

Modern Symmetric Cryptography

SIO

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

João Paulo Barraca

Welcome to this lecture on modern symmetric cryptography. The single word that organises everything here is symmetric: it means the same secret key is used to both encrypt and decrypt. That makes it the oldest and the fastest way to keep data confidential, and it is the engine sitting under almost all bulk encryption you will ever meet. Wherever large amounts of data need to be kept secret at a reasonable cost, symmetric cryptography is doing the work.

We will build the vocabulary first (cryptography, cryptanalysis, cryptology), then look at how a cipher actually works: you choose an algorithm and feed it a key, and it turns plaintext into ciphertext and back. Along the way we meet the two families of symmetric ciphers, stream and block, and the famous examples, DES, AES, RC4 and ChaCha20. We then spend a large part of the lecture on something that surprises many students: the cipher is only half the story. A block cipher must be run in a mode of operation (ECB, CBC, GCM, XTS), and choosing the wrong mode can be just as fatal as choosing a weak cipher.

The lens to carry through the whole lecture is that security is the product of the algorithm, the key, the mode, and the randomness. A weak choice anywhere, an ECB mode, a reused IV, a low-entropy key, collapses the entire system even if the algorithm itself is strong. The practical stakes are high: TLS on the web, full-disk encryption, and mobile and IoT communication all rely on these ideas. By the end you should be able to look at a real encryption choice and judge not just whether the cipher is good, but whether it is being used well.

A practical orientation to hold onto: by the end, the goal is not to memorize cipher names but to reason about choices. When you open a config or a codebase and see an encryption call, you will want to be able to ask the right questions: which cipher, which mode, where did the key come from, is the IV fresh, is integrity covered. This lecture builds the vocabulary and the intuition to ask those questions and to recognize when the answers are weak. That is the difference between someone who trusts that it is encrypted and someone who can verify that the encryption is actually sound.

Lecture notes were created with Gen AI and reviewed by faculty. If you find an error, please report it.

Terminology

- **Cryptography**
 - Art or science of hidden writing (confidential writing)
 - From Gr. *kryptós*, hidden + *graph*, r. *de graphein*, to write
 - Initially used to enforce the confidentiality of information
- **Cryptanalysis**
 - Art or science of breaking cryptographic systems or encrypted information
- **Cryptology**
 - Cryptography + cryptanalysis

Start with the words, because they frame the whole field. Cryptography literally means hidden writing, from the Greek *kryptos*, hidden, and the root of *graph*, to write. It began as a practical art of confidentiality: how do I send a message so that only the intended reader can understand it? For most of human history that was a military and diplomatic concern, and much of the vocabulary we use today carries that origin.

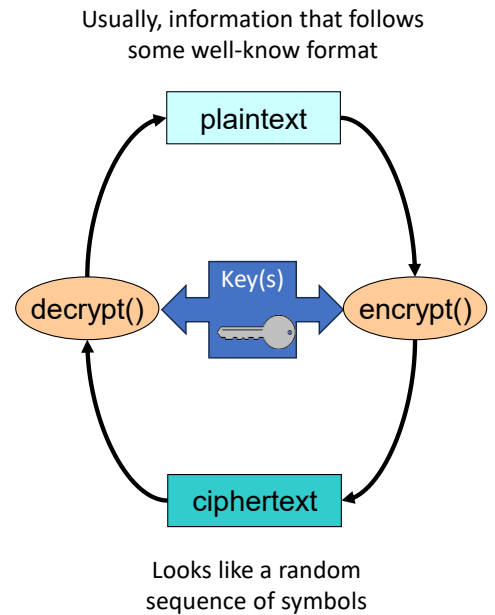
Cryptanalysis is the other half of the field: the art and science of breaking cryptographic systems, or of decrypting a message without the proper key. It is the adversary's perspective. Cryptology is the umbrella term that covers both cryptography and cryptanalysis. This pairing is essential, because the two disciplines are not separate activities but two sides of the same coin, and the people who do one usually understand the other deeply.

The productive tension between them is the key idea to internalise. We never truly prove a cipher is safe; instead we accumulate evidence that the best known attacks fail. So when we say a cipher is strong, we mean that cryptanalysis has not found a shortcut and that brute force remains infeasible. This adversarial framing is what drives the field forward and is exactly the standard we will use to evaluate every system in this lecture.

One more framing that will pay off: because cryptanalysis is part of the definition of the field, every secure claim in this lecture should be read as secure against every attack we know, and brute force remains infeasible. That is an empirical, falsifiable stance, not a proof. It also means the field is inherently adversarial: the best practitioners think like attackers. If you carry that mindset, you will evaluate systems the way their designers hoped, and you will spot the hidden assumptions that make a system weaker than it claims to be.

Cryptography: how does it work?

- Select a cipher (or cipher algorithm)
 - Specific cryptographic technique
 - Apply the cipher with a key
 - **Encryption:** original information → cryptogram
 - **Decryption:** cryptogram → original information
 - **Key:** algorithm parameter
 - Influences algorithm execution
- Original information aka plaintext or cleartext
- Cryptogram aka ciphertext



A cryptographic system is built from two pieces. The first is the cipher algorithm, a precise, deterministic procedure, the exact sequence of mathematical operations that performs the transformation. The second is the key, a parameter that feeds into that procedure and steers it. Encryption runs the algorithm forward with the key to turn the original information, the plaintext or cleartext, into a cryptogram, the ciphertext; decryption runs it backward to recover the plaintext. The beauty of the design is that the algorithm is public and fixed, while the key is what varies.

The key is what makes the whole thing flexible and secure. The same algorithm produces completely different ciphertexts for different keys, and it is the secrecy of the key, not of the algorithm, that we ultimately rely on. That division of labour is deliberate: it lets us openly review and harden the algorithm while concentrating all the secrecy into one manageable secret, the key. It is the foundation of the principle we will discuss next.

Notice the contrast between the two ends of the process. The plaintext is usually data with a well-known structure, a text file, an image, a network packet, with patterns and regularities that a human or a program can read. The ciphertext, by design, should look like an unintelligible, random-looking sequence of symbols with no readable structure. That readable-input-versus-noise-like-output gap is the goal of every cipher we will meet, and the quality of that gap is a useful way to judge a cipher's effectiveness.

A useful way to hold the two pieces apart: the algorithm is a public recipe that everyone can read and check, while the key is the private ingredient that makes the result unique. If you imagine the algorithm as a lock's publicly documented design and the key as the only matching object, you can see why the security effort concentrates on the key. Every later idea, from Kerckhoffs's principle to the emphasis on random keys and IVs, is a consequence of this split between public algorithm and secret key. Keep it in mind, because it is the load-bearing idea of the entire lecture.

Use cases for (symmetric) ciphers

- Self protection with secret key **K**

- Alice encrypts plaintext **P** with key **K**
- Alice decrypts ciphertext **C** with key **K**
- **P'** should be equal to **P** (requires checking)
- Only Alice needs to know **K**

→ Alice: $C = \{P\}_K$
→ Alice: $P' = \{C\}_K$

- Secure communication with secret key **K**

- Alice encrypts plaintext **P** with key **K**
- Bob decrypts ciphertext **C** with key **K**
- **P'** should be equal to **P** (requires checking)
- **K** needs to be known by Alice & Bob

→ Alice: $C = \{P\}_K$
→ Bob: $P' = \{C\}_K$

There are two basic use cases, and they differ in who needs the key. In the self-protection case, Alice encrypts data with a secret key and later decrypts it herself; only she needs to know the key, so it is a single-user confidentiality scenario, much like locking a file on a disk that only you ever open. In the secure-communication case, Alice encrypts with the key and Bob, who shares that same key, decrypts; now both parties must know the key and must have established it beforehand, which is a fundamentally harder problem.

That shared-secret requirement is exactly the practical difficulty of symmetric cryptography, and the reason key distribution is such a central theme across the whole course. The mathematics of encryption is easy; the real engineering challenge is getting the same secret key to all the right people without an eavesdropper seeing it. This is precisely the gap that asymmetric cryptography, which we touch on later, is designed to fill, which is why real systems combine the two.

In both cases there is a small but important detail: the plaintext recovered after decryption must be checked to be equal to the original. A corrupted or tampered ciphertext will decrypt to garbage, and the receiver needs a way to notice that something went wrong. This quietly introduces a second security goal beyond confidentiality, integrity. A cipher only guarantees that the data is secret, not that it has not been altered, and distinguishing those two goals will matter again when we reach the authenticated modes.

The contrast also shows up in the real world in a specific way. Self-protection, where a single user locks their own data, is behind full-disk encryption and encrypted file containers; the key is tied to the device or to a passphrase the owner knows. Secure communication, where two parties share a key, is behind TLS and messaging protocols, and it is where the hard key-distribution problem bites hardest. Recognizing which of the two you are dealing with tells you where the security burden really lies, and why the communication case needs extra machinery, usually asymmetric, to bootstrap the shared secret in the first place.

Goals of cryptanalysis

- **Reveal the plaintext hidden in a ciphertext**
 - Usually requires discovering the key that produced the ciphertext
- **Sometimes requires discovering the cipher algorithm**
 - Usually algorithms are not secret, but there are exceptions
 - Sometimes using reverse engineering
 - Lorenz, A5 (GSM), RC4 (WEP) , Crypto-1 (Mifare)
 - Algorithms for DRM (Digital Rights Management) in Videos and Games

The primary goal of cryptanalysis is to recover the hidden plaintext, and in practice that usually means first discovering the key that produced the ciphertext. But there is a second, subtler goal that is easy to overlook: identifying the cipher algorithm itself. We tend to assume, and the next principle says we should, that algorithms are public. Yet history is full of systems that shipped with secret, proprietary algorithms, and those secrets eventually fell through reverse engineering or outright analysis.

Classic examples include the Second World War Lorenz cipher, the GSM A5 stream ciphers, the RC4 cipher used in old WEP Wi-Fi, and the Mifare Crypto-1. Digital-rights-management schemes in video and games are a recurring target precisely because their algorithms are deliberately hidden. In each case the secrecy bought only a short delay, not real security, and once the design leaked it was broken relatively quickly. The pattern is consistent and instructive.

The lesson is that keeping an algorithm secret is a losing long-term strategy. Secrets spread, people leave companies, devices are disassembled, and obscurity decays over time. This is exactly why the modern approach is to design for a world where the attacker already knows the algorithm, and to put all the security into the key. Understanding this history is not a curiosity; it explains the foundational design principle we are about to state, and it is a warning against the temptation to hide rather than to design.

It is worth adding the reverse-engineering angle, because it is how most secret algorithms actually fall. A device that uses a hidden cipher can be opened, its firmware extracted, and the algorithm recovered; that is how A5, RC4 in WEP, and Crypto-1 were all exposed. Reverse engineering is not a theoretical risk but a routine part of modern cryptanalysis, especially for consumer hardware. This is the concrete mechanism behind the general principle: a secret that lives in a shipped device is, in practice, not secret for long, which is why relying on it is a poor foundation for security.

Kerckhoffs's principle

- A cipher must be secure even if **everything about it, except the key, is public**
 - Stated by Kerckhoffs in 1883; also Shannon's maxim: 'the enemy knows the system'
 - Security depends only on the secrecy of the key
- Security must **not rely on keeping the algorithm secret**
 - Secret algorithms hide weaknesses behind obscurity
 - Once leaked they are often broken (Lorenz, A5, RC4, Crypto-1)
- Public algorithms are scrutinised, tested and improved **by the community**
 - Open review finds weaknesses before deployment
 - AES was selected after a public international competition
- Contrast with 'security by obscurity': **secret algorithms are fragile**
 - Weaknesses remain hidden until reverse engineering
 - DRM schemes are typically broken shortly after release

SIO

This is the most important design principle in all of cryptography. Kerckhoffs stated it in 1883: a system should remain secure even if everything about it, the algorithm, the design, the implementation, is known to the enemy, with the single exception of the key. Shannon later phrased it as the enemy knows the system, and it became the standard against which all modern ciphers are judged. It is a strong claim, and it is one of the reasons modern cryptography is both robust and trustworthy.

The intuition is practical rather than philosophical. The key is something you can keep secret, change regularly, and distribute carefully to a small number of parties. An algorithm, by contrast, is hard to keep secret: it is embedded in millions of devices, implemented by many vendors, and eventually someone reverse-engineers it. Relying on algorithm secrecy, called security by obscurity, is fragile, because the weaknesses stay hidden only until the design leaks, and then they are exposed all at once.

Public algorithms, by contrast, are attacked, tested and improved by the entire cryptographic community before they are deployed. The more eyes on a design, the more likely a flaw is found by defenders rather than discovered by attackers later. The AES standard was chosen through exactly such a public international competition, in which candidate ciphers were openly critiqued for years. So the principle is not an ideal we aspire to but a deliberate risk-management decision: open review turns a secret that will eventually fail into a design that has already survived a global siege.

The principle also has a practical consequence for how new ciphers get approved. Because we assume the algorithm will be public, the standard process is to publish it and let the community attack it before it is trusted. That is how AES came from a public competition, and it is why a new cipher should be considered experimental until it has survived that public scrutiny. So the principle is not only a design rule; it is a governance rule too. It tells you both how to build a cipher and how to judge whether a given cipher has earned its place as a standard you can rely on.

Cryptanalysis attacks

- Brute force
 - Exhaustive search of the key space until finding a match
 - GPUs are great for this!
 - Usually unfeasible for large key spaces
 - Key space: set of all possible keys with the same size
 - e.g., 128-bit keys allow a key space of 2^{128} values
 - Key randomness is fundamental!
 - Means that any key has the same probability of being the right one
- Clever attacks
 - Reduce the search space to a smaller set of potential candidates
 - Words, numbers, restricted size or alphabet
 - Identify patterns in different operations, etc.

Attacks fall into two broad families, and distinguishing them is useful. Brute force, or exhaustive key search, simply tries every key in the key space until the ciphertext decrypts correctly. The key space is the set of all possible keys of the same size, so a 128-bit key gives 2 to the 128 possible values. For large key spaces brute force is computationally infeasible, which is exactly why long keys are the core defence. It is also why key randomness is fundamental: if the key is not uniformly random, then it is not truly a full-strength key in the attacker's eyes.

Think about the scale. 2 to the 128 is an almost incomprehensible number; even with enormous parallel hardware, brute-forcing AES-128 is far beyond reach with any foreseeable technology. That is the quiet power of a long, random key. But the defence depends on the key being genuinely random. A key built from a word, a short number, or a weak random source has a far smaller effective key space, and brute force suddenly becomes realistic. This connects directly to the key-generation slides we will see later.

The second family is clever cryptanalysis, where instead of trying all keys the attacker looks for structure. They exploit weak choices such as words, short or restricted keys, or small alphabets, and they look for patterns across many encryption or decryption operations to narrow the set of candidates. This is where understanding the mathematical structure of a cipher pays off: a well-designed cipher has no exploitable patterns, while a weak one gives the attacker a shortcut that beats brute force. The practical takeaway is a double one: use long random keys, and use ciphers that have withstood structural analysis.

A concrete way to see why both families matter: the DES key was small enough that brute force, done by a purpose-built machine, was feasible, which is the brute-force family in action. But many ciphers fall not to brute force but to a clever structural attack that finds a shortcut, the clever family. A cipher can be safe against one and vulnerable to the other. This is why evaluating a cipher requires asking two separate questions: is the key space large enough to make brute force infeasible, and does the algorithm have any exploitable structure? Only when both answers are favorable can you trust the design.

