tampered ciphertext is rejected by the MAC check first, the vulnerable padding-error behavior that a padding oracle attack depends on is never even reached.

AEAD — What GCM Was Actually Doing All Along

Chapter 6 introduced GCM as bundling confidentiality, integrity, and authenticity into one primitive, without fully explaining how. Now it can be stated precisely: GCM's authentication tag is built from **GMAC**, a MAC construction specialized for GCM's own CTR-mode-based encryption — and the overall structure of GCM is, in essence, **encrypt-then-MAC baked directly into a single, efficient primitive**, rather than two separate operations bolted together by hand. That's the real technical answer behind Chapter 6's "mode-of-operation-level" preview.

Associated Data — Authenticating Without Encrypting

Sometimes part of a message genuinely needs to stay *readable* — a network packet's routing header, say, needs to be visible before decryption can even happen — while still being protected from tampering. AEAD ciphers support this directly via **associated data**: extra data that's included in the authentication tag's computation (so tampering with it is detected) but is never encrypted at all. A network packet with an unencrypted header (needed for routing) and an encrypted payload, both covered by one authentication tag, is the textbook example.

> **CLOSING THE LOOP FROM CHAPTER 1**
>
> Chapter 1's very first exercise asked what encryption alone does and doesn't guarantee — confidentiality, and nothing else. This chapter is the other half of that answer: a MAC alone gives integrity and authenticity but no confidentiality at all; encryption alone (Chapters 5-6) gives confidentiality but no integrity or authenticity; AEAD (GCM, or a properly-built encrypt-then-MAC construction) is what actually satisfies three of Chapter 1's four goals together in one primitive.

Hands-On Exercises

EXERCISE 1

Walk through, step by step, why sending a message alongside a plain (unkeyed) SHA-256 hash provides no real protection against a tampering attacker, and explain specifically what a MAC adds that a plain hash lacks.

View solution

EXERCISE 2

A developer proposes building a MAC as simply `SHA256(key || message)` rather than using HMAC. Explain, conceptually, why this naive construction is considered unsafe, and what specific problem HMAC's nested double-hash structure is designed to prevent.

View solution

EXERCISE 3

Of encrypt-then-MAC, MAC-then-encrypt, and encrypt-and-MAC, which scheme is safest against Chapter 6's padding oracle attack, and why specifically? Explain what happens differently at the moment a tampered ciphertext arrives.

View solution

CHAPTER 8 QUICK REFERENCE

- A **plain hash** provides no authenticity — anyone can recompute one for tampered data; needs a secret to be meaningful
- **MAC** = a keyed hash-like function; only the key holder can produce a valid one for given data
- **HMAC** — the standard way to build a MAC from a hash function; its nested structure defends against length-extension attacks
- MAC comparison must be **constant-time** — a naive early-exit comparison leaks timing information exploitable byte by byte
- **Encrypt-then-MAC** is the recommended combination order — lets tampered ciphertext be rejected before decryption is ever attempted, closing off padding oracle attacks
- **GCM** (Ch.6) is, underneath, essentially encrypt-then-MAC built into one efficient AEAD primitive, using GMAC
- **Associated data** — authenticated but left unencrypted, for data (like packet headers) that must stay readable
- **Next chapter:** Public-Key Cryptography — RSA, Diffie-Hellman & Elliptic Curves, leaving the shared-secret world of symmetric crypto behind

CHAPTER 9 OF 12

Public-Key Cryptography — RSA, Diffie-Hellman & Elliptic Curves

Cryptography Fundamentals

Chapter 9 · Public-Key Cryptography — RSA, Diffie-Hellman & Elliptic Curves

Every cipher covered since Chapter 2 — including AES and every mode in Chapters 5-6 — is **symmetric**: the same secret has to reach both parties before anything can happen. Chapter 1 named this "the hard problem symmetric crypto never solved," and previewed asymmetric cryptography as the eventual answer. This chapter is that answer, in full.

The Problem Symmetric Crypto Never Solved

How do two parties who have never met establish a shared secret, over a channel an eavesdropper might be listening to the entire time? Historically, the only real answer was physical: couriers, sealed codebooks, or — as Chapter 3 covered directly — Enigma's own daily key settings, physically distributed on paper to every operator in advance. That approach doesn't scale to two strangers connecting over the open internet for the first time, which is exactly the situation every HTTPS connection (`https1`) faces constantly.