Cryptanalysis: attack models

- **Ciphertext-only:** attacker has access only to ciphertext
 - Weakest model; hardest to exploit
 - Relies on statistical structure or cipher weaknesses
 - Classical substitution ciphers are broken this way
- **Known-plaintext:** attacker knows some plaintext/ciphertext pairs
 - E.g. standard headers, protocol fields, known file formats
 - Enables deriving the key or the keystream
- **Chosen-plaintext:** attacker can choose plaintext and obtain its ciphertext
 - Attacker feeds chosen inputs to the encryption oracle
 - Reveals relationships between input and output
- **Chosen-ciphertext:** attacker can choose ciphertext and obtain its plaintext
 - Attacker queries the decryption oracle
 - Basis of padding-oracle and malleability attacks

SIO

Cryptographers classify attacks by how much the attacker can obtain, from the weakest to the strongest, and it is worth being precise because it tells you how much you can trust a cipher. Ciphertext-only is the weakest and hardest case: the attacker has only the encrypted data and must rely purely on statistical structure or a cipher weakness. This is how classical substitution ciphers fall, because the letter frequencies of the language survive in the ciphertext.

Known-plaintext means the attacker has some matched plaintext and ciphertext pairs, often from predictable content such as standard headers, protocol fields, known file formats, or even a guess about what the beginning of a message might say. With enough matched pairs the attacker can often recover the key or the keystream. Chosen-plaintext goes further: the attacker can feed inputs of their own choosing to an encryption oracle and study the resulting ciphertexts, exposing the relationship between input and output.

Chosen-ciphertext is the strongest model, where the attacker can query a decryption oracle directly; this is the basis of powerful padding-oracle and malleability attacks, some of which have broken real systems. The pattern is clear: the higher you go, the more the attacker controls, and the more the cipher must resist. So when someone says a cipher is secure, the sharp question is secure against which model. A cipher that resists ciphertext-only but fails against chosen-ciphertext is not safe for a protocol that exposes a decryption function, and this is exactly the kind of reasoning we apply when choosing modes later.

The practical takeaway is that the model you are exposed to depends on the protocol, not just the cipher. If your system offers a decryption function, a padding function, or a way to check whether a message is valid, you have given an attacker an oracle, and you are in the chosen-plaintext or chosen-ciphertext world, where far more is possible. This is why protocol design matters as much as cipher choice: many real breaks come not from a weak cipher but from a protocol that inadvertently hands the attacker an oracle. When you review a system, look specifically for any endpoint that decrypts or validates data for an untrusted caller.

Computer ciphers

- Operate by making substitutions
 - Original information is a sequence of **symbols**
 - Each symbol is replaced by a **substitution symbol**
 - Usually with the same size
 - **Polyphonic substitution**: several, larger substitution symbols for each original symbol
 - Substitution symbols are picked from a **substitution alphabet**
- Usual symbols
 - Bit
 - Block of bits
- Strategies
 - **Monoalphabetic** substitution: key → one substitution alphabet
 - **Polyalphabetic** substitution: key → several substitution alphabets (one for each symbol)

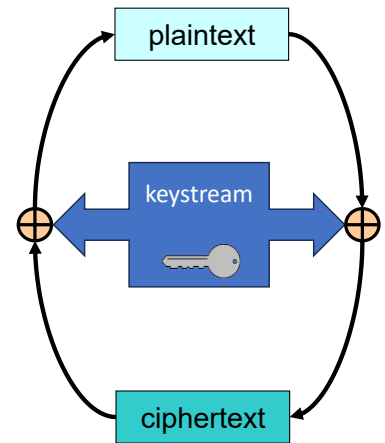
Modern computer ciphers are fundamentally about substitution: the original data is a sequence of symbols, and each symbol is replaced by another. Usually the substitution symbol is the same size as the original, though a polyphonic substitution maps one original symbol onto several possible, larger ones. The replacement symbols are drawn from a substitution alphabet, and in digital systems the basic symbols are typically a single bit or a block of bits. This substitution framing is the bridge from the classical paper world to the digital one.

The strategy is set by the key. A monoalphabetic substitution uses one fixed substitution alphabet, while a polyalphabetic substitution uses several, choosing one per symbol so that the same plaintext symbol is not always mapped the same way. That last property, that the same input is not always treated identically, is what defeats simple frequency analysis, and it is the idea that stream ciphers generalise to the bit level. Understanding the mono versus poly distinction is a key that unlocks the rest of the lecture.

Here is the unifying observation worth pausing on: stream ciphers turn out to be polyalphabetic (a fresh keystream symbol per bit), block ciphers are monoalphabetic (one big substitution table defined by the algorithm and key), and even asymmetric ciphers can be viewed as a kind of homophonic substitution. So what look like three very different mechanisms are really variations on one theme of substitution. Whenever you meet a new cipher, asking what kind of substitution it really is often predicts both its strengths and its weaknesses, and it keeps the whole landscape from feeling like a disconnected list of names.

Computer ciphers: stream ciphers

- **Encrypt/decrypt by mixing streams**
 - They consider the data to cipher or decipher as a bit stream
 - Each plaintext/ciphertext bit is **XORed** ($\oplus$) with each keystream bit
 - Usually explored in low-level communication protocols
- **Polyalphabetic ciphers**
 - Each bit (0 or 1) is not always encrypted the same way
- **Keystream**
 - Randomly produced, as long as the processed data
 - Vernam cipher (or **one-time pad**)
 - The only perfect cipher (but rarely used, very impractical)
 - Pseudo-randomly produced from a limited key
 - Ordinary stream ciphers



A stream cipher treats the data as a continuous stream of bits. It produces a keystream, a sequence of bits ideally as long as the data, and encrypts by XOR-ing each plaintext bit with the corresponding keystream bit. Because XOR is its own inverse, the exact same operation decrypts, which is an elegant and practical property: encryption and decryption are literally the same circuit. This makes a stream cipher a polyalphabetic cipher in the sense we just met, because the same plaintext bit is not always encrypted the same way, it meets a different keystream bit each time.

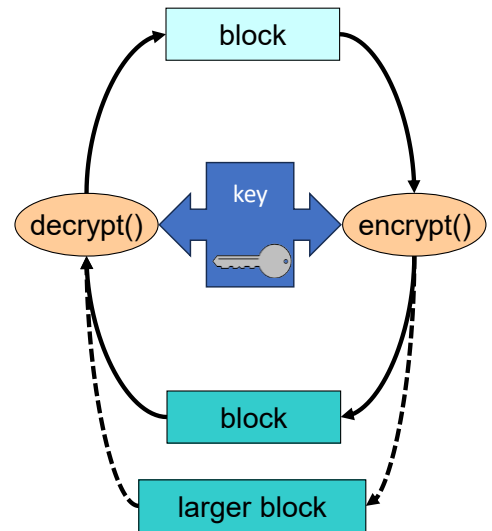
The ideal keystream is truly random and used only once. That is the Vernam cipher, or one-time pad, the only information-theoretically perfect cipher, meaning it cannot be broken even with unlimited computing power, provided the key is truly random, at least as long as the message, and never reused. In practice the one-time pad is rarely usable, because it requires generating, securely transmitting and destroying a secret random key as long as the data, which is impractical for most applications.

Real stream ciphers instead use a pseudo-random generator seeded by a short key, which is far more convenient but no longer provably perfect; its security now depends on the quality of that generator. This is a recurring theme: a stream cipher's strength is entirely the strength of its keystream generator. If the generator is predictable, the keystream is predictable, and the whole XOR protection collapses. So the rest of our discussion of stream ciphers is really a discussion of how to build a keystream generator that is as close to truly random as we can make one, and how to use it without breaking that guarantee.

The one-time pad also teaches a deeper lesson about the source of security. Its perfect secrecy comes not from a clever algorithm but from the key being genuinely random and as long as the message. The moment you relax either condition, reuse the key or let the keystream be pseudo-random, you move from provable security to security-by-the-quality-of-the-generator. That is the whole arc of real stream ciphers: practical compromises that trade the one-time pad's unattainable guarantees for a short key and a strong generator. Understanding that trade, rather than just the XOR, is the key to reasoning about every stream cipher we meet.

Computer ciphers: block ciphers

- Encrypt/decrypt sequences of blocks
 - Symbols are fixed-length blocks of bits
 - Usually use byte blocks as symbols
- Are monoalphabetic substitution ciphers
 - Transform each symbol into another symbol
 - The key defines the transformation (substitution alphabet)



A block cipher, in contrast, works on fixed-size chunks. The data is split into blocks of a fixed length, in practice usually whole bytes such as 64 or 128 bits, and each block is transformed into another block of the same size. In substitution terms this is a monoalphabetic substitution: there is one big mapping from input blocks to output blocks, and the key determines that mapping. This is a fundamentally different structure from a stream cipher, and it leads to different strengths, weaknesses and uses.

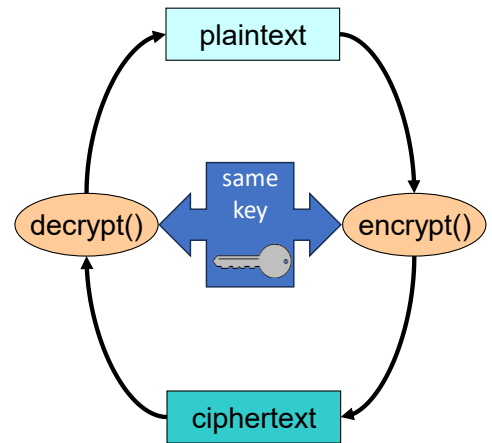
Because the transformation is a single deterministic function, the same plaintext block always gives the same ciphertext block under the same key. That sounds like a flaw, and on its own it is: encrypt the same block twice and you get the same ciphertext twice, leaking structure. The resolution is not to change the block cipher but to change how the blocks are combined, which is exactly what cipher modes solve later. So the monoalphabetic nature of a block cipher is not the end of the story; it is the starting point that motivates chaining modes.

Some ciphers are also homophonic substitutions: they deliberately assign multiple possible output symbols to a single plaintext symbol, usually by randomizing the process, so that identical plaintext does not always yield identical ciphertext. That randomization can hide patterns, but it also complicates decryption and requires extra state to be shared. Block ciphers are the workhorses of modern encryption, and the rest of the lecture focuses on two things: how a block cipher is built internally, and how it is chained together into secure modes of operation.

The fixed-block structure also explains a practical constraint you will meet constantly: the data must be a multiple of the block size, which is where padding comes from, and it is why block ciphers need modes to handle arbitrary-length messages. A stream cipher, by contrast, naturally consumes a byte or bit stream with no alignment concern. So one of the practical differences between the two families is not just speed but shape: block ciphers work in neat chunks and need help with ragged edges, while stream ciphers flow continuously. That difference drives several of the design choices, including padding and the choice of mode, that we will meet next.

Computer ciphers: symmetric

- Encrypt/decrypt with the same key
 - The oldest strategy
- Also called **secret key ciphers**
- Most common mechanism to provide **confidentiality**
 - Relatively simple to be implemented in software and hardware
 - Very good performance
 - Widely available across systems and platforms



Symmetric ciphers are the oldest strategy: the same key is used for both encryption and decryption. They are also called secret-key ciphers, and they are the most common practical mechanism for providing confidentiality. The reasons are practical and hard to beat. They are relatively simple to implement, both in software and in hardware, they are very fast, and they are widely available across virtually every system and platform, from data-center servers to battery-powered sensors.

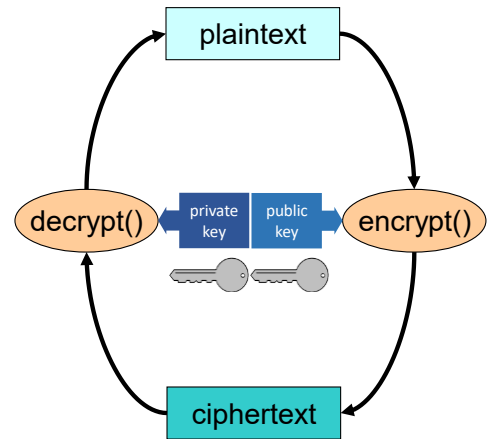
For bulk data encryption, files on disk, network streams, mobile communication, symmetric ciphers are almost always the tool you reach for. Asymmetric schemes, the two-key approach we touched on earlier, are far slower, often by orders of magnitude, so they are used only where their unique ability to work without a shared secret is needed. The standard architecture in real systems is to use the fast symmetric cipher for the data and the slow asymmetric scheme only to exchange or protect the short symmetric key. That combination is the reason both families coexist.

The catch, as we saw in the use cases, is that all parties must share the same secret key, so the whole scheme depends on being able to establish and keep that key secret. That shared-secret requirement is what makes symmetric cryptography both the default choice and the source of its main operational difficulty. It is a beautiful trade: the simplest and fastest mathematics in exchange for the hard problem of getting the key to the right people. Keeping that trade in mind helps you understand why key management is treated with so much care in every real system.

The speed advantage is not just a convenience; it is why symmetric ciphers are the default for the data itself in almost every secure system. Even TLS, which uses asymmetric cryptography to set up the connection, switches to a symmetric cipher for the actual data transfer, because encrypting a network stream with an asymmetric scheme would be far too slow. So the pattern you will see again and again is asymmetric to establish trust and exchange a key, then symmetric to move the data. Once you internalize that division of labour, the role of each family becomes obvious rather than something to memorize.

Computer ciphers: asymmetric

- Encrypt/decrypt with two different keys
 - Private component, public component
 - Public computed from private (or from secret data)
- An approach that was first published in 1978
 - Diffie–Hellman 1976; RSA invented 1977, published 1978
- Different algorithms work in different ways
 - Mostly public and we



Asymmetric ciphers use two different keys: a public one and a private one, with the public key derivable from the private one, or from some secret data. The private key stays secret; the public key can be shared openly, which removes the need to pre-share a secret. This was a genuine breakthrough, first published in 1978, with Diffie–Hellman in 1976 and RSA, invented in 1977, published in 1978. It solved the key-distribution problem that had limited symmetric cryptography.

The trade-off is that asymmetric operations are much slower than symmetric ones, often by orders of magnitude. That is why real systems almost always combine the two: use the fast symmetric scheme to protect the bulk of the data, and use the slower asymmetric scheme only to exchange or protect the short symmetric key. This hybrid design, asymmetric for the handshake and key exchange, symmetric for the data, is the backbone of secure communication protocols such as TLS. It is the best of both worlds.

Different asymmetric algorithms achieve the two-key relationship in different mathematical ways, each relying on a hard problem, and we will explore those in a later lecture on asymmetric cryptography. For now the key point is the role asymmetric ciphers play in this symmetric-focused lecture: they solve the key-distribution problem that plagues the symmetric ones. So even though we will spend most of our time on symmetric ciphers, you should hold in mind that the two families are complementary, and that the shared-secret requirement we keep coming back to is precisely what the asymmetric family is built to remove.

It is also worth noting the practical implication for this lecture: because asymmetric is slow, you should never use it to encrypt bulk data directly. The correct pattern is to use it to protect a short symmetric key or a random value, and then use that symmetric key for the actual data. This hybrid approach is the reason the two families are complementary rather than competing, and it is why a full understanding of symmetric cryptography is a prerequisite for understanding how secure protocols actually work end to end. The two halves only make sense when you hold both.

Computer ciphers: combinations

- (Symmetric) stream ciphers
 - Polyalphabetic ciphers
 - Keystream defined by the key
 - Keystream and XOR implement a polyalphabetic transformation
- Symmetric block ciphers
 - Monoalphabetic ciphers
 - Substitution alphabet is defined by the algorithm & key
- Asymmetric (block) ciphers
 - Homophonic substitution ciphers
 - Not by nature, but for security reasons
 - The functionalities of these ciphers are not homogeneous

This is a conceptual map that ties the substitution idea back to the three cipher families, and it is a good moment to consolidate what we have built. Symmetric stream ciphers are polyalphabetic: the key defines a keystream, and the XOR with that keystream implements a polyalphabetic transformation where each bit meets a different substitute. Symmetric block ciphers are monoalphabetic: the algorithm and key together define one substitution alphabet, the block-to-block mapping, which is why chaining modes are needed to hide repeated blocks.