Trapdoor Functions — The Core Idea

Asymmetric cryptography is built on **trapdoor functions**: a function that's easy to compute in one direction, but computationally infeasible to reverse — *unless* you possess a specific secret piece of information (the "trapdoor") that makes reversal easy. The public key effectively *is* the function itself, fully public; the private key is the trapdoor.

> **A STRIKING EXTENSION OF CHAPTER 1'S KERCKHOFFS'S PRINCIPLE**
>
> Chapter 1 established that an algorithm can be fully public without compromising security, as long as the key stays secret. Public-key cryptography takes this further than any classical cipher ever could: here, the "algorithm" (the public key itself) isn't just publishable — it's *meant* to be handed to literally anyone, and the system's security still holds completely.

RSA — Trapdoors from Prime Factorization

RSA's trapdoor is built from a genuinely simple asymmetry in difficulty: **multiplying** two large prime numbers together is fast and easy, but **factoring** that product back into its two original primes is computationally infeasible once the primes are large enough (2048+ bits in practice).

- **Public key** — the product of the two primes (the "modulus"), plus a public exponent.
- **Private key** — derived using the two original prime factors, which only the key's owner knows.

Encryption and decryption both use modular exponentiation against the modulus. Knowing the two original prime factors lets you efficiently compute the specific exponent needed to reverse the operation; without them, an attacker is left facing the same hard problem — factor an enormous number — that made RSA usable in the first place.

RSA in Practice — Key Sizes & Speed

2048-bit RSA keys are the accepted minimum today, with 3072 or 4096 bits used for higher security margins. RSA is computationally expensive compared to Chapter 5's symmetric ciphers — exactly why Chapter 1 flagged asymmetric crypto as "much slower, used sparingly." In practice, RSA is typically used only to encrypt or exchange a much shorter symmetric key, after which the actual bulk data is handled by fast AES or ChaCha20 (Chapters 5-6) — precisely what `https1`'s TLS handshake does.

Diffie-Hellman Key Exchange — A Different Approach

Diffie-Hellman (DH), published in 1976, doesn't encrypt or decrypt anything at all. It lets two parties who have never met derive an *identical shared secret* over a public channel, even if an eavesdropper sees every single message exchanged between them.

> **THE CLASSIC PAINT-MIXING ANALOGY**
>
> Both parties agree publicly on a shared starting color. Each secretly mixes in their own private color, producing a new mixed color, and sends *that mixture* (not their private color) to the other party over the public channel. Each party then mixes their own private color into what they received. Both arrive at the exact same final combined color — but an eavesdropper who only ever saw the two publicly-exchanged mixtures can't feasibly work backward to recover either party's private color, because "un-mixing" paint isn't a practical operation.

The real mathematics uses modular exponentiation over a large prime, and the hard problem underneath is the **discrete logarithm problem**: given the public values exchanged, finding either party's private exponent is computationally infeasible for large enough numbers.

RSA vs. Diffie-Hellman — Different Jobs

RSA can both encrypt data directly and produce digital signatures (Chapter 10). Diffie-Hellman solves *key exchange specifically* — it never encrypts a message itself, only derives a shared secret both parties can then use with a symmetric cipher.

DH's major practical advantage is **forward secrecy**, when used with fresh, temporary ("ephemeral") values for every session — commonly written DHE. Even if a party's long-term private key is compromised at some later date, past session keys derived via ephemeral DH cannot be recovered from that long-term key alone, since the ephemeral values used to derive them were never transmitted anywhere and are discarded immediately after use.

Elliptic Curve Cryptography (ECC) — Smaller Keys, Same Security

ECC is built on a different hard problem entirely — the elliptic curve discrete logarithm problem — believed to be considerably harder, per key bit, than either RSA's factoring problem or plain Diffie-Hellman's discrete logarithm problem. The practical payoff is dramatic: a **256-bit** ECC key offers roughly the same real-world security as a **3072-bit** RSA key.

Why the gap is so large: no algorithm anywhere near as effective as the best known factoring or discrete-log algorithms has ever been found against elliptic curve math, so ECC needs far fewer bits to reach an equivalent difficulty level — smaller keys, faster computation, and less bandwidth and storage, all for the same practical security. **ECDH** is the elliptic-curve version of Diffie-Hellman; **ECDSA**, the elliptic-curve signing equivalent, is covered fully in Chapter 10.