Asymmetric block ciphers can be seen as homophonic substitutions, not by their mathematical nature, but for security reasons, because their operation is randomized so that the same input can yield different outputs. The reason for presenting the landscape this way is to show that what look like very different mechanisms are really variations on a single theme of substitution, differing only in whether the substitution is mono-, poly- or homophonic. Once you see that, the families stop being a list to memorise and become a coherent spectrum.

This is a useful lens for the rest of the course. Whenever you meet a new cipher, you can ask what kind of substitution it really is, and that question often predicts both its strengths and its weaknesses. A monoalphabetic scheme will leak repeated blocks, so it needs chaining; a polyalphabetic scheme hides those repetitions but depends on a good keystream; a homophonic scheme adds randomization at some cost in complexity. Asking the substitution question is a cheap but powerful diagnostic, and it will make the specific ciphers and modes that follow much easier to place in the bigger picture.

The reason to spend time on this map is that it is the organizing principle for the rest of the course, and it will keep recurring. Whenever you later meet a specific cipher or mode, you can slot it into this substitution-based classification and immediately predict its behavior. A monoalphabetic scheme will repeat, so it needs chaining; a polyalphabetic one hides repetitions but depends on a keystream; a homophonic one adds randomization at a cost. The map is small, but it is a powerful diagnostic, and the payoff is that the many specific ciphers and modes that follow become a set of predictable variations rather than a disconnected list to memorize.

Techniques used by ciphers

- **Confusion**

- Complex relationship between the key, plaintext and the ciphertext
- Output bits (ciphertext) should depend on the input bits (plaintext + key) in a very complex way

- **Diffusion**

- Plaintext statistics are dissipated in the ciphertext
 - If one plaintext bit toggles, then the ciphertext changes substantially, in an unpredictable or pseudorandom manner
- Creates an avalanche effect

Almost all good ciphers, and especially block ciphers, are built from two ingredients, named by Shannon. Confusion creates a complex relationship among the key, the plaintext and the ciphertext, so that the ciphertext appears to depend on the key in a very intricate way; an attacker should not be able to deduce the key from the ciphertext. Diffusion spreads the statistical structure of the plaintext across the whole ciphertext, so that changing a single plaintext bit causes a substantial, unpredictable change in the ciphertext, the so-called avalanche effect.

Think of the two goals as answering two different questions. Confusion answers how the key influences the output, and it makes that influence as opaque as possible. Diffusion answers how the plaintext is distributed in the output, and it makes a tiny change in the input ripple through the whole result. Between them they destroy the statistical regularities that cryptanalysis would otherwise exploit. A cipher with good confusion but poor diffusion leaks structure; one with good diffusion but poor confusion leaks key relationships. You need both.

Every round of a modern block cipher is essentially an application of confusion and diffusion, which is why these two words are the vocabulary for reading a cipher's design. When you later look at AES, you will recognise the S-box as the source of confusion (nonlinear substitution) and the mixing and shifting steps as the source of diffusion (spreading changes across the state). Understanding these two goals first, before the specific mechanisms, lets you understand why each part of AES exists, rather than treating it as an arbitrary sequence of operations.

A concrete way to see why both are needed: imagine a cipher with strong confusion but weak diffusion. Then the key relationship is opaque, but a change in one input bit stays local in the output, so the attacker can isolate and study small pieces. Now imagine the opposite, strong diffusion but weak confusion. Then changes spread everywhere, but the way the key influences the output is simple, so the attacker can recover the key. Each failure mode is a real, exploitable weakness. The point is that confusion and diffusion are complementary, and a strong cipher must have both, which is why every good block cipher devotes explicit structure to each.

(Symmetric) stream ciphers: examples

- A5/1, A5/2
 - Cellular communications
 - Initially secret, reverse engineered
- E0
 - Bluetooth communications
 - Keys up to 128 bits
- RC4
 - Wi-Fi communications (WEP, deprecated)
 - Initially secret, reverse engineered, never officially published
 - Keys with 40 to 2048 bits
- Other
 - Salsa20, Chacha20, etc.

A5/1 and A5/2 protect cellular (GSM) voice traffic; they were originally kept secret and were later reverse engineered. E0 protects Bluetooth communication, with keys up to 128 bits. RC4 was the stream cipher behind Wi-Fi's WEP, now deprecated, and behind many other systems; it too was secret at first, was reverse engineered, and was never officially published, with key sizes from 40 up to 2048 bits.

RC4 is a cautionary tale: once its design leaked, weaknesses were found that led to its retirement from most protocols.

Then there is the modern generation, Salsa20 and its improved successor ChaCha20, which are designed for fast, secure software implementation without relying on hardware acceleration. They are now central to current protocols and are a strong choice where AES-NI is not available. The contrast between the two generations is instructive: the older, historically important ciphers were often kept secret and then broken once their design became known, while the newer ones are public by design and have survived years of public scrutiny.

The pattern worth internalising is that secrecy correlates with fragility, and openness correlates with durability. That is Kerckhoffs's principle playing out in real systems. When you are choosing or evaluating a stream cipher, the age and public track record of the design matter as much as its speed or key size. A cipher that has been publicly attacked for a decade without being broken is a far safer bet than a newer secret design that no one has had a chance to examine, no matter how clever it looks on paper.

There is a broader industry lesson hidden in this list. The older ciphers that were kept secret tended to be broken once their design became known, while the newer ones, designed openly, have endured. This is not a coincidence; it is Kerckhoffs's principle at the level of industry history. It gives you a practical heuristic for evaluating any cipher: a design that has been public and attacked for years without being broken is a far safer bet than a newer or secret design whose weaknesses no one has yet found. The track record under open scrutiny is one of the strongest signals of a cipher's real strength.

(Symmetric) stream ciphers: approach

- Use a cryptographically secure, pseudo-random bit **generator**
 - This generator produces the keystream
 - The generator implements a state machine
 - The generator is usually controlled by two values:
 - **Initialization Vector** (defines the initial state of the state machine)
 - **Key** (defines how one state moves to the next to produce the keystream)
- Cryptographically secure, pseudo-random means:
 - Statistically, the keystream looks like a totally random sequence of zeros and ones
 - If an attacker learns a part of the keystream, it cannot infer:
 - Past keystream values
 - Future keystream values

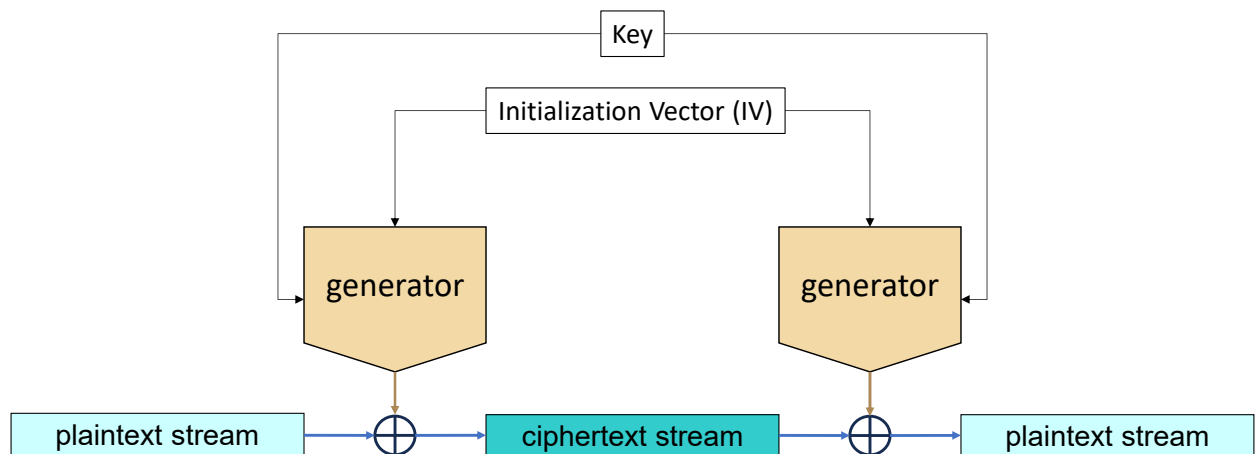
The core of any real stream cipher is a cryptographically secure pseudo-random bit generator, a CSPRNG, that produces the keystream. Internally this generator is a state machine: it has an internal state and, at each step, it emits bits and moves to the next state. Two inputs control it. The key determines how the state transitions, that is, how one state produces the next and the bits in between. The initialization vector, or IV, sets the starting state. Separating these two roles is important: the key is the long-term secret, while the IV is a per-message value that varies the starting point.

Cryptographically secure pseudo-random means two things, and the second is the one that really matters. First, statistically, the keystream looks like a genuinely random sequence of zeros and ones. Second, if an attacker ever learns part of the keystream, they must not be able to infer either the past or the future values of the keystream; knowing the middle must reveal nothing about the ends. This forward and backward secrecy is what makes a stream cipher safe against known-plaintext attacks.

Why does that second property matter so much? Because in the real world an attacker will often learn some of the keystream, for example from a guess about part of the plaintext, and if that lets them predict the rest, the whole message falls. A strong generator ensures that a leaked fragment is useless, it is like learning one page of an unguessable book without learning the others. So the generator is not a minor detail; it is the entire security of the stream cipher, and the next diagram shows exactly how that state machine produces the keystream and drives the XOR.

It is also worth connecting the IV and key roles to the reuse problem we will meet explicitly later. The key is the long-term secret shared between parties, while the IV is a per-message value that varies the starting state, and the two play different security roles. The key must stay secret; the IV must be fresh and unpredictable, but it does not need to be secret. Confusing these two requirements is a common mistake, and it leads to real bugs, such as reusing an IV and thereby reusing a keystream. Keeping the key and IV requirements separate in your mind is a small habit that prevents a large class of stream-cipher failures.

(Symmetric) stream ciphers: approach (diagram)



This diagram shows the structure we just described in motion. The key and the IV feed the pseudo-random generator, which maintains its internal state. At each step the generator outputs a block of keystream bits and updates its state to the next one, so the keystream is an endless chain of outputs, each depending on the previous state. The plaintext stream is XORed with this keystream bit by bit to produce the ciphertext. The whole encryption is just this loop: produce a keystream block, XOR it, advance the state, repeat.

Because the same key-and-IV state machine produces the keystream, decryption is simply running the identical generator again and XORing the ciphertext with the same keystream to recover the plaintext. That symmetry is the practical elegance of stream ciphers: the same simple loop encrypts and decrypts, which is why they are cheap to implement, even in constrained hardware. It also means the two parties only need to agree on the key and the starting state.

What the diagram makes clear is that the whole security rests on the generator. If its state machine can be predicted, if its outputs leak its internal state, or if the same key and IV are ever reused, then the keystream is no longer secure and the XOR protection collapses. So when you look at any stream cipher, focus your attention on the generator: how it is seeded, how it advances, and whether it has the forward and backward secrecy we just defined. Get the generator right and the cipher is strong; get it wrong and even a simple XOR cannot save you.

One more practical point from the diagram: because encryption and decryption run the identical loop, the two parties must agree on exactly the same starting point and the same key, or the keystreams will not match and decryption produces garbage. This is why the IV is sent openly alongside the ciphertext, so the receiver can reproduce the same keystream. It also means the failure mode is silent: a mismatching IV does not raise an error, it just decrypts to nonsense, which is why an integrity check is needed to catch it. The diagram shows not just how it works but where it can quietly fail.

Secure key & IV generation

- Keys and IVs must come from a cryptographically secure random source
 - Use a CSPRNG (OS random, /dev/urandom, hardware RNG)
 - Never use weak sources: time, process id, rand(), humans
- Weak entropy => predictable keys => cipher broken
 - Predictable keys shrink the effective key space
 - Poor entropy led to broken keys in the wild (weak SSH keys)
- IVs must be unique and unpredictable for each encryption
 - Reusing an IV with the same key reuses the keystream
 - A random per-message IV is the safest choice

SIO

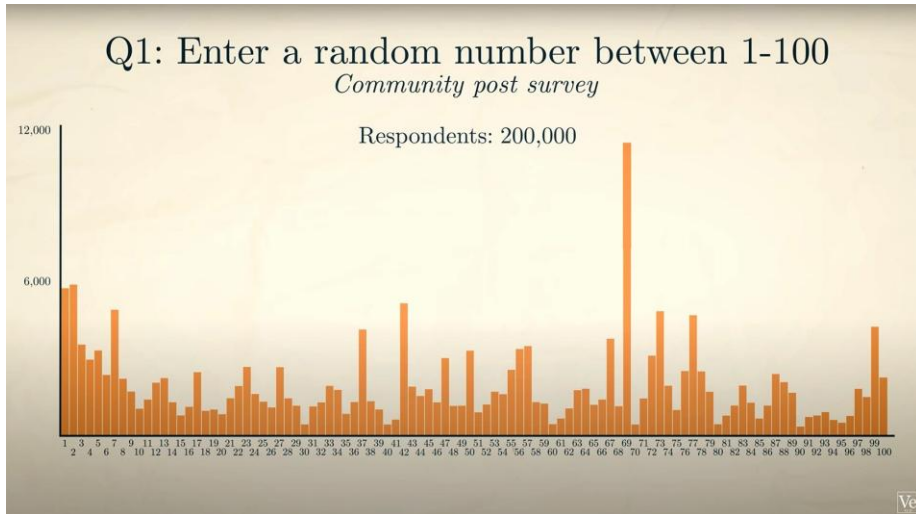
Everything we have said about stream ciphers only holds if the key and the IV are genuinely random, so where they come from is critical. They must come from a cryptographically secure random source, a CSPRNG such as the operating-system random interface, /dev/urandom on Unix systems, or a hardware random-number generator. Weak sources must be avoided: the current time, a process identifier, a naive rand() function, or values a human thinks up. These look random to a programmer but are highly predictable to an attacker who knows the system.

Weak entropy means the keys live in a far smaller space than they appear to, so an attacker can enumerate the likely candidates far more easily; predictable keys are effectively broken ciphers, no matter how strong the algorithm. This is not theoretical. There are documented real-world incidents of weak keys, such as certain weak SSH host keys generated from poor entropy, which let attackers take over systems. The lesson is that a strong cipher plus a weak key is a weak system, and the randomness is the part people most often get wrong.

The IV has its own distinct requirement: it must be unique and unpredictable for each encryption. Reusing an IV with the same key reuses the keystream, which, as we will see explicitly, destroys everything. A fresh random IV per message is therefore the safest choice, and it is worth emphasising because it is a common source of real bugs. So the practical recipe is simple to state and important to follow: use a proper CSPRNG for both the key and the IV, and never let a weak or human source dilute the entropy.

A practical way to operationalize this: treat randomness as an input you must verify, not assume. In code, the safe choices are the platform's documented cryptographic random interface, not the convenience random function in your language's standard library, which is often intended for shuffling, not for secrets. And when you are unsure where a key or IV came from, ask whether an attacker with the same context could predict it, because that is the exact test. Most real-world key breaks trace back to this step, not to the cipher, which is why it is worth the care.

Secure key & IV generation (with humans)



From: <https://www.youtube.com/watch?v=d6iQrh2TK98>

SIO

This is a short video illustrating the difficulty of getting genuine randomness from people. Humans are notoriously bad at producing unpredictable, uniformly distributed bits; when asked to be random, we fall into patterns and avoid numbers we find unlikely, which is exactly the bias a CSPRNG must not have. The point of the video is to make concrete why we never rely on human input for keys or IVs, and to build respect for the entropy sources that a computer can actually trust, the operating system and hardware generators.

In practice, when a person's contribution is unavoidable, for example when someone types a passphrase, we do not use that passphrase directly as a key. Instead we run it through a strong key-derivation function, a KDF, that stretches the limited human entropy into a full-strength key and resists guessing. This is a crucial distinction: a human passphrase is a low-entropy secret, and a KDF is what turns it into something with the full strength of a random key. The KDF also makes brute-forcing the passphrase expensive, which is the whole point.

The broader lesson is that randomness is a finite, precious resource, and the whole discipline of secure key and IV generation is about making sure the randomness you use is as close to ideal as possible, and never diluted by a weak or human source. It is a humbling reminder that the security of a system often depends not on the cleverest part but on the most unglamorous part: where the random bits actually come from. Treat that part with the same seriousness as the algorithm, because in the real world it is usually the part that fails.

The deeper lesson is about the limits of human contribution to security. Humans are good at choosing a secret that an attacker cannot guess, a passphrase, but they are bad at choosing random bits. So the right division of labour is to let humans provide secrecy and let machines provide randomness, and to join the two with a key-derivation function. Trying to use a human as a random-number source is asking them to do the one thing they are worst at. Recognizing this division is a practical skill: it tells you where a person is the right input and where a secure random generator is.

Stream ciphers: exploitation considerations

- No two messages (P1,P2) should be encrypted with the same key and IV
 - Because they will be encrypted with the same keystream (KS)
 - The knowledge about one message reveals the other

$$\begin{array}{l} C1 = P1 \oplus KS \\ C2 = P2 \oplus KS \end{array} \quad \longrightarrow \quad P2 = C2 \oplus KS = C2 \oplus C1 \oplus P1$$

- Knowledge about P1 => immediate knowledge about P2
 - Known/chosen-plaintext attacks become very effective!
- Keystreams may be periodic (have a cycle)
 - Depends on the type and quality of the generator
 - Same problem as the one above (KS is reused)
 - Plaintext should be shorter than the period length

The single most important rule for stream ciphers is: never encrypt two different messages with the same key and IV. If you do, both messages are XORed with the very same keystream. That has a dramatic consequence: if you ever learn one plaintext, you immediately recover the other, because the XOR of the two ciphertexts cancels the shared keystream and leaves the XOR of the two plaintexts. In other words, a known-plaintext or chosen-plaintext attack becomes devastatingly effective.

Work through why that is so powerful. You have $C1 = P1 \oplus KS$ and $C2 = P2 \oplus KS$. XORing the two ciphertexts gives $C1 \oplus C2 = P1 \oplus P2$, with the keystream gone. If you now know $P1$, you can recover $P2$ immediately. The keystream, which was supposed to be a secret, has effectively been exposed by its own reuse. This is one of the cleanest examples of how a correct algorithm used incorrectly is catastrophically broken, and it is why the reuse rule is stated so strongly.

There is a second, more subtle danger: some generators produce keystreams that are periodic, repeating after some cycle. If the period is short relative to the message length, the same keystream is reused within a single message, creating the same kind of weakness even without a second message. The defence is twofold: keep the plaintext shorter than the generator's period, and in practice use a fresh random IV for every message so the keystream is never reused at all. Both of these are reasons stream ciphers are more fragile than they first appear, and why careful use matters more than the choice of generator alone.

This reuse failure is worth stressing because it is so common and so catastrophic. It has broken real systems, and many stream-cipher deployments fail because a counter or IV is reset between messages. The lesson is that the security of a stream cipher is conditional on the key and IV being used correctly, and that condition is easy to violate in software, where state can be lost or reused. So the practical rule is not just do not reuse, but verify that your implementation actually generates a fresh IV for every message and never resets the keystream, because the algorithm alone cannot save a deployment that gets this wrong.

Stream ciphers: exploitation considerations

- Ciphertexts can be deterministically manipulated
 - Stream ciphers are simple and have no capability to detect manipulation
 - Each cipher bit depends only on one plaintext bit

$$C' = C \oplus \Delta \Rightarrow P' = P \oplus \Delta$$

- It is fundamental to have integrity control elements
 - In the ciphertext; or
 - In the plaintext
 - Objective is to detect accidental or malicious changes to P

A further weakness of stream ciphers is that the ciphertext can be manipulated deterministically, and the cipher itself cannot detect it. Because each ciphertext bit depends only on one plaintext bit via a simple XOR, an attacker can flip any bit in the ciphertext and know exactly which bit in the recovered plaintext will flip. There is no built-in integrity check; the cipher is a pure transformation, and it will happily decrypt a tampered block to a correspondingly altered plaintext with no indication that anything is wrong.

This is a general and important lesson that carries beyond stream ciphers: confidentiality and integrity are separate properties, and a plain cipher gives you only the first. An attacker who cannot read your message can still change it in a targeted way, for example flipping a bit that changes a transfer amount or a permission flag. The receiver, seeing a valid decryption, has no way to know the data was altered. This is why confidentiality alone is not enough for most real applications, which also need to detect tampering.

To protect against accidental or malicious changes you need an integrity-control mechanism, either embedded in the ciphertext as an authentication tag, or carried alongside in the plaintext as a message authentication code or a digest. We will see later how modern authenticated modes provide exactly this protection, combining the encryption with a tag that changes if any bit is altered. For now the key point is to hold the two goals separately in your mind: a plain cipher hides the data, and only an authenticated construction also guarantees the data has not been changed.

The reason this matters in practice is that confidentiality and integrity failures look different to an attacker. An attacker who cannot read your traffic still has a strong incentive to alter it, flipping a bit that changes a status flag or an amount, because the change is free and undetectable by a plain cipher. This is why any system that carries more than secrets, commands, amounts, permissions, needs integrity protection, not just encryption. The practical rule is to never ship a plain cipher where an attacker's ability to tamper matters, and to reach for an authenticated construction that makes tampering detectable.

Symmetric Block ciphers: examples

- **DES (Data Encryption Standard)**
 - Proposed in 1974, standard in 1977, nowadays deprecated
 - Input/output: 64-bit blocks
 - Key: 56 bits
- **AES (Advanced Encryption Standard)**
 - Proposed in 1998 (Rijndael), standard since 2001
 - Input/output: 128-bit blocks
 - Key: 128, 192 or 256 bits
 - Most commonly used symmetric cipher in applications
- **Other**
 - IDEA, CAST, Twofish, Blowfish, RC5, RC6, Kasumi, etc.

The two block ciphers you must know are DES (for historical reasons) and AES (the current workhorse). DES, the Data Encryption Standard, was proposed in 1974 and standardized in 1977; it works on 64-bit blocks with a 56-bit key. That key is now far too short, so DES is deprecated and should not be used for anything new. AES, the Advanced Encryption Standard, was proposed in 1998 as the Rijndael algorithm and has been the standard since 2001. The shift from one to the other is a central story of modern cryptography.

AES works on 128-bit blocks and supports 128-, 192- or 256-bit keys, and it is by far the most commonly used symmetric cipher in applications today. Beyond these there is a long tail of other designs, such as IDEA, CAST, Twofish, Blowfish, RC5, RC6 and Kasumi, each with its own trade-offs, but none has displaced AES as the default. The historical arc is instructive: DES was the first widely deployed standard, it aged badly because its key was too short, and AES was chosen through a public competition to replace it.

That story is a case study in the principle that a cipher's lifetime is limited by its weakest dimension. DES's design was sound, but its 56-bit key eventually became breakable, and no amount of cleverness in the rest of the algorithm could compensate. When you evaluate any cipher, the honest question is not whether the design is clever but whether its weakest link, usually the key length or the mode of use, is strong enough for the threat you face and the time horizon you need. That is the standard that led the world from DES to AES.

The contrast between DES and AES is also a lesson in how standards age. DES was not a bad design; it was a good design for its time with a key size that was eventually too small. That is the typical failure mode of a cipher: not a flaw in the algorithm but a dimension, usually the key, that the world outgrows. It is why modern standards choose key sizes with a long horizon in mind, and why 128-bit is now the floor rather than the ceiling. When you evaluate any cipher, the honest question is not is it clever but will its weakest dimension still be strong enough in ten or twenty years.

Key length vs security strength

- Security strength: the work (in bits) needed to break the cipher
 - n-bit strength means $\sim 2^n$ operations
- For symmetric ciphers, strength roughly equals key length (AES-128 $\sim$ 128 bits)
 - AES-128 $\sim 2^{128}$ work; AES-256 $\sim 2^{256}$
 - Brute force is the best known attack on AES
- Effective strength can be lower: 3DES 168-bit key $\sim$ 112 bits (2-key $\sim$ 80 bits)
 - Meet-in-the-middle attacks reduce effective strength
 - Key length is not always equal to security strength

SIO

Key length is not the same thing as security strength, and it is worth separating the two carefully. Security strength is the amount of work an attacker must do to break the cipher, measured in bits: n-bit strength means roughly 2^n operations. For a well-designed symmetric cipher the strength is roughly equal to the key length, so AES-128 gives about 128 bits of strength because the best known attack is simply brute force over 2^{128} keys. That equivalence is a good property, but it is not guaranteed.

But the effective strength can be lower than the nominal key size, and this is the point that trips people up. For example, triple DES with a 168-bit key only gives about 112 bits of effective strength, and about 80 with a two-key variant, because of a meet-in-the-middle attack. This is a concrete example of why you cannot just add up key sizes: three 56-bit keys do not buy you 168 bits of security, because the structure of multiple encryption creates a shortcut. The number you should care about is the effective strength, not the sum of the key lengths.

Historically, this matters because the 56-bit DES key was broken by a dedicated brute-force machine called Deep Crack in 1998 in roughly two days, which is exactly what the strength analysis predicted was feasible. That demonstration is what pushed the field towards 128-bit keys. 128-bit AES is secure today, with 256-bit recommended for long-term caution, especially for data that must remain secret for decades. The practical takeaway is to think in terms of strength, not key length, and to remember that a cipher's advertised key size is an upper bound on, not a guarantee of, its security.

A practical heuristic follows from all this: when a vendor or a standard quotes a key size, mentally convert it to effective strength, not to bits, and discount for known structural attacks. The meet-in-the-middle reduction of triple DES is the canonical example, but the principle is general: the structure of a construction can create shortcuts that no single key length captures. This is also why security advisories talk about strength levels rather than raw key sizes. Learning to think in effective strength, and to be suspicious of key lengths that add up to more than they should, is a small skill that prevents a common and expensive mistake.

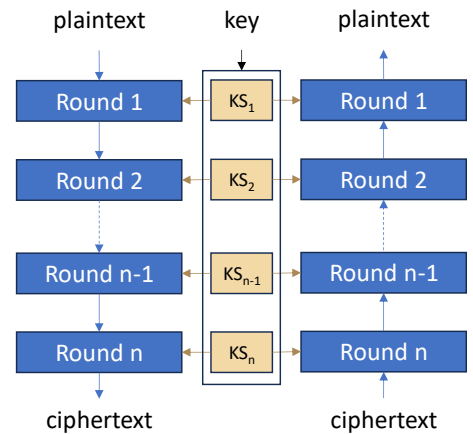
Symmetric block ciphers: approach

- Use a pipeline of transformation **rounds**

- Each round adds confusion and diffusion
- Each round is usually controlled by a **subkey**
 - Aka **key schedule**
 - A value derived from the encryption/decryption key

- Rounds need to be reversible

- To allow decrypting what was encrypted
- Make use of well-known structures:
 - Feistel networks
 - Substitution-permutation networks



A modern block cipher is built as a pipeline of transformation rounds. Each round applies a little bit of confusion and diffusion, and each round is driven by a round-specific subkey. The process of deriving those subkeys from the main encryption or decryption key is called the key schedule. The overall design philosophy is that no single round is strong enough, but many rounds of well-mixed transformation together destroy all exploitable structure. It is the repetition, not any single step, that creates the security.

Two design requirements matter. First, the rounds must be reversible, so that the encryption process can be run backward to decrypt; a block cipher that could not be reversed would be useless. Second, the rounds are built from well-known, well-studied building blocks rather than ad-hoc operations, because those blocks have been analysed and their contribution to confusion and diffusion is understood. Building from proven pieces is what lets a new cipher gain the community's trust.

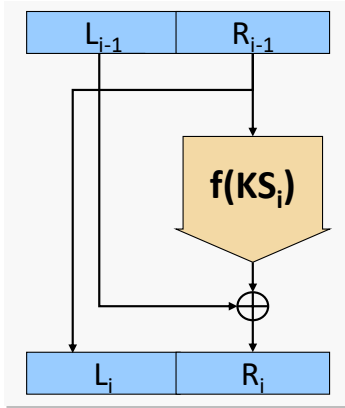
The two main families of building blocks are the Feistel network and the substitution-permutation network. A Feistel network is especially attractive because it lets you use any function inside the rounds without worrying that it must itself be reversible; the overall structure guarantees reversibility. A substitution-permutation network applies substitutions and permutations in a repeating pattern, with the S-boxes providing confusion and the P-boxes providing diffusion. Nearly every important block cipher, including AES and its predecessors, is built from one of these two structures, so understanding them unlocks the specific ciphers that follow.

The round-based structure also explains why ciphers are parametrized by rounds, and why more rounds are safer. Each round adds a little more confusion and diffusion, so the total security grows with the number of rounds, which is why AES uses more rounds for longer keys. There is, however, a subtlety: simply adding rounds to a weak round function does not fix a weak cipher, because the rounds all share the same structural weakness. The rounds are only as good as the building blocks inside them, which is why the design of the round matters as much as the count. Understanding this is what lets you read a cipher's specification and judge whether it is sound.

Feistel network

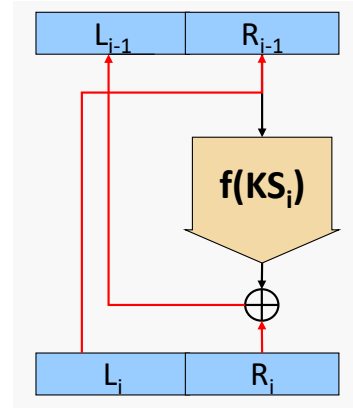
$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$



$$R_{i-1} = L_i$$

$$L_{i-1} = R_i \oplus f(L_i, K_i)$$



The function $f(KS_i)$ doesn't need to be reversible!

The Feistel network is the structure that made the use-any-function idea work, and it is one of the cleverest ideas in block-cipher design. In each round the block is split into two halves, and one half is combined with the output of a function applied to the other half and the round subkey. The elegant property is that the function does not need to be reversible at all. The clever arrangement of the two halves and the swaps guarantees that the whole round is reversible, so you can decrypt by running the same rounds with the subkeys in reverse order.

This is a genuinely powerful design freedom. In most of mathematics, if you want to be able to undo an operation, you need the operation itself to be invertible. The Feistel network shows that you can build a reversible transformation out of non-reversible parts, by arranging them in a specific symmetric pattern. This means the designer can choose the strongest, most confusing function available for the round function without worrying whether it can be inverted. That freedom is why Feistel ciphers are so popular.

DES is the classic example of a Feistel network, and many other block ciphers follow the same pattern. The key takeaway to carry forward is that the reversibility of the overall cipher is a structural property of the network, not a requirement placed on the individual round function. That is a very different and more flexible design constraint, and it is the reason a Feistel cipher can be built from strong one-way-looking components and still decrypt correctly. It is a small insight that explains a large family of ciphers.

The Feistel idea is worth appreciating because it is a general trick, not a DES-specific detail. It appears in a long line of ciphers, and its lesson, that you can build a reversible transformation from non-reversible parts by arranging them symmetrically, is a deep one. It frees the designer to use the strongest available function for confusion without worrying about invertibility, which is a real constraint in other designs. Once you see the pattern, you recognize it quickly in any Feistel cipher, and you understand why these ciphers are so popular and so amenable to analysis, which is exactly the property you want in a standard.