Putting It Together — What TLS Actually Does

Now fully informed, `https1`'s handshake can be stated precisely: modern TLS 1.3 uses **ECDHE** (ephemeral elliptic-curve Diffie-Hellman) for key exchange, giving every session forward secrecy by default, then immediately switches to a fast symmetric AEAD cipher — AES-GCM or ChaCha20-Poly1305 (Chapters 5, 6, and 8) — for all the actual bulk data. RSA's role in modern TLS has shifted almost entirely to signing the server's certificate (Chapter 10) rather than performing the key exchange itself.

A REMARKABLY RECENT BREAKTHROUGH

Symmetric ciphers date back millennia — Caesar's shift cipher (Chapter 2) is over 2,000 years old. Public-key cryptography wasn't invented until 1976 (Diffie-Hellman) and 1977 (RSA) — meaning the entire idea that two strangers could establish a shared secret over an open, eavesdropped channel is barely half a century old, and directly closes the "hard problem" Chapter 1 flagged as symmetric crypto's fundamental limitation.

QUANTUM COMPUTING THREATENS THIS CHAPTER FAR MORE THAN CHAPTER 5

Shor's algorithm, if ever run at sufficient scale on a real quantum computer, can efficiently solve *both* RSA's factoring problem and Diffie-Hellman/ECC's discrete logarithm problem — breaking every algorithm in this chapter outright,

not just weakening them. Chapter 5's Grover's algorithm, by contrast, only halves AES's effective key strength, a manageable degradation. This sharp asymmetry — devastating for RSA/DH/ECC, merely inconvenient for AES — is exactly why the post-quantum cryptography preview in Chapter 12 is focused specifically on replacing this chapter's algorithms.

Hands-On Exercises

EXERCISE 1

In your own words, explain what a trapdoor function is, and explain why RSA's security specifically depends on multiplication being easy while factoring is hard — not the other way around.

View solution

EXERCISE 2

Using the paint-mixing analogy, explain how two parties end up with the same shared secret while an eavesdropper who saw every exchanged message cannot. Then explain what "forward secrecy" means, and why it specifically requires ephemeral (not static/reused) private values.

View solution

EXERCISE 3

Explain why Shor's algorithm poses a much more serious threat to RSA and Diffie-Hellman than Grover's algorithm poses to AES, even though both are quantum algorithms aimed at cryptography.

View solution

CHAPTER 9 QUICK REFERENCE

- **Trapdoor function** — easy one direction, infeasible to reverse without a secret; public key = function, private key = trapdoor
- **RSA** — trapdoor from prime factorization (easy to multiply, hard to factor); used mainly for key exchange/signing since it's slow, not bulk encryption
- **Diffie-Hellman** — derives a shared secret over a public channel without encrypting anything; hard problem = discrete logarithm; DHE gives forward secrecy
- **ECC** — a harder problem per bit than RSA/DH, so a 256-bit ECC key ≈ a 3072-bit RSA key; ECDH and ECDSA are its key-exchange and signing forms

- **Modern TLS** — ECDHE for key exchange (forward secrecy), AES-GCM/ChaCha20 for bulk data, RSA mostly just for certificate signing now
- **Quantum threat** — Shor's algorithm breaks RSA/DH/ECC outright; Grover's algorithm only halves AES's key strength — a sharp asymmetry driving Chapter 12's post-quantum preview
- **Next chapter:** Digital Signatures & the Trust Chain — signing vs. encrypting, and revisiting `https1`'s certificates from the primitive level

CHAPTER 10 OF 12

Digital Signatures & the Trust Chain

Cryptography Fundamentals

Chapter 10 · Digital Signatures & the Trust Chain

Chapter 9 built RSA and ECC entirely around encryption — confidentiality. This chapter covers the reverse-direction use of the exact same key pairs: proving *who produced a message* and that it wasn't altered. By the end of this chapter, every one of Chapter 1's four goals will finally have a concrete mechanism behind it — and `https1`'s certificate chapters can be revisited with full understanding of what's actually happening underneath.

Signing vs. Encrypting — Not the Same Operation

These two operations use the same key *pair*, but in opposite roles:

	ENCRYPTING	SIGNING
Goal (Ch.1)	Confidentiality	Integrity, authenticity, non-repudiation
Uses which key to act?	Recipient's PUBLIC key	Signer's PRIVATE key
Uses which key to reverse?	Recipient's PRIVATE key (to decrypt)	Signer's PUBLIC key (to verify)
Who can perform the action?	Anyone (public key is, well, public)	Only the private key holder
Who can reverse the action?	Only the private key holder	Anyone (public key is public)

Notice the pattern flips entirely: encrypting is "anyone can lock it, only the owner can unlock it"; signing is "only the owner can produce it, anyone can check it." Same key pair, opposite direction of use.