Substitution-Permutation network

- SBox – Substitution Box

- Table with an output for an input (index)

$$\text{output} = \text{SBox}[\text{input}]$$

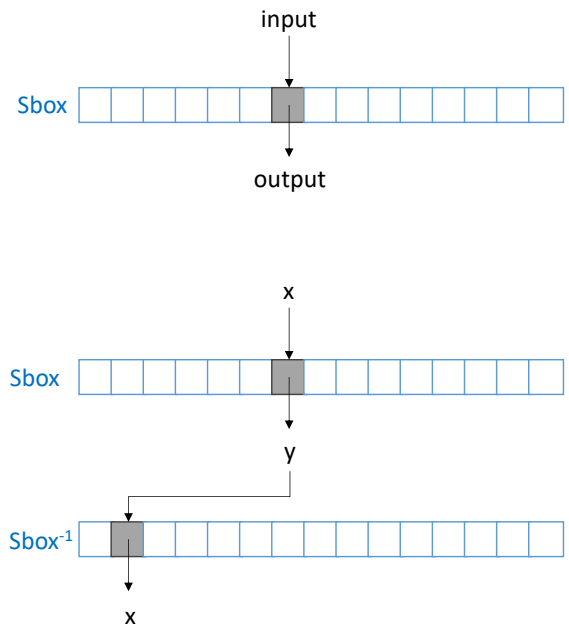
- SBoxes may be constant or key-dependent

- DES and AES use constant Sboxes
- Blowfish and Twofish use variable, key-dependent SBoxes

- In SP networks, SBoxes must be reversible

- Bijective transformations

$$y = \text{SBox}[x] \quad x = \text{SBox}^{-1}[y]$$



A substitution-permutation network is built from S-boxes and P-boxes, and the S-box is where the nonlinear power lives. An S-box, or substitution box, is a table: you feed in an index, the input, and you read out the corresponding value, so the output is simply the table entry for that input. S-boxes provide the confusion, the nonlinear substitution of the round. Nonlinearity is essential, because a purely linear cipher can be attacked with linear algebra, so the S-box is what defeats that class of attack.

S-boxes can be constant, fixed tables, as in DES and AES, or key-dependent, where the table itself is computed from the key, as in Blowfish and Twofish. The trade-off is interesting: constant S-boxes are simple and well-studied, while key-dependent S-boxes add more key involvement and can make some attacks harder, at the cost of more computation. In an SP network the S-boxes must be reversible, that is bijective, because the overall transformation has to be invertible for decryption.

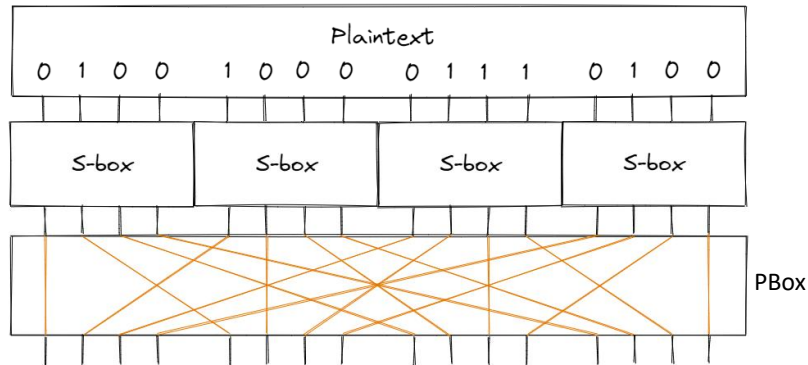
Mathematically, if the output y is the table entry for x , then x is the inverse-table entry for y ; a bijective table has exactly one entry per value, so it can be run in both directions. This reversibility requirement is a real design constraint that the S-box must satisfy, unlike in a Feistel round function. Together the S-boxes scramble the values and the P-boxes, which we look at next, shuffle the positions, and repeating them over many rounds is what gives the network its diffusion and confusion. So the S-box and P-box are the two hands of the same transformation, one changing values, the other changing positions.

The nonlinearity requirement is the key insight behind the S-box, and it is worth stating plainly: a cipher made only of linear operations, XORs and permutations, can be solved with linear algebra, so the S-box is what makes the cipher hard. Its job is to introduce the nonlinearity that defeats that class of attack. The fact that it must also be bijective, for reversibility, is a design constraint that the S-box must satisfy, unlike in a Feistel round. So the S-box sits at the intersection of two requirements, nonlinearity for security and bijectivity for reversibility, and getting both right is what makes an SP network work.

Substitution-Permutation network

- **Permutation Box**

- Changes the positions of the input bits
- Bits are not modified, only the position is modified



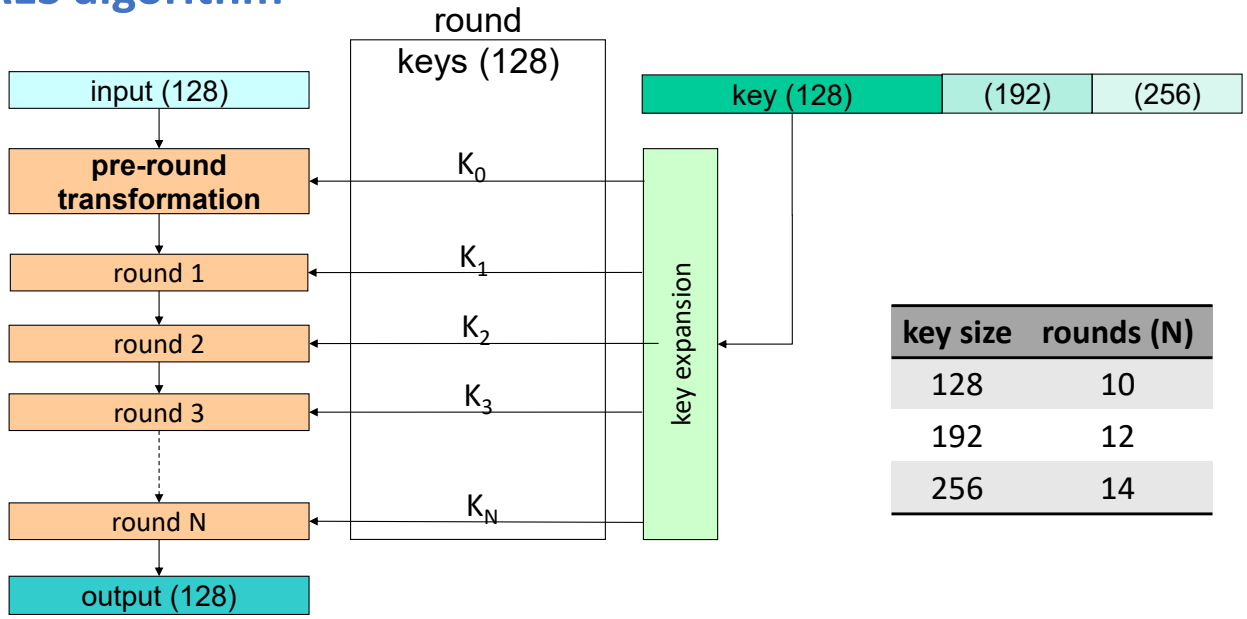
This is the other building block, the P-box, or permutation box. A P-box changes only the positions of the bits; it does not alter the bits themselves, only where they end up. This is the mechanism that provides diffusion in an SP network. By moving bits around between the S-boxes, a small change in one input bit is spread across many positions in the next round, so that it influences many S-box outputs rather than staying local. Without this shuffling, a change would propagate slowly and predictably.

Because a P-box is a fixed rearrangement of positions it is trivially reversible, you just apply the inverse permutation, so it does not complicate decryption. That is a nice property: the P-box gives you all the diffusion you want at essentially no cost to reversibility. The interplay is the whole point. S-boxes change the values, providing confusion, while P-boxes change the positions, providing diffusion, and alternating them over many rounds is what turns a simple input into a thoroughly mixed output where every bit depends on everything.

The diagram here is credited to the linear-cryptanalysis post by kevinliu.me, which illustrates the structure well. It is worth pausing to see how a few rounds of substitution-then-permutation spread a single flipped bit across the entire block. That visual is the avalanche effect we discussed, made concrete: it is why a block cipher can hide the location of a change in the plaintext. Together with the S-box, the P-box completes the SP-network toolkit, and it is exactly this toolkit that AES and its relatives use, just arranged in a slightly different geometry.

The P-box is easy to underestimate because it is so simple, but it is the component that makes diffusion actually happen. Without it, a change in one bit would stay in one place and propagate slowly, giving an attacker a local signal to work with. The permutation is what takes that local change and scatters it, so that after a few rounds every bit of the output depends on every bit of the input. That global dependence, the avalanche effect, is the property that hides where a change occurred, and it is the P-box, working with the S-box, that produces it. So the humble position-swapper is doing some of the most important work in the cipher.

AES algorithm



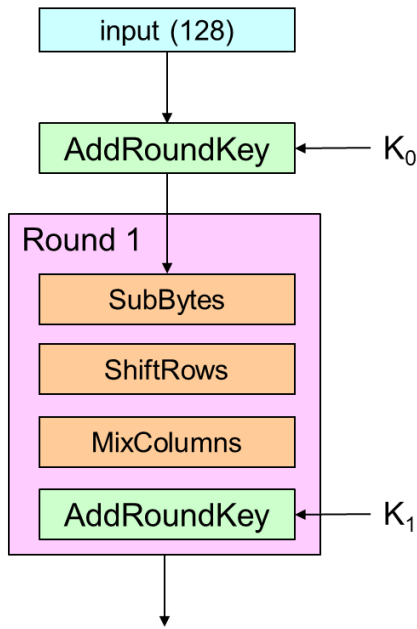
AES is a substitution-permutation network, and one of its defining features is that the number of rounds depends on the key size, as shown in this table. With a 128-bit key it runs 10 rounds, with 192-bit it runs 12, and with 256-bit it runs 14. More rounds means more confusion and diffusion and thus more security, which is why the longer the key, the more rounds are used. This is a clean illustration of the general design philosophy: security is bought with repeated, well-designed transformation, and the key size sets how many times you pay for it.

AES works on a 4 by 4 matrix of bytes, a 128-bit state, and each round applies a small sequence of operations to that matrix. Those operations, adding a round key, substituting bytes, shifting rows, and mixing columns, are exactly the confusion and diffusion steps we discussed, arranged into an SP-network structure. The matrix representation is a useful way to visualise what is happening, because each operation has a clear geometric meaning on the grid of bytes, and together they mix the values and positions in a well-studied way.

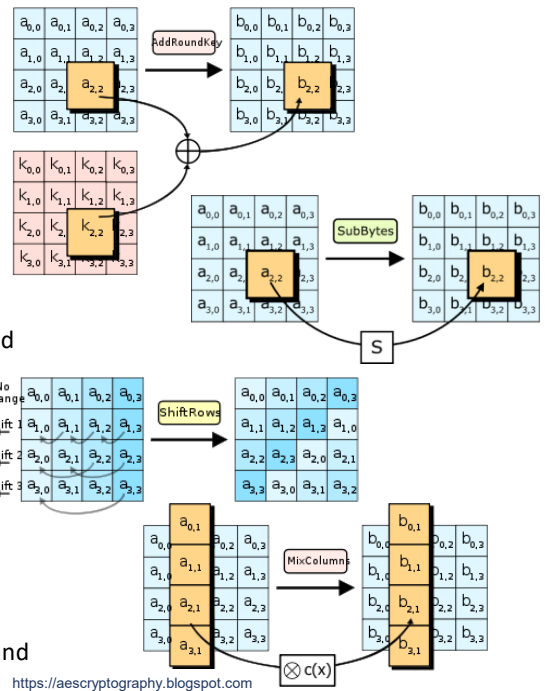
The slides that follow walk through each of those operations in turn, so it is worth keeping in mind the goal of each one before we meet it. AddRoundKey injects the key, SubBytes provides nonlinear confusion, ShiftRows starts to spread bytes across columns, and MixColumns completes the diffusion within each column. Seeing each operation as a deliberate step towards confusion or diffusion, rather than an arbitrary instruction, is what makes AES legible. It is a beautifully engineered balance of simple operations that, repeated a handful of times, produce a cipher that has withstood the entire world's cryptanalysis.

The table of rounds is a good place to internalize a general principle: security in a block cipher is purchased with repeated, well-designed transformation, and the parameter that controls how much you pay is the key size. Longer keys buy more rounds and therefore more mixing, which is why the security grows smoothly with the key size. It is also a reminder that AES is not a single cipher but a family, three ciphers that differ in rounds and key size, and that the 128-bit member is the one most people use. Seeing the rounds table this way, as a dial for security, connects the specific numbers back to the general design philosophy we have been building.

AES (encryption) round



- **AddRoundKey**
 - 128-bit XOR
 - Output is a 4x4 byte matrix
- **SubBytes**
 - 256-element S-box
 - Each matrix byte is substituted
- **ShiftRows**
 - Rows are rotated left
 - Byte shifts vary (0, 1, 2 & 3)
- **MixColumns**
 - Each column is transformed
 - Not performed in the last round



<https://aescryptography.blogspot.com>

João Paulo Barraca, André Zúquete

SIO

30

Each AES encryption round applies four operations in sequence, and it is worth understanding what each one contributes. AddRoundKey XORs the state with a round subkey, a 128-bit XOR that injects the key material; it is the only step where the key directly acts, and it is what ties the transformation to the secret. SubBytes replaces every byte of the state using a fixed 256-element S-box, providing the nonlinear confusion that defeats linear attacks.

ShiftRows cyclically rotates the rows of the matrix, by different amounts for each row, 0, 1, 2 and 3 byte shifts, which starts to spread the bytes across columns so that a byte in one column begins to influence the others. MixColumns then transforms each column mathematically, mixing the bytes within it to provide full diffusion, so that each output byte depends on all four input bytes of the column. One detail: MixColumns is deliberately skipped in the final round, because its mixing job is unnecessary after the last substitution and it would only complicate decryption.

Taken together, these four steps make one round of AES; the whole encryption is just a repeated application of this round, with a key addition before the first round and one after the last. Each step is individually simple, but repeated over ten to fourteen rounds they produce a very strong, well-mixed transformation. The elegance is that there is nothing exotic here, only XOR, table lookup, rotation and a linear mix, yet combined and repeated they form a cipher that has resisted the entire field. That is the essence of good block-cipher design: simple, analysable parts, applied with discipline.

One more thing to notice about the round is how deliberately each step targets one of the two goals. AddRoundKey connects the transformation to the key, SubBytes provides the nonlinearity, and ShiftRows plus MixColumns together provide the diffusion, with the column mix ensuring that each output byte depends on all four input bytes of its column. The fact that the final round skips MixColumns is a small but telling detail, a deliberate simplification that does not reduce security but makes decryption cleaner. Reading each step this way, as a purposeful move towards confusion or diffusion, is what turns AES from a sequence of operations into a design you can reason about.

AES in CPU instruction sets make encryption cheap/fast

- Intel AES New Instructions (AES-NI)

AESENC	Perform one round of an AES encryption flow
AESENCCLAST	Perform the last round of an AES encryption flow
AESDEC	Perform one round of an AES decryption flow
AESDECLAST	Perform the last round of an AES decryption flow
AESKEYGENASSIST	Assist in AES round key generation
AESIMC	Assist in AES Inverse Mix Columns

- ARMv8 Cryptographic Extension

- ... and other

A practical point that makes AES especially attractive is that modern processors have AES built into their instruction sets. Intel added the AES New Instructions, or AES-NI, extension, and ARM added a cryptographic extension in ARMv8, with others following. These dedicated instructions implement the core AES operations, the S-box substitution, the state mixing, and the key schedule, in hardware, so encryption that would otherwise take many software operations is done in a handful of fast cycles.

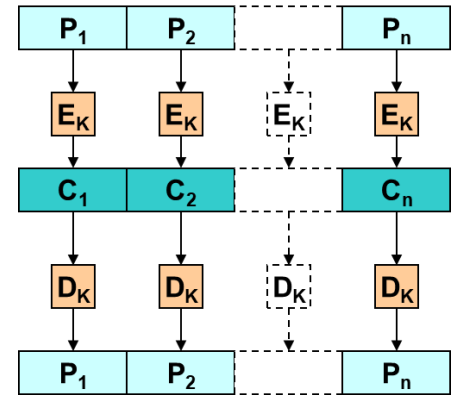
The consequence is that AES can be so fast that the cost of encrypting is no longer a meaningful bottleneck in most systems. This is a big reason AES became the universal default: it is not only cryptographically strong but also cheap to run, even on small and battery-powered devices. It is also why using AES is now such sound, low-effort advice; the performance objection that once favoured lighter or ad-hoc schemes has largely disappeared. When a standard algorithm is both secure and fast, there is little reason not to use it.