How Signing Actually Works — Hash-Then-Sign

Chapter 7 mentioned that signing algorithms actually sign a message's *hash*, not the raw message — here's why, and exactly how it works:

1. The signer computes a cryptographic hash of the message (Chapter 7) — messages can be any length, but signing algorithms operate on fixed-size numeric inputs, so hashing first makes the input size uniform and small.
2. The signer applies their **private key** to that hash, producing the signature.

3. The signer sends the message plus the signature.
4. The recipient independently hashes the *received* message themselves.
5. The recipient uses the signer's **public key** to check whether the signature corresponds to that hash. If it matches, the signature is valid; if not, either the message was altered or the signature wasn't produced by the claimed signer's private key.

RSA Signatures vs. ECDSA vs. EdDSA

RSA can sign directly, using the same trapdoor idea from Chapter 9 run in the signing direction. **ECDSA** is the elliptic-curve signature algorithm — offering the same dramatically smaller key and signature sizes for equivalent security that Chapter 9 covered for ECDH, and it's now the dominant choice for new certificate issuance. **EdDSA** (commonly seen as Ed25519) is a newer, related alternative — faster, and critically, **deterministic**: it doesn't need a fresh random value for every signature the way ECDSA does.

> **ECDSA'S NONCE-REUSE TRAP — A REAL, CATASTROPHIC FAILURE**
>
> ECDSA requires a fresh, secret random value (a "nonce") for every single signature. If that nonce is ever reused across two different signatures — whether from a broken random number generator or a design flaw — the private key can be mathematically recovered from just those two signatures. This isn't theoretical: Sony's PlayStation 3 firmware-signing private key was famously extracted in 2010 exactly this way, after security researchers discovered Sony's signing implementation reused the same nonce for every signature instead of generating a fresh one each time. EdDSA's deterministic design (deriving the nonce from the message and key rather than generating it randomly each time) exists specifically to make this entire class of failure impossible.

What a Signature Actually Proves

Tying directly back to Chapter 1's four goals:

- **Integrity** — Chapter 7's avalanche effect means even a single altered bit of the message produces a completely different hash, which will no longer match the signature. Any tampering is detected.
- **Authenticity** — only someone holding the corresponding private key could have produced a signature that verifies successfully against a given public key.
- **Non-repudiation** — because only the private key holder could have produced it, they cannot credibly deny having signed it later, assuming their private key was never compromised.

The Trust Chain — From Signatures to Certificates

A signature alone only proves "produced by whoever holds *this specific* private key" — it says nothing about who that key actually belongs to in the real world. That's exactly the gap `https1`'s certificate chapters (`https1-4`, `https1-5`) exist to close, and it can now be explained fully.

A **Certificate Authority (CA)** signs a certificate that binds a specific public key to a specific identity — a domain name, for instance — using exactly the hash-then-sign process described above, with the CA's own private key. Browsers and operating systems ship with a small, curated set of trusted **root CA** public keys built in. The full **chain of trust** is literally a chain of signatures: a root CA signs an intermediate CA's public key, and that intermediate CA signs the actual website's certificate — each link individually verifiable via the exact signature-verification process described earlier in this chapter, all the way back to a root key the browser already trusts.

Signatures Beyond TLS

Certificates are only one application. **Code signing** lets software be verified as genuinely coming from its claimed publisher, unaltered — which is exactly what Chapter 7's 2012 Flame malware defeated, by forging an MD5 collision specifically to produce a fraudulent, still-valid-looking Microsoft code signature. **Software update mechanisms** and **package managers** both rely on signature verification to ensure a downloaded update or package genuinely came from its claimed source. **Git commit signing** applies the same idea to source code history itself.

> **CHAPTER 1'S FOUR GOALS, NOW FULLY ACCOUNTED FOR**
>
> Confidentiality: Chapters 5, 6, and 9. Integrity: Chapters 7 and 8. Authenticity: Chapters 8 and this chapter. Non-repudiation: this chapter specifically, via digital signatures. Every goal named on the very first page of this course now has a concrete, working mechanism behind it.

Hands-On Exercises

EXERCISE 1

For each of the four operations — encrypt, decrypt, sign, verify — state which specific key is used (signer's or recipient's, public or private) to perform it.

View solution

EXERCISE 2

Explain why signing algorithms sign a message's hash rather than the raw message itself, and explain — using Chapter 7's avalanche effect specifically — why this still fully protects the message's integrity, rather than weakening it.