There is a deeper lesson here about how security actually gets adopted. A cipher that is theoretically perfect but slow will not be deployed, and people will reach for something faster, sometimes less secure. The fact that hardware makers built AES into the chips removed the performance excuse and pushed the entire industry towards a strong standard. It is a reminder that the success of a cryptographic standard depends not only on mathematics but on engineering and economics, and that making the secure choice also the fast choice is the single best thing you can do to get it adopted.

There is a strategic lesson here beyond performance. When the fast path is the secure path, insecure alternatives lose their main excuse. Before hardware acceleration, a common argument for rolling your own or using a lighter scheme was that AES was too slow; once hardware made AES fast, that argument evaporated, and the industry converged on the standard. This is a general pattern in security: the most secure choice usually wins in the end not just because it is secure but because it becomes cheap to run. It is a reminder that the best cryptographic advice is often the one that is also the easiest to implement correctly, and that making the secure option the default is the single most effective way to improve real-world security.

Cipher Modes: Electronic Code Book (ECB)

- Direct encryption of each block: $C_i = E_K(P_i)$
- Direct decryption of each block: $P_i = D_K(C_i)$
- Blocks are processed independently
 - Parallelism is possible
 - Uniform random access exists
- Problem:
 - Exposes patterns existing in the clear text
 - If $P_i = P_j$ then $C_i = C_j$



A cipher mode decides how to turn a block cipher, which only encrypts one block at a time, into something that can encrypt an arbitrary amount of data. ECB, Electronic Code Book, is the simplest possible mode: each plaintext block is encrypted directly and independently, and decryption is the reverse. Because the blocks are independent, you can encrypt them in parallel and you can jump to any block directly, so ECB supports parallelism and uniform random access. On paper, those are attractive properties.

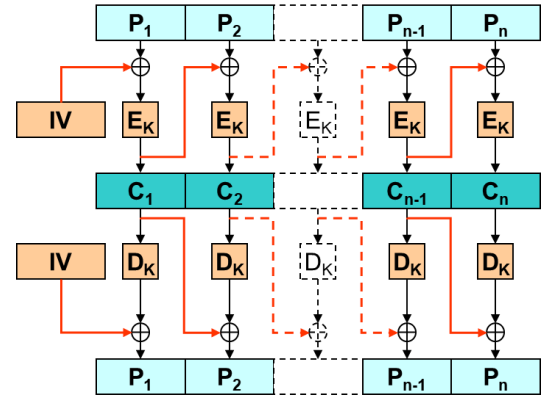
But ECB has a serious weakness that makes it unusable for real data: it exposes patterns in the plaintext. Because each block is encrypted on its own with the same key, if two plaintext blocks are identical then their ciphertext blocks are also identical. An attacker can therefore read the structure of the data even without decrypting it. The classic demonstration is encrypting an image in ECB mode, where the outline of the subject is still clearly visible in the ciphertext. That pattern leakage is a confidentiality failure, not a minor one.

The lesson is that simplicity is not the same as security. ECB is the easiest mode to understand and implement, which is part of why it appears in textbooks and in early implementations, but it is the one mode we should actively avoid in practice. It teaches us the most important lesson about modes: the mode, not just the cipher, determines whether your ciphertext hides the structure of your data. Every more sophisticated mode that follows is essentially a response to this specific weakness, so understanding ECB's failure is the foundation for understanding why the rest exist.

The reason ECB is so instructive, even though we never use it, is that it isolates the core idea of a mode: how do independent blocks relate to one another? ECB answers that they do not, which is both its virtue, parallelism and random access, and its fatal flaw, pattern leakage. Every other mode is essentially a different answer to that same question, introducing dependencies between blocks precisely to kill the pattern. So ECB is the baseline from which all the others are defined, and understanding it is the key to understanding why the more sophisticated modes exist and what they are trading off. It is the control experiment of the whole modes section.

Cipher Modes: Cipher Block Chaining (CBC)

- Encrypt each block T_i combined with C_{i-1} : $C_i = E_K(P_i \oplus C_{i-1})$
- Decrypt each block C_i combined C_{i-1} :
 $P_i = D_K(C_i) \oplus C_{i-1}$
 - Parallelism and uniform random access is possible (decryption)
- First block uses an IV (Initialization Vector)
 - Better not reuse for the same key
 - Random value, sequence value, etc.
 - May be sent openly
- Polyalphabetic transformation
 - The feedback prevents equal blocks from being equally processed
 - Seems like we have a different key per block



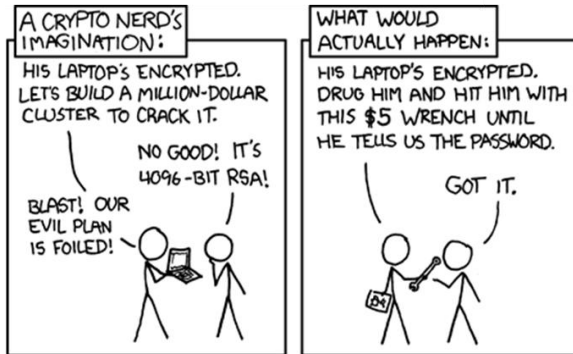
CBC, Cipher Block Chaining, fixes ECB's pattern problem by chaining the blocks together. Instead of encrypting a block on its own, CBC XORs each plaintext block with the previous ciphertext block before encrypting. Decryption reverses this: each plaintext block is the decryption of the ciphertext block, XORed with the previous ciphertext block. The very first block has no previous ciphertext, so it is XORed with an initialization vector, which should not be reused for the same key and can be sent openly alongside the ciphertext.

This chaining has a crucial effect: because each block depends on the one before it, two identical plaintext blocks now produce different ciphertext blocks. In substitution terms, CBC turns the monoalphabetic block cipher into a polyalphabetic transformation, it is as if each block were encrypted with a different, effective key. That is exactly the property that defeats ECB's pattern leakage. The price is a loss of independence: the blocks are no longer isolated, and the chaining introduces dependencies that the other modes handle in different ways.

One practical note worth remembering: in CBC, decryption can be parallelized and random access is possible, but encryption is inherently sequential, because each block needs the previous ciphertext before it can be computed. That asymmetry matters for performance in some settings. CBC was for a long time the default block-cipher mode, and it is still perfectly sound for confidentiality when used with a fresh random IV. It is a good example of a small, elegant modification, a single XOR with the previous block, that fixes a whole class of pattern-leakage problems.

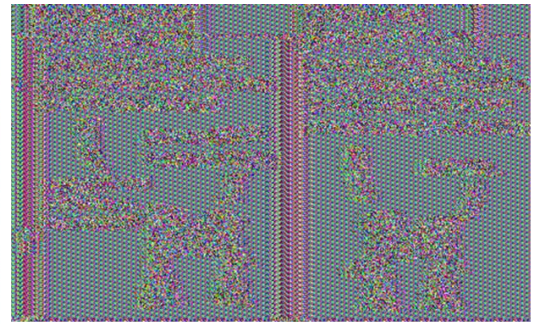
CBC is also a good place to appreciate the IV's role more deeply. The IV is the seed that makes the first block behave like the rest, so that the whole message is chained from a random start rather than from a fixed one. If you reuse an IV with the same key, the first block of two messages is processed identically, leaking whether they started the same, and weakening the rest of the chain. That is why the guidance is a fresh random IV per message, and why the IV can be sent openly, it is a nonce, not a secret. Getting the IV right is not a minor detail of CBC; it is the condition on which the whole mode's pattern-hiding depends.

ECB vs CBC: pattern exposure



<https://xkcd.com/538/>

ECB



CBC



This image makes the ECB-versus-CBC difference concrete and memorable, and it is worth studying. On one side, the data is encrypted in ECB mode and the structure of the original image survives; you can clearly see the silhouette of the subject in the ciphertext, because identical plaintext blocks produced identical ciphertext blocks. On the other side, the same image encrypted in CBC mode looks like uniform noise; the chaining has destroyed the block-level patterns, and nothing about the original is visible.

This is the single most important intuition to carry away about modes: the mode, not just the cipher, determines whether your ciphertext hides the structure of your data. ECB, using the same cipher and the same key, leaks the image; CBC, with no extra key material, hides it completely. The only difference between the two is the chaining. That makes the choice of mode as consequential as the choice of cipher, which is a point that is easy to underappreciate when everyone focuses on the algorithm.

It is also a vivid reminder that simply using AES is not enough; you must also use it in a mode that does not reveal patterns. A common mistake, especially in early code, is to reach for the simplest mode, ECB, because it is easy to implement, and then wonder why the ciphertext looks suspicious. This image is the antidote to that mistake. When you see a recognizable shape in your ciphertext, you are looking at ECB, and the fix is to switch to a chaining or counter mode. The visual is a tool you can use to spot a real, common bug at a glance.

The image is also a diagnostic tool you can use in practice, and it is worth building that habit. If you ever see structure in your ciphertext, recognizable shapes, repeating blocks, you are almost certainly in ECB mode or in a mode that is leaking patterns, and that is a bug to fix. Conversely, good confidentiality should look like uniform noise with no visible structure at all. So the test is almost visual: encrypt a known image and look at the output. If the subject is visible, the mode is wrong. It is a cheap, intuitive way to catch a common and serious configuration error, and one that many people would never think to run.

ECB/CBC cipher modes: contents not block-aligned

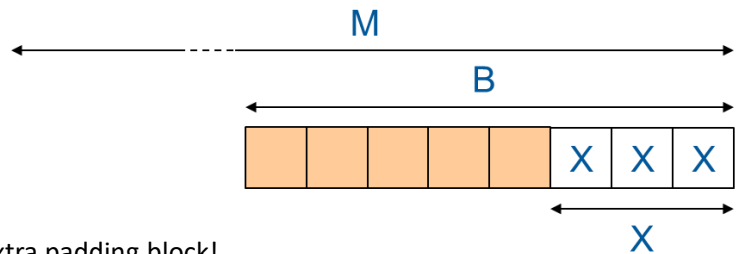
- ECB and CBC require block-aligned inputs
 - Final sub-blocks need special treatment

- Alternatives

- Padding
 - Of last block, identifiable
- Different processing for the sub-block
 - Adds complexity, rarely used

- PKCS #7 padding

- $X = B - (M \bmod B)$
- X extra bytes, with the value X
- PKCS #5: Equal to PKCS #7 with $B = 8$
- Drawback: perfectly aligned inputs get an extra padding block!



Block ciphers and modes like ECB and CBC require their input to be an exact multiple of the block size, but real data rarely lines up neatly. You need to deal with the final partial block, and there are two general strategies. The clean one, and the one almost always used, is padding: you fill the last block out to a full size in a way that can be recognized and removed later. The other option is to treat the final sub-block specially, which adds complexity to the cipher and is rarely used. Padding is almost always the right answer.

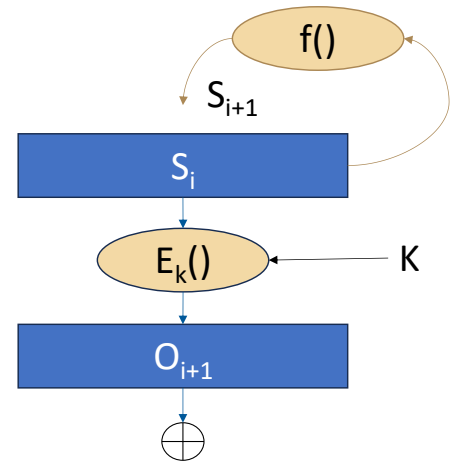
PKCS #7 is the standard padding rule, and it is elegant in its simplicity. You add X extra bytes, each with the value X , where X is computed as the block size minus the message length modulo the block size. For example, if you need five more bytes to complete the block, you add five bytes, each equal to five. This makes the padding self-describing, so the receiver knows exactly how much to strip, and it works for any message length. PKCS #5 is the same scheme with a fixed 8-byte block, used for 64-bit ciphers like DES.

One drawback worth noting: if the input is already perfectly block-aligned, PKCS #7 still adds a whole extra block of padding, so the padded output is always strictly longer than the input. That is a deliberate and safe choice, because it keeps the rule uniform and unambiguous. Padding is a small detail, but getting it wrong can cause real bugs, including padding-oracle attacks, which are a whole class of chosen-ciphertext attack we touched on earlier. So the humble act of filling the last block is not trivial; it is a place where correct, standardised handling matters for security.

Padding is also where a subtle but real attack class lives, the padding oracle. If a system, when given a ciphertext, tells you whether the padding is valid, an attacker can use that feedback, one bit of information at a time, to decrypt blocks without the key. This has broken real TLS implementations. The lesson is that even the unglamorous act of handling the last block is a place where a careless design leaks information. So the practical guidance is to use a standard padding scheme, to treat padding validation as part of the integrity check, and, even better, to prefer authenticated modes that make the whole class of padding-oracle problems disappear.

Stream cipher modes

- Stream ciphers use a pseudorandom generator
 - There are multiple techniques to implement them
 - Some techniques are specially suited for hardware implementations
 - Typically used in mobile, battery-powered devices
 - Other techniques are more suitable for CPU-based implementations
- Stream cipher modes
 - They use a block cipher to implement a stream cipher generator
 - The fundamental idea is:
 - The generator is a state machine with state S_i on iteration i
 - The output of the generator for state S_i is $O_{i+1} = E_k(S_i)$
 - The state S_i is updated to S_{i+1} using some transformation function f
 - S_0 is defined by an IV
 - The generator only uses block cipher encryptions (or decryptions)



So far we have looked at stream ciphers, built on a dedicated pseudo-random generator, and block ciphers as separate families. There is a clever bridge that unifies them: you can use a block cipher to build a stream-cipher generator. The idea is to model the generator as a state machine with a state at each iteration. The output for a given state is the block cipher applied to it, and the state is then updated to the next value by some transformation function, with the initial state set from an IV. The keystream is just the sequence of outputs.

Because the generator only ever uses block-cipher encryptions, or decryptions, you can get a keystream from any good block cipher, including AES. This is a powerful idea, because it means you do not need a separately designed stream cipher; you can derive one from the block cipher you already trust. Different choices for the update function give different modes, and each has different properties and trade-offs that we will see next.

Practically, some of these modes are better suited to hardware, for example the mobile and battery-powered devices that benefit from dedicated logic, while others favour CPU-based software implementations. This hardware versus software distinction matters in the real world, because it influences which mode ends up on which device. The modes that follow, OFB, CFB, CTR and GCM, are all different instantiations of exactly this state-machine-driven-by-a-block-cipher idea, so this slide is the template for all of them. Hold the general picture, and the specifics will slot into place.

The practical payoff of this unification is that it lets you build a stream from a block cipher you already trust, which is a powerful and economical idea. You do not need to design a new generator from scratch; you can derive one from AES. This is also why the modes differ mainly in how they update the state, because everything else, the block cipher, the XOR, is shared. So when you compare OFB, CFB, CTR and GCM, you are really comparing different state-update rules on the same underlying machine. Holding that common frame lets you see the differences as small, meaningful variations rather than as unrelated ciphers, which is the clearest way to master the family.