View solution

EXERCISE 3

Using this chapter's signature-verification process, explain precisely what a Certificate Authority is vouching for when it signs a website's certificate, and explain what would go wrong if an attacker managed to get a malicious root CA added to a browser's trust store.

View solution

CHAPTER 10 QUICK REFERENCE

- **Encrypting** — recipient's public key locks, recipient's private key unlocks; **signing** — signer's private key produces, signer's public key verifies
- **Hash-then-sign** — hash the message (Ch.7), then sign the fixed-size hash rather than the arbitrary-length message directly
- **RSA** signs directly; **ECDSA** gives smaller keys/signatures for equal security; **EdDSA** is deterministic, avoiding ECDSA's nonce-reuse trap
- A reused ECDSA nonce leaks the private key outright — the real cause of Sony's 2010 PS3 signing-key extraction
- A signature proves integrity (Ch.7's avalanche effect), authenticity, and non-repudiation together — all four of Chapter 1's goals are now fully covered
- **Trust chain** — a chain of signatures from a trusted root CA down to a website's own certificate, each link verified the same way
- **Next chapter:** Key Management in Practice — entropy, storage, rotation, and HSMs, the operational discipline every primitive in this course ultimately depends on

CHAPTER 11 OF 12

Key Management in Practice — Entropy, Storage, Rotation & HSMs

Cryptography Fundamentals

Chapter 11 · Key Management in Practice — Entropy, Storage, Rotation & HSMs

Every primitive this course has covered — AES (Ch.5), hashing (Ch.7), MACs (Ch.8), RSA/DH/ECC (Ch.9), signatures (Ch.10) — is only as strong as the keys behind it. This chapter is the full payoff of the throughline first named in Chapter 1 and demonstrated concretely in Chapter 4's Enigma breakdown: real cryptographic failures are almost always about **key management**, not the underlying mathematics.

Entropy — Where Keys Actually Come From

A cryptographic key needs to come from a source of genuine **entropy** — true unpredictability, not merely output that "looks random." A cryptographically secure pseudorandom number generator (CSPRNG), properly seeded from an operating system's real entropy sources (hardware noise, timing jitter, and similar unpredictable physical inputs), is the correct tool. A pseudorandom number generator built for simulations, games, or statistical sampling — even one that passes ordinary randomness tests — is a completely different, unsuitable tool: such generators are typically deterministic and predictable once their internal state or seed is known, which is exactly what a cryptographic attacker needs.

THE 2008 DEBIAN OPENSSL DISASTER

In 2006, a Debian maintainer, trying to silence a code-analysis warning, accidentally removed nearly all the sources of real entropy feeding OpenSSL's key generation on Debian and Ubuntu systems. For roughly two years, until the flaw was discovered and fixed in 2008, every SSL/SSH key generated on an affected system was drawn from a drastically shrunk pool of possible values — small enough that the *entire set* of possible keys for common key sizes could be pre-computed and exhaustively enumerated. Every affected key, across a huge number of real servers, was retroactively rendered guessable. It remains one of the most severe real-world key-management disasters in modern cryptography — not because any cipher or algorithm in this course was broken, but because the entropy feeding key generation was quietly, catastrophically weakened.

Key Storage — Where Keys Live Once Generated

Keys should never be committed directly into source code or version control — a rule this site's own `pipelines1-5` states plainly: a committed credential is compromised forever, since version history retains it

indefinitely even after deletion. Environment variables are a step up but still leave keys sitting in plaintext in process memory and configuration. The modern standard is a dedicated **secrets manager** (a cloud provider's KMS, or a tool like Vault) — purpose-built for storing, retrieving, and auditing access to sensitive key material, with the keys themselves encrypted at rest, echoing `dbsec1-5`'s encryption-at-rest coverage applied specifically to key material rather than application data.

Hardware Security Modules (HSMs)

An HSM is a dedicated, tamper-resistant hardware device that generates and stores private keys such that they **never leave the device in plaintext form at all**. Rather than retrieving a key to use it elsewhere, cryptographic operations — signing, decryption — are performed *inside* the HSM itself; only the input and the result cross the boundary, never the raw key material. This is a categorically different security model from software-only storage: even a fully compromised host system, with complete access to its own memory and disk, still cannot extract the private key — it can only request operations using it, and only for as long as it retains legitimate access to do so. HSMs are standard for root CA private keys (Chapter 10), banking infrastructure, and as the backend for most cloud KMS offerings.

Key Rotation

Keys shouldn't live forever. Regular rotation limits the blast radius of an undetected compromise (an old, retired key is useless to an attacker even if it was silently stolen), reduces how much data any single key ever protects, and forces regular verification that key-management processes actually work end to end, rather than being untested until an emergency. Rotation is easy to state and genuinely hard to execute well — replacing a key used by many dependent systems or services requires careful coordination, echoing the real complexity behind TLS certificate revocation.

> **FORWARD SECRECY IS ROTATION TAKEN TO ITS LOGICAL EXTREME**
> Chapter 9's ephemeral Diffie-Hellman (DHE/ECDHE) generates a brand-new key for literally every single session, then discards it immediately — the most aggressive possible form of key rotation, and precisely why it delivers forward secrecy: there's no long-lived key left around to later compromise.

Key Derivation — One Master Key, Many Purposes

Rather than independently generating and managing a large number of separate keys, real systems often derive several purpose-specific keys from one master secret, using a **key derivation function (KDF)**. HKDF, a standard example, is built internally using HMAC (Chapter 8) — a direct callback to the exact primitive that chapter established. This reduces how much raw entropy needs to be generated and carefully managed

directly, while still keeping keys used for different purposes (encryption, authentication, and so on) cryptographically separated from one another.

The Human Side — Access Control & the Principle of Least Privilege

`dbsec1-3`'s least-privilege account design applies directly to keys themselves — not every service or person needs access to every key a system uses. For especially high-value keys (a root CA key, for instance), real systems sometimes use **dual control** or **split-knowledge** schemes, requiring several independent custodians to jointly participate before a key can be used or reconstructed at all — the same principle underlying real root CA key ceremonies and much cryptocurrency custody today.

What Actually Breaks Key Management in Practice

Looking back across this entire course, nearly every "real-world" failure this course has covered was a key-management failure, not a mathematical one:

- Chapter 4 — Enigma's "cillies" (predictable settings) and message-key reuse.
- Chapter 5 — a reused stream-cipher key/nonce, reintroducing the two-time-pad vulnerability.
- Chapter 6 — a reused CBC initialization vector, leaking whether two messages start identically.
- Chapter 9 — a compromised static (non-ephemeral) private key retroactively exposing every past session that used it.
- Chapter 10 — Sony's 2010 ECDSA nonce reuse, directly leaking a signing private key.
- This chapter — the 2008 Debian OpenSSL entropy disaster, making entire classes of keys exhaustively guessable from the moment of generation.

> **THE THROUGHLINE, FULLY CLOSED**
>
> Chapter 1 stated it as a prediction; Chapter 4 demonstrated it concretely with Enigma. This chapter is the moment that prediction gets confirmed across the *entire* course at once: every cited real-world failure, from a 1930s rotor machine to a 2010 game console to a 2008 Linux distribution, traces back to key management and operational discipline — never to the underlying cryptographic mathematics failing.

Hands-On Exercises

EXERCISE 1

Explain why cryptographic key generation specifically requires a CSPRNG seeded from real OS-level entropy, rather than a "random-looking" but non-cryptographic PRNG (like the kind used to shuffle cards in a simple video game). What could go wrong if the latter were used to generate an AES key?

View solution

EXERCISE 2

Using the 2008 Debian OpenSSL disaster as your example, explain why sharply reducing the entropy feeding key generation doesn't just make keys "somewhat weaker" — explain conceptually why it can make an entire class of keys exhaustively enumerable.

View solution

EXERCISE 3

A company stores its signing key either (a) encrypted at rest inside a cloud KMS running as software on a general-purpose server, or (b) inside a dedicated HSM. If an attacker fully compromises the host server's operating system in both scenarios, what can they do in case (a) that they cannot do in case (b)?

View solution

CHAPTER 11 QUICK REFERENCE

- **Entropy** — keys need a CSPRNG seeded from real OS-level randomness; the 2008 Debian OpenSSL bug made keys exhaustively enumerable by shrinking the entropy pool
- **Storage** — never in source control; secrets managers (Vault, cloud KMS) are the modern standard, encrypted at rest
- **HSMs** — keys never leave the device in plaintext; operations happen inside it, protecting keys even from a fully compromised host
- **Rotation** — limits blast radius of an undetected compromise; ephemeral DH (Ch.9) is rotation taken to its logical extreme
- **KDFs (HKDF)** — derive many purpose-specific keys from one master secret, built on HMAC (Ch.8)
- **Least privilege** and **dual control/split-knowledge** apply directly to key access, not just data access (`dbsec1-3`)
- Nearly every real-world failure cited across this entire course was a key-management failure, never a mathematical one
- **Next chapter:** Capstone — Cryptography in the Real World, a post-quantum preview and a full worked system combining every prior chapter

CHAPTER 12 OF 12

Capstone: Cryptography in the Real World

Cryptography Fundamentals

Chapter 12 · Capstone — Cryptography in the Real World

The final chapter, in two parts: the post-quantum preview flagged back in Chapters 5 and 9, and a single worked scenario that combines every primitive this course has built, in the order a real system would actually use them.