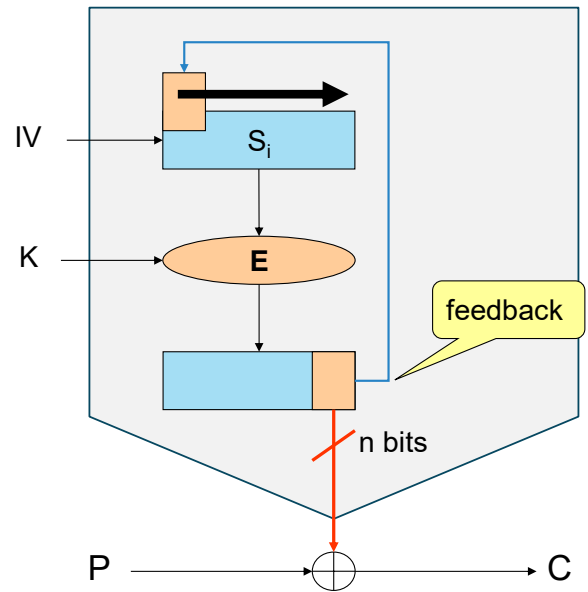
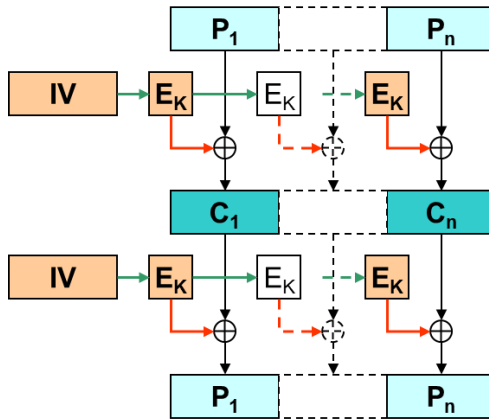
Stream cipher modes: n-bit OFB (Output Feedback)

$$C_i = T_i \oplus E_K(S_{i-1})$$

$$T_i = C_i \oplus E_K(S_{i-1})$$

$$S_{i+1} = f(S_i, E_K(S_i))$$

$$S_0 = IV$$



OFB, Output Feedback, is one instantiation of the block-cipher-as-generator idea. Here the block-cipher output is fed back to become the next state: the ciphertext is the data XORed with the block-cipher output of the previous state, and the next state is produced from the current state and its cipher output, with the initial state set to the IV. The important structural feature is what gets fed back, and in OFB it is the block-cipher output, not the data and not the ciphertext.

The key characteristic of OFB is that it is a true stream-cipher mode: it produces a keystream by repeatedly encrypting the state, and it XORs that keystream with the data. Because the keystream generation is independent of the actual data being encrypted, OFB can be used in a streaming fashion and, in principle, supports parallel pre-processing of the keystream, which is useful in some high-throughput settings. It can also operate on sub-blocks, n bits, rather than full blocks.

The price of this flexibility is that, like all pure stream modes, it provides confidentiality only. It does not by itself detect tampering with the ciphertext, so you need a separate integrity mechanism, such as a message authentication code, if you want that property. This is the recurring theme: a plain mode hides the data but does not guarantee it was not changed. OFB is a useful example of how the general state-machine template yields a practical, well-understood mode, and comparing its feedback rule to CFB's, which we see next, will sharpen your understanding of what the choice of feedback really changes.

A useful way to remember OFB is that the keystream is generated independently of the data, which is both its strength and its limitation. The strength is that you can pre-compute the keystream in parallel and even recover from errors, because the data does not feed back. The limitation is that, precisely because the data does not feed back, a tampered ciphertext changes exactly the corresponding bits of the plaintext and nothing else, with no error spreading and no detection. So OFB is a clean, parallel stream mode, but it is a confidentiality-only mode, and you must add integrity separately. That balance is the signature you should associate with OFB.

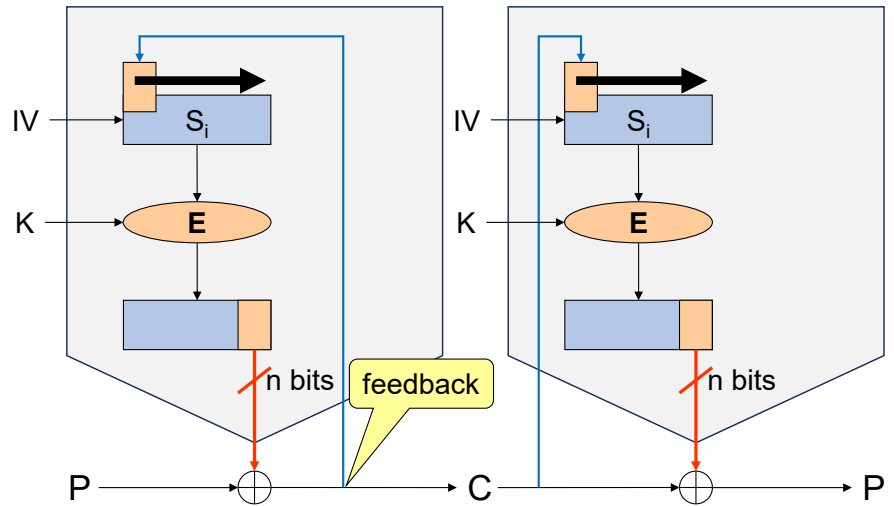
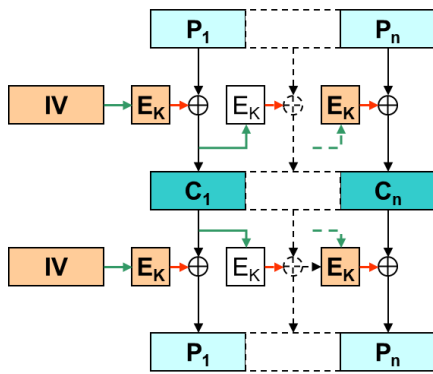
Stream cipher modes: n-bit CFB (Ciphertext Feedback)

$$C_i = T_i \oplus E_K(S_{i-1})$$

$$T_i = C_i \oplus E_K(S_{i-1})$$

$$S_{i+1} = f(S_i, C_i)$$

$$S_0 = IV$$



CFB, Ciphertext Feedback, is another block-cipher mode, and the difference from OFB is what gets fed back. In CFB the ciphertext, not the raw block-cipher output, is fed back to produce the next state. The ciphertext is the data XORed with the block-cipher output of the previous state, and the next state is produced from the current state and the ciphertext just generated, with the initial state set to the IV. That single change in the feedback, feeding the ciphertext instead of the cipher output, gives a distinct mode with different properties.

Because the ciphertext is part of the feedback, CFB is self-synchronizing in a useful way and can operate on smaller chunks, n bits, than the full block, which makes it flexible for various data sizes and hardware. Self-synchronizing means that if a bit of the ciphertext is lost or corrupted in transit, the synchronization error does not persist indefinitely, it decays after a bounded number of steps. That is a genuinely useful property for noisy communication channels.

Like OFB, CFB turns a block cipher into a keystream generator and thus provides confidentiality, but on its own it does not provide integrity; the ciphertext can still be manipulated in a way that deterministically changes the plaintext, so a separate integrity check is needed. CFB is a classic, well-established mode, and a good illustration of how a small change in the feedback rule gives a distinct, useful mode. When you study these modes, the most efficient way to tell them apart is to ask three questions: what is fed back, what can be parallelized, and does it provide integrity, and CFB has its own particular answer to each.

CFB's distinctive property is self-synchronization, which is genuinely useful in noisy channels. If a bit of the ciphertext is corrupted in transit, the effect on the decrypted output decays after a bounded number of blocks rather than persisting, because the corrupted ciphertext is fed back and then washed out by the subsequent correct feedback. That is a practical advantage over modes where an error propagates indefinitely. The cost is that, like OFB, CFB is a confidentiality-only mode, so you still need integrity separately. The takeaway is that the feedback rule is not an arbitrary choice; it is what gives CFB its self-synchronizing behavior, and that behavior is a real design feature, not a side effect.

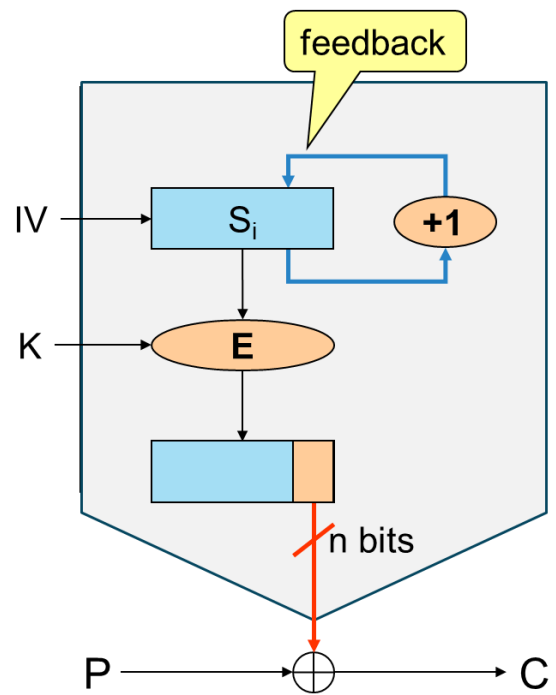
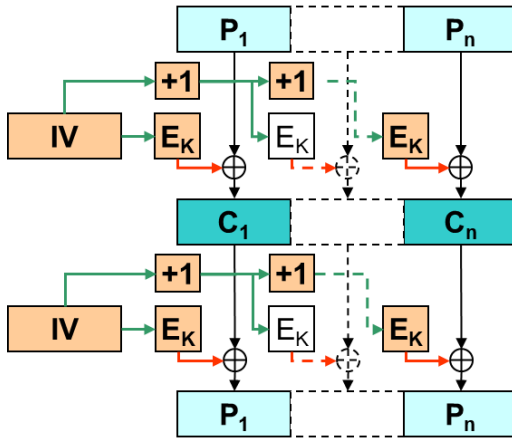
Cipher modes: n-bit CTR (Counter)

$$C_i = T_i \oplus E_K(S_{i-1})$$

$$T_i = C_i \oplus E_K(S_{i-1})$$

$$S_{i+1} = S_i + 1$$

$$S_0 = IV$$



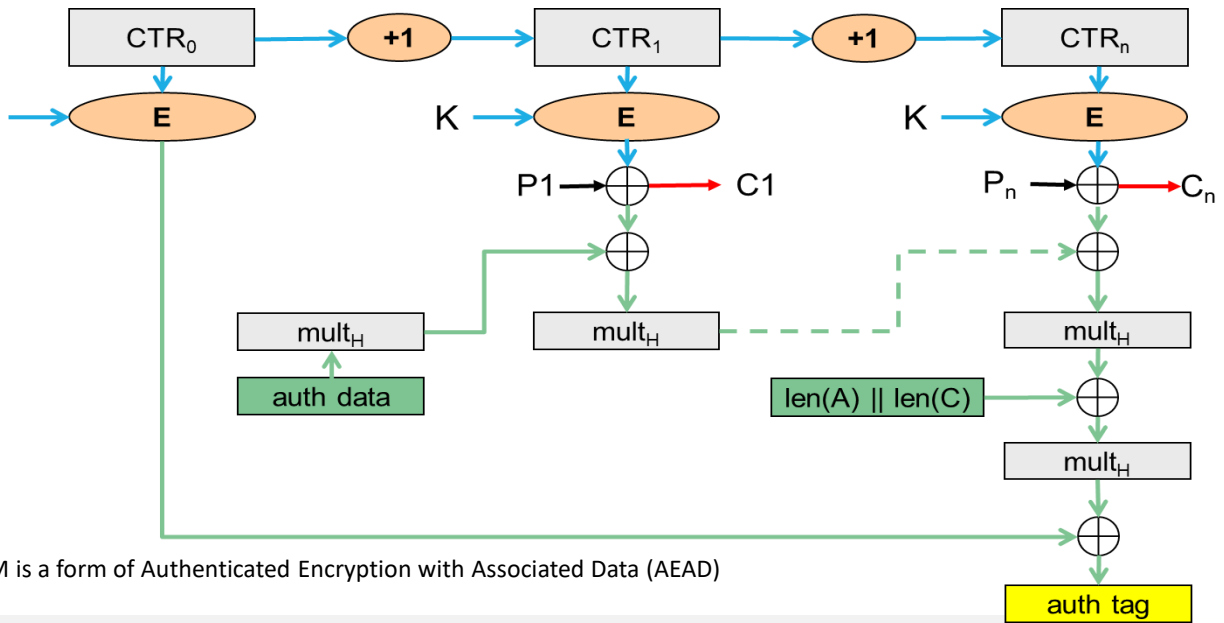
CTR, Counter mode, is one of the most important and widely used block-cipher modes, and its elegance is worth appreciating. Its trick is that instead of feeding back the previous state, you feed a counter. The state is simply a counter value, and each step you increment it, starting from an initial value set to the IV. The keystream is produced by encrypting each counter value, and the data is XORed with it. There is no chaining, no feedback, just a running counter.

This has several big advantages that make CTR stand out. Because each counter value is independent, every block can be encrypted in parallel, and you can jump directly to any block; so CTR gives both full parallelism and uniform random access, which is exactly what you want for fast, large, or random-access data. It also turns a block cipher into a stream cipher, so it handles data of any length without padding. For these reasons CTR is a workhorse mode and the confidentiality engine inside GCM.

The important caveat, consistent with everything we have said, is that CTR on its own provides confidentiality only, so integrity must be handled separately. And because the keystream depends on the counter, reusing the same key and the same counter, the IV, is just as disastrous as reusing a stream cipher's keystream, producing the same catastrophic cancellation we saw earlier. So the practical recipe for CTR is: use it for its parallelism and random access, pair it with integrity for full protection, and never reuse the key-counter combination. It is a mode that is both powerful and easy to get wrong if you forget those two cautions.

CTR is worth singling out because it is the mode that best combines the properties we have been asking for: full parallelism, uniform random access, no padding, and a clean turn of a block cipher into a stream. That is why it is the confidentiality core of GCM and a common choice on its own. But it carries the same two cautions as every stream-like mode: reuse of the key-counter combination is catastrophic, and confidentiality without integrity leaves the data tamperable. So the practical recipe for CTR is to use it for its speed and random access, never reuse the key and counter, and pair it with integrity when that matters.

Stream cipher modes: Galois with Counter Mode (GCM)



João Paulo Barraca, André Zúquete

SIO

40

GCM, Galois/Counter Mode, is a modern, recommended mode that combines the best of both worlds, and it is the default in many current systems. It uses CTR mode, as we just saw, to provide the confidentiality, and it adds an authentication mechanism based on a Galois-field multiplication to provide integrity and authenticity at the same time. That makes GCM a form of authenticated encryption with associated data, or AEAD: it encrypts your data and, in the same pass, produces an authentication tag that proves the data has not been tampered with.

GCM can also authenticate related data that you do not want encrypted, the associated data, such as protocol headers that should be verified but not hidden. The practical upshot is that GCM gives you both confidentiality and tamper detection from a single, well-studied construction, which is exactly what a modern protocol wants. It is fast, it reuses CTR's parallelism, it is authenticated, and it is simple to use correctly if you follow the standard. The authentication tag is the key feature: if any bit of the ciphertext or the associated data is altered, the tag check fails and the message is rejected.

This is why GCM is the default in TLS 1.3 and in many current systems, and it is a concrete example of the broader modern trend towards preferring authenticated modes over plain encryption. The design lesson is that integrity and confidentiality are best delivered together, in one well-reviewed construction, rather than bolted on as separate, error-prone pieces. When you are designing or reviewing a system today, the default answer to how should I encrypt this data should almost always be an AEAD mode like GCM or its software-friendly equivalent ChaCha20-Poly1305, rather than a bare encryption mode.

GCM is a good moment to state the modern default clearly: for new systems, authenticated encryption should be the starting point, not the exception. The reason is that most real data carries meaning beyond secrecy, and an attacker who cannot read it can still corrupt it, which a plain mode cannot detect. GCM removes the burden of wiring integrity separately by giving you a tag in the same construction. That is why it is the default in TLS 1.3 and in many APIs. The takeaway for practice is simple: unless you have a specific reason not to, choose an AEAD mode, and treat a bare encryption mode as something you must justify, not the default.