Post-Quantum Cryptography — Preparing for Shor's Algorithm

Chapter 9 explained the asymmetry precisely: Shor's algorithm, on a sufficiently powerful quantum computer, breaks RSA, Diffie-Hellman, and ECC outright — not a manageable degradation like Grover's algorithm's effect on AES (Chapter 5), which can simply be countered with a larger key. Machines capable of running Shor's algorithm at the scale needed to threaten real-world key sizes don't exist yet — but "post-quantum cryptography" is the active, ongoing work of replacing this course's Chapter 9 algorithms with new ones built on *entirely different* hard math problems, believed resistant to both classical and quantum attack alike.

The leading approach is **lattice-based cryptography**. Mirroring Chapter 5's AES competition and Chapter 7's SHA-3 competition, NIST ran another open, public, multi-year competition (2016-2024) — a third demonstration of Chapter 1's Kerckhoffs's Principle at genuinely massive scale. The results, standardized in 2024: **ML-KEM** (formerly CRYSTALS-Kyber) for key exchange, and **ML-DSA** (formerly CRYSTALS-Dilithium) for signatures — direct post-quantum replacements for Chapter 9's Diffie-Hellman/ECDH and Chapter 10's RSA/ECDSA signatures respectively.

> **"HARVEST NOW, DECRYPT LATER" — WHY THIS MATTERS BEFORE QUANTUM COMPUTERS EXIST**
>
> An adversary can record encrypted traffic *today* and simply store it, waiting until a sufficiently powerful quantum computer eventually exists to decrypt it retroactively. For data that needs to stay confidential for years or decades — government secrets, long-term medical records, some financial data — this makes post-quantum migration an urgent *present-day* concern, not something to defer until quantum computers actually arrive.

A Full Worked System — Securing a Message End to End

Alice needs to send Bob a message, securely and verifiably, over an untrusted network. Here's every chapter of this course, contributing one piece each, in order:

```
1. KEY EXCHANGE (Ch.9)
   Alice and Bob perform ECDHE — ephemeral elliptic-curve Diffie-Hellman —
   deriving a shared secret with forward secrecy, using much smaller
   keys than RSA would need for equivalent strength.

2. KEY DERIVATION (Ch.11)
   That shared secret is fed into HKDF (built on HMAC, Ch.8) to derive
   two SEPARATE purpose-specific keys: one for encryption, one for
   authentication — rather than reusing one raw secret for everything.

3. AUTHENTICATED ENCRYPTION (Ch.5, Ch.6, Ch.8)
   The message is encrypted with AES-256-GCM, using a FRESH,
   never-reused nonce (Ch.6's warning). GCM's built-in GMAC tag
   (Ch.8) provides integrity and authenticity for this session in
   one step.

4. IDENTITY VERIFICATION (Ch.10)
   If Bob needs to confirm the message truly came from Alice
   specifically (not just "whoever Bob happened to key-exchange
   with"), Alice signs a hash of the message (Ch.7) with her own
   long-term private key. Bob verifies that signature using Alice's
   public key, itself vouched for by a certificate chain (Ch.10)
   rooted in a CA Bob's system already trusts.

5. KEY MANAGEMENT (Ch.11)
   Every key involved — Alice's long-term signing key especially —
   was generated via a proper CSPRNG, stored in a secrets manager or
   HSM rather than in code, and the ephemeral session keys from
   step 1 are discarded the moment the session ends.
```

This is, in essence, exactly what a modern TLS 1.3 connection with client certificate authentication does — every step traceable to a specific chapter of this course, none of it magic.

What This Course Doesn't Cover

In the interest of an honest accounting, matching the precedent this site's `ts5` course set for its own closing chapter: full protocol design (how TLS assembles these primitives into an actual wire protocol — that's `https1`'s subject specifically), cryptocurrency-specific cryptography, zero-knowledge proofs, homomorphic encryption, and formal, mathematically rigorous security proofs are all real, substantial topics this course intentionally left out of scope. This course is the foundation those subjects build on, not a replacement for studying them directly.

This Course's Throughline, Restated One Last Time

Chapter 1 predicted it; Chapter 4 demonstrated it with Enigma. By this final chapter, the pattern has repeated six times across nearly a century of real cryptographic history:

CHAPTER	REAL-WORLD CASE	WHAT ACTUALLY FAILED
4	Enigma, 1930s-40s	Predictable settings ("cillies"), message-key reuse — never the underlying math
5	Stream cipher misuse	Key/nonce reuse, reintroducing a two-time-pad break
6	CBC IV reuse	Predictable first-block leakage from a reused IV
9	Static key compromise	No forward secrecy — one stolen key exposes every past session
10	Sony PS3, 2010	ECDSA nonce reuse, directly leaking the signing private key
11	Debian OpenSSL, 2008	Collapsed entropy, making generated keys exhaustively enumerable

Six case studies, from a 1930s rotor machine to 2010s consumer electronics to open-source infrastructure — every one a key-management or operational failure, never a mathematical one.

> **THE ENIGMA DETOUR WAS NEVER JUST A HISTORICAL ASIDE**
>
> Chapters 3 and 4 established nearly every idea this course kept reusing afterward: Kerckhoffs's Principle in concrete action (Ch.1, Ch.3), polyalphabetic substitution as the direct conceptual ancestor of modern stream ciphers (Ch.2, Ch.3, Ch.5), and above all, the central lesson — reinforced six more times since — that real cryptographic systems fail operationally, not mathematically.

> **ONE LAST TIME: NEVER ROLL YOUR OWN CRYPTO**
>
> This warning appeared in Chapter 5 (ciphers), Chapter 8 (MAC construction), and implicitly throughout Chapter 11 (key handling) — worth stating once more, together: every primitive in this course reached its current, trusted form through open competition, years of public cryptanalysis, and real-world failures discovered and fixed in public. A custom, unreviewed design skips every part of that process.

Where to Go From Here

This course is the foundation underneath several others already on this site: `https1`'s entire handshake is now fully explainable, chapter by chapter, using nothing but this course's own primitives. `bc1-2`'s password hashing is Chapter 7 and Chapter 11 applied specifically to authentication. `dbsec1-5` / `dbsec1-6`'s encryption at rest and in transit are Chapter 5 and Chapter 6 applied specifically to databases. `owasp1-2`'s entire "Cryptographic Failures" category can now be read with real technical grounding behind every item on the list, rather than as an abstract warning.

Hands-On Exercises

EXERCISE 1

Explain "harvest now, decrypt later" in your own words, and explain why it means post-quantum migration is an urgent present-day concern even though no quantum computer capable of running Shor's algorithm at threatening scale currently exists.

View solution

EXERCISE 2

In this chapter's worked Alice-and-Bob system, if a large-scale quantum computer became practical tomorrow, which specific step(s) would need to be replaced immediately, and which step(s) would be comparatively unaffected? Explain why, referencing Chapter 9's quantum asymmetry.

View solution

EXERCISE 3

Pick any two of the six real-world case studies from this chapter's closing table (Enigma, stream cipher reuse, CBC IV reuse, static key compromise, Sony's ECDSA reuse, the Debian OpenSSL bug). Despite being separated by decades and completely different technologies, explain in your own words what they have in common structurally.

View solution

CHAPTER 12 QUICK REFERENCE

- **Post-quantum crypto** — new algorithms (ML-KEM, ML-DSA) built on different hard problems, standardized by NIST in 2024 after another open competition
- **"Harvest now, decrypt later"** — recorded ciphertext today can be decrypted retroactively once quantum computers arrive, making migration urgent now
- **Full worked system:** ECDHE (Ch.9) → HKDF (Ch.11/Ch.8) → AES-256-GCM (Ch.5/6/8) → signature + certificate chain (Ch.7/10) → proper key handling throughout (Ch.11)
- Out of scope: full protocol design, cryptocurrency-specific crypto, zero-knowledge proofs, homomorphic encryption, formal security proofs
- Six real-world case studies (Ch.4, 5, 6, 9, 10, 11) — every one a key-management/operational failure, never a mathematical one
- This course underlies `https1`, `bc1-2`, `dbsec1-5/6`, and `owasp1-2` directly
- **Course complete** — Cryptography Fundamentals, 12 chapters, from Caesar's shift cipher to post-quantum lattice cryptography