Cipher Modes: comparison

	Block		Stream			
	ECB	CBC	OFB	CFB	CTR	GCM
Input pattern hiding		✓	✓	✓	✓	✓
Same key for different messages	✓	other IV	other IV			
Tampering difficulty	✓	✓ (...)		(...)		✓
Pre-processing			✓		✓	✓
Parallel processing	✓	decrypt	With pre-proc	decrypt	✓	✓
Uniform random access						
Cryptogram single bit error propagation on decryption	same block	same & next block		a few next bits		detected
Capacity to recover from losses	some	some		some		detected

This table compares the block modes (ECB, CBC) and the stream-cipher modes (OFB, CFB, CTR, GCM) across several practical dimensions, and it is the best summary of when to use which, so it is worth reading carefully. On pattern hiding, ECB is the only one that fails; it leaks plaintext structure, while all the others hide it. On reusing the same key for different messages, the safe modes all require a different IV each time, whereas ECB has no IV at all and is simply unsafe.

On tamper detection, only GCM is authenticated by itself; the others give confidentiality and rely on external integrity, which means if you use them in a security-sensitive context you must add a message authentication code. On pre-processing, parallel processing and uniform random access, CTR and GCM stand out as the most parallel and the most random-access friendly, which is why they suit high-throughput and random-access scenarios, while CBC encrypts sequentially. The single-bit tamper row and the error-recovery row capture subtler differences in how a flipped bit propagates, which matters for noisy channels.

The practical conclusion the table drives home is that ECB is avoided, the older modes are fine for confidentiality when you also add integrity, and GCM, and its software equivalent ChaCha20-Poly1305, is the modern default because it gives you both security properties at once. When you are choosing a mode, this table is a decision aid: start from your requirements, do you need parallelism, random access, integrity, and then the short list narrows quickly. The skill is not memorising the table but learning to read it against a concrete set of requirements, which is exactly how the choice is made in practice.

The skill the table is really teaching is how to turn a set of requirements into a short list of acceptable modes. Start from what you need: do you need parallelism, random access, integrity, operation in noisy channels, and then the table narrows the field quickly. The trap to avoid is choosing a mode because it is familiar or easy to implement, rather than because it satisfies the requirements, which is exactly how people end up with ECB. So the table is less a reference to memorize and more a method to apply, and the method is: state the requirements, eliminate the modes that fail them, and let the survivors be your candidates. That is how the choice is actually made in real design work.

Cipher Modes: multiple encryption

- Invented for extending the lifetime of DES

- DES was never cryptanalyzed
- But its key was too short (56 bits only)
- Its key could be discovered by brute force

- Triple encryption EDE, or 3DES-EDE

$$C_i = E_{K3}(D_{K2}(E_{K1}(P_i))) \quad P_i = D_{K1}(E_{K2}(D_{K3}(C_i)))$$

- With $K1 \neq K2 \neq K3$, it uses a key with 168-bit of effective security
- With $K1 = K3 \neq K2$, it uses a key with 112-bit of effective security
- If $K1 = K2 = K3$, then we get simple encryption, although the key is 168 bits.
- In all cases, **3 times slower than DES**

Multiple encryption was invented to extend the lifetime of DES after its 56-bit key proved too short, and it is a good case study in how to buy more security by paying in speed and complexity. The idea is to apply the cipher several times with different keys. The standard form is triple DES in EDE form: encrypt, then decrypt, then encrypt, with the reverse order for decryption. The middle decryption is not arbitrary; it is chosen for backwards compatibility, so that triple DES with equal keys reduces to ordinary single DES.

The number of effective key bits depends on how the keys are related, and this is where the naive arithmetic fails. With three distinct keys you get 168 bits of nominal key, though a meet-in-the-middle attack means the effective security is a bit lower than 168. With the first and third keys equal but different from the second you get 112 bits. If all three are equal you have just done single encryption three times over, giving no real gain despite a 168-bit key. In every non-trivial case the cost is three times slower than plain DES.

Triple DES was a stopgap that kept DES usable until AES was ready, and its history is instructive on two counts. First, it shows that you can extend a cipher's life by layering, but at a real performance cost. Second, and more subtly, it shows that the effective security of a construction can be less than the sum of its parts, because the structure of multiple encryption admits a meet-in-the-middle shortcut. That is the same lesson as the key-length-versus-strength slide: count effective strength, not nominal key bits. Today you would almost always choose AES over triple DES, but understanding triple DES explains how the field bridged the gap.

The EDE structure, encrypt-decrypt-encrypt, is worth a moment because the middle decrypt is not a mistake. It is there for backwards compatibility: if all three keys are equal, the decrypt in the middle cancels the first encrypt and you are left with ordinary single DES, so the new mode can decrypt old single-DES data. That is a clever compatibility trick, but it also illustrates the general theme that multiple encryption is a pragmatic bridge rather than an elegant design. The real lesson is that once you understand why 3DES exists, you understand it as a historical compromise, strong enough to hold the line until AES was ready, and not a pattern you would choose for new work.

Cipher Modes: DESX

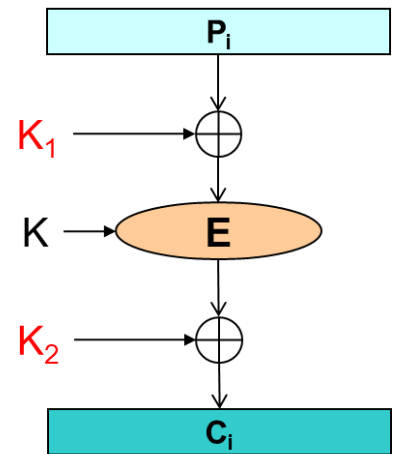
- Another solution for extending the lifetime of DES

- Much faster than 3DES
- Two extra keys are used to add confusion
 - Before the cipher input
 - After the cipher output

$$C_i = E_K(K_1 \oplus P_i) \oplus K_2 \quad P_i = K_1 \oplus D_K(K_2 \oplus C_i)$$

- The equivalent key length is 184 bits

- 64 + 64 + 56 bits
- But never a standard, and debated effective security



DESX was another way to extend DES's lifetime, and it is a useful contrast to triple DES because it takes a completely different approach. Instead of running the cipher three times, it runs it once and adds two extra 64-bit keys, XORed into the input before encryption and into the output after. This gives an equivalent key length of 184 bits: 64 plus 64 plus the original 56. The extra XOR keys add confusion at the edges of the transformation, making the effective key much larger without the triple speed cost.

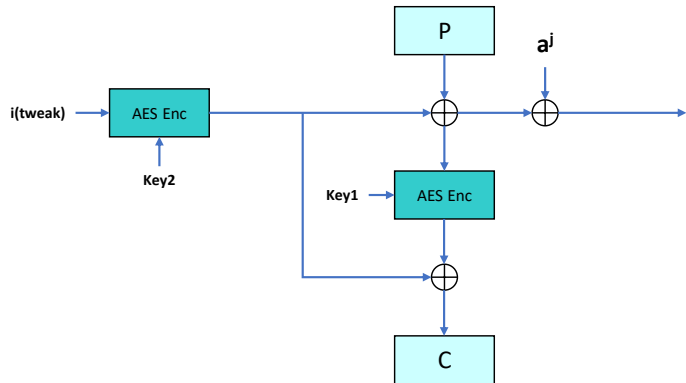
The appeal of DESX is clear: you get a much larger effective key at roughly the speed of single DES, rather than three times slower. That is a tempting trade. However, DESX was never adopted as a standard, and its effective security has been debated. In particular, it is vulnerable to certain meet-in-the-middle-style attacks that reduce its practical strength below the nominal 184 bits. So the 184-bit headline overstates the real security, which is a recurring caution: a larger key does not automatically mean a proportionally larger strength.

It is a useful example of a clever but ultimately not-quite-good-enough enhancement. It shows both the appeal of a fast extension and why the community still preferred to move on to a fresh, well-designed standard like AES, rather than patching DES in clever ways. The broader lesson is about how standards actually evolve: sometimes the right answer is not to fix the old thing as much as you can, but to recognise that a fundamentally new design, chosen through open competition, is the durable solution. DESX is the near miss, and AES is the winner, and the contrast between the two is a good lesson in engineering judgement.

DESX is a good example of a near miss, and near misses are instructive. It is clever and fast, and it does increase the effective key, but it was never standardized and its effective security has been questioned because of meet-in-the-middle-style attacks. So it illustrates two things at once: that a larger key does not automatically buy proportional strength, and that the community often prefers a fresh, well-reviewed standard to a clever patch on an old design. When you see a new enhancement to an old cipher, DESX is the cautionary template: ask whether it is strong enough to standardize, and whether a new standard is not simply a better answer.

Cipher Modes: XTS

- XEX-based Tweaked-codebook mode with Ciphertext Stealing
- Developed for at-rest data
 - Uses a known ID (Sector ID)
 - Hides Patterns with repeated disk sectors
 - Does not increase data size
- Uses two keys
 - Key 1: encrypts data
 - Key 2: generates whitening values
- Other arguments
 - $i(\text{tweak})$: storage sector number
 - a^j : a is 2, j an index.



XTS, XEX-based Tweaked codebook mode with Ciphertext Stealing, was designed specifically for encrypting data at rest, such as whole disks or storage devices. The key challenge there is that the same sector or block may contain the same data repeatedly, and a naive mode like ECB would leak those repetitions. XTS solves this with a tweak: it takes a known identifier, a sector or track ID, and uses a second key to generate a per-sector whitening value, which is combined with the data so that even identical sectors in different locations encrypt to different ciphertexts.

It uses two keys: one to encrypt the data, and one to generate the whitening values, and the separation of these roles is deliberate. XTS also keeps the data the same size, it does not grow it, which matters for storage where you cannot easily add padding, and the ciphertext stealing component is what lets it handle the final partial block without expanding the output. It hides the patterns caused by repeated disk sectors, which is exactly the problem that plagues at-rest encryption of real disks full of zeros and repeated structures.

In short, XTS is the go-to mode for disk and storage encryption: it gives pattern-hiding confidentiality for random-access, at-rest data, which is exactly the setting that block-storage encryption demands. The design lesson is that the right mode depends on the access pattern and the threat. XTS was built for the specific needs of storage, random access, fixed-size sectors, repeated data, and that is why it is the standard there rather than a general-purpose mode. When you later see full-disk encryption products, the mode working underneath is very likely XTS, and now you can see why it was shaped the way it was.

XTS is also a good example of how a mode is shaped by its setting. It was not designed to be a general-purpose mode; it was designed for the specific constraints of storage, fixed-size sectors, random access, repeated data, no size growth, and a known sector identifier. Every feature we described, the tweak, the two keys, the ciphertext stealing, exists because of one of those constraints. The lesson is that the right mode depends on the access pattern and the threat model, and that a mode built for storage is not the same as one built for transit. When you choose a mode, start from the setting, and you will usually arrive at the right answer, which in the at-rest case is XTS.

Choosing a cipher and mode

- Use a modern cipher (AES or ChaCha20) with a modern mode - **never ECB**
 - ECB leaks patterns; equal plaintext blocks give equal ciphertext
 - ChaCha20 is a modern stream cipher with excellent software speed
- **In transit:** AES-GCM or ChaCha20-Poly1305 (authenticated encryption)
 - AEAD combines encryption and integrity
 - Used by TLS 1.3 for transport security
- **At rest:** disk encryption, e.g. AES-XTS
 - XTS is designed for storage with random access
 - Each sector is encrypted with a tweak
- **Prefer authenticated modes:** they detect tampering
 - GCM and ChaCha20-Poly1305 detect malicious changes
 - Avoid unauthenticated modes unless integrity is handled separately

SIO

This slide brings everything together into practical advice, and it is the slide to remember. First, use a modern cipher, AES, or ChaCha20 for software-centric contexts, and a modern mode, and never, ever use ECB, because ECB leaks plaintext patterns and is the one mode we should actively avoid. ChaCha20 is worth calling out as a modern stream cipher with excellent pure-software speed, which is why it is a strong choice where AES hardware acceleration is not available, such as on many phones and embedded devices.

For data in transit, prefer authenticated encryption: AES-GCM or ChaCha20-Poly1305, both AEAD constructions that combine encryption and integrity, and both of which are exactly what TLS 1.3 uses for transport security. For data at rest, use disk encryption such as AES-XTS, designed for storage with random access and per-sector tweaks. The pattern is that the context, transit versus rest, points to a specific mode, and the default in both cases is an authenticated, modern construction rather than a bare encryption mode.

The overarching principle is to prefer authenticated modes that detect tampering, and to avoid unauthenticated modes unless you handle integrity separately, because leaving integrity to a separate, hand-wired piece is where real bugs live. Choosing correctly is what turns a strong cipher into a strong system, and it is a skill that comes from internalising the trade-offs we have covered. If you remember only three things from this lecture, let them be: use AES or ChaCha20, never use ECB, and prefer authenticated modes. Those three rules cover the vast majority of real decisions you will face.

The reason these three rules cover so much is that they encode the main failure modes we have seen. Using a modern cipher avoids weak or broken algorithms; avoiding ECB avoids the pattern-leakage that is the most common mode bug; and preferring authenticated modes avoids the integrity gap that a plain cipher leaves open. So the rules are not arbitrary; each one is a direct answer to a specific, common way systems fail. The deeper takeaway is that good cryptographic practice is mostly about avoiding a small set of well-understood mistakes, and that once you know those mistakes, the choice between the remaining sound options matters far less.

Summary

- **Symmetric ciphers:** stream vs block and security strength $\sim$ key length
 - Stream: XOR with a keystream; block: process fixed-size blocks
 - AES-128 $\sim$ 128-bit strength; 128-bit is the modern minimum
- AES (Rijndael) is the modern standard block cipher
 - 128-bit blocks, 128/192/256-bit keys
 - Hardware support (AES-NI) makes it fast
- Choose authenticated modes (GCM), and avoid ECB
 - GCM detects tampering; ECB exposes patterns
 - Use XTS for disk encryption
- Cryptography is only as strong as its key management and randomness
 - Use a CSPRNG and KDFs for keys
 - Never reuse keys or IVs

SIO

To close the lecture, here are the key ideas to carry forward, organised as a short mental model. Symmetric ciphers come in two families, stream ciphers, which XOR the data with a keystream, and block ciphers, which process fixed-size blocks, and for a well-designed symmetric cipher the security strength is roughly equal to the key length, so 128 bits is the modern minimum. AES, based on the Rijndael algorithm, is the standard block cipher: 128-bit blocks, 128/192/256-bit keys, and hardware support through AES-NI that makes it fast everywhere.

The mode matters as much as the cipher. Choose authenticated modes such as GCM and avoid ECB, and reach for XTS when you are encrypting disks. The reason is that a strong cipher used in a weak mode, or used with a reused IV, or a weak key, can still be broken, so the mode and the key management are not secondary concerns, they are part of the security. The difference between a secure system and a broken one is often not the algorithm but one of these surrounding choices.

And remember the meta-point that underlies all of it: a cryptographic system is only as strong as its key management and its randomness. Use a cryptographically secure random source and proper key-derivation functions for keys, and never reuse keys or IVs. Get those fundamentals right and the strong algorithms we have covered will protect your data; get them wrong and even the best cipher will fail. That is the final lesson of the whole lecture: the mathematics is only as good as the discipline around it, and the discipline is where you, as the engineer, actually have control.

If you take one thing from the whole lecture, let it be the separation of the algorithm from its use. The algorithms, AES, ChaCha20, the modes, are strong because the field has attacked them and they have held. But a system fails not when the algorithm is weak but when the use is wrong: ECB mode, a reused IV, a weak key, or missing integrity. That is why the closing point is that cryptography is only as strong as its key management and randomness, because those are the parts you control as the engineer. Master the algorithms and, more importantly, master the discipline around them, and you will build systems that are secure in practice, not just in theory